MÁSTER DE AUTOMÁTICA E INFORMÁTICA INDUSTRIAL

Modelado y Reconstrucción de Fragmentos Arqueológicos a partir de Correspondencia de Patrones

Autor
Carlos Sánchez Belenguer

Director
Eduardo Vendrell Vidal

Valencia, 30 de octubre de 2010

Índice

Índice de figuras

Capítulo 1

Introducción

La correspondencia de patrones es un problema que ha sido estudiado durante muchos años y que sigue sin ser resuelto con un coste computacional acotado. Poder establecer de forma automática similitudes entre patrones permite dar solución a multitud de problemas, como la reconstrucción a partir de fragmentos, el ensamblado de piezas industriales, la síntesis de proteínas, el estudio del ADN…

La complejidad de la solución es una consecuencia inmediata de que no puedan realizarse búsquedas locales garantizando la exactitud de los resultados. Así, la dimensionalidad del problema estudiado implica una explosión combinatoria en el espacio de búsqueda que afecta gravemente a la eficiencia de las técnicas automáticas.

Existen en la actualidad dos grandes grupos de metodologías que permiten abordar el problema de la correspondencia: los algoritmos ingenuos y los aleatorios. Los primeros realizan una búsqueda exhaustiva, garantizando la exactitud de los resultados a expensas de costes computacionales muy elevados. Los segundos, reducen aleatoriamente la talla del problema minimizando los costes, aunque corren el riesgo de detener la búsqueda ante mínimos locales. Existen, además, técnicas híbridas que pretenden compensar las carencias de una metodología con las bondades de la otra, o técnicas semi-automáticas en las que un usuario introduce la inicialización de la búsqueda manualmente, restringiendo considerablemente el espacio de soluciones. Aunque cada una tiene sus ventajas y sus inconvenientes, ninguna soluciona globalmente el problema.

El presente documento pretende abordar la problemática asociada a las técnicas de correspondencia de patrones centrándose en el análisis de imágenes. Este planteamiento presenta dos ventajas principales: en primer lugar, la baja dimensionalidad del problema permite que el espacio de búsqueda sea lo bastante reducido como para poder operar de forma cómoda ante complejidades algorítmicas elevadas. En segundo lugar, trabajar sobre imágenes bidimensionales permite disponer de datos de prueba de forma cómoda y económica. Además, durante el proceso de desarrollo, operar en un espacio de dos dimensiones permite elaborar trazas gráficas que resultan de gran ayuda para comprender las características del problema abordado.

1.1. Precedentes

El desarrollo que en esta memoria se detalla surge como extensión a un proyecto precompetitivo de I+D para equipos de investigación de la Generalitat Valenciana que fue otorgado a la Escuela Técnica Superior de Ingeniería Informática (UPV) en el año 2008.

Bajo la denominación "*CATALOGARQ: Catalogación, Reconocimiento Y Clasificación De Piezas Arqueológicas*", este proyecto plantea un procedimiento de catalogación de fragmentos de pintura mural semi-automático y de bajo coste. En él, a partir de una imagen del anverso y del reverso de cada pieza, se lleva a cabo una extracción de características básicas y se vincula la información introducida por el usuario a cada fragmento. Posteriormente, mediante un proceso de búsqueda de correspondencias, se corrigen las alineaciones relativas de las dos imágenes disponibles, y se realiza una reconstrucción 3D automática de la pieza. Los datos asociados a cada pieza catalogada, así como las características extraídas de las imágenes capturadas, son elementos que se utilizarán para una posterior clasificación y ayuda a la reconstrucción final de la pieza original.

La motivación del proyecto deriva de la multitud de yacimientos arqueológicos existentes en la Comunidad Valenciana, de los que se extraen una gran cantidad de piezas y restos de diversa índole que, de acuerdo a las normativas vigentes, deben ser clasificados antes de su exposición en museos o de su posterior almacenamiento.

No obstante, la arqueología ha sido tradicionalmente un campo con una metodología muy artesanal, que no cuenta con demasiadas herramientas informáticas que den soporte al trabajo que realizan los arqueólogos. Así, el procedimiento de catalogación y clasificación suele ser manual, apoyándose en formatos de ficha más o menos estandarizados o bien en base a desarrollos particularizados. Este proceso conlleva un ingente trabajo al que se dedican los arqueólogos y técnicos que trabajan en los yacimientos.

Recientemente, han comenzado a surgir técnicas de catalogación y clasificación más o menos automatizadas, que se apoyan en de dispositivos de adquisición de datos como sensores de visión o escáneres 3D. En [1][2] se comentan dos proyectos actuales de aplicación de estas técnicas que ilustran el potencial que pueden ofrecer las herramientas informáticas aplicadas a esta temática.

Disponer de una base de datos con imágenes e información asociada a fragmentos permite llevar a cabo procedimientos automáticos que ayuden al proceso de reconstrucción. El desarrollo de una técnica de búsqueda que permita comparar estos datos permitirá acelerar considerablemente el proceso de recuperación del patrimonio, a la par que reducir los costes de manera considerable, ya que las personas únicamente deben intervenir en la etapa inicial de adquisición y catalogación de información vinculada a los fragmentos.

1.2. Objetivos

Partiendo de la experiencia adquirida durante el desarrollo de *CATALOGARQ*, en este documento se pretende desarrollar una técnica de búsqueda de correspondencias entre patrones obtenidos a partir de fragmentos arqueológicos.

Tal y como se comenta en [4], uno de los aspectos más cruciales en la resolución de problemas de correspondencia de formas tiene que ver con la manera en que se representan éstas. Por ello, deberá estudiarse con detenimiento el criterio de representación de los contornos, en búsqueda de un sistema de coordenadas que facilite este proceso, a la par que garantice la exactitud de los resultados.

Pese a que la aplicación final de los resultados se llevará a cabo en un ámbito muy específico (reconstrucción de fragmentos arqueológicos), el problema de la correspondencia deberá afrontarse de manera general. De este modo, los resultados obtenidos podrán ser aplicados a otros ámbitos, o extendidos a problemas similares pero de mayor dimensionalidad.

En este sentido, pese a que el objetivo principal de este proyecto sea desarrollar una técnica formal para afrontar problemas de correspondencia sobre datos de entrada bidimensionales, resulta también importante sentar bases sólidas que permitan afrontar problemas tridimensionales en una futura tesis doctoral.

Respecto a la complejidad algorítmica de los resultados obtenidos, ésta debería ser competitiva con técnicas similares que hayan sido desarrolladas previamente. Para poder llevar a cabo esta comprobación, será preciso documentar minuciosamente el estado del arte, y ser capaz de caracterizar el problema solucionado dentro del contexto general de la correspondencia de patrones.

1.3. Estructura de la memoria

Este documento comienza con un análisis exhaustivo del estado del arte en técnicas de correspondencia de patrones. El objetivo principal de este capítulo será ofrecer una perspectiva general de los avances en este terreno, así como presentar la caracterización de las distintas metodologías existentes y los costes computacionales asociados a ellas. En base a los conceptos que aquí se establezcan, será posible categorizar adecuadamente la técnica que se desarrolle, y así poder contrastar los resultados obtenidos con los de otras técnicas similares. Por otro lado, disponer de una perspectiva general del problema resultará de gran utilidad al lector para comprender la magnitud del problema afrontado, y familiarizarse con los términos específicos de esta disciplina.

Una vez presentado el estado del arte se procederá a una descripción formal de la técnica desarrollada en el capítulo 3. Así, en primer lugar, será necesario especificar el problema abordado en base a los criterios introducidos en el capítulo 2, y caracterizar la técnica. Hecho esto, se comentará el sistema de representación escogido que, en gran medida, condicionará la calidad de los resultados que se obtengan y determinará la forma en la que se lleva a cabo el resto de operaciones. Posteriormente, en base a las características específicas de la representación de los patrones, se comentará el algoritmo de búsqueda de alineaciones y cálculo de correspondencias. Para finalizar el capítulo, se calculará la complejidad computacional teórica del algoritmo desarrollado, con el fin de poder contrastar los resultados con los de técnicas similares.

Una vez comentada la técnica desarrollada, se estudiará su implementación sobre un lenguaje de programación en el capítulo 4. Dicha implementación será fiel a todos los conceptos introducidos, y servirá para corroborar la validez de los resultados obtenidos, así como los cálculos de complejidad teóricos que cierran el capítulo 3.

Finalmente se presentarán las conclusiones alcanzadas durante el desarrollo del proyecto y se valorará en qué medida se han alcanzado los objetivos presentados en este capítulo. Dado que el presente desarrollo pretende ser el punto de partida para la elaboración de una tesis doctoral, se incluirá un último apartado en el que se sugerirán ampliaciones futuras que incrementen el potencial de la técnica desarrollada.

Capítulo 2

Estado del arte

2.1. Introducción

La correspondencia de patrones ha sido un área activa de investigación durante muchos años y cuenta con aplicaciones en diversas disciplinas (como visión por computador y biología molecular). En correspondencia de patrones, el objetivo consiste en llevar a cabo una transformación sobre una forma y medir su similitud con otra forma, apoyándose en alguna métrica específica de similitudes. Las propiedades de esta métrica dependen del problema concreto de correspondencia que se pretenda solucionar [3].

En líneas generales, existen dos métodos para la comparación de imágenes: los basados en intensidades (color y texturas), y los basados en la geometría (forma) [4]. En la actualidad las investigaciones se centran más en el análisis de la forma, y la disciplina se conoce como *"correspondencia de patrones"* o *"correspondencia de formas"*. Pese a que estos términos se emplean indistintamente en la literatura, visto desde un punto de vista más formal, una forma asociada a un patrón es uno de los múltiples resultados que se pueden obtener al aplicar alguna transformación sobre éste.

En la actualidad, la búsqueda de correspondencias entre patrones está considerada como uno de los campos más complejos dentro de las técnicas de búsqueda. De hecho, sistemas como el QBIC de IBM, tal vez uno de los sistemas de búsqueda sobre imágenes más avanzado, proporciona resultados relativamente correctos en búsquedas indexadas por color y textura, mientras que la calidad de éstos se reduce drásticamente cuando se llevan a cabo las búsquedas indexadas por forma.

Existen numerosas técnicas para abordar el problema de la correspondencia, pero este documento se centra únicamente en los métodos propios de la computación geométrica. Ésta es una sub-área de la algorítmica que trata del diseño y análisis de algoritmos que operan sobre problemas geométricos, manipulando objetos como puntos, líneas, polígonos y poliedros. El objetivo consiste en el desarrollo de algoritmos que proporcionen soluciones exactas, eficientes y correctas. Los aspectos más cruciales en este tipo de soluciones tienen que ver con la forma en que se representan los patrones, las transformaciones posibles que se consideran, y la métrica de similitud empleada.

2.1.1. Problemática asociada

Dadas dos formas y una métrica de similitud, los tipos de problemas a los que se han de enfrentar las técnicas de correspondencia de patrones se pueden clasificar como:

1. Problemas de computación: relacionados con el cálculo de la similitud entre dos patrones.
2. Problemas de decisión: dado un umbral de tolerancia, es necesario decidir si una similitud es mayor o menor que éste.
3. Problemas de decisión: dado un umbral de tolerancia, decidir si existe una transformación que aumente la similitud entre la forma transformada y la otra forma, para que esta quede por encima del umbral establecido.
4. Problema de optimización: encontrar la transformación que maximice la similitud entre la forma transformada y la otra forma.
5. Aproximación del problema de optimización: a menudo, la complejidad de solucionar los problemas anteriores es extremadamente elevada. Para estos casos se emplean algoritmos aproximados cuyo objetivo es encontrar una transformación que permita que la similitud entre ambas formas esté a una distancia constante de la similitud máxima.

Existen, además, múltiples variaciones sobre los problemas planteados. Un patrón puede ser comparado con otro patrón, o con varios, en cuyo caso será necesaria una estructura de indexado que permita acelerar las comparaciones. Otra variación consiste en tener en cuenta perturbaciones, como puede ser el ruido en la imagen, o el error cometido durante un proceso de discretización, que implican llevar a cabo correspondencias parciales.

2.1.2. Aplicaciones

Las soluciones que proporcionan las técnicas de correspondencia de patrones pueden ser aplicadas en diversos campos. En términos generales, éstos pueden clasificarse como [4]:

- Recuperación de formas: búsqueda de formas sobre un conjunto de formas, que sean similares a una forma dada. Para llevar a cabo estas búsquedas, se suelen emplear estructuras de indexación (basadas en las propiedades de las formas), que permitan llevar a cabo exclusiones de forma rápida durante el proceso de comparación.
- Reconocimiento y clasificación de formas: permiten determinar si una forma dada es suficientemente similar a otra forma o, en su defecto, encuentran la forma más similar dentro de un conjunto de formas disponibles. Dentro de esta categoría se engloban, entre otras, las técnicas de reconocimiento de caracteres, reconocimiento del habla…
- Alineación: llevan a cabo la transformación de una forma dada para obtener la mejor correspondencia con una segunda forma. Esta área de aplicación se corresponde con el objetivo del presente documento, que pretende definir una técnica para reconstruir restos arqueológicos fragmentados mediante correspondencia de patrones.
- Aproximación de formas y simplificación: el objetivo consiste en crear nuevas formas a partir de una dada, cuya complejidad sea menor, pero que mantenga una gran similitud con la forma original.

- Reconstrucción de superficies variantes en el tiempo: se basa en la recuperación de un único modelo que represente una forma y su proceso de deformación, a partir de gran cantidad de datos recogidos en diferentes intervalos de tiempo.
- Interpolación de formas: el objetivo es transformar gradualmente una forma en otra, teniendo en cuenta que esta transformación debe satisfacer ciertos requisitos.
- Modelado estadístico: el análisis de estructuras anatómicas como órganos y huesos se simplifica considerablemente si se disponen de un modelo estadístico. Éstos resultan útiles para la extracción de formas a partir de imágenes, ya que son capaces de describir variaciones válidas en el aspecto y tamaño de la forma. Estos modelos pueden ser construidos mediante un conjunto de formas que representen la misma estructura, por lo que resulta importante poder establecer la correspondencia entre éstas.
- Detección de cambios: el objetivo de estas aplicaciones consiste en mantener un registro de las variaciones que se producen sobre una forma. Los ejemplos más típicos se encuentran en el terreno médico y en sensorización remota.

2.1.3. Medidas de similitud

Por lo general, las medidas de similitud deben tener ciertas propiedades específicas que las conviertan en útiles para la correspondencia de contornos, pese a que estas propiedades puedan variar según la aplicación desarrollada. Tómense como ejemplo tres formas *X*, *Y* y *Z*, entre las que la distancia se expresa como $d(X, Y)$. Las propiedades básicas que debe tener una medida de similitud se pueden clasificar como:

1. Propiedades métricas: necesarias para poder establecer una relación de orden.
    - No negatividad: $d(X, Y) \geq 0$.
    - Identidad: $d(X, X) = 0$.
    - Unicidad: $d(X, Y) = 0 \rightarrow X = Y$.
    - Desigualdad triangular: $d(X, Y) + d(X, Z) \geq d(Y, Z)$
    - Simetría: $d(X, Y) = d(Y, X)$

    Si una función de distancia cumple la identidad, unicidad y desigualdad triangular, se dice que dicha función es métrica. Si únicamente satisface la desigualdad triangular, se dice que se trata de una semi-métrica.
2. Propiedades de continuidad: en las medidas de similitud, la robustez se considera como una forma de continuidad. Éstas, permiten a las medidas de similitud ser robustas ante errores de discretización o la presencia de perturbaciones en las medidas.
3. Propiedades de invarianza: una función de distancia *d*, se considera invariante bajo un grupo de transformaciones *G*, si $\forall g \in G, d\big(g(X), g(Y)\big) = d(X, Y)$

Las medidas más comunes de similitud son las siguientes:

- Métrica Discreta: d(X, Y) = 0, si X es igual que Y. En otro caso d(X, Y) = 1. Una desventaja de esa métrica es que si una forma X es ligeramente distorsionada, formado una nueva forma X', la distancia discreta d(X, X') siempre será 1. Emplear la métrica discreta para obtener la mínima d(X, Y) sobre un conjunto de transformaciones G es equivalente a buscar la transformación g ϵ G tal que g(X) = Y. Esto se conoce como "match de congruencia exacta".
- Distancia de Minkowski (distancia L_p): dados dos puntos *x*, *y* en R^k, la distancia L_p se define como: $L_p(x, y) = \left(\sum_{i=0}^{k} |x_i - y_i|^p\right)^{\frac{1}{p}}$.
- Distancia de cuello de botella: asumiendo que *A* y *B* son dos conjuntos de puntos de tamaño *n*, y con *d(a, b)* como la distancia entre dos puntos *a* y *b*, la distancia de cuello de botella *F(A,B)* es el mínimo sobre todas las correspondencias uno a uno *f* entre *A* y *B* de la distancia máxima *d(a, f(a))*.
- Distancia de Hausdorff: sean *P* y *Q* dos conjuntos de puntos en R^d. La distancia dirigida de Hausdorff desde *P* hasta *Q*, expresada como *h(P, Q)*, es:

$$\max_{p \in P} \min_{q \in Q} \|p - q\|$$

  La distancia de Hausdorff entre *P* y *Q*, expresada como *H(P, Q)*, es:

$$max\{h(P, Q), h(Q, P)\}$$

  Intuitivamente, la función *h(P, Q)* encuentra el punto $p \epsilon P$ que está más alejado de cualquier punto en *Q*, y mide la distancia desde *p* a su vecino más próximo en *Q*. No obstante, esta distancia es muy sensible al ruido. Como alternativa, para mejorar la robustez, se suele emplear la distancia parcial de Hausdorff, definida como:

$$H_k(P, Q) = max\{h_k(P, Q), h_k(Q, P)\}$$

  donde $h_k(P,Q)$ es el k-ésimo valor en orden ascendente de distancia desde un punto en *P* hasta *Q*. El principal inconveniente de esta distancia es que no es una métrica, por no satisfacer la desigualdad triangular.
- Distancia de Fréchet: se emplea generalmente para medir la similitud entre curvas. La distancia de Fréchet entre dos curvas se define de la siguiente manera:

$$Fr(P, Q) = \inf_{\alpha,\beta} \max_{t \epsilon [0,1]} \left\| P(\alpha(t) - Q\left(\beta(t)\right) \right\|$$

  donde P,Q :[0, 1] → R^2 son parametrizaciones de las dos curvas, y α, β :[0, 1]→ [0, 1] el rango sobre todas las funciones continuas y monótonamente crecientes.

En general, las transformaciones pueden clasificarse como rígidas o como afines. En una transformación rígida, las distancias y orientaciones se preservan. Existen en R^3 dos tipos de transformaciones rígidas: las traslaciones y las rotaciones. Las primeras mantienen los vectores diferencia, mientras que las segundas mantienen el origen. Como ejemplos de las transformaciones afines está la escala.

2.2. Técnicas de correspondencia de patrones

Existe gran variedad de problemas que pueden clasificarse bajo la denominación de *correspondencia de patrones*. Como se verá más adelante, aparecen diferencias considerables entre ellos, y los paradigmas computacionales empleados para solucionarlos también pueden diferir bastante. No obstante, dado que todos ellos llevan a cabo la misma tarea básica, se puede asumir que todos ellos están unificados considerando el problema como: *dada una serie de formas de entrada S_1,S_2,…,S_N, establecer una relación significativa R entre sus elementos*[5]. Cuando dos elementos están relacionados, se dice que están en correspondencia, o que existe un match entre ellos. Esta relación puede plantearse de diferentes formas, tales como uno a uno, uno a varios, o varios a varios.

De este modo, la correspondencia de formas se define generalmente como la identificación de elementos homólogos entre dos o más formas, es decir, elementos que tienen una estructura idéntica o similar en términos de su apariencia local y contexto. La definición de dichos *elementos*, y de la *estructura similar* dependerá del problema a resolver. No obstante, una forma estará siempre formada por múltiples elementos (puntos, segmentos, caras…). Para poder establecer la similitud entre estos elementos será necesario llevar a cabo ordenaciones de medidas de similitud geométrica o estructural.

Una distinción importante que hacer consiste en la diferencia entre problemas de correspondencia de formas, y problemas de recuperación de formas. La recuperación de formas consiste en que, dada una forma de consulta, se desea encontrar formas de una base de datos que sean similares a la consulta, tolerando gran variedad de transformaciones y, posiblemente, considerando similitudes parciales, ya que únicamente interesa cuantificar la similitud entre formas. En los problemas de correspondencia de formas, lo que se pretende es relacionar explícitamente los elementos de una forma con los elementos de otra. La evaluación de similitudes entre formas, para problemas de recuperación, se suele llamar *correspondencia de formas*[6].

El problema de correspondencias planteado hasta el momento puede clasificarse de diferentes formas, que agruparemos en las siguientes categorías:

Correspondencia parcial frente a correspondencia completa: continuando con la definición genérica del problema planteada al inicio de este apartado, esta clasificación de técnicas se basa en las propiedades de inclusión de la relación de correspondencia *R*, definida entre los elementos de las formas. Ésta puede requerir que una correspondencia sea completa (definida para todos los elementos de las formas) o parcial (definida para un subconjunto de elementos). La motivación de una correspondencia parcial se basa en que las formas están constituidas por partes diferentes (es posible que una forma tenga más o menos elementos, en comparación con otras), por lo que no tendría ningún sentido establecer la correspondencia entre todas las partes. Estas partes, además, pueden diferir pos su geometría, escala, y punto de conexión con la totalidad de la forma.

Calcular una correspondencia parcial es más complicado que computar una correspondencia completa, ya que se produce una explosión combinatoria sobre el espacio de soluciones. Si se consideran todas las correspondencias uno a uno entre formas con *n* elementos, el espacio de

soluciones estará formado por *n!* correspondencias. Si además se añade la posibilidad de la correspondencia parcial, el espacio de soluciones incluirá, además, todos los subconjuntos posibles de esas *n!* correspondencias. Además, la búsqueda del subconjunto adecuado incrementa la complejidad del problema y además requiere una definición muy precisa del criterio de optimalidad.

Correspondencia densa frente a correspondencia dispersa: otro aspecto a tener en cuenta es la *densidad* de la relación *R*. Una *correspondencia densa* se define para todos los elementos o primitivas de la forma (por ejemplo, puntos). Una *correspondencia dispersa* se define para un pequeño número de elementos pre-seleccionados. Los elementos suelen ser un conjunto de características. Por ejemplo, para inferir la semántica de contornos humanos, basta con establecer la correspondencia entre puntos representativos colocados en las piernas, brazos, cabeza y cuerpo (correspondencia dispersa). Por otro lado, aplicaciones como el morphing de dos formas precisan de una correspondencia densa, para poder garantizar la suavidad global del resultado.

Está generalmente aceptado que calcular una correspondencia dispersa entre puntos representativos tiene la misma complejidad que calcular una correspondencia densa entre dos formas, ya que en ambos problemas es necesario considerar la estructura global de las formas [7]. Sin embargo, una forma típica de proceder ante una correspondencia densa consiste en la obtención de una solución a partir de una correspondencia dispersa mediante técnicas de parametrización e interpolación, siendo necesario tomar decisiones a niveles locales [8].

Correspondencia de grupos: una forma especializada de correspondencia de formas tiene que ver con el tratamiento de grupos de formas simultáneos ($N > 2$ en la definición propuesta). Pese a que este problema puede ser solucionado directamente calculando las correspondencias entre todos los pares de formas en el grupo, considerar todas las formas colectivamente puede tener ventajas en el proceso. Así, la información del grupo puede actuar como refuerzo para distinguir qué estructuras o partes son comunes a todas las formas y deben ser consideradas en la correspondencia, y qué partes pueden caracterizarse como ruido y ser ignoradas en el cálculo [9]. Una correspondencia robusta entre un grupo de formas es la entrada requerida para tareas como la construcción de descriptores estadísticos de un conjunto de formas [10].

2.3 Visión general de los problemas de correspondencia

En líneas generales, se puede obtener una correspondencia directamente sobre la similitud de los elementos que forman una forma, o se puede alinear primero las formas, y posteriormente derivar la correspondencia por la proximidad de los elementos alineados. Estas dos opciones afectan directamente a la manera en que se soluciona el problema de la correspondencia. Cabe destacar que la alineación entre las formas es un resultado colateral del cálculo de la correspondencia, y que su conocimiento puede ser esencial para la aplicación que se esté desarrollando.

La siguiente clasificación de problemas considera, en primer lugar, el problema sin alineación de los elementos (correspondencia basada en similitud), después se distinguirán dos tipologías que incorporan la alineación (alineación rígida y alineación no-rígida) y, finalmente, se añadirá la dimensión del tiempo (alineación variante en el tiempo).

Correspondencia basada en la similitud: una de las formas más básicas que existen para calcular la correspondencia consiste en estimar la similitud entre pares de elementos de las formas. Alternativamente, se pueden obtener las correspondencias a través de características obtenidas de las mismas, técnica conocida como *correspondencia de características*. La correspondencia se obtiene seleccionando pares de elementos mientras se optimiza una función objetivo formada por dos términos. El primero intenta maximizar la similitud entre los descriptores de los elementos correspondientes, mientras que el segundo intenta minimizar la distorsión que se introduciría en las formas si estuvieran deformadas para ser alineadas. No obstante, el segundo término puede ser estimado por la correspondencia sin la necesidad de alinear explícitamente las formas. Idealmente, satisfacer estos dos objetivos debería proporcionar la solución al problema, que generalmente se obtiene mediante métodos de optimización.

Este método puede ser aplicado a cualquier contexto en el que sea posible calcular un conjunto de descriptores de los elementos. Aplicaciones típicas incluyen la correspondencia de escaneos 3D [11], correspondencia de esqueletos [12]. Además, esta técnica no está restringida a su propio dominio, y puede ser combinada con métodos basados en la alienación para proporcionar inicializaciones adecuadas [13] o limitar el tamaño del espacio de soluciones [14][15][16][17]

Alineación rígida: en ciertos problemas, la búsqueda de una correspondencia puede consistir en encontrar una transformación geométrica que alinee las formas. Una aplicación de ejemplo para la alineación rígida son los escaneos 3D, en los que el objetivo es obtener una representación digital de un objeto del mundo real. Durante el proceso de escaneo es posible que no se pueda capturar la totalidad del objeto en una sola pasada, debido a oclusiones propias o a limitaciones físicas del escáner, por lo que será necesario tomar múltiples muestras, y alinearlas de forma óptima para reconstruir el objeto [18][13][14][15]. La característica clave de este problema de alineación consiste en que la forma de los objetos no cambie entre un escaneado y otro. Así, es posible asumir que cada muestra puede ser transformada mediante una transformación rígida simple para alinearla perfectamente con las otras muestras. Las transformaciones rígidas se apoyan, básicamente, en traslaciones y rotaciones, y una de sus características claves para solucionar el problema consiste en la baja dimensionalidad del problema.

Los escaneos 3D son sólo un ejemplo de muchas aplicaciones en las que se puede asumir rigidez en los datos. Si las formas de entrada se proporcionan como conjuntos de puntos 3D, el problema puede enunciarse como: para cada conjunto de puntos *S*, maximizar el número de puntos en *S* que se alinean con los puntos de otros conjuntos. Este objetivo depende generalmente de una tolerancia ε que indica cuándo dos puntos están lo bastante cerca para considerar que hacen un match [19]. Dado que encontrar la transformación que permita la mejor alineación puede ser una tarea necesaria, se puede hacer uso de la correspondencia de características como función de ayuda para buscar la alineación óptima.

Alineación no rígida: en ocasiones será necesario obviar la asunción de que cada muestra puede ser perfectamente alineada con el resto mediante una transformación rígida. Los ejemplos más significativos incluyen la correspondencia entre formas articuladas [16][20][21][22][23], en la que ciertas partes de las formas se pueden doblar independientemente, la correspondencia de formas anatómicas [24], que pueden deformarse de forma elástica e introducir deformaciones en zonas localizadas de la forma y, finalmente, la correspondencia entre formas con diferentes geometrías pero que representa la misma clase de objetos o que tienen partes semánticamente relacionadas [25][26]. En este último caso, se puede apreciar que el problema de la correspondencia difiere de los doblamientos locales o de las deformaciones elásticas.

En este tipo de problemas, es necesario añadir más grados de libertad en cómo las formas pueden ser correspondidas entre ellas. Esto puede lograrse generalizando dos aspectos del problema. En primer lugar, hay que desviarse del caso rígido y permitir transformaciones no rígidas (posiblemente no lineales). En segundo lugar, estas transformaciones pueden ser aplicadas independientemente a zonas locales de la forma. Por ejemplo, se puede representar la transformación aplicada a una forma como el vector de desplazamiento asociado a cada vértice de la forma [27]. Más tarde, habrá que calcular las transformaciones que coloquen cada vértice en correspondencia con la forma de destino. La principal diferencia con el caso rígido es que el espacio de transformaciones considerado es, dimensionalmente hablando, mucho mayor.

Alineación variante en el tiempo: dados los avances tecnológicos recientes, una aplicación en auge es la reconstrucción de formas 3D adquiridas en un espacio de tiempo mientras se mueven y deforman. En estos problemas es necesario disponer de un determinado número de muestras por unidad de tiempo, y éstas deben ser correladas entre ellas para permitir la reconstrucción del objeto y de su secuencia de movimiento [25][26][27][28][29][30][31][32][33][34]. Pese a que este problema se parezca mucho a la alineación no rígida, existen ciertas particularidades que convierten al problema en único. En el problema de correspondencia clásico, se asume que todas las muestras pueden ser correladas para formar un único objeto. Por otro lado, la variación del tiempo introduce una dificultad adicional que consiste en que la forma puede deformarse considerablemente entre una muestra y otra. De este modo, muestras posteriores en el tiempo sólo se corresponderán con muestras anteriores si se tiene en cuenta la deformación. Además, existen nuevos retos debidos a la gran cantidad de información no disponible (debido a oclusiones) que puede aparecer en cada muestra [28] o en muestras capturadas en diferentes instantes de tiempo [32][34].

2.4 Clasificación de los métodos de correspondencia

En esta sección se presentarán varias maneras de clasificar los métodos de correspondencia. El objetivo de esta clasificación es permitir al lector comparar los métodos no sólo por sus aspectos algorítmicos, sino también por sus propiedades o tipo de problemas que pueden solucionar.

El criterio de clasificación estará basado en los componentes de la definición del problema: *dada una serie de formas de entrada S_1,S_2,…,S_N, establecer una relación significativa R entre sus elementos*. Con el ánimo de ser más específicos, a continuación se detalla el significado de cada uno de los componentes que aparecen en ella:

1. Formas: la primera categoría de la clasificación propuesta analizará los métodos de correspondencia en base a cómo se representan las formas de entrada.
2. Relación: los métodos se analizarán en base a cómo se representa la correspondencia de salida y qué propiedades tiene.
3. Significativa: clasificación de métodos en base a la correspondencia elegida (qué correspondencia está más cerca del objetivo planteado).
4. Establecer: qué técnica computacional se emplea para calcular la correspondencia.

La respuesta a cada una de estas preguntas da lugar a la clasificación que a continuación se presenta.

2.4.1 Formas de entrada

La geometría de las formas de entrada puede estar representada por conjuntos de puntos, puntos orientados, superficies, esqueletos… Su dimensionalidad puede ser 2D, 3D, y puede incluirse el tiempo.

Los métodos clásicos de correspondencia [35][36][37] generalmente trabajan sobre conjuntos de puntos, mientras que los métodos modernos basados en deformación [38][7], formas articuladas [16] y aplicaciones gráficas basadas en la correspondencia de plantillas [39][24][30] suelen trabajar con superficies. Las superficies variantes en el tiempo son la representación de formas de entrada más habitual en los trabajos que se centran en la reconstrucción de movimiento de superficies deformables [25][26][27][28][30][31][34].

Los datos de entrada pueden obtenerse desde gran variedad de fuentes, como escáneres 3D que proporcionan nubes de puntos, modelado manual o imágenes y volúmenes obtenidos mediante diferentes dispositivos de adquisición. Los esqueletos son obtenidos, generalmente, a partir de superficies con un procesamiento extra [40] y almacenan información más relacionada con la estructura.

En lugar de trabajar directamente sobre la representación original de los datos de entrada, es común llevar a cabo una extracción de sus puntos representativos (características), y calcular a partir de ellos los descriptores de las formas. Estos descriptores serán típicamente valores escalares o vectores de escalares que representen alguna propiedad de la forma en torno al punto de interés [41]. Los descriptores de formas se utilizan para establecer indirectamente la similitud

entre formas. De manera ideal, si dos descriptores son similares, sus puntos correspondientes también lo serán. Alternativamente, los descriptores pueden usarse para guiar la búsqueda en la inicialización, mientras que la verificación se lleva a cabo sobre el conjunto de puntos originales [15].

Existe una gran cantidad de descriptores propuestos en la literatura. En [4][42][43] se detallan las características de éstos, mientras que en [44] se comparan. La tabla 2.4.1.1 proporciona una visión parcial de los conjuntos de descriptores más habituales. Éstos suelen ser las entradas típicas para algoritmos de optimización y búsqueda utilizados en visión y gráficos [23][45][46] [7][14][47]. Otro problema interesante relacionado con el problema consiste en cómo elegir el conjunto de descriptores que proporcione los mejores resultados de correspondencia, problema conocido como *selección de características* en la literatura de machine learning [48][49]

Descriptor	Tipo de datos
Contexto de la forma [50][51]	Conjunto de puntos
Imágenes rotadas [52]	Puntos orientados
Características multi-escala [53]	Puntos orientados
Basado en armónicos esféricos [54]	Puntos orientados
Mapas de curvatura [55]	Superficies
Invariantes integrales [14][56]	Superficies
Características de salientes [57]	Superficies

Tabla 2.4.1.1: Descriptores más habituales y tipos de datos manejados.

2.4.2 Correspondencia de salida

La correspondencia calculada por los métodos puede representarse de diferentes maneras, y también puede diferir en otras propiedades, como si se trata de una correspondencia completa, parcial, densa o dispersa. Se puede representar la correspondencia como una transformación aplicada a ambas formas o como una simple relación entre elementos de cada forma.

Correspondencia + transformación: cuando se emplea una única transformación (o un conjunto de ellas), un factor distintivo entre las técnicas se basa en la naturaleza de las transformaciones empleadas. Éstas pueden ser ordenadas incrementalmente por su número de grados de libertad: traslación, transformaciones rígidas (incluye rotación), transformaciones de similitud (incluye escalado isotrópico), transformaciones afines y transformaciones no lineales.

Una transformación rígida mantiene los pares de distancias entre los puntos transformados, y se puede descomponer en traslaciones, rotaciones y reflexiones. Suele ser la elección más común ante problemas de correspondencia de datos escaneados [13][14]. Las transformaciones de similitud añaden la posibilidad del escalado uniforme, que puede ser necesario en ciertos contextos como la correspondencia de patrones con porciones limitadas de otros. Las transformaciones afines amplían el conjunto anterior teniendo en cuenta la posibilidad de estiramientos lineales, que pueden ser empleados global [15][19] o localmente [39]. Finalmente, en el caso más general de correspondencia entre formas no rígidas, debe ser posible mover los elementos de la forma libremente para que encaje con sus homólogos de la otra forma. Esto se

consigue mediante el uso de transformaciones de deformación no-lineales para cada elemento de forma individual [24].

Sólo correspondencia: en este caso, la relación *R* puede representar una biyección (relación uno a uno), una inyección (debe estar definida tomando como referencia cada elemento de una de las formas, y puede ser una relación uno a muchos) o puede ser una relación completa (muchos a muchos).

Algunos métodos permiten elegir el tipo de mapeado a aplicar, mientras que otros asumen que el objetivo está centrado en un tipo de correspondencia específica [45][46]. En estos últimos, el problema se entiende como un proceso de optimización en el que se deben restringir las relaciones a uno a uno.

Un tipo específico de métodos de sólo correspondencia son aquellos que asignan un valor de confianza a cada asignación de un par de elementos. Estas asignaciones pueden estar caracterizadas por un valor discreto (existe o no existe correspondencia), o pueden tener un grado de confianza asociado a ellas (una medida de probabilidad). En el primer caso se trata de una valoración binaria, mientras que en el segundo el valor asignado a una correspondencia depende del algoritmo que se esté empleando.

Correspondencia parcial frente a correspondencia completa: algunos métodos son sólo válidos para contextos en los que se consideran todos los puntos de las formas, mientras que otros permiten calcular correspondencias parciales. En general, si un método puede encontrar soluciones para el caso parcial, también será aplicable al caso de correspondencia completa.

Correspondencia densa frente a correspondencia dispersa: la ventaja de definir el problema en términos de correspondencia dispersa consiste en que la complejidad de la computación, en tiempo y espacio, se reduce considerablemente. Algunas técnicas se han diseñado teniendo esto en cuenta, como los métodos basados en búsquedas descritos en [7][14]. Pese a que su espacio de búsqueda asociado es exponencial, estos métodos pueden utilizarse en la práctica considerando únicamente un conjunto reducido de puntos característicos extraídos a partir de la forma. Otros métodos no hacen una distinción fuerte entre estos dos casos, y pueden intercambiar los datos para calcular tanto correspondencias densas como dispersas. Finalmente, existe gran cantidad de métodos que calculan una correspondencia densa a partir de una correspondencia dispersa inicial [8][24].

2.4.3 Función objetivo

La función objetivo proporciona una medida de cuán buena es una correspondencia dada, o cuán lejos está de la solución deseada. En ocasiones se hace referencia a ella como la *medida de error*, la *función de coste*, o la *energía*, en el caso de los métodos que formulan el problema como la minimización de determinadas funciones de energía. Esta formulación depende del tipo de datos de entrada considerados, y de los problemas específicos que se deseen solventar.

En este apartado, se clasificarán los métodos de correspondencia de formas en base a los criterios establecidos en el apartado 2.3 (correspondencia basada en la similitud, alineación rígida, alineación no rígida y alineación variante en el tiempo).

Correspondencia basada en la similitud: cuando el objetivo es encontrar la correspondencia entre dos conjuntos de datos sin alinear antes las formas, es necesario emplear descriptores y medidas intrínsecas para medir la calidad de la correspondencia. Así, entre dos formas P y Q y una relación de correspondencia R, la función objetivo toma la forma:

$$Obj(P,Q,R) = Sim(P,Q,R) + \propto Distor(P,Q,R)$$

con un término de similitud lineal al número de puntos y un término de distorsión, generalmente cuadrático al número de puntos. El peso α controla la influencia de cada término de la función objetivo. Poder establecer de forma automática este término para adaptarse a la intención del usuario supone un problema adicional [58].

El término de *similitud* codifica la similitud entre los descriptores de forma de los puntos de correspondencia. Estos descriptores pueden tener atributos geométricos, como los vectores normales, que pueden proporcionar información acerca de si la orientación de los puntos correspondidos es coherente [23].

El término de *distorsión* mide cuánto deberían ser deformadas las formas para poder establecer una correspondencia. Una medida común para la disparidad consiste en las distancias de los puntos encajados. La disparidad es una forma de aproximar la medida de la distorsión provocada por la correspondencia sin haber alienado antes las formas, y puede expresarse como:

$$Dispar(P,Q,R) = \sum_{\substack{\{p1,p2\}\subset P \\ \{q1,q2\}\subset Q}} Dispar(p1,p2,q1,q2)$$

donde $\{p1,q1\}\epsilon R$ y $\{p2,q2\}\epsilon R$. El término de disparidad entre dos pares $\{p1,p2\}$y $\{q1,q2\}$ puede obtenerse como la diferencia de distancias entre los pares de puntos. Cualquier medida de distancia puede emplearse. Un ejemplo común es la distancia Euclídea [16]:

$$Dispar(p1,p2,q1,q2) = \; |\|p1-p2\| - \|q1-q2\||$$

o la distancia geodésica en caso de trabajar sobre superficies:

$$Dispar(p1,p2,q1,q2) = \; \left|geod_P(p1,p2) - geod_Q(q1,q2)\right|$$

donde $geod_S(s_1,s_2)$ es la distancia geodésica entre s_1 y $s_2$sobre la superficie S.

La deformación es una forma más elaborada de cuantificar la distorsión, y consiste en que, una vez conocidos los puntos que deben ser encajados, se deforma una forma hasta la otra de manera que estos puntos queden encajados. Las técnicas descritas en [7][22] hacen uso de estas medidas. Nótese que este caso es diferente a las deformaciones no rígidas ya que aquí ya se dispone de la correspondencia, y únicamente es necesario estimar su distorsión.

Alineación rígida: para el problema de la alineación rígida, el objetivo suele estar definido en términos del número de puntos que encajan, o mediante una métrica que cuantifique cómo de bien se alinea una forma con otra.

La técnica más habitual se conoce como *LCP* (acrónimo de la expresión inglesa *Largest Common Pointset*), y su objetivo consiste en encontrar una transformación que ponga la mayor cantidad posible de puntos en correspondencia [15][19], dada una tolerancia ε que indica si dos puntos están lo bastante cerca y pueden considerarse como un match. De este modo, el objetivo es maximizar la cardinalidad de este conjunto de puntos encajados, lo que puede ser expresado para dos formas *P* y *Q* como:

$$LCP(P,Q) = \sum_{p \in P} Match(p,Q),$$

donde

$$Match(p,Q) = \begin{cases} 1 \rightarrow \exists q \in Q, \|p-q\| < \varepsilon \\ 0 \rightarrow en\ cualquier\ otro\ caso \end{cases}$$

La distancia geométrica es otra función objetivo común que no precisa de un parámetro ε, minimizando el error de alineamiento dado por la suma cuadrática de las distancias entre los puntos. Esto significa que para cada punto encuentra su punto más próximo en la otra forma, y suma la distancia entre estos a la medida de error. Suele emplearse en algoritmos como el *ICP* [13] (acrónimo de la expresión inglesa *Iterated Closest Point*). Su expresión viene dada como:

$$Dist(P,Q) = \sum_{p \in P} Dist(p,Q),$$

donde

$$Dist(p,Q) = \min_{q \in Q} \|p-q\|$$

La técnica *LCP* tiene la ventaja de que las correspondencias parciales pueden ser tratadas directamente sobre la función objetivo, mientras que la suma de distancias al cuadrado necesitará considerar todos los puntos.

Alineación no rígida: si las formas deben encajarse mediante deformaciones, la función objetivo deberá incorporar términos que cuantifiquen cuándo una transformación tiene sentido. Esto significa que, si un vértice se puede desplazar libremente en base a sus propias transformaciones, se debe proporcionar cierta consistencia global (regularización). Esta regularización puede obtenerse limitando el número de grados de libertad de las transformaciones, o penalizando las transformaciones grandes. La medida de error en este caso debe incorporar dos términos: la calidad de la alineación, y la regularización.

La calidad de la alineación es una medida que cuantifica cuán bien quedan alineadas las formas tras las transformaciones introducidas, proporcionando una medida de distancia geométrica similar a la introducida para el caso de alineación rígida [24][59], o distancia al plano para el caso de superficies [39].La regularización es un término que enfatiza en la consistencia global de las transformaciones exigiendo que las transformaciones de puntos vecinos sean similares (lo que proporciona una transición de transformaciones suave). Esta similitud puede ser medida de forma directa [39] [59] o en base a las diferencias entre transformaciones [24].

2.4.4 Tipo de solución

Existe gran cantidad de técnicas que pueden emplearse para la búsqueda de la mejor correspondencia. Desde el punto de vista de la solución, hay métodos que buscan una serie de transformaciones que alineen las formas, métodos que únicamente consideran los pares de asignaciones entre elementos y encuentra una solución utilizando métodos de optimización bien conocidos o técnicas de búsqueda, y métodos que llevan a cabo una búsqueda híbrida, alternando entre la alineación y el cálculo de correspondencias. Clasificaremos estas técnicas en base a las propiedades de las soluciones que aportan:

Totalmente automáticas frente a semi-automáticas: los métodos semi-automáticos precisan entradas del usuario, como una inicialización adecuada, mientras que los métodos automáticos no precisan de ninguna interacción con el usuario, a parte de unos pocos parámetros.

Búsquedas globales frente a locales: la distinción se lleva a cabo en función de si el método explora todo el espacio de soluciones en búsqueda de una buena solución [7] [14], o de si los resultados del método dependen totalmente de la inicialización. Ésta puede ser dada por el usuario, caso semi-automático, o se puede incluir inicializaciones automáticas que proporcionen un punto de inicio para el algoritmo de búsqueda local.

Uno de los ejemplos más efectivos en la categoría de búsquedas locales es el *ICP* [13] que alterna los cálculos de correspondencias entre puntos, con el cálculo de una transformación de alineación. Dado que este proceso iterativo sigue una única trayectoria en el espacio de soluciones, tiene el inconveniente de que puede finalizar en un mínimo local. Además, el estado inicial influencia claramente el resultado final del algoritmo y, por lo tanto, existen diversos métodos de inicialización propuestos.

Otro ejemplo de algoritmos de búsqueda local son los métodos para alineación no rígida basados en el cálculo explícito de la transformación para cada elemento de la forma. Dado que estas transformaciones son calculadas en base a métodos convexos de optimización, la inicialización también condicionará la correspondencia final obtenida [24][39][59].

Correspondencia entre pares o entre grupos: los métodos de correspondencia entre grupos predominan en la comunidad de computación anatómica [60] en la que es importante obtener una correspondencia coherente entre grupos de formas para construir un modelo estadístico. Estos métodos buscan optimizar el grupo de correspondencias para que el modelo estadístico generado sea lo más compacto posible [10]. Esta optimización, generalmente consiste en llevar a cabo ajustes locales sobre las formas pre-alineadas.

Pese a que el término *correspondencia entre grupos* no se emplea en reconstrucciones variantes en el tiempo, ciertas técnicas aplicadas a este problema pueden verse como una variante de estos métodos, ya que todas las muestras se consideran simultáneamente. La principal diferencia con el caso de las formas anatómicas es que, en reconstrucciones variantes en el tiempo, cada muestra se puede deformar entre fotogramas, y se pierden gran cantidad de datos, mientras que en el caso de la anatomía generalmente se busca una correspondencia completa entre formas completas, que son tomadas como variaciones de la misma forma media de un órgano o hueso.

2.4.5 Resumen de los criterios de clasificación.

El siguiente diagrama muestra, a modo de resumen, los criterios de clasificación presentados a lo largo de este apartado.

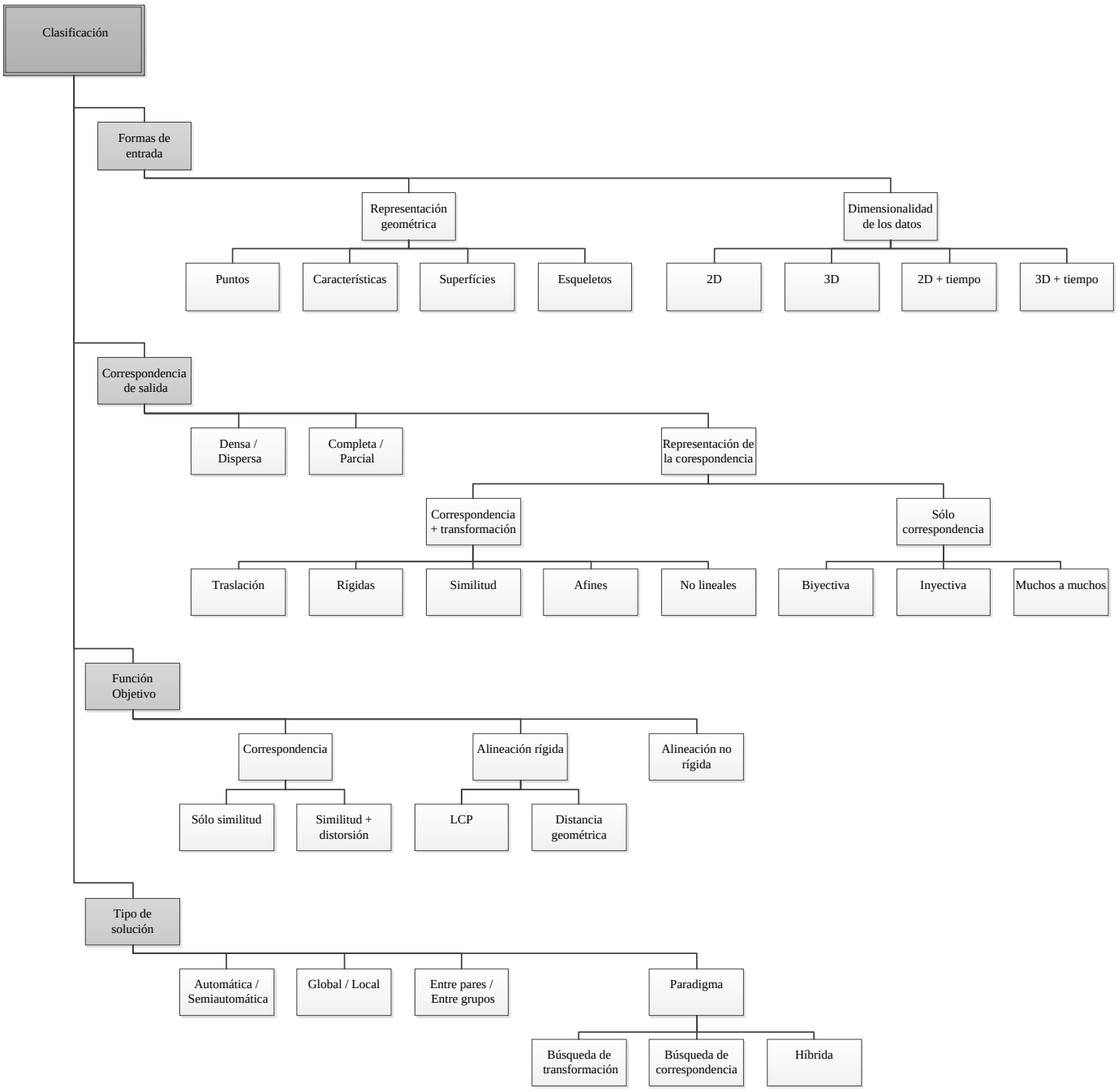

Figura 2.4.5.1: Resumen de los criterios de clasificación de los métodos de correspondencia.

2.5 Métodos más representativos

Con el ánimo de estudiar los métodos más representativos con cierto orden, este apartado se apoya en la clasificación basada en el método en que se obtiene la correspondencia. Así, los distintos métodos se clasificarán de forma primaria en tres categorías: los que buscan una transformación de alineación, los que buscan directamente una correspondencia sin calcular la alineación, y los métodos híbridos. Finalmente, se comentarán los métodos que hacen uso de correspondencias parciales.

2.5.1 Búsqueda de transformación y alineación

Antes de introducir los métodos más representativos de esta categoría, es precios recordar que este tipo de métodos buscan, en primer lugar, una transformación que alinee las formas y, posteriormente, derivan la correspondencia basada en la proximidad de los elementos alineados.

Alineación rígida: los métodos en esta clase se basan en el hecho de que las transformaciones empleadas para el alineamiento pueden obtenerse a partir de un pequeño conjunto de puntos. Por ejemplo, si se pretende alinear dos conjuntos de puntos en tres dimensiones, el cálculo la transformación se puede llevar a cabo mediante una configuración inicial de tres puntos para cada forma. Tras muestrear el resultado de la transformación, se puede *verificar* la calidad en la alineación o, alternativamente, *votarla*.

Los métodos que hacen uso del muestreo y la verificación, también conocidos como métodos *ingenuos* [38], aplican directamente la idea propuesta: toman tres puntos de la primera forma, tres puntos de la segunda (caso 3D), derivan una transformación rígida y comparan la calidad de la alienación Tras comprobar todos los tripletes posibles de puntos en las dos formas, se devuelve la mejor transformación. Este tipo de algoritmos, trabajando sobre muestras 3D, presentan una complejidad de $O(m^3 n^3)$ para muestrear los tripletes, y $O(m \log n)$ para la verificación, teniendo una complejidad total de $O(m^4 n^3 \log n)$.

Claramente, estos algoritmos distan de la eficiencia. No obstante, se han propuesto diferentes modificaciones que mejoran este coste. Las técnicas basadas en la filosofía RANSAC (acrónimo de la expresión inglesa *Random Sample Consensus*) tomar muestras aleatorias en los diferentes pasos [61]. Así, en lugar de muestrear todos los tripletes posibles, se puede considerar únicamente una muestra aleatoria de puntos en cada forma, reduciendo la complejidad en un factor de $O(m^3)$. Además, también es posible aleatorizar el proceso de verificación, reduciendo la complejidad en otro factor de $O(m)$ en un caso típico [19].

Alternativamente, es posible considerar las invariantes geométricas que mantienen las transformaciones. Un caso de este estilo es el radio entre tres puntos coplanares, que se mantiene al aplicar transformaciones rígidas y afines. Así, el problema de buscar tripletes de puntos que proporcionen una transformación óptima puede convertirse en encontrar cuatro conjuntos de puntos coplanares que compartan los mismos radios [62]. Pre-procesando estas invariantes se pueden desarrollar métodos eficientes que reducen la complejidad a $O(n^2 + k)$ [15], donde k es el tamaño de la salida proporcionada.

Los métodos que hacen uso del muestreo y votación, en lugar de muestrear una transformación y evaluar su calidad, simplifican el paso de verificación votado la transformación. Para estos métodos, conocidos como *pose clustering* (cuya traducción aproximada será *planteamiento de agrupación*) se hace uso de una tabla de acumulación [37]. Tras enumerar dos tripletes de puntos y calcular una transformación, se almacena en la tabla una votación indexada por los parámetros de la transformación. Al final de este proceso de $O(m^3 n^3)$ las celdas con más votos corresponden a la transformación que mejor alinea los puntos. Nótese que procesar la tabla de acumulación requiere un coste extra $O(h)$ dependiente del tamaño h de la tabla.

Los algoritmos de *hashing* geométrico son métodos basados en el sistema de votaciones, que emplean el concepto del pre-procesado para acelerar la alineación [36]. La idea principal consiste en almacenar en una tabla hash todas las configuraciones posibles de un grupo de puntos de referencia de manera que cuando se busque el conjunto que mejor encaja con un conjunto dado, esta búsqueda se realice de forma eficiente. Informalmente, se puede decir que este método convierte el coste $O(m^3 n^3)$ de los algoritmos ingenuos en un coste $O(m^3 n)$. El aumento de velocidad se obtiene a costa de emplear más memoria.

A modo de resumen de los costes de alineación rígida, se recoge en la tabla 2.5.1.1 las características de las técnicas comentadas en este apartado.

Técnica	Método	2D		3D	
		Tiempo	Espacio	Tiempo	Espacio
Alineación	Algoritmo ingenuo	$O(m^3 n^2 \log n)$	-	$O(m^4 n^3 \log n)$	-
	Aleatorizado	$O(mn^2 \log n)$	-	$O(mn^3 \log n)$	-
	Verificación aleatoria	$O(n^2 \log n)$	-	$O(n^3 \log n)$	-
	4 puntos coplanares	$O(n^2 + k)$	$O(n)$	$O(n^2 + k)$	$O(n)$
Pose clustering	Algoritmo ingenuo	$O(m^2 n^2 + h)$	$O(h)$	$O(m^3 n^3 + h)$	$O(h)$
	Aleatorizado	$O(m n^2 + h)$	$O(h)$	$O(m n^3 + h)$	$O(h)$

Técnica	Método	Pre-procesado	Espacio	Consulta
Hashing geométrico 2D	Algoritmo original	$O(m^3 \log n)$	$O(m^3)$	$O(n^3 \log n)$
	Aleatorizado	$O(r^3 \log r)$	$O(r^3)$	$O(n^3 \log n)$

Tabla 2.5.1.1: complejidad de los métodos de alineación rígida para dos conjuntos con m y n puntos. r es el tamaño del conjunto de puntos empleado para la verificación, k es el tamaño de la salida, y h denota el tamaño de la tabla de acumulación en la técnica de pose clustering.

Alineación rígida de trozos: los métodos comentados en el apartado anterior emplean una única transformación global para hacer coincidir una forma con otra. Un tipo diferente de métodos generaliza esta idea aplicando transformación a porciones locales de las formas. En [16], estas transformaciones se aplican a trozos de piezas de forma rígida para establecer una correspondencia entre formas articuladas. El problema se formula como etiquetar los vértices de las formas con transformaciones candidatas. Así, representando los vértices únicamente como transformaciones sobre un conjunto reducido, se simplifica considerablemente la búsqueda de la

solución en contraste con los métodos que permiten que cualquier transformación se asigne a cualquier vértice.

Alineación no rígida:en este tipo de métodos, a cada vértice se le asigna una transformación, quedando el problema definido como encontrar la mejor transformación que lleve a cada vértice de una forma de referencia cerca de su correspondiente en una forma objetivo. La optimización se suele llevar a cabo con métodos basados en Newton, y se añade un término de regularización para enfatizar la similitud entre transformaciones asociadas a vértices vecinos. La dificulta en estos métodos consiste en evitar los mínimos locales en el espacio de soluciones. Esto se consigue inicializando los métodos con un conjunto diferente de puntos, y solucionando la optimización a múltiples niveles.

En lugar de calcular las deformaciones locales o desplazamientos para los vértices, en [63] el desplazamiento se obtiene de forma implícita mediante el aprendizaje de una función que adapta una forma a la otra. Esta adaptación se obtiene solucionando un problema de optimización similar al empleado en las técnicas de machine learning, y consigue evitar los mínimos locales.

Una extensión de estos métodos es la propuesta en [64], donde la alineación entre dos formas se lleva a cabo mediante dos transformaciones diferentes: una transformación rígida que alinea a groso modo ambas formas, y una segunda transformación afín por vértice que las alinea completamente. Es posible también obtener una alineación robusta deformando una forma hacia la otra en base a deformaciones Laplacianas de mallas [29].

2.5.2 Búsqueda de correspondencias

La principal característica de estos métodos consiste en que operan principalmente entre pares de puntos de características sin considerar las transformaciones que alinean las formas. El problema de correspondencia se plantea típicamente como un proceso de optimización de una función objetivo de la forma $Obj(P,Q,R) = Sim(P,Q,R) + \propto Distor(P,Q,R)$, que se basa en la calidad de los pares de asignaciones (término lineal) y la compatibilidad entre esos pares de asignaciones (término cuadrático). La solución se alcanza mediante métodos de optimización bien conocidos.

La optimización cuando únicamente se trabaja sobre el término de similitud se formula como un problema de asignación lineal (LAP). Este objetivo simplificado se puede solucionar mediante el algoritmo simplex, que es un caso especial de la programación lineal. No obstante, si se limita la correspondencia a uno a uno el problema se convierte en encontrar el match óptimo sobre un grafo ponderado y bipartido, que puede solucionarse de manera más eficiente con el algoritmo Húngaro $O(n^3)$, siendo n el número de puntos característicos de cada forma.

Por otro lado, si el objetivo contempla ambos términos (el lineal de similitud y el cuadrático de distorsión), se plantea un problema de asignación cuadrática (QAP) que es NP-duro. Muchas técnicas proponen calcular soluciones aproximadas a este problema. Un grupo de ellas lo plantea como una optimización entera, lo que relaja el problema, y puede ser solucionado con técnicas de programación cóncava [45], aproximaciones basadas en programación lineal [46] o etiquetado [65]. También puede plantearse el problema mediante términos probabilísticos y solucionarlo como un problema de optimización convexa [66].

Otro grupo de métodos soluciona el problema de forma discreta, estando basada una de las técnicas más comunes en calcular un etiquetado óptimo para el grafo, por ejemplo, trasladando el problema a términos de una red de Markov en la que las etiquetas corresponden con los puntos que encajan en la forma de destino [34].

Una parte importante de la literatura se centra en el caso específico de la correspondencia de contornos en dos dimensiones. Varias técnicas han sido desarrolladas teniendo en cuenta el hecho de que los vértices de un contorno pueden ser linealmente ordenados. Esto es empleado sobre técnicas de optimización como la programación dinámica, que también permiten solucionar el problema expresado en términos de encontrar el camino más corto [56].

Por último, existe un grupo específico de métodos de optimización discreta que encuentran la solución haciendo uso de técnicas de búsqueda basadas en árboles [14][54]. Durante la expansión del árbol, cada nodo representa una solución parcial, mientras que la solución completa se encuentra siguiendo el camino desde la raíz del árbol hasta una de sus hojas. Estas técnicas generalmente llevan a cabo tres pasos importantes: la expansión de un nodo que representa una nueva solución parcial, la estimación de cuán alejada está la estimación de la solución óptima, y la eliminación de nodos que no conducen a la solución óptima.

Generalmente las soluciones proporcionadas son colecciones de asignaciones entre pares de características, y el proceso de expansión supone añadir nuevos pares de asignaciones a la solución. Cuando se puede extraer una estructura jerárquica de la forma la búsqueda de la solución contempla esta información. En ese caso en particular, los esqueletos son muy usados.

2.5.3 Búsquedas híbridas

En este apartado se comentará el método *ICP* (*Iterated Closest Point*), que obtiene el cálculo de la correspondencia alternando entre dos pasos. En el primero, busca una alineación entre las formas, mientras que en el segundo, obtiene una correspondencia a partir del alineamiento. Este proceso se repite empleando la correspondencia para estimar una nueva transformación de alineación. *ICP* es un método de búsqueda híbrido porque busca tanto la alineación como la correspondencia, y ambos resultados afectan al otro. Las diferentes variantes de este algoritmo se obtienen modificando los métodos de solución de los dos pasos [13].

Alineación rígida: en la variante clásica del algoritmo ICP para alineaciones rígidas, dados dos conjuntos de p untos P y Q, se establece la correspondencia entre cada punto $p \in P$ y su punto más cercano en Q, en base a una métrica de distancia. Después, para todos los pares de asignaciones definidos en el paso anterior, se estima la mejor transformación rígida que alinea los dos conjuntos de puntos (solucionando un sistema lineal) y se re-alienan las formas en base a esta transformación. Finalmente, se repiten estos dos pasos, acabando cuando no se producen cambios significativos en la transformación estimada.

Como se puede deducir de este planteamiento, la posición inicial de las formas influencia considerablemente el resultado final, ya que la primera correspondencia se obtiene a partir de éstas. Por esto, un paso crucial en los métodos basados en *ICP* consiste en pre-alinear las formas de manera que el algoritmo no quede atrapado en un mínimo local. Se han propuesto diferentes formas de llevar a cabo este pre-alineamiento. Las soluciones clásicas se basan en la correspondencia entre un conjunto de puntos característicos, una configuración inicial

proporcionada por el usuario, o la alineación automática mediante *PCA* (acrónimo de la expresión inglesa *Principal Component Analysis*) [13].

Alineación no rígida: el método *ICP* puede ser empleado para problemas de alineación no rígida modificando alguna de sus componentes. Un conjunto de métodos calcula una correspondencia ponderada en la que cada asignación tiene asociada un valor de confianza. Estos valores son cruciales para que el método sea robusto. Tras alinear de forma rígida las formas, éstas son deformadas de forma no rígida para hacer que encajen [67][68]. En [22], la transformación rígida original del algoritmo *ICP* es reemplazada por una deformación basada en componentes rígidas.

Variaciones recientes de *ICP* están siendo utilizadas para el contexto de superficies variantes en el tiempo, para alinear la geometría de fotogramas adyacentes [26][28][30]. Si el número de muestras tomada por unidad de tiempo es suficientemente grande, se puede asumir que únicamente aparecen pequeños cambios en la configuración espacial de las formas, por lo que la alineación inicial de cada fotograma no resulta tan crucial como en los casos anteriores.

2.5.4 Correspondencias parciales

Dado que el cálculo de las correspondencias parciales es una especialización importante de la correspondencia de formas, en este apartado se comentarán las principales estrategias propuestas para esta área. El problema se define como encontrar un subconjunto de elementos de una forma para los que se obtenga una correspondencia. Esta tarea puede descomponerse en dos sub-problemas: buscar el subconjunto óptimo de k puntos y encontrar la correspondencia entre esos k puntos.

Una técnica para determinar el subconjunto óptimo consiste en examinar la función objetivo en búsqueda de incrementos pronunciados en el error de alineación, que aparecen cuando un punto de un saliente se añade al conjunto de puntos encajados [7][14].

Otra estrategia para determinar el conjunto de puntos consiste en una votación [17]. Aquí se calculan una serie de correspondencias candidatas y se vota cada par de asignaciones. Al final del proceso, las asignaciones que tengan mayor votación serán elegidas para el conjunto de puntos a encajar. El motivo por el que se muestrean las correspondencias candidatas es que este procedimiento actúa como refuerzo, en el que únicamente se vota una asignación si puede formar parte de una correspondencia consistente.

Otro enfoque común en este problema consiste en tomar la correspondencia parcial como un problema de encajado de dos grafos. Se consideran los puntos de una forma como los nodos del grafo, y se conecta cada par mediante una arista cuyo peso es proporcional a una métrica geométrica. De este modo, el encajado parcial se convierte en un problema de isomorfismo de sub-grafos, problema bien conocido, NP-completo. No obstante, dado que se han propuesto diferentes heurísticas para solucionar este problema, es posible hacer uso de estos métodos para calcular la correspondencia. Por ejemplo, la noción de encajar dos grafos encontrando un conjunto de operaciones que transformen un grafo en el otro se usa para derivar algoritmos heurísticos para correspondencia de esqueletos en 2D [69] y 3D [12]. Así, el conjunto de puntos que encajan viene dado por los nodos que no se han borrado durante la edición del grafo.

Capítulo 3

Técnica desarrollada

3.1. Introducción

Con el objetivo de introducir ordenadamente las principales características de la técnica desarrollada, se analizarán éstas en base a los criterios establecidos a lo largo del capítulo 2. No obstante es preciso puntualizar que, pese a que el objetivo final de la técnica es su aplicación en un ámbito muy concreto (recomposición de restos arqueológicos a partir de fragmentos), en este apartado se comentará una técnica de un ámbito mucho más genérico. Las restricciones específicas que impone el problema final se comentarán durante el capítulo 4, en el que se tratará la implementación de la técnica que a continuación se detalla aplicada a los restos arqueológicos.

De esta forma, de acuerdo con el criterio de clasificación planteado en el apartado 2.4, para caracterizar las propiedades de la técnica desarrollada se analizará qué datos se toman a partir de las formas de entrada, qué tipo de correspondencia se obtiene al finalizar la ejecución, qué función objetivo guía el proceso de búsqueda, y qué tipo de solución se ofrece al problema planteado.

3.1.1. Caracterización en base a las formas de entrada

Para hacer frente al problema de la correspondencia se ha optado por llevar a cabo la búsqueda contemplando únicamente las coordenadas de cada punto de la forma, y teniendo en cuenta también las coordenadas de sus. En cuanto a la dimensionalidad de los datos, el ámbito de aplicación de la técnica propuesta se limita a las dos dimensiones aunque, como más adelante se explicará, su aplicación a problemas tridimensionales es prácticamente inmediata.

La extracción de los contornos de las formas se lleva a cabo sobre mapas de bits obtenidos mediante el escaneado / fotografiado de los fragmentos reales, y se asume que se dispone de una imagen ya segmentada de la pieza. La única restricción que se aplica durante el proceso de captura tiene que ver con el tamaño de las piezas: dado que más adelante no se contemplarán transformaciones de escalado, resulta vital que todas las piezas de un mismo conjunto estén representadas en la misma escala. De esta manera, se reducirá el espacio de búsqueda, evitando un grado de libertad si se contempla el escalado isotrópico, o dos en caso de contemplar el escalado anisotrópico, simplificando así considerablemente el coste final del algoritmo.

El motivo de no incluir las técnicas de segmentación en el proceso es bastante sencillo: existe gran cantidad de técnicas ya desarrolladas cuyos resultados son de alta calidad, y cuyo ámbito de estudio se aleja del objetivo que aquí se persigue. Además, sus resultados pueden ser empleados indistintamente con la técnica de búsqueda propuesta, en la medida en que únicamente es necesaria una imagen blanco/negro para poder trabajar sobre las formas. No obstante, sí resulta interesante comentar brevemente el criterio con el que se extraen los puntos que definen el contorno de la pieza, ya que sus características condicionarán la ejecución del resto de procesos.

En primer lugar, se contemplará la vecindad de los puntos en base al criterio de 8-conexión, que estipula que un punto $p = \{p1, p2\}$ es vecino de otro punto $q = \{q1,q2\}$sujeto a $p \neq q$, si ambos forman parte de la frontera de la forma, y cumplen que $|p1 - q1| \leq 1 \wedge |p2 - q2| \leq 1$. Se dice que un punto perteneciente a la figura forma parte de la frontera si alguno de sus 8-vecinos es de tipo *fondo*. De este modo, los posibles vecinos de un punto que ocupe la posición *5* en la figura 3.1.1.1 deberán encontrarse en las posiciones [1..4], [6..9].

1	2	3
4	5	6
7	8	9

Figura 3.1.1.1: posibles posiciones de los 8-vecinos del punto 5.

En segundo lugar, para unificar el criterio con el que se establece qué punto *p* ocupa la primera posición del contorno *P*, se estipula que este será aquel que verifique:

$$\forall q \in P, q \neq p \begin{cases} q1 > p1 \\ q1 = p1, q2 > p2 \end{cases}$$

Esto implica que el primer punto de una forma será el que tenga la menor coordenada *x* y, en caso de que haya varios, será el que tenga la menor coordenada *y*. En adelante, se asumirá que una imagen tiene su origen en la esquina superior izquierda, y que las coordenadas aumentarán a medida que se avance hacia abajo y hacia la derecha.

En último lugar comentar que, inicialmente, se descartará toda la información asociada al color de los puntos. Posteriormente, cuando se detalle la implementación del algoritmo se replanteará esta afirmación, pero por el momento resulta interesante centrarse únicamente en la búsqueda basada en la forma, y no en las intensidades u otros atributos.

3.1.2. Caracterización en base a la correspondencia de salida

Tal y como se introdujo en el capítulo anterior, existen diversos criterios de clasificación en base a la correspondencia de salida que proporcione el algoritmo. Así, se puede distinguir entre correspondencias densas frente a dispersas, parciales frente a completas, y en base a cómo se representan éstas (correspondencias + transformaciones o sólo correspondencias).

Respecto a la primera distinción, la técnica desarrollada busca establecer una relación densa, intentando obtener la máxima cantidad de asignaciones entre pares de puntos de las dos formas estudiadas. En principio, no se extraen ningunas características representativas de los puntos ni se simplifican las formas por lo que se puede decir que, cuanto mayor sea la densidad de la relación establecida entre dos formas, más posible es que su correspondencia sea buena. Además, una relación dispersa no representa el tipo de correspondencia que se pretende obtener ya que resulta importante que, si dos piezas coinciden, lo hagan a lo largo de una zona concreta y continua, no en puntos específicos esparcidos por sus perímetros.

Por otro lado, diremos que la técnica desarrollada busca establecer una correspondencia parcial de los contornos, ya que se espera que las dos formas coincidan únicamente a lo largo de una zona de su perímetro, no totalmente. No obstante recuérdese del capítulo 2 que, las técnicas de correspondencia parcial, permiten solucionar el problema de correspondencia total, ya que ésta es la correspondencia entre dos subconjuntos de dos figuras que contienen todos sus puntos.

Pese a esta última observación, es necesario considerar que la correspondencia total no se contempla debido a que, durante la extracción del contorno, no se permite la presencia de agujeros en las piezas, que son la única manera de permitir que se dé una correspondencia total. La figura 3.1.2.1 ilustra este concepto, en el que el caso *a)* ilustra una correspondencia parcial, mientras que el caso *b)* ilustra una correspondencia total.

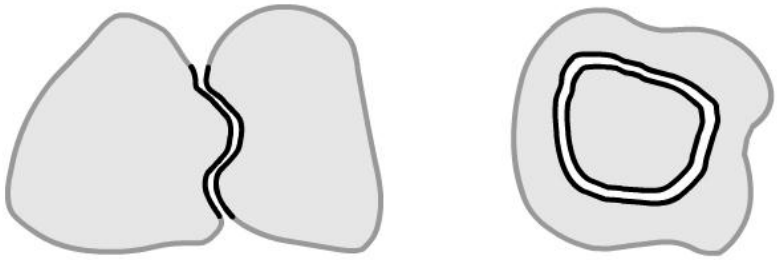

Figura 3.1.2.1: diferencia entre las correspondencias totales y parciales

Es interesante recordar que la búsqueda de correspondencias parciales complica computacionalmente el problema, ya que se produce una explosión combinatoria en el espacio de soluciones. Dado que se buscan correspondencias uno a uno, si se dispone de *n* puntos existen *n!* correspondencias en el espacio de soluciones y, como se contemplan las correspondencias parciales, se concluye que el espacio de soluciones está formado por todos los subconjuntos posibles de esas *n!* correspondencias.

Finalmente, en cuanto al criterio de representación de la correspondencia, salta a la vista que el objetivo final consiste en obtener la correspondencia junto con la transformación que alinea las formas. Además, por la naturaleza de los datos proporcionados, no tiene sentido hablar de deformaciones locales en las piezas y, en base a la restricción planteada durante la fase de adquisición de datos que establece que los tamaños relativos han de mantenerse constantes, se puede decir que la técnica únicamente hará uso de transformaciones rígidas, limitando el problema a un conjunto de soluciones formado por la posición y el ángulo de cada pieza.

3.1.3. Caracterización en base a la función objetivo

En cuanto a la función objetivo empleada durante la búsqueda, como ya se ha establecido, el problema está caracterizado como una alineación rígida, en la que las dos soluciones más comunes se adaptan bien. Mediante *LCP* (*Largest Common Poinset*) el objetivo sería maximizar el número de puntos que están en correspondencia, mientras que apoyándose en la distancia geométrica, la mejor solución sería aquella que minimiza la distancia entre las formas, maximizando al mismo tiempo la longitud del perímetro en contacto.

La primera solución presenta el inconveniente de que el usuario deberá establecer un valor de tolerancia ε que establezca cuándo dos puntos están lo bastante cerca para considerar que hacen un match. El valor de este parámetro dependerá, sobre la aplicación final, de la erosión que hayan sufrido los fragmentos, o de la regularidad que presenten sus perímetros. Además, será necesario considerar que, durante las iteraciones del algoritmo (como se explicará más adelante), será necesario interpolar linealmente puntos intermedios, por lo que será importante distinguir qué puntos son reales, y cuáles intermedios.

En términos generales, se puede decir que el objetivo será maximizar la cardinalidad de puntos en correspondencia para dos formas *P* y *Q* de la siguiente manera:

$$LCP(P,Q) = \sum_{p \in P} Match(p,Q),$$

donde

$$Match(p,Q) = \begin{cases} 1 \rightarrow \exists q \in Q, \|p-q\| < \varepsilon \\ 0 \rightarrow en\ cualquier\ otro\ caso \end{cases}$$

El principal problema de la segunda solución consiste en que se trata de un problema de optimización multi-objetivo, en el que es necesario maximizar la longitud en contacto, pero minimizando al mismo tiempo el error cometido.

No obstante, en torno a la idea general de la distancia geométrica, se puede expresar una solución híbrida de ambas técnicas en la que el objetivo será maximizar la puntuación del match, puntuando en base a la distancia de aquellas asignaciones de pares en las que la ésta sea inferior a una tolerancia dada por el usuario. La puntuación asignada a cada correspondencia válida vendrá dada por la expresión ϵ - *dist(p, q)*.

En términos generales, se puede decir que el objetivo será maximizar la puntuación para dos formas *P* y *Q* de la siguiente manera:

$$Puntuación(P,Q) = \sum_{p \in P} Match(p,Q),$$

donde

$$Match(p,Q) = \begin{cases} \varepsilon - \|p-q\| \ \rightarrow \exists q \in Q, \|p-q\| < \varepsilon \\ 0 \rightarrow en\ cualquier\ otro\ caso \end{cases}$$

De esta manera, se obtiene un mayor poder descriptivo en la función objetivo, ya que cada correspondencia puntuará proporcionalmente a la distancia entre los puntos que la forman, no de manera discreta como en *LCP*.

3.1.4. Caracterización en base al tipo de solución

El objetivo de la técnica es que no sea necesaria la participación del usuario durante el proceso de búsqueda. Así, la inicialización y evolución de la búsqueda se llevarán a cabo de forma totalmente automática, aunque la interpretación de los resultados obtenidos quedará en manos del usuario. Es importante recalcar las palabras *interpretación* y *resultados*, debido a que según sean las formas, pueden obtenerse varios resultados muy similares en base a la función objetivo, y será el usuario quien determine en última instancia qué opción es la más adecuada.

Respecto al ámbito de la búsqueda, debido a que se pretende alinear las formas, resulta necesario considerar la totalidad de los puntos, ya que es preciso verificar que no se producen situaciones ilegales (penetración entre las formas). Esto incrementa el coste computacional de la técnica pero es una restricción necesaria para garantizar la corrección de los resultados. Así, consideraremos que la técnica proporciona una solución global, evitando finalizaciones ante mínimos locales.

Como ya se ha comentado, para acabar con la caracterización de la técnica, la búsqueda se realiza entre pares, contemplando una transformación de alineación.

3.1.5. Caracterización resumida

A modo de resumen de este primer apartado, se presenta la siguiente caracterización completa de la técnica desarrollada:

- *Por forma de entrada:*
    - Representación geométrica: puntos.
    - Dimensionalidad de los datos: 2D.
- *Por correspondencia de salida:*
    - Densa.
    - Parcial.
    - Representación de la correspondencia: correspondencia + transformación, únicamente transformaciones rígidas.
- *Por función objetivo:*
    - Alineación rígida:
        - LCP.
        - Híbrido (LCP + geométrico).
- *Por tipo de solución:*
    - Automática.
    - Global.
    - Entre pares.
    - Paradigma: búsqueda de transformación.

3.2. Representación de las formas

Tal y como se comenta en [3], los aspectos más cruciales en la resolución de problemas de correspondencia de formas tienen que ver con la manera en que se representan éstas, las transformaciones posibles que se consideran, y la métrica de similitud empleada.

Dejando la métrica de similitud para apartados posteriores, a la hora de decidir cómo representar las formas resulta vital considerar qué transformaciones posibles se consideran. Tal y como se ha comentado durante la caracterización de la técnica desarrollada, el problema propuesto consiste en una búsqueda de alineación y correspondencia, haciendo uso únicamente de transformaciones rígidas. Éstas están formadas por la traslación y la rotación.

Considerando que la dimensionalidad delos datos problema es 2D, el espacio de soluciones tendrá seis grados de libertad: para cada pieza se debe establecer una traslación (dos grados de libertad), y una rotación (un grado de libertad). No obstante, con el ánimo de reducir el espacio de búsqueda del algoritmo, nótese como se puede reducir la complejidad fijando la posición y orientación de una pieza, y actuando únicamente sobre la otra. De este modo, el espacio de soluciones queda restringido a tres grados de libertad, no habiendo perdido ningún poder descriptivo al llevar a cabo esta simplificación.

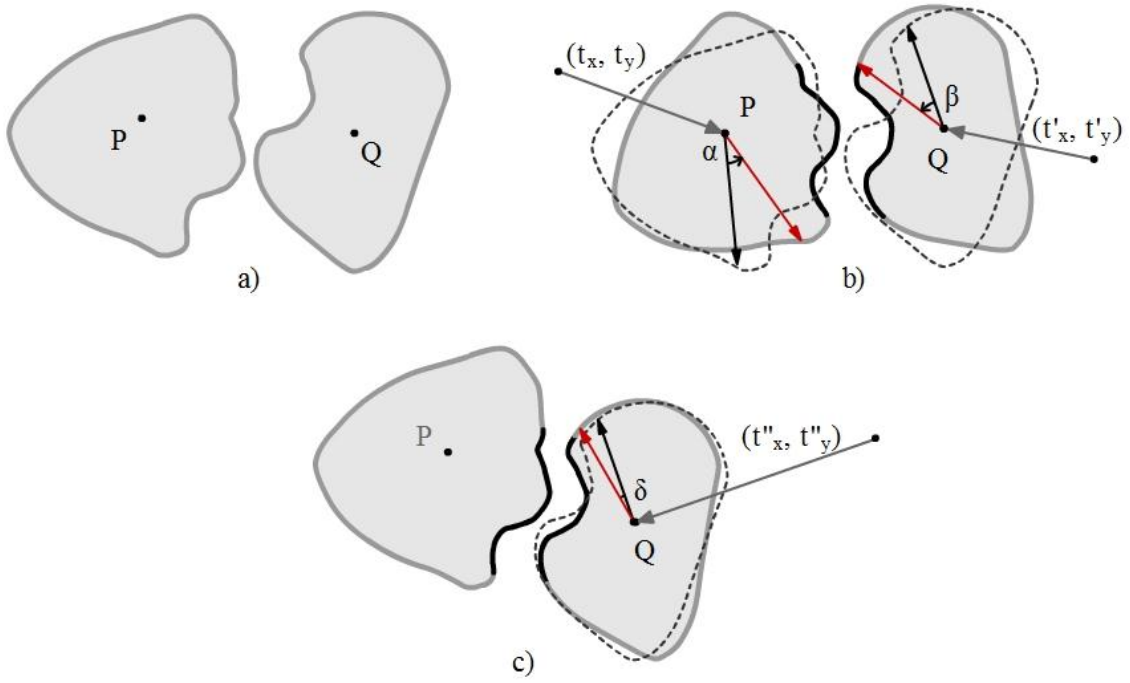

Figura 3.2.1: Reducción de dimensionalidad del espacio de soluciones.

Tal y como se muestra en la figura 3.2.1, para encontrar la correspondencia entre las dos piezas *P* y *Q*, mostradas en *a)*, se pueden deducir dos traslaciones (*T* y *T'*), y dos rotaciones (*α* y

β) obteniendo una solución de seis dimensiones *b)*, o se puede deducir una única traslación (*T''*), y una única rotación (*δ*), obteniendo una solución de tres dimensiones *c)*, que cumple:

$$T'' = T' - T$$
$$\delta = \beta - \alpha$$

De esta forma, cada solución obtenida en el espacio de soluciones de tres grados de libertad equivaldrá a un número infinito de soluciones en el espacio de seis grados de libertad que verifiquen las igualdades propuestas. Además, no sólo se consigue reducir dimensionalidad, sino que se evita la redundancia de soluciones, unificando todas las configuraciones equivalentes en clases únicas, caracterizadas por el incremento de desplazamiento y el incremento de rotación.

3.2.1 Sistema de coordenadas y propiedades

Una vez establecida la dimensionalidad del espacio de soluciones y las transformaciones consideradas para llevar a cabo la alineación de las formas, es preciso centrarse en la representación de éstas. Recordando el capítulo anterior, se plantea una técnica de *hashing* geométrico, cuya idea principal consiste en almacenar en una tabla todas las configuraciones posibles de un grupo de puntos de referencia de manera que, cuando se busque el conjunto que mejor encaja con un conjunto dado, esta búsqueda se realice de forma eficiente. El coste colateral que tiene esta técnica es que se reduce el coste computacional a expensas de aumentar el coste espacial. No obstante, dado que cada vez se compararán dos formas, se asume que la penalización espacial pasará desapercibida. Esta última asunción se lleva al extremo, pre-procesando a priori todos los emparejamientos de puntos dentro de una misma forma, lo que implica un coste espacial de orden cuadrático a la talla del contorno (número de puntos).

Por último, queda por responder la pregunta más importante de la caracterización de las formas, que tiene que ver con qué características emplear para describir de manera eficiente el contorno. Para contestar esta pregunta, resulta muy útil prestar atención a las relaciones entre puntos que permanecen invariantes durante todas las transformaciones: su distancia, y su ángulo relativo. Este camino conduce directamente a una representación polar de los puntos, en lugar de la representación cartesiana en que vienen expresados.

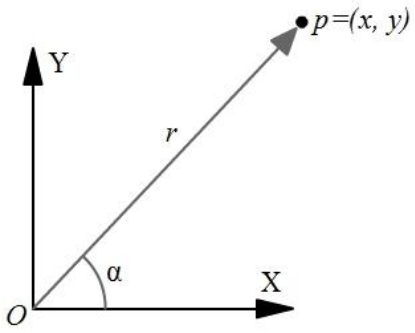

Figura 3.2.1.1: Correspondencia entre coordenadas polares y cartesianas.

La representación polar de un contorno implica que todos sus puntos vienen expresados como una tupla*p = (r, α)*, en la que *r* representa la distancia desde el origen de coordenadas hasta el punto *p*, y *α* el ángulo del vector que conecta el origen *O* con el punto *p*. La figura 3.2.1.1 ilustra esta correspondencia entre sistemas de coordenadas, que dado un punto *p*, a partir de sus coordenadas cartesianas *(x, y)*, se calcula como:

$$(x,y) = (r, \propto) \leftrightarrow \begin{cases} r = \sqrt{(x_1 - X_O)^2 + (y_1 - Y_O)^2} \\ \propto = atan2\big((y_1 - Y_O), (x_1 - X_O)\big) \end{cases}$$

$$atan2(y,x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & x > 0 \\ \pi + \arctan\left(\frac{y}{x}\right) & y \geq 0, x < 0 \\ -\pi + \arctan\left(\frac{y}{x}\right) & y < 0, x < 0 \\ \frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ indefindo & y = 0, x = 0 \end{cases}$$

La función *atan2(y,x)* es muy conocida en problemas geométricos, y devuelve el ángulo de un vector teniendo en cuenta el signo de sus coordenadas. El resultado proporcionado se encuentra siempre comprendido en el intervalo *]-π, π]*, que puede normalizarse a *[0, π/2[* si se incrementa en *2π* los ángulos negativos.

Una de las ventajas de esta conversión supone que, dado que los datos de entrada están discretizados, se puede hacer uso de *LUTs* (acrónimo de la expresión inglesa *Look Up Tables*) para acelerar los cálculos. De esta manera, la tabla pre-calculada de distancias debería tener tantas filas como la distancia máxima en *X* de dos puntos, y tantas columnas como la distancia máxima en *Y*, es decir, las dimensiones de la imagen de entrada. Cada valor se consultaría indexando la tabla con las distancias sobre cada coordenada. Así, la distancia entre un punto *p=(3,7)* y otro punto *q=(8,9)* se consultaría como *LUT[abs(8-3)][abs(9-7)] = LUT[7][4]*, donde *abs(x)* es una función que devuelve el valor absoluto del parámetro dado *x*. La figura 3.2.1.2 ilustra una LUT de distancias, en las que la distancia máxima en ambos ejes es de 5 unidades.

0	1	2	3	4
1	1.4	2.2	3.2	4.1
2	2.2	2.8	3.6	4.4
3	3.2	3.6	4.2	5
4	4.1	4.4	5	5.7

Figura 3.2.1.2: LUT de distancias para una longitud máxima de 5 unidades por eje.

La LUT de ángulos se calculará de forma análoga, teniendo en cuenta que su dimensión será *2n-1* filas, siendo *n* el número de filas de la LUT de distancias, y *2m-1* columnas, siendo *m* el número de columnas de la LUT de distancias, ya que en el caso de los ángulos es necesario tener en cuenta el signo de los valores.

Todavía queda una pregunta que responder en cuanto a la representación de los contornos: qué origen de coordenadas se toma para caracterizar el perímetro de la figura. Como más adelante se verá, para buscar la alineación de los contornos, será necesario llevar a cabo muchas operaciones de cálculo de distancia angular, por lo que para cada figura se calcularán *n* representaciones polares, siendo *n* el número de puntos que la forman. Esto supondrá una aceleración considerable durante el proceso de búsqueda, pero añade un coste de pre-proceso por figura de $O(n^2)$ tanto espacial como temporal. Como se ha comentado, este es el principal problema de las técnicas de hashing pero, dado que únicamente se comparan dos contornos, las ventajas que plantea la técnica superan los inconvenientes colaterales.

Las principales ventajas de este sistema de representación son su invarianza ante las traslaciones, la inmediatez en el cálculo de rotaciones y su linealidad respecto al escalado. Esto es de gran importancia ya que, únicamente eligiendo este sistema de representación, se discretizan dos de los tres grados de libertad del problema (traslación), quedando únicamente por solucionar en el espacio continuo la rotación de una pieza. Así, una configuración se caracterizará por los puntos de inicio en ambos contornos, y por el ángulo de rotación de uno de ellos. Las propiedades respecto al escalado no aportan ninguna ventaja en este caso particular, pero podrían aprovecharse en técnicas más generales.

La figura 3.2.1.3 ilustra las propiedades del sistema de representación ante las traslaciones, mientras que la figura 3.2.1.4 lo hace para las rotaciones, y la figura 3.2.1.5 para los escalados isotrópicos.

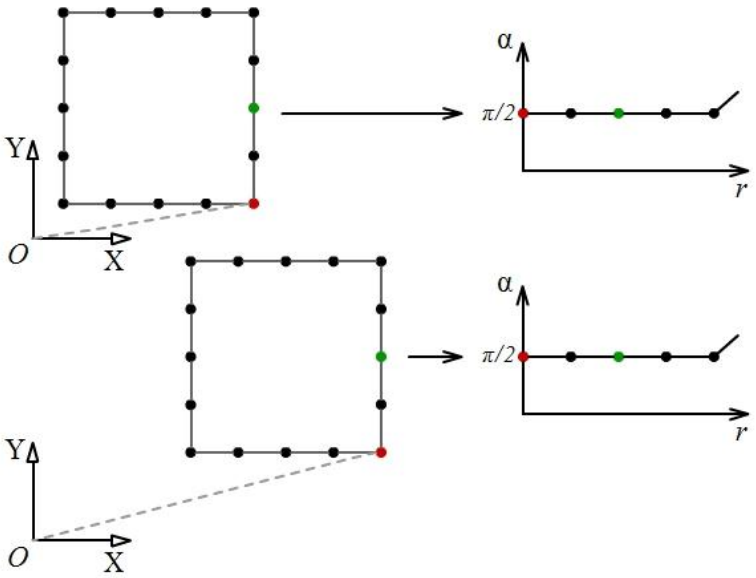

Figura 3.2.1.3: Invarianza de la representación ante traslaciones.

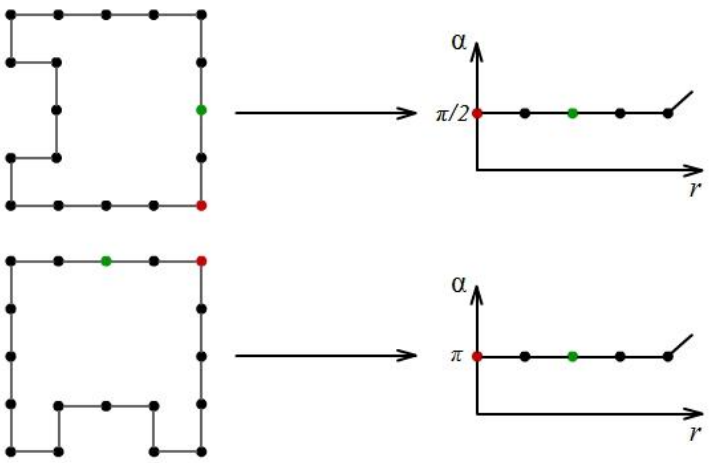

Figura 3.2.1.4: Inmediatez de cálculo ante rotaciones.

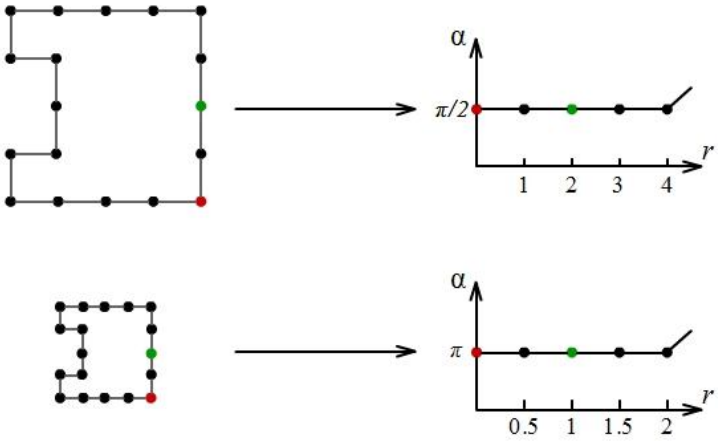

Figura 3.2.1.5: Linealidad ante escalado isotrópico.

3.2.2 Construcción de la representación polar

Una vez consideradas las propiedades del sistema de representación escogido, resulta importante comentar las particularidades de la construcción completa de la representación polar de un contorno, ya que ésta aporta cierta dificultad añadida al problema.

El principal inconveniente deriva de la representación de los ángulos, ya que ésta se apoya en un sistema de coordenadas circular, en el que un ángulo α es equivalente a cualquier otro ángulo $\alpha + 2k\pi$, siendo k un número entero. Este problema aparece cuando un punto q perteneciente al contorno está alineado sobre el eje Y con el punto de inicio p, mientras que su sucesor/antecesor s está inmediatamente debajo de p. Así, el segmento que une a q y a s en la representación polar

tendrá un punto de origen *p'* con distancia r_1 y ángulo 0, mientras que el punto de destino *s'* tendrá una distancia r_2 y un ángulo ligeramente inferior a 2π (en función de la distancia r_2), introduciendo una discontinuidad no deseada en la representación, tal y como ilustra la figura 3.2.2.1.

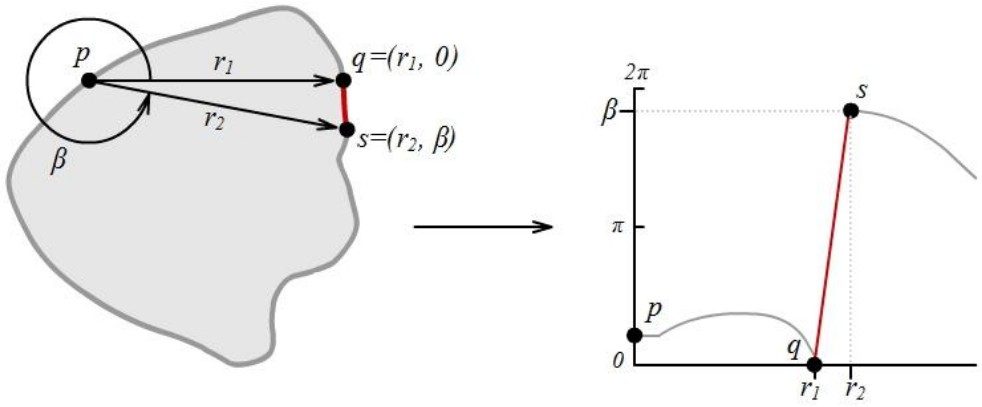

Figura 3.2.2.1: El problema de las discontinuidades.

Aunque no se podrá evitar la presencia de discontinuidades en determinadas configuraciones, conviene llevar a cabo la mayor parte del pre-procesado sobre una representación continua, y trabajar sobre las discontinuidades al final.

Para poder trabajar sin discontinuidades es necesario relajar las restricciones sobre el espacio de representación de los contornos, permitiendo que los ángulos estén expresados sobre el intervalo $]-2\pi, 4\pi[$. De esta forma, la función *atan2* garantiza que, en una primera pasada, los ángulos estarán todos representados en el intervalo $[0, 2\pi[$. Tras llevar a cabo este cálculo, comenzando por el punto de inicio de la representación, se recorrerán todos los sucesores comprobando que el valor absoluto de la diferencia de ángulos entre puntos consecutivos nunca supere π radianes. De ser así, se incrementará o decrementará a los puntos sucesores en 2πradianes para evitar la discontinuidad

Dado que el contorno de la figura es cerrado, desde el punto de inicio hasta el resto debe ser posible representar las coordenadas polares de forma continua dentro de un intervalo de ángulos en el que sus extremos disten menos de 360 grados. Manteniendo un registro del ángulo mínimo obtenido, se puede realizar un desplazamiento posterior de todos los puntos, para garantizar que la representación polar del contorno está contenida entre $[0, 4\pi[$. Como se ha comentado, posteriormente se actuará sobre este intervalo para reducirlo a $[0, 2\pi[$.

El único punto cuya representación es ligeramente diferente del resto es el inicial, ya que a una distancia de cero unidades forma dos ángulos diferentes que sí pueden diferir en más de π radianes entre ellos sin que por ello exista discontinuidad. Por ello, el cálculo de coordenadas polares se llevará a cabo sobre todos los puntos menos el inicial. Finalizado este proceso, y antes de llevar a cabo el desplazamiento que corrija el intervalo de representación, se añadirá un punto al comienzo de la representación polar con $r=0$, y α igual al ángulo del punto posterior al inicial, y otro punto al final con $r=0$, y α igual al último punto insertado en la representación

polar. El siguiente fragmento de pseudocódigo muestra este proceso, dado un punto inicial *i* en una figura *P*.

```
PARATODO Punto p ϵ P, p ≠ i HACER
    p' = CalculaPolar(p-i)
    SI (Anterior(p').α – p'.α) ≤ π ENTONCES p'.α -= 2π
    SI (Anterior(p').α – p'.α) ≥ π ENTONCES p'.α += 2π
    SI p'.α < αmin ENTONCES αmin = p'.α
    Polar.AñadirPunto(p')
FIN PARA
Polar.AñadirPuntoInicio(r = 0, α = Polar[0].α)
Polar.AñadirPunto(r = 0, α = Polar[Polar.nPuntos - 1].α)

PARATODO Punto p ϵ Polar HACER
    SI αmin < 0 ENTONCES p.α += 2π
    SI αmin > 2π ENTONCES p.α -= 2π
FIN PARA
```

Al finalizar la ejecución de este algoritmo, se dispone de una lista ordenada *Polar*, en la que el criterio de orden de los puntos tiene que ver con su consecutividad en el contorno original. De esta manera, para la representación polar de una figura tomando como punto de partida un punto *i*, la posición 0 del vector *Polar*, corresponderá con *i*, la posición 1 con el punto siguiente a *i* siguiendo un sentido de rotación horario, la penúltima posición del vector *Polar* corresponderá con el punto inmediatamente anterior a *i*, y la última posición volverá a corresponder con *i*. Como se ha comentado, el punto inicial tiene dos representaciones diferentes, en base al ángulo que forma con sus puntos anterior y siguiente. La figura 3.2.2.2 muestra el resultado obtenido al calcular la representación polar sin discontinuidades de un triángulo. Nótese como el punto inicial *(amarillo)* está representado dos veces, en base a su ángulo con el sucesor *(rojo)* y antecesor *(verde)*. Además, nótese como la corrección del intervalo de representación ha provocado que los puntos hayan sido desplazados $+2\pi$ radianes.

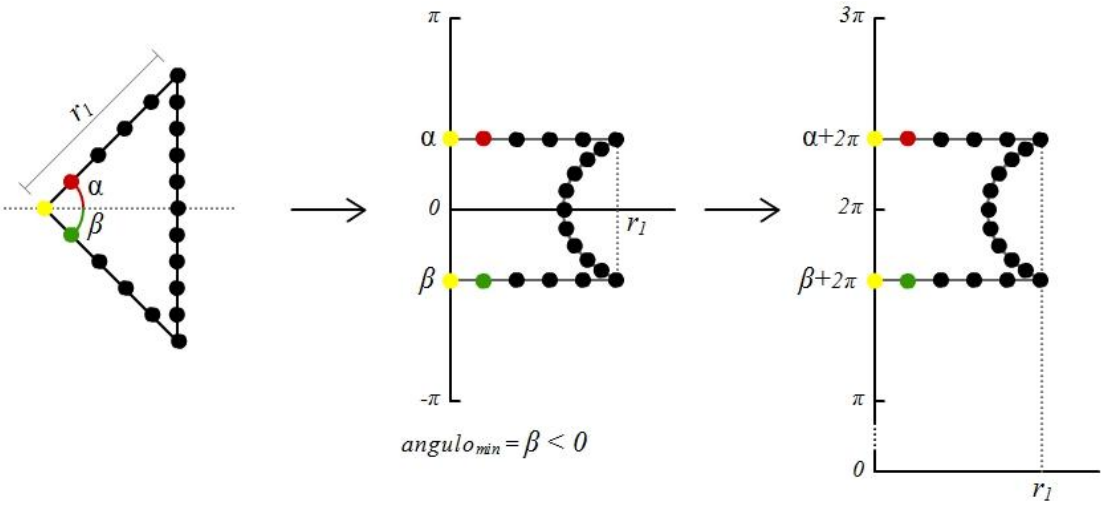

Figura 3.2.2.2: Conversión a sistema polar sin discontinuidades.

La figura 3.2.2.3 muestra el resultado obtenido al calcular la representación polar de la misma figura, tomando un punto de inicio diferente *(amarillo)*. Nótese como el ángulo con el siguiente *(rojo)* es menor que cero, *-π/2*, y como durante todo un primer tramo el ángulo se encuentra en coordenadas negativas. Al alcanzar un punto que iguala la coordenada *Y* de *i*, recordando la definición de la función *atan2(y, x)*, se obtiene un ángulo de *π* radianes, lo que provoca una discontinuidad *(azul)*, siendo necesario incrementar los ángulos de los puntos siguientes en *2π* radianes. Recordando el pseudocódigo planteado a lo largo de este capítulo para obtener la representación polar de un contorno, SI (Anterior(p').α – p'.α) ≤ π ENTONCES p'.α -= 2π.

Una vez llevada a cabo la corrección de discontinuidades, al obtenerse un ángulo mínimo menor que cero, será necesario desplazar todos los puntos *2π* radianes, garantizando que la representación queda contenida en el rango *[0, 4π[*.

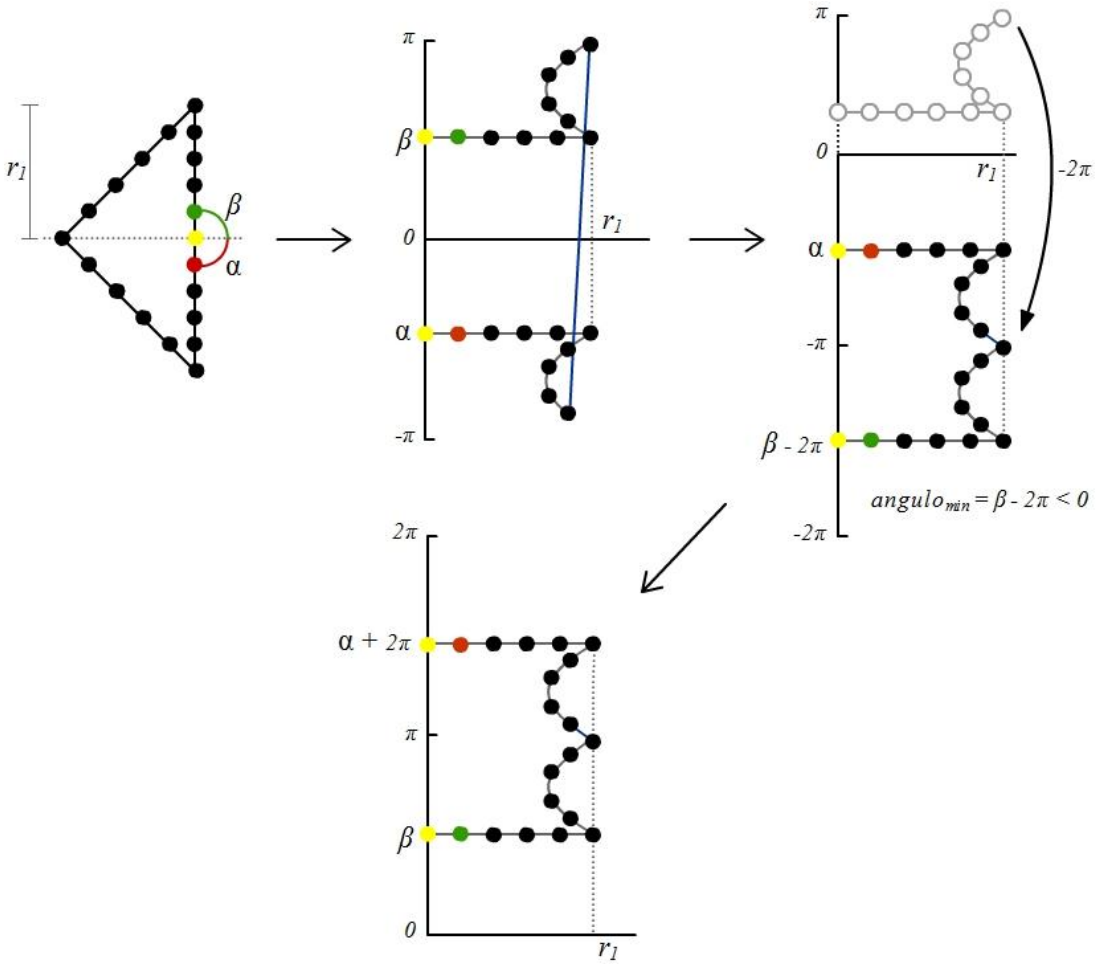

Figura 3.2.2.3: Conversión a sistema polar con discontinuidades.

De los resultados obtenidos en las figuras 3.2.2.2 y 3.2.2.3, llama la atención la gran cantidad de puntos que no aportan ningún poder descriptivo. Así, los puntos que forman los segmentos *[0, r_1]* de ambos ejemplos son redundantes entre ellos, y podrían eliminarse todos, salvo los extremos, eliminando cierta complejidad al problema, con un coste añadido de *O(n)*.

Nótese también como las distancias máximas de cada configuración han sido destacadas mediante el valor r_1. Cuando se lleven a cabo búsquedas de alineación, será interesante disponer de esta medida, por lo que durante la fase de conversión del sistema de coordenadas resulta interesante almacenarla vinculada al contorno. El coste temporal extra que esto añade es constante y, por tanto, despreciable durante el análisis de complejidad de la técnica. Lo mismo ocurre con el coste espacial.

Tanto en la figura 3.2.2.2 como en la 3.2.2.3 se ha obtenido un vector de puntos ordenado según la posición que éstos ocupan en el perímetro del contorno, recorriéndolo circularmente desde el origen, en sentido horario. No obstante, como más adelante se verá, este orden de los puntos no es demasiado conveniente para la búsqueda.

Dado que lo único que interesa del orden natural de los puntos es conservar la relación *siguiente(p)*, y *anterior(p)*, para cualquier punto $p \in P$, en cada punto se anotará una referencia a su siguiente *p.siguiente* y su anterior *p.anterior*, y para cada representación del contorno se guardará una referencia a su punto de inicio *i*. Así, recursiva o iterativamente se podrá recorrer todos los puntos en orden, no permitiendo el acceso aleatorio a cualquier punto. A modo de ejemplo, en la representación actual el tercer punto del contorno es accesible como *Polar[4]*, mientras que en la representación planteada será necesario hacer *i.siguiente.siguiente.*

Representar de esta manera los puntos aporta un relativo coste espacial extra, aunque el peso de una referencia es muy reducido, pero por otro lado, al no permitirse el acceso aleatorio a los puntos de manera ordenada permite organizar el vector con un criterio mucho más interesante. Ya que se buscan puntos que encajen, será necesario detectar que éstos estén próximos entre sí. La proximidad entre puntos sobre un sistema de representación polar tiene que ver con sus distancias al origen y el ángulo que forman (como se explicará más adelante). De este modo, parece interesante ordenar el vector de puntos en base a estos dos criterios.

En la técnica desarrollada, se ha optado por un orden primario en base a la distancia, y un orden secundario en base al ángulo, por lo que el punto que ocupe la primera posición en el vector será el más próximo al origen y, en caso de que haya varios, será aquel que tenga menor ángulo (típicamente los dos primeros puntos serán el inicio *i* en sus dos representaciones, ya que la distancia $r = 0$).

Para llevar a cabo la ordenación de los puntos, será necesario apoyarse en alguna técnica algorítmica bien conocida que proporcione buenos resultados. En la implementación que más adelante se comentará, se ha hecho uso de un algoritmo de inserción ordenada basado en *quicksort*, cuyo coste es de $O(n \log n)$. El origen de este coste deriva de que es necesario recorrer todos los elementos del vector (de ahí *n*), y para cada uno de ellos llevar a cabo una inserción sobre una lista ordenada. Para poder hacer esta inserción, es necesario encontrar la posición que el nuevo elemento debe ocupar que, mediante bipartición supondrá, una media de $\log n$ iteraciones (cada comprobación elimina la mitad de resultados).

Desde esta conversión del vector de puntos, será vital preservar el orden ante cualquier modificación, ya que el objetivo del pre-procesamiento que se está llevando a cabo es el de preparar los datos para su posterior recuperación eficiente.

3.2.3 División del contorno en intervalos convexos.

Hasta el momento, se ha obtenido una representación de los contornos mediante un conjunto de puntos expresados en coordenadas polares, que mantienen la relación de orden y que pueden ser rápidamente localizados en el vector si se conoce su distancia y ángulo. Este cambio de sistema de representación ya supone una gran ventaja respecto al problema original, pero todavía es posible mejorar más la solución.

Dado que los contornos son representados como figuras cerradas, debe ser posible distinguir entre el interior y el exterior para evitar que en la búsqueda de la transformación se produzcan penetraciones que invaliden la solución. Ya que el objetivo de la representación de la fase de pre-procesado consiste en acelerar el proceso de búsqueda, sería interesante que se pudiera distinguir entre estos dos casos de forma inmediata. Para ello, una representación basada en puntos no resulta de gran ayuda.

Para solventar de manera eficiente este problema, se propone representar el contorno mediante polígonos convexos definidos en el espacio polar. Como se detallará a continuación, esta operación supone un coste de *O(n)*, y aporta grandes beneficios.

En primer lugar, resulta necesario especificar con mayor detalle la topología de estos polígonos. En vista a los resultados obtenidos en la etapa anterior, tómese como ejemplo la figura 3.2.2.2, el trapecio parece la forma geométrica convexa que mejor se adapta al objetivo propuesto: resulta fácil de obtener, de representar, y aporta todas las ventajas propias de los polígonos convexos. La figura 3.2.3.1 ilustra el criterio con el que se pretende descomponer el contorno en trapecios, tomando como punto de partida el resultado obtenido en la figura 3.2.2.1.

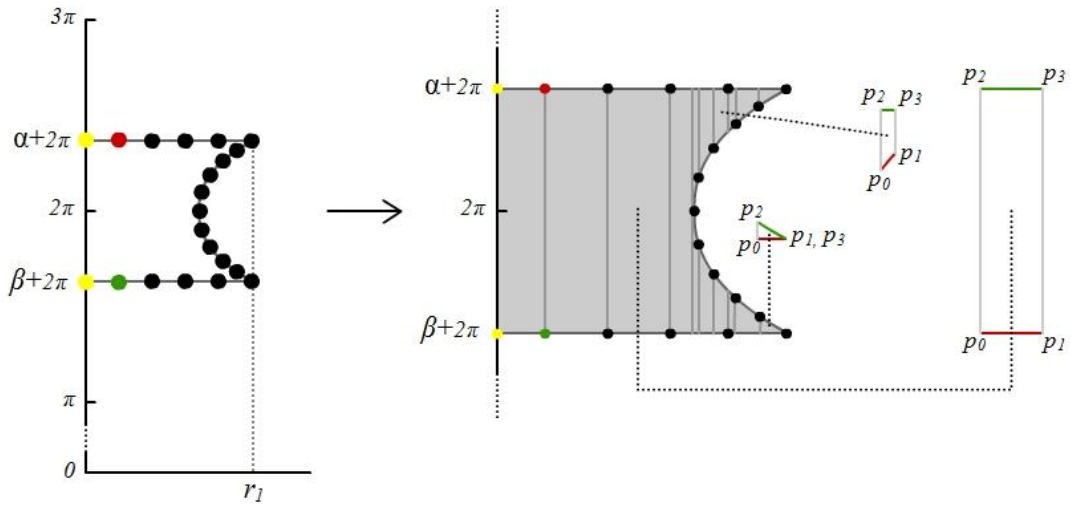

Figura 3.2.3.1: Topología de los polígonos convexos.

Como se puede apreciar, cada polígono convexo está caracterizado por cuatro puntos (p_0, p_1,p_2,p_3), que están alienados dos a dos sobre el eje de las distancias. De esta forma, las distintas formas que se pueden tomar van desde el trapezoide, en el que dos ejes son paralelos y los otros

dos no, hasta el paralelogramo / rectángulo / cuadrado, en los que ambos pares de ejes son paralelos, teniendo como caso especial el triángulo, en el que se asumirá que un par de puntos ocupa la misma posición. Todas estas formas tienen en común que pueden ser definidas mediante dos segmentos p_0p_1 y p_2p_3 cuyos orígenes y finales están alienados sobre el eje de las distancias, y a los que llamaremos *segmento inferior* y *segmento superior*, respectivamente.

Nótese como los puntos que definen cada polígono no tienen por qué corresponder con los puntos de la figura aunque, necesariamente, todos los puntos de la figura pertenecerán a un polígono, por lo menos. Esto hace que sea necesario interpolar nuevos puntos intermedios para poder llevar a cabo la subdivisión.

La figura 3.2.3.2 ilustra la correspondencia entre los polígonos definidos sobre el espacio polar y sus equivalentes cartesianos. Al corresponder un polígono polar con una sección de disco en el espacio cartesiano, su poder descriptivo será de gran ayuda ante rotaciones respecto al punto de inicio, ya que ambas circunferencias exterior e interior están definidas tomando como centro la posición del centro de rotación.

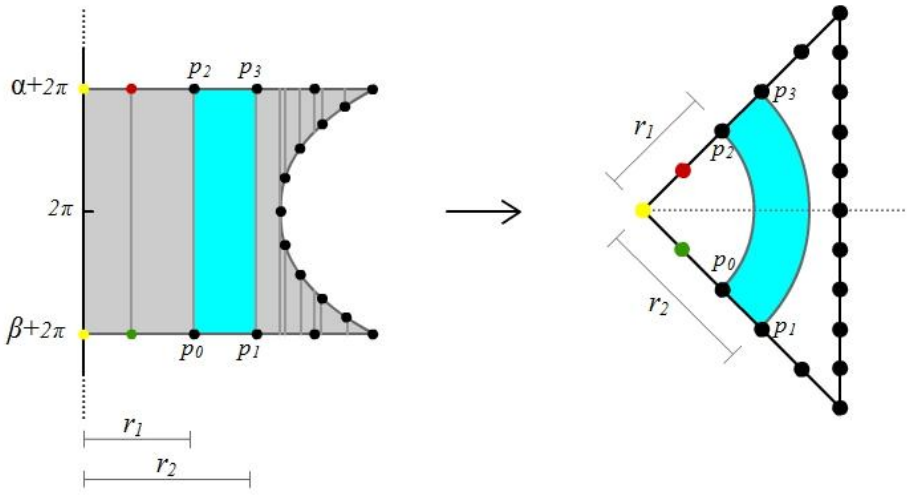

Figura 3.2.3.2: Equivalencia entre polígonos polares y sus homólogos cartesianos.

Respecto a la interpolación decir que, con el ánimo de simplificar la notación y los gráficos, se asume que puede llevarse a cabo de forma lineal sobre el espacio polar aunque, en realidad, sería necesario llevarla a cabo sobre un espacio cartesiano, ya que el ángulo que se obtiene no es lineal respecto al segmento que ocupa el nuevo punto.

Para simplificar el proceso de la subdivisión, se procederá en tres fases: en primer lugar, se crearán todos los puntos intermedios necesarios, y se actualizarán las referencias *siguiente* y *anterior* de cada punto. En segundo lugar, se crearán los polígonos a partir de los puntos obtenidos. Finalmente, se normalizará el resultado obtenido para que la representación de los ángulos esté contenida en el intervalo $[0, 2\pi[$. Como se irá viendo, el hecho de haber ordenado los puntos en base a sus coordenadas hará que cada fase tenga únicamente un coste de $O(n)$.

Para crear todos los puntos intermedios necesarios, se debe garantizar que ningún punto tendrá sobre/bajo sí ninguna arista, únicamente puntos. Dicho de manera más formal:

$$\forall p \in P, \nexists s, t \in P \rightarrow (t.siguiente = s \ \vee t.anterior = s) \wedge \big((t.\propto > p.\propto \wedge s.\propto < p.\propto) \vee (t.\propto < p.\propto \wedge s.\propto > p.\propto)\big)$$

Para ello, en primer lugar, se recorrerán todos los puntos almacenando en un vector todas las distancias sin duplicados. Hecho esto, siguiendo el criterio de ordenación del vector *Polar*, se recorrerán de izquierda a derecha, y de abajo a arriba todos los puntos del contorno. Para cada uno de ellos se comprobará si el segmento que forma con su sucesor/predecesor de la derecha debe ser dividido y, de ser así, se calcularán los puntos intermedios, actualizando las relaciones *anterior* y *siguiente*, para poder mantener el orden.

Para ejecutar este algoritmo con coste lineal serán necesarios dos índices, *j* y *k*, sobre el vector *Polar* y sobre el vector de distancias respectivamente. Nótese que, como el vector de distancias ha sido generado consultando ordenadamente los puntos del vector *Polar*, se encuentra directamente ordenado de forma creciente.

El índice *j* se inicializará a cero, y recorrerá todos los puntos del vector *Polar* (*Polar.nPuntos*). El punto *Polar[j]* será, en cada iteración, el punto de menor distancia del segmento estudiado. De esta manera, se podrán dar 4 casos diferentes:

1. El punto *Polar[j].siguiente* está más a la derecha, por lo que se debe tratar el segmento definido como *(Polar[j], Polar[j].siguiente)*.
2. El punto *Polar[j].anterior* está más a la derecha, por lo que se debe tratar el segmento definido como *(Polar[j], Polar[j].anterior)*.
3. Ambos puntos están más a la derecha, por lo que es necesario tratar los dos segmentos.
4. Ninguno de los dos puntos está más a la derecha (están más a la izquierda o son iguales), por lo que no será necesario refinar ningún segmento para este punto.

Respecto al índice *k*, su objetivo es establecer a qué distancias se deben dividir los segmentos y se inicializará a 1, ya que el primer valor del vector *distancias* siempre tendrá el valor cero, no siendo necesario considerar este punto. Cuando se haya decidido sobre qué segmento se actúa, el valor de *k* deberá incrementarse hasta alcanzar el primer elemento que verifique que *distancias[k] >Polar[j].r*. *k* nunca se desplazará hacia detrás ya que, como los puntos están ordenados por su distancia, una vez visitado *Polar[j]*, cualquier punto *Polar[j+x]* tendrá una distancia superior. De esta manera, ambos índices avanzarán al mismo tiempo hasta que se hayan recorrido todos los puntos, garantizando el coste lineal.

Una vez seleccionado el segmento a tratar y decididas las distancias en que debe dividirse, mediante una interpolación lineal se calcularán los puntos intermedios de la forma:

$$p_{nuevo}.x = distancias[k]$$
$$p_{nuevo}.y = y_0 + \left(\frac{p_{nuevo}.x - x_0}{p_1.x - p_0.x} * (p_1.y - p_0.y)\right)$$

Por último, para mantener enlazados los puntos con *siguiente* y *anterior*, será necesario distinguir dos casos: que el segmento estudiado se defina como *(Polar[j], Polar[j].siguiente)*, o que se defina como *(Polar[j], Polar[j].anterior)*. En el primer caso, únicamente se podrá garantizar que p_{nuevo}*.anterior* será el último punto que se haya insertado (*Polar[j]* en la primera iteración). En el segundo caso únicamente se podrá garantizar que p_{nuevo}*.siguiente* será el último punto que se haya insertado (*Polar[j]* en la primera iteración). Para completar el resto de relaciones, será necesaria una segunda pasada analizando los datos disponibles tras esta iteración. Si p_{nuevo}*.anterior* = *p* entonces *p.siguiente* = p_{nuevo}.

Como ambas pasadas recorren todos los puntos, nunca retroceden, y llevan a cabo operaciones de coste constante, la complejidad de esta primera parte se puede decir que es *O(n)*. En el siguiente capítulo, cuando se comente la implementación, se detallará más el funcionamiento de esta subdivisión de segmentos.

Una vez se dispone de la representación polar que garantiza que sobre/bajo cualquier punto únicamente hay otros puntos, no segmentos, se puede comenzar con la segunda fase del proceso, que consiste en crear los polígonos convexos.

El algoritmo de subdivisión comenzará con una lista vacía, *Poligonos*, sobre la que se irán haciendo inserciones ordenadas de nuevos polígonos, y con un polígono también vacío, caracterizado por cuatro puntos. Para que se mantenga el orden propuesto en los puntos (izquierda a derecha, arriba a abajo), basta con estudiar los puntos tal y como están ordenados en el vector *Polar*. Para cada punto, se considerarán únicamente su predecesor y/o antecesor cuya distancia sea mayor que el punto estudiado (esté más a la derecha), lo que plantea 3 casos:

1. Únicamente el punto anterior o siguiente están más a la derecha, con lo que se define un único segmento.
2. Tanto el predecesor como el sucesor están más a la derecha que el punto estudiado, por lo que se definen dos segmentos.
3. Ni el sucesor ni el predecesor están más a la derecha, por lo que no se define ningún segmento.

La figura 3.2.3.3 ilustra estos tres casos, distinguiendo el caso 1 en dos variantes *1.a, 1.b*, que son las que dan lugar a los polígonos triangulares. El punto marcado en amarillo corresponde con el punto que se está estudiando, mientras que los puntos marcados en verde corresponden con los anteriores y/o sucesores que se están considerando. Los segmentos en azul son los que se obtienen en la iteración ilustrada.

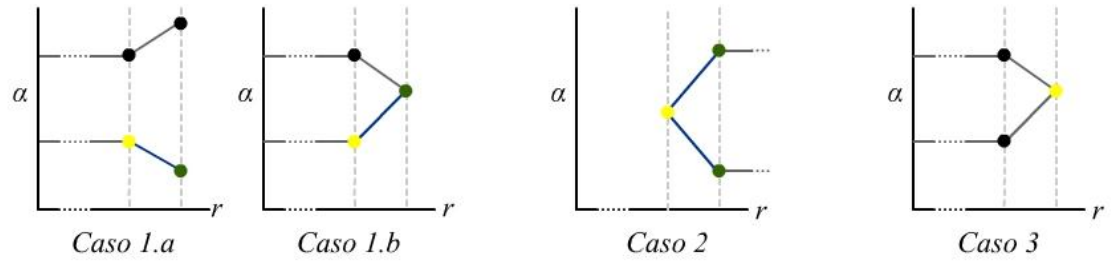

Figura 3.2.3.3: distintos casos posibles en la creación de polígonos convexos.

Como se puede observar, cada punto estudiado puede originar entre cero y dos segmentos, que formarán parte de algún polígono convexo. El problema consiste en que la división de casos planteada no permite determinar qué segmentos serán *segmento inferior*, y cuales *segmento superior*. Para poder eliminar esta ambigüedad, es necesario contemplar el orden en que vayan apareciendo los segmentos. Según [70], si una figura es cerrada, y se intersecta mediante dos líneas paralelas, los pares de segmentos que aparezcan dos a dos, siguiendo la dirección de una de las líneas, forman polígonos cerrados, contenidos dentro de la figura. De este modo, será necesario recorrer los puntos de abajo a arriba, identificar los segmentos que se forman y, procediendo de menor a mayor, insertarlos en el polígono activo (recuérdese que el algoritmo comienza con un polígono vacío). Cada vez que el polígono activo tenga cuatro puntos, se almacenará en la lista *Polígonos* y se creará uno nuevo vacío.

La figura 3.2.3.4 muestra la evolución parcial del algoritmo sobre todos los puntos que comparten la misma distancia.

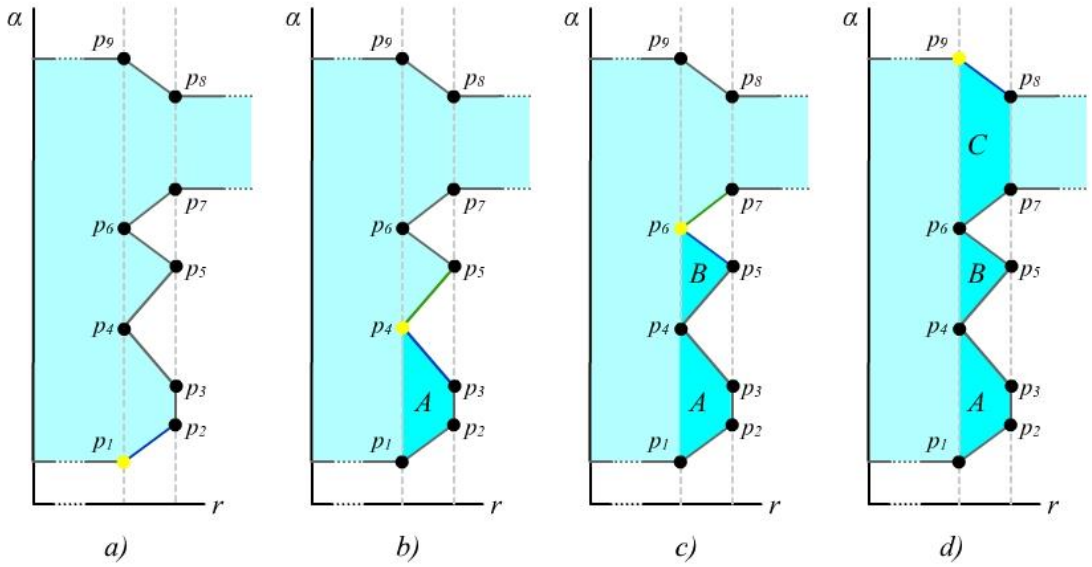

Figura 3.2.3.4: distintos casos posibles en la creación de polígonos convexos.

Nótese como en el paso *a)*, se ha comenzado por el punto de menor ángulo (ya que así está ordenado el vector *Polar*), y en los pasos sucesivos se ha avanzado hacia arriba. En *a)* se han insertado dos puntos en el polígono activo (p_1 *y* p_2) pero, al no haber todavía cuatro puntos, el algoritmo ha continuado con el siguiente punto. En *b)*, ha sido necesario ordenar los dos segmentos obtenidos (caso 2 de la figura 3.2.2.3), para insertar primer el segmento (p_4 *y* p_3), marcado en azul, y posteriormente el segmento (p_4 *y* p_5), marcado en verde. En este punto, es necesario remarcar dos detalles: en primer lugar, tras la primera inserción, el polígono activo contaba con cuatro puntos, por lo que se ha añadido a la lista (polígono *A*), y se ha creado un nuevo polígono activo vacío sobre el que se ha insertado el segmento (p_4 *y* p_5). En segundo lugar, nótese como la nomenclatura empleada para definir los segmentos durante las inserciones se corresponde con la definición de la topología de los polígonos (el primer punto insertado es el inferior izquierdo, el segundo es el inferior derecho, el tercero es el superior izquierdo, y el cuarto es el superior derecho). En el paso *c)* se ha actuado del mismo modo que en el *b)*, insertando primero el segmento inferior, que cierra el polígono *B*, y creando un nuevo polígono

activo en el que se ha insertado el segmento superior. La particularidad de *B* consiste en que se trata de un triángulo, formado por los segmentos (p_4, p_5) y (p_6, p_5). Nótese que el punto p_5 aparece dos veces en la representación del polígono, permitiendo así generalizar la descripción de éstos. Finalmente, en *d)*, se ha insertado el último segmento que completa al polígono activo. Resulta importante destacar que, tras cada conjunto de puntos en la misma distancia, el último polígono haya sido cerrado, ya que se violaría la definición de éstos si los orígenes y finales de sus segmentos no estuvieran alineados. Tal y como se demuestra en [70], si la forma representada estaba cerrada, esto siempre sucederá.

Una vez finalizada la ejecución, se dispone de una representación del contorno en forma de polígonos polares convexos que, además, están ordenados según el mismo criterio que los puntos polares (el primer contorno es aquel cuyo p_0 tiene una distancia menor. En caso de que haya múltiples, el primer contorno será aquel cuyo p_0 tenga un ángulo mínimo). De este punto en adelante, el vector *Polar*, ya no será necesario, por lo que se podrá eliminar liberando recursos.

No obstante, queda un último paso: dado que los puntos polares están representados en el intervalo $[0, 4\pi[$, los polígonos obtenidos también lo estarán. Con el ánimo de facilitar los cálculos durante la búsqueda, resulta conveniente que no existan ambigüedades en el sistema de representación. Esto implica que cada punto debe tener una única representación, cosa que no ocurre actualmente, ya que $k = k + 2\pi$. Así, será necesario normalizar todos los polígonos para que estén comprendidos en el intervalo $[0, 2\pi[$.

Para llevar esto a cabo, para cada polígono, habrá que comparar su ángulo mínimo α_{min} con su ángulo máximo α_{max} incluidos en los segmentos inferior y superior, respectivamente. Sus valores permitirán distinguir tres casos:

1. $\alpha_{min} \geq 0 \wedge \alpha_{max} < 2\pi$: si el polígono está dentro del intervalo $[0, 2\pi[$, no es necesario hacer ningún cambio.
2. $\alpha_{min} \geq 2\pi \wedge \alpha_{max} < 4\pi$: si el polígono está totalmente fuera del intervalo $[0, 2\pi[$, no será necesario dividirlo, pero tendrá que ser desplazado -2π radianes para garantizar que quede dentro de $[0, 2\pi[$.
3. $\alpha_{min} < 2\pi \wedge \alpha_{max} \geq 2\pi$: si el polígono intersecta con la recta $\alpha = 2\pi$, será necesario dividirlo en dos polígonos: el primero de ellos estará definido por el segmento inferior del polígono original, y por un nuevo segmento superior cuyo origen y fin corresponderá con los del segmento inferior en *r*, pero cuyo ángulo será 2π. El segundo polígono tendrá un segmento superior expresado como ($p_3 - 2\pi$, $p_4 - 2\pi$) y un segmento inferior cuyo origen y fin corresponderá con los del segmento superior en *r*, pero cuyo ángulo será *0*.

Nótese como el sistema de representación final estará libre de ambigüedades para todos los valores de α, excepto los extremos ($0 = 2\pi$). Esto tiene cierto sentido si se considera que únicamente aparecerán puntos ambiguos en segmentos que hayan tenido que ser divididos, lo que en cierta forma sirve para expresar la continuidad del espacio ocupado.

La figura 3.2.3.5 muestra gráficamente los tres casos comentados. Nótese que los cálculos realizados tiene un coste constante, por lo que la subdivisión de polígonos tiene un coste total de *O(n)*, no incrementando la complejidad del pre-procesado.

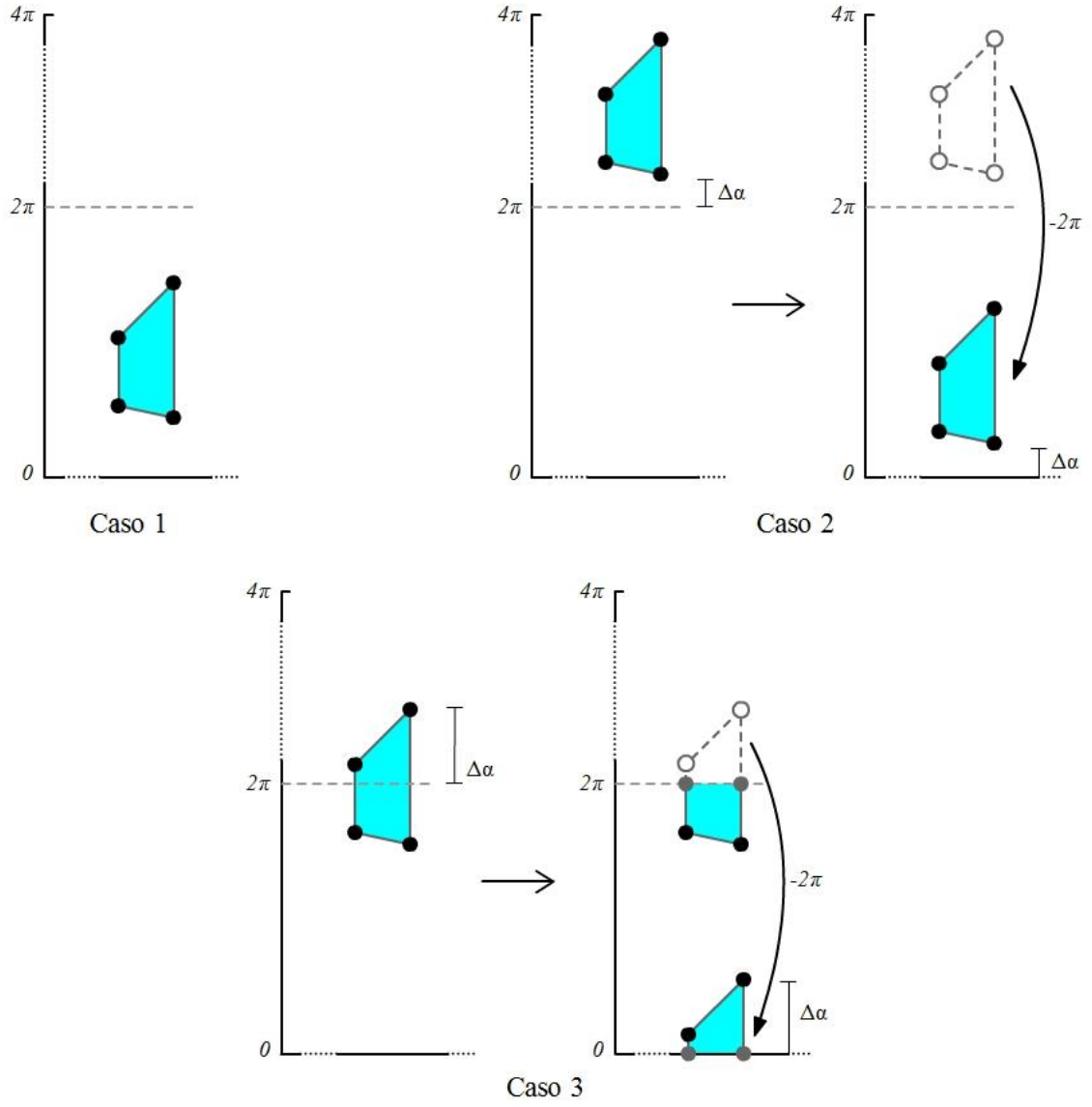

Figura 3.2.3.5: distintos casos posibles durante la normalización de los polígonos convexos.

Obtenida esta representación diremos que la forma está totalmente pre-procesada para el punto de inicio escogido. El coste total del pre-procesamiento será $O(n^2 \log n)$:

- *n*: tomar como punto de partida todos los puntos.
- *n*: para cada punto es necesario recorrer todos los demás puntos durante:
  - Cálculo de coordenadas polares de los puntos, sin discontinuidades.
  - Normalización de la representación al intervalo *[0, 4π[*
  - Ordenación de los puntos (añade un coste extra de *log n*).
  - Cálculo de polígonos convexos.
  - Normalización de la representación al intervalo *[0, 2π[*

3.3. Búsqueda de alineaciones

Una vez se dispone de ambos contornos representados en polígonos polares convexos, para cada posible punto de inicio, se puede comenzar el proceso de búsqueda. Como se ha comentado durante el capítulo 2, este se va a dividir en dos partes: para cada *configuración* se estudiará primero las transformaciones que alinean los contornos, y posteriormente se analizarán las correspondencias. A lo largo de este apartado, se analizará la búsqueda de alineaciones y en el apartado siguiente se analizarán las correspondencias. Así, el objetivo a alcanzar durante esta etapa será obtener un conjunto finito de transformaciones que alineen cada *configuración*.

El término *configuración* hace referencia al criterio con el que se está estudiando la alineación y las correspondencias. La idea de la técnica desarrollada consiste en enfrentar cada par posible de puntos de ambas figuras, *P y Q*, buscar una alineación y calcular su correspondencia. Esta característica, que automáticamente supone un coste cuadrático, encasilla al algoritmo bajo la denominación *ingenuo*, ya presentada en el capítulo 2. La principal bondad de este tipo de algoritmos consiste en que garantizan soluciones exactas, no corriendo el riesgo de detener la búsqueda ante mínimos locales. El inconveniente, el coste.

A los puntos enfrentados se les conocerá como puntos de inicio i_p, i_q y quedará implícito que su enfrentamiento supone que ambos ocupan la misma posición (se asumen las traslaciones), y que las rotaciones se llevarán a cabo bajo el sistema de coordenadas local cuyo centro está precisamente en la posición que los puntos ocupan. En este punto es cuando todo el pre-procesamiento cobra sentido: si ambos puntos de inicio están en la misma posición, sus representaciones polares pueden superponerse y estudiar la alineación y correspondencia sobre este espacio, en el que ya se ha visto que las traslaciones no afectan a los puntos y las rotaciones se calculan como desplazamientos uniformes sobre el eje de ordenadas.

Así, una *configuración* estará caracterizada únicamente por los puntos de inicio de ambas figuras $C(i_p, i_q)$ donde $i_p \epsilon P$, $i_q \epsilon Q$. La figura 3.3.1 ilustra una configuración entre dos contornos de forma triangular, en la que los puntos de inicio están marcados en amarillo.

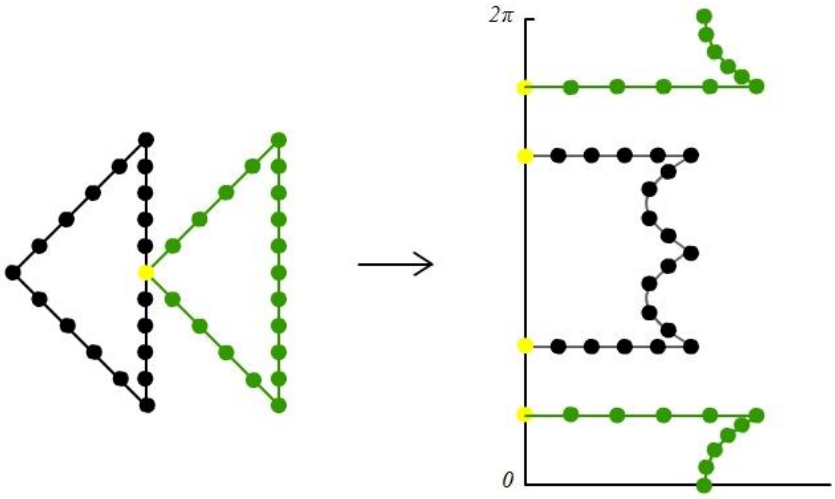

Figura 3.3.1: configuración entre dos contornos, y correspondencia de coordenadas polares.

3.3.1 Alineaciones

Se considera que dos formas *P, Q* están alineadas cuando existen, al menos, dos pares de puntos (p_1, q_1), (p_2, q_2) tal que $p_1, p_2 \in P$, $q_1, q_2 \in Q$, $p_1 = q_1$, $p_2 = q_2$ sin que exista una penetración, es decir, cuando al menos dos puntos de cada una de ellas están en contacto con sus homólogos de la otra. Se considera que la alineación es máxima en este caso pues, una rotación que las aproximase más incurriría en una penetración, considerada como situación ilegal en el problema.

Una alineación se caracteriza por la configuración sobre la que se está trabajando, y por el ángulo de rotación de una de las dos piezas que pone en contacto a los contornos $A(i_p, i_q, \alpha)$. Así, tal y como ilustra la figura 3.3.1.1, para la configuración mostrada en la figura 3.3.1, el número de alineaciones posibles no es sólo uno, sino que pueden darse alineaciones múltiples. Obsérvese como en la representación polar el cálculo de las alineaciones es mucho más sencillo que en la representación cartesiana, ya que no hay que calcular rotaciones sino traslaciones sobre un único eje.

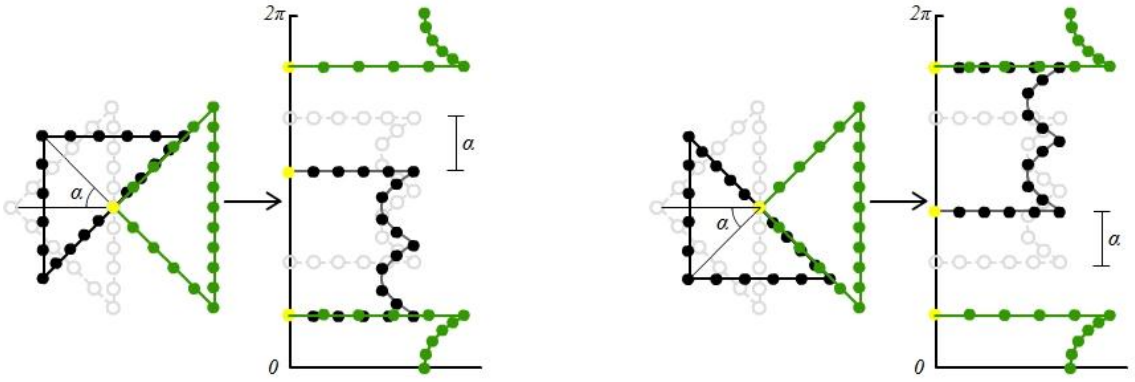

Figura 3.3.1.1: alineaciones posibles para una configuración dada.

A la vista de que puede haber más de una solución en el cálculo de la alineación, será interesante conocer qué casos pueden darse para afrontar adecuadamente el problema. De hecho, aunque los casos típicos serán cero alienaciones, una, o dos, pueden darse casos en los que el número de alineaciones posibles sea superior a dos por lo que, para afrontar el problema de una manera genérica, es necesario considerar que el número de resultados no está acotado superiormente.

La figura 3.3.1.2 ilustra los tres casos más típicos de alineación y un cuarto caso que, pese a ser atípico, debe ser considerado. Así, la imagen *a)* ilustra una configuración para la que no es posible obtener una alineación, consecuencia de que para cualquier ángulo existe una penetración entre contornos. La imagen *b)* ilustra una configuración para la que sólo hay un resultado debido a que, en algún punto, el espacio libre que deja el contorno fijo (*negro)* corresponde con el espacio que ocupa el contorno móvil (verde*)*. La imagen *c)* ilustra una configuración en la que son posibles dos alineaciones, debido a que el espacio libre que deja el contorno fijo es mayor que el espacio que ocupa el contorno móvil. Este será el caso más habitual. Finalmente, la imagen *d)* muestra una configuración muy atípica en la que los

contornos permiten obtener múltiples alineaciones. Nótese como, en este último caso, no es posible acotar superiormente el número de alineaciones posibles.

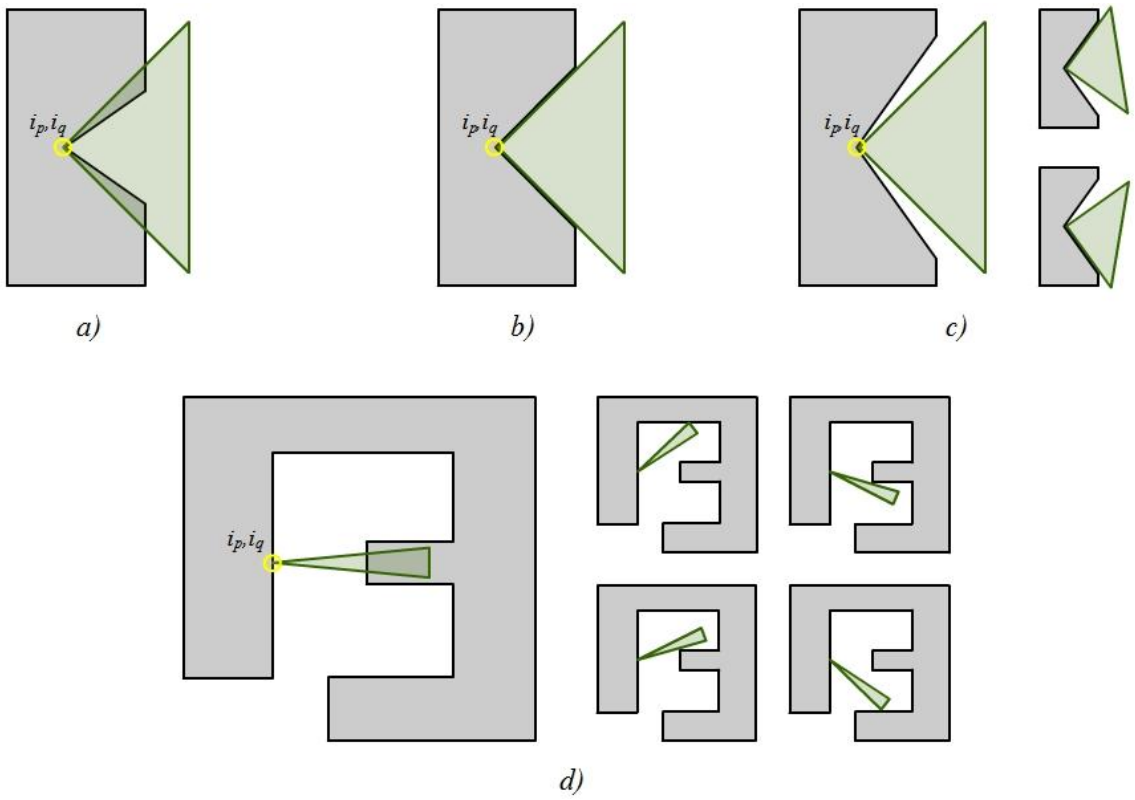

Figura 3.3.1.2: diferentes casos en base al número de soluciones posibles.

3.3.2 Algoritmo de búsqueda

Una vez definidas las alineaciones es el momento de comentar el algoritmo que, dada una configuración, devuelve una lista con todas las alineaciones posibles, pudiendo ésta estar vacía si se da el caso *a)* ilustrado en la figura 3.3.1.2. Por los motivos que se han comentado anteriormente, la búsqueda se realizará sobre el espacio polar haciendo uso de los polígonos convexos obtenidos durante el pre-procesado, que están almacenados en un vector *Polígono*, ordenado en base a la distancia (clave primaria) y al ángulo (clave secundaria) en sentido ascendente. Será en este punto donde el pre-procesado cobrará sentido, ya que la consulta de los polígonos en el vector tendrá un coste constante *O(k)*, permitiendo calcular la alineación con un coste lineal al número de puntos *O(n)*.

Si se observa con detenimiento la figura 3.3.1.1, se constata que la alineación se produce cuando dos puntos de la representación polar están en contacto sin incurrir en una penetración. Así, será necesario detectar estas situaciones ilegales para evitar obtener falsas alineaciones y, adicionalmente, será conveniente distinguir cuanto antes las configuraciones que no permiten ninguna alineación para poder detener la búsqueda.

Por otro lado, pese a que no queda claramente reflejado en la figura 3.3.1.1, es necesario considerar que los puntos de inicio de ambas figuras i_p, i_q siempre están en contacto. La justificación es bastante obvia: dado que ambos se encuentran en el origen de coordenadas, su posición debe ser la misma. Además, ya que el valor de *r* en ambos casos es igual a 0, independientemente de su ángulo estarán en contacto. Una última consideración a cerca de estos puntos es que, pese a estar representados por duplicado ambos, se considera que sólo hay un contacto. Las dos representaciones únicamente sirven para identificar los ángulos de los dos segmentos que conectan el punto de origen con su predecesor y su antecesor.

De este modo, el problema de buscar dos puntos en contacto se convierte en el problema de buscar un solo punto de contacto sin considerar el punto de inicio. Para ello, será necesario conocer qué intervalos de rotación de la pieza móvil permiten situaciones legales. Las soluciones del problema de alineación se encontrarán en los extremos de estos intervalos. Retomando el caso propuesto en la figura 3.3.1.1, la figura 3.3.2.1 ilustra este concepto.

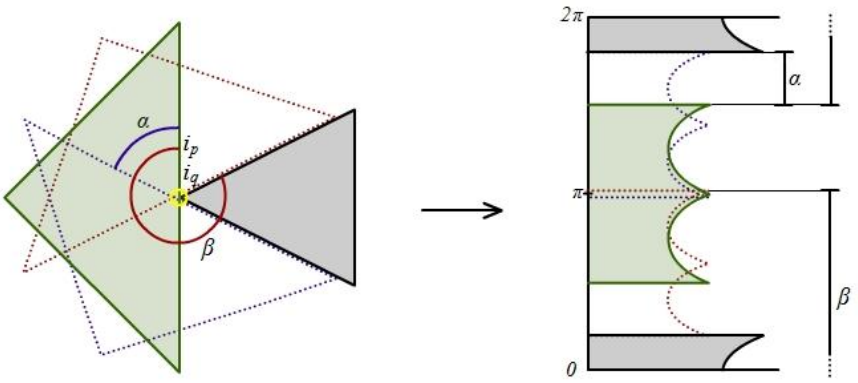

Figura 3.3.2.1: intervalos de rotación que proporcionan situaciones legales.

En base a los resultados mostrados en la figura 3.3.2.1, diríamos que el único intervalo de rotación válido es *[0..α], [β, 2π]*. Pese a que este intervalo esté expresado en dos sub-intervalos, nótese como $2\pi = 0$, por lo que ambos forman un único intervalo de rotación, *[β, α]*, cuyos extremos son las soluciones al problema de la alineación.

Para poder resolver de forma automática el problema de la alineación, se procede en dos fases: en primer lugar se buscan los ángulos de contacto (que pueden ser más de dos) y en segundo lugar se calculan los intervalos en los que hay situaciones legales. Además, en lugar de calcular primero los ángulos legales para toda la figura y, posteriormente, calcular los intervalos que proporcionarán el resultado, se ejecutarán ambas fases a medida que se va estudiando cada grupo de puntos de la representación polar que comparten su coordenada *r*, de izquierda a derecha (distancias ascendentes) y de abajo a arriba.

Alternar entre la búsqueda de ángulos y el cálculo de intervalos tiene una ventaja: cuando un conjunto de puntos no proporcione ningún ángulo válido, el cálculo de espacio libre detectará que la configuración no permite ninguna alineación y finalizará la búsqueda sin necesidad de consultar los puntos posteriores. Cuanto menos regular sea una forma, más casos como este se darán.

Ya que se van a comprobar los ángulos de alineación para cada conjunto de puntos que compartan la distancia *r* al origen, será necesario contemplar todos los polígonos convexos de ambos contornos que estén definidos en *r*. El objetivo será estudiar, para cada polígono convexo del perfil móvil que esté definido en *r*, las rotaciones que son legales. Para ello, habrá que contrastar sus extremos con los de todos los polígonos convexos del perfil fijo que estén definidos en *r*, obteniendo como resultado un conjunto de intervalos válidos.

Como globalmente sólo serán válidos los intervalos de rotación que sean legales para todos los polígonos convexos del perfil móvil definidos en *r*, será necesario intersectar los resultados individuales entre ellos para obtener los intervalos posibles de rotación del perfil móvil, dada una distancia *r* al origen.

La figura 3.3.2.2 ilustra el proceso de obtención de intervalos válidos considerando un perfil fijo representados mediante un único polígono (*negro*), y un perfil móvil representado mediante dos polígonos (*verde*). Como puede observarse, en primer lugar, se estudia por separado los intervalos válidos para cada polígono de forma independiente y, en segundo lugar se intersectan todos los resultados individuales para obtener el resultado global.

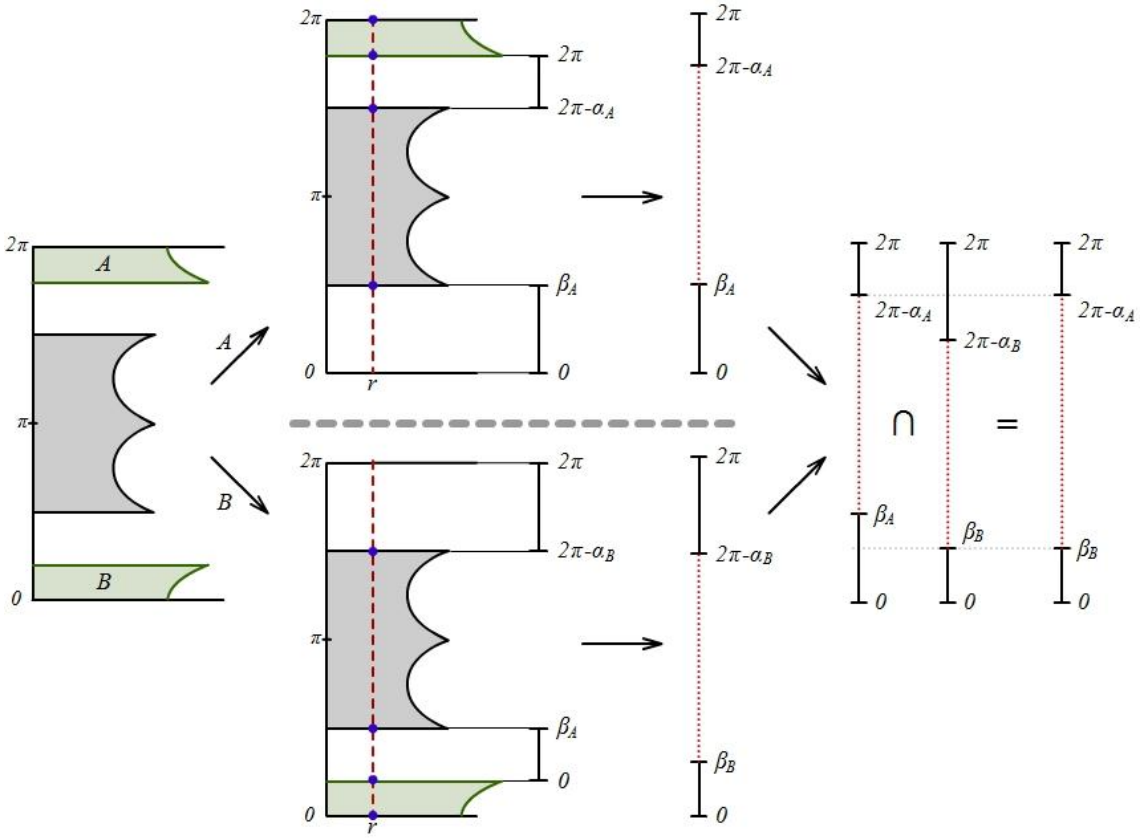

Figura 3.3.2.2: Cálculo de ángulos válidos para dos figuras, a una distancia r del origen.

Los puntos mostrados en *azul* en la figura 3.3.2.2 corresponden con las coordenadas polares que se están considerando para obtener los intervalos locales de rotación. Nótese que estos puntos no tienen por qué estar definidos en su totalidad dentro de los polígonos considerados de ambas figuras. Durante el pre-procesado, se ha garantizado la división individual de cada figura en polígonos convexos, pero no se garantiza que éstos estén alineados respecto a los de la figura con la que se está comprobando la alineación. Así, para llevar a cabo la búsqueda de forma eficiente, resulta conveniente subdividir los polígonos de ambas figuras de manera que se garantice que éstos estén siempre alineados entre ellos.

La forma en la que se obtiene dicha subdivisión es muy similar a la ya comentada cuando se pretendía garantizar que cada punto de la representación polar tuviera siempre sobre/bajo él otros puntos, y no aristas: en primer lugar se recorrerán todos los polígonos de ambas figuras, almacenando en un vector todas las distancias *r* de sus puntos sin duplicados. Hecho esto, y siguiendo el criterio de ordenación propuesto, se recorrerán de izquierda a derecha y de abajo a arriba todos los polígonos de los dos contornos, comprobando si sus segmentos superior e inferior deben ser divididos. En caso de que un polígono deba ser dividido se calcularán, mediante interpolación lineal, los puntos intermedios y se crearán nuevos polígonos cuya unión sea igual al polígono original, pero cuyos segmentos estén alineados con los de la otra figura. Nótese como la convexidad de los polígonos originales garantiza que sus intersecciones con una recta sean tan sólo dos puntos, con lo que el cálculo se simplifica considerablemente.

La figura 3.3.2.3 ilustra el proceso de subdivisión, que comienza con la obtención de las coordenadas *r* de todos los puntos de ambos contornos *a)*, posteriormente selecciona los polígonos que deben ser divididos *b)*, y finalmente obtiene los sub-polígonos que garantizan la alineación de segmentos en ambas representaciones *c)*. Como el avance sobre los polígonos es ordenado de izquierda a derecha, el avance sobre las coordenadas *r* también lo es, por lo que ambos índices se incrementan al mismo tiempo obteniendo un coste lineal *O(n)*.

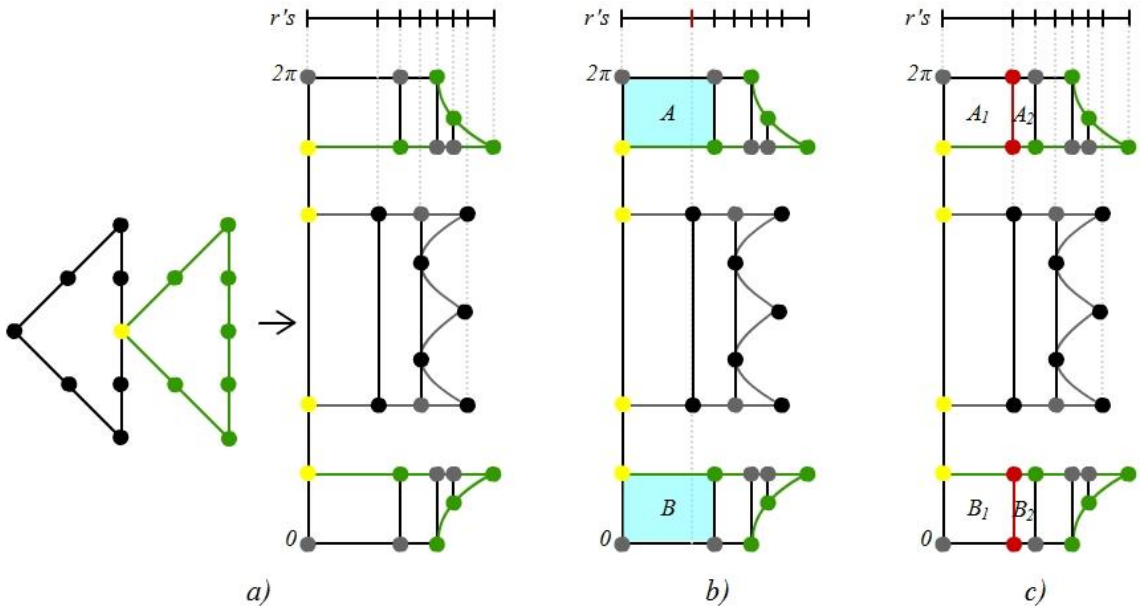

Figura 3.3.2.3: Subdivisión de polígonos para garantizar la alineación de segmentos.

Observando detalladamente la figura 3.3.2.3, se podrá observar que, durante el cálculo de la alineación, no es necesario comprobar todos los polígonos: dado que se busca el contacto entre homólogos, las distancias *r* de éstos deben ser iguales.

Cualquier punto de un contorno cuya distancia *r* sea mayor que la distancia máxima del otro polígono r_{max} no aporta ninguna información, por lo que puede descartarse. Como la subdivisión de polígonos y la comprobación de la alineación se van a realizar en orden de distancia ascendente, se trabajará únicamente sobre el intervalo $[0, r_{MAX}]$, siendo $r_{MAX} = Mín(r1_{max}, r2_{max})$. En términos de complejidad, esta simplificación no aporta ninguna ventaja considerable, pero en la práctica puede reducir el tiempo de cálculo considerablemente si se trabaja con piezas de tamaños dispares. Durante el apartado de representación de formas de este capítulo, se había comentado que la distancia máxima se asociaba a cada configuración. En este punto, dicha afirmación queda justificada.

Volviendo a la figura 3.3.2.2, se puede observar que durante el estudio de ángulos libres se debe solucionar un problema de una única dimensión (grados de rotación). Además, se pone en evidencia que para buscar alineaciones, el segmento inferior de un polígono del perfil fijo debe estar en contacto con el segmento superior de otro polígono del perfil móvil y viceversa.

Teniendo en cuenta estos conceptos, y dados un polígono de la figura móvil $A \in P$ y todos los polígono de la figura fija que estén definidos para *r*, $B_i \in Q$, $p_0, p_1 \in B_i$, $p_0.r = r$, $p_1.r > r$, el problema local de la alineación se solucionará en dos partes.

En primer lugar, se reducirá la dimensionalidad obteniendo los intervalos $I_A = [p_{0\_A}, p_{2\_A}]$ e $I_{Bi} = [p_{0\_Bi}, p_{2\_Bi}]$, tal y como ilustra el apartado *b)* de la figura 3.3.2.4. Los puntos finales de los segmentos (p_3, p_1) se considerarán cuando se estudien los polígonos adyacentes por la derecha.

En segundo lugar, una vez obtenidos los intervalos de rotación unidimensionales, se definirá $h_A = p_{2A} - p_{0A}$ y $h_{Bi} = p_{0\_B(i+1)} - p_{2\_Bi}$. El apartado *c)* de la figura 3.3.2.4 muestra el cálculo de estas alturas. Si $h_A < h_{Bi}$ el polígono móvil *A* no cabrá en el espacio libre que deja el polígono fijo B_i con su sucesor. Obsérvese que, en el ejemplo propuesto, $h_{B2} < h_A$, por lo que no se considerará esta pareja de valores (el segmento del polígono A nunca podrá caber en el hueco descrito por h_B).

Si el segmento estudiado cabe en un hueco, el intervalo de rotaciones válido para la pareja estudiada $[\alpha, \beta]$ se calculará como:

$$\alpha = p_{2_{B_{i+1}}} - p_{0_A}$$
$$\beta = \alpha + h_{B_i} - h_A$$

El apartado *c)* de la figura 3.3.2.4 ilustra el cálculo de α, mientras que el apartado *d)* ilustra el cálculo de β. Nótese que, en ocasiones, α tendrá un valor negativo, y que β siempre será mayor o igual que α dado que $h_A \geq h_{Bi}$ Para normalizar el resultado a un conjunto de ángulos controlado, tanto α como β se incrementarán en 2π radianes si el valor del primero es inferior a cero, quedando garantizado que $\alpha \in [0, 2\pi[$, y que $\beta \in [0, 4\pi[$. La ambigüedad en los valores de β se contemplará durante la intersección de ángulos válidos para todos los polígonos del contorno móvil que están definidos en *r*.

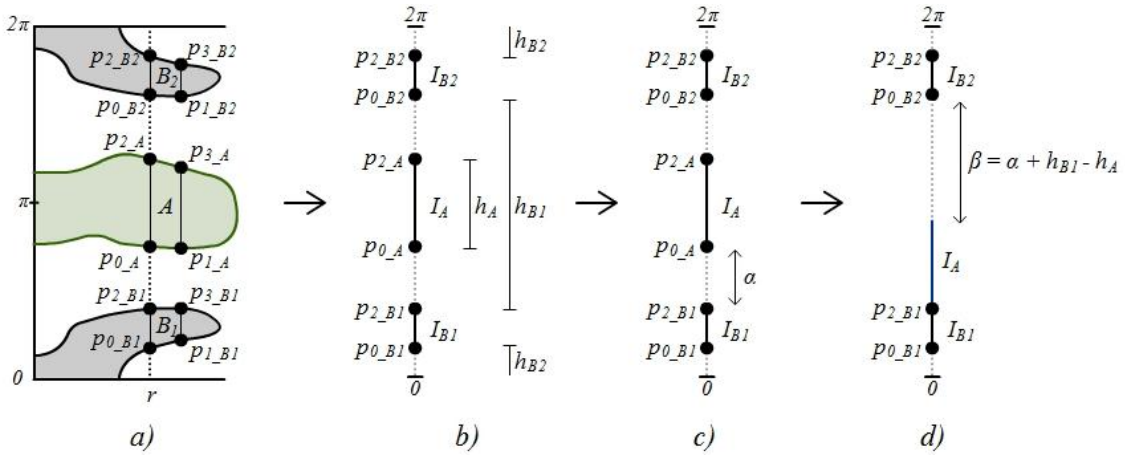

Figura 3.3.2.4: Cálculo de intervalo de rotación válido para un polígono del contorno móvil.

Una vez se ha establecido el criterio con el que obtener los intervalos de rotación válidos para una distancia *r* determinada, queda por comentar el criterio con el que se fusionan todos estos datos para construir el resultados final.

Asúmase que, inicialmente, el intervalo de rotación válido (al que se denominará *espacio libre*) se inicializa a *[0, 2π]*, es decir, se permiten todas las rotaciones. A medida que la búsqueda vaya evolucionando a lo largo del eje *r*, el espacio libre se intersectará con las restricciones que imponga cada conjunto de polígonos definidos en el *r* que se está estudiando. De esta manera, el espacio libre se irá reduciendo hasta que se dé uno de los dos casos de terminación: si el espacio libre es el conjunto vacío, no hay ninguna alineación posible, por lo que se detiene la búsqueda. Si por el contrario, la coordenada *r* estudiada es mayor que r_{MAX} , que se había definido como $Mín(r1_{max}\ ,\ r2_{max})$, tampoco será necesario contemplar más polígonos. El espacio libre que quede será el resultado de la búsqueda, y los extremos de los intervalos definidos serán los ángulos que alinean los contornos.

Para finalizar con el proceso de búsqueda, resulta interesante analizar el coste en términos de complejidad. Dada una alineación, será necesario subdividir los intervalos de ambos contornos para garantizar que todos los polígonos de ambos estén alineados entre ellos. Para esto, únicamente es necesario almacenar, en una primera pasada, las coordenadas *r* de todos los puntos *O(n)*, y en una segunda pasada, gracias a la ordenación de los polígonos durante el pre-procesado y a las propiedades de la convexidad, sólo será necesario recorrer de nuevo todos los polígonos y aplicar operaciones de coste constante *O(k)*, por lo que se obtendrá un coste de *O(n)*. Una vez hecho esto, será necesario desplazarse sobre el eje *r*, intersectando el espacio libre con las restricciones que aporte cada conjunto de intervalos, cuyo cálculo es de coste constante *O(k)*, lo que proporcionará un peor caso de *O(n)*, si se han de recorrer todos los puntos porque $r1_{max} = r2_{max}$, y un mejor caso en el que se detecta la incompatibilidad en las primeras iteraciones, por lo que el coste es despreciable. Normalmente, el coste será ligeramente inferior a *O(n)*, porque las distancias máximas no coincidirán, pero en términos de complejidad debe considerarse que este último paso también tendrá un coste *O(n)*. Así pues, se concluye que la complejidad total del proceso de búsqueda para una determinada configuración es lineal al número de puntos, por lo que se trata de un algoritmo de coste *O(n)*.

3.4. Cálculo de las correspondencias

Una vez calculadas las posibles alineaciones para una configuración dada, es necesario definir una función *Match* que, a partir de una alineación (caracterizada por los puntos de inicio de cada contorno i_p, i_q y por el ángulo de rotación del contorno móvil α) devuelva un valor numérico que exprese la similitud entre los contornos para la alineación estudiada.

$$Match(i_p, i_q, \propto) \rightarrow \mathbb{R}$$

Dicha función, en calidad de relación de orden, debe cumplir las propiedades introducidas en el apartado 2.1.3 de identidad, unicidad y desigualdad triangular para ser considerada como métrica, y únicamente la desigualdad triangular para ser considerada como semi-métrica.

A lo largo del capítulo 2 se han introducido muchas métricas comunes, y al comienzo de este capítulo se han sugerido las dos más habituales: LCP y la distancia geométrica. En términos generales, cualquier función que cumpla los requisitos comentados proporcionará resultados válidos, ya que el proceso de alineación devuelve resultados exactos, y el cálculo de distancias no supone un problema. Las características de los contornos que se estén estudiando, y el grado de tolerancia que permita la definición de correspondencia harán que unas técnicas resulten más favorables que otras. Tómese como ejemplo el caso ilustrado en la figura 3.4.1. Ante los dos contornos presentados en *a)*, las soluciones mostradas en *b)* y en *c)* podrían considerarse como válidas. El caso *b)* debería ser mejor que *c)* si se buscase una alineación perfecta, mientras que *c)* sería mejor que *b)* si se estudiase la correspondencia con una tolerancia mayor.

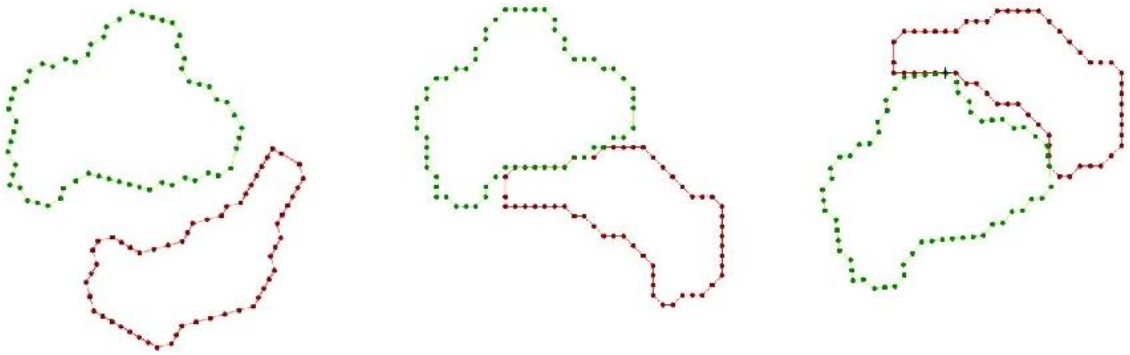

Figura 3.4.1: Posibles correspondencias en base al criterio establecido.

Por comentar las dos técnicas más populares, mediante LCP sería necesario que el usuario indicase una tolerancia máxima, y la función *Match* debería comprobar todos los puntos para poder establecer cuántos emparejamientos satisfacen la condición de distancia. Típicamente, el valor de la función *Match* se correspondería con esta cardinalidad, cumpliendo así las propiedades de orden necesarias. Es necesario precisar que, dado que se han creado puntos

intermedios para permitir la subdivisión en intervalos convexos, LCP únicamente debería recorrer los puntos originales de las figuras, descartando los interpolados.

Si se desea añadir potencia a LCP, sin incrementar el coste, se podría hacer uso de todos los puntos obtenidos durante los pasos anteriores y de las ventajas que aporta la representación polar. Así, definición alternativa de distancia consistiría en estudiar ésta en base a los puntos cuya coordenada *r* polar es idéntica. La distancia entre un punto $p \in P$ y el contorno opuesto Q podría expresarse como:

$$d = 2r * \sin\left(\frac{(p_\alpha + \alpha) - q_\alpha}{2}\right)$$

donde $q \in Q$ es el punto más próximo a p que cumple que $q_r = p_r$. Nótese que encontrar este punto es extremadamente sencillo sobre la representación polar, ya que está sobre la misma vertical, y es aquel cuya distancia es menor. La figura 3.4.2 ilustra esta alternativa para LCP, en la que es posible medir distancias con segmentos de manera rápida.

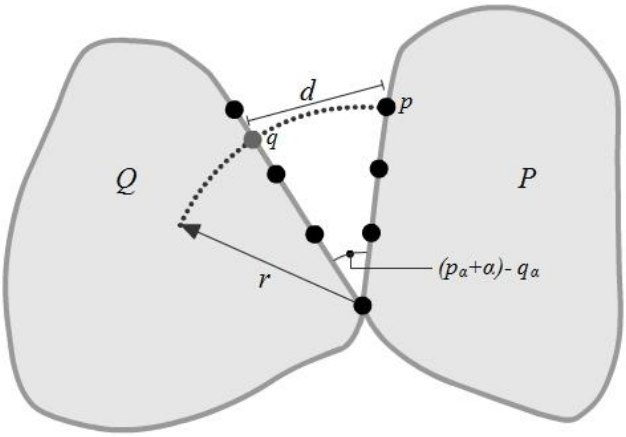

Figura 3.4.2: Posibles correspondencias en base al criterio establecido.

En cuanto al método de las distancias geométricas, como ya se ha introducido al comienzo del capítulo, se presenta un problema típico de optimización multi-objetivo. Así, se trata de maximizar la longitud del contorno en contacto, a la par que se minimiza el error cometido. Como alternativa simplificada, se utilizar una función híbrida en la que, cada punto original del contorno incrementa el valor del match en base a su distancia. A menor distancia, mayor puntuación. Todos los puntos que queden fuera de una tolerancia fijada por el usuario serán descartados, quedando la función *Match* expresada como:

$$Match(P, Q) = \sum_{p \in P} Puntuación(p, Q),$$

donde

$$Puntuación(p, Q) = \begin{cases} \varepsilon - \|p - q\| \ \rightarrow \exists q \in Q, \|p - q\| < \varepsilon \\ 0 \rightarrow en\ cualquier\ otro\ caso \end{cases}$$

Como alternativa más potente, se plantea la posibilidad de estudiar el área de espacio vacío que queda entre las figuras en relación a la longitud de contorno que se considera lo bastante próxima como para ser una correspondencia. Así, la definición del área asociada a un segmento sería una extensión de la ampliación de la técnica LCP. La figura 3.4.3 ilustra este concepto.

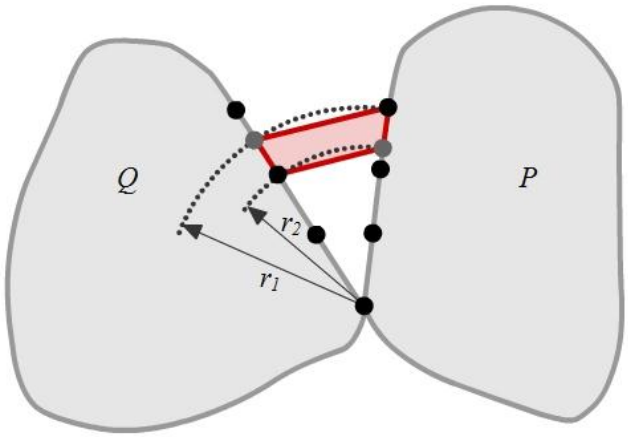

Figura 3.4.3: Obtención de áreas vinculadas a segmentos.

La ventaja que aporta, en este caso, la representación polar y los puntos intermedios obtenidos consiste en que, sin coste alguno, permite vincular un área a cada segmento de manera que se garantice que la unión de todas las áreas sea igual al espacio vacío entre los contornos, y que el área definida puede calcularse muy fácilmente mediante la descomposición del trapezoide en dos triángulos, definidos por vértices cuyas posiciones son conocidas de antemano.

Para finalizar, un análisis de complejidad de cualquiera de las soluciones propuestas ofrece un coste lineal a la talla del problema *O(n)*, ya que durante la fase de alineación se han creado todos los puntos intermedios necesarios para que los cálculos en esta parte sean la aplicación inmediata de las métricas deseadas. Además, el orden de los polígonos convexos permite que no sea necesario llevar a cabo ninguna búsqueda para conocer el punto homólogo en la otra figura del punto estudiado.

3.5. Análisis de complejidad

A modo de resumen de las propiedades de la técnica desarrollada, se aúnan en esta sección todos los análisis de complejidad llevados a cabo a lo largo del capítulo:

- $Coste_{total} = Coste_{pre-procesado} + Coste_{búsqueda}$
  - $Coste_{pre-procesado} = nRepresentaciones * Coste_{configuración}$
    - $nRepresentaciones = n$
    - $Coste_{configuración} = Coste_{puntos\_polares} * Coste_{ordenación} + Coste_{polígonos}$
      - $Coste_{puntos\_polares} = n$
      - $Coste_{ordenación} = \log(n)$
      - $Coste_{polígonos} = n$
    - $Coste_{configuración} = n * \log(n) + n \approx n \log(n)$
  - $Coste_{pre-procesado} = n * n \log(n) = n^2 \log(n)$
  - $Coste_{búsqueda} = nConfiguraciones * (Coste_{alineación} + Coste_{correspondencias})$
    - $nConfiguraciones = n * m \approx n^2$
    - $Coste_{alineación} = Coste_{subdivision} + Coste_{búsqueda}$
      - $Coste_{subdivision} = n$
      - $Coste_{búsqueda} = n$
    - $Coste_{alineación} = n + n \approx n$
    - $Coste_{correspondencias} = n$
  - $Coste_{búsqueda} = n^2 * (n + n) \approx n^2 * n \approx n^3$
- $Coste_{total} = n^2 \log(n) + n^3 \approx n^3$

Capítulo 4

Implementación

4.1. Introducción

Una vez introducida la técnica desarrollada, es momento de analizar los aspectos concretos que subyacen de su implementación en un lenguaje de programación. A lo largo de este capítulo se comentarán con mayor grado de detalle las operaciones *automáticas* necesarias para trasladar las ideas ya expuestas a una aplicación real.

La implementación que aquí se va a comentar no persigue la eficiencia en términos de tiempo de computación, sino que pretende ser lo más próxima posible a los conceptos ilustrados manteniendo, eso sí, la complejidad algorítmica analizada durante el capítulo anterior. En este sentido, los lenguajes de programación de bajo nivel, cuyo rendimiento es superior a otros más próximos al lenguaje natural, no aportan grandes ventajas ya que la magnitud de las abstracciones que éstos implican "ensombrecen" la naturaleza del proceso que se está llevando a cabo. Además, nótese que el proceso de implementación ha sido paralelo al desarrollo de la técnica, por lo que la capacidad de adaptación del lenguaje de programación ha facilitado considerablemente la tarea. Cuanto más genérica sea la definición de los datos y operaciones involucrados en el desarrollo más fácil será el replanteamiento de la técnica, no siendo necesario detenerse en aspectos específicos de la programación. Por contra, el carácter genérico tiene implícito un menor rendimiento durante la ejecución.

Tal y como se ha planteado el desarrollo de la técnica, el paradigma de programación que mayores ventajas aporta es la programación imperativa, por su potencia y grado de aceptación dentro de la comunidad informática. Siendo más específicos, el paradigma de programación orientado a objetos aporta ventajas considerables respecto a la programación funcional, ya que su capacidad de abstracción, encapsulamiento y, especialmente, las posibilidades que ofrece el polimorfismo dotan a la implementación del carácter genérico que se persigue. Además, debe tenerse en cuenta que uno de los principales objetivos de la implementación ha sido proporcionar un *feedback* visual que permita comprender con mayor grado de detalle los aspectos concretos del problema, para poder abordar la resolución de éste de forma genérica.

En base a todo lo comentado, el lenguaje de programación que se elija deberá ser imperativo, orientado a objetos, y deberá estar integrado en un *IDE* (acrónimo de la expresión inglesa *Integrated Develpoment Environment*) que permita el desarrollo rápido de interfaces de usuario.

4.2. Lenguaje de programación: C#

En base a los criterios de selección del lenguaje de programación comentados durante la introducción, la disyuntiva está entre *Java*, o *C++* o *C#*, dado que son los más comunes que cumplen todos los requisitos.

Se descartará C++ en primer lugar dado que, pese a ser considerado como un lenguaje de programación orientado a objetos, su caracterización queda a medio camino entre la programación funcional y ésta. De ahí que no se le considere *puro*. Muchas de las operaciones que se llevan a cabo son a un nivel de abstracción demasiado bajo (distante del lenguaje natural), por lo que se ensombrecería mucho la naturaleza del proceso que se esté llevando a cabo.

Entre *Java* y *C#*, cualquier elección sería buena (ambos son orientados a objetos *puros*, y disponen de herramientas de desarrollo que permiten la definición de interfaces de usuario de manera cómoda). No obstante, se ha decidido emplear *C#* por dos motivos principales: *Java*, pese a tener la ventaja de ser interpretado y, por tanto, multiplataforma, tiene el principal inconveniente de que presenta una eficiencia bastante peor. Además, *C#* está integrado en el IDE de Microsoft, *Visual Studio*, que proporciona opciones de desarrollo muy potentes en términos de diseño de interfaces, depuración, e incluso ayudas durante la codificación del programa.

C# (pronunciado C-sharp en inglés) es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET, que después fue aprobado como un estándar por la ECMA e ISO. Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos de la plataforma.NET, similar al de Java aunque incluye mejoras derivadas de otros lenguajes (entre ellos Delphi).

La creación del nombre del lenguaje, C#, proviene de dibujar dos signos positivos encima de los dos signos positivos de "C++", queriendo dar una imagen de salto evolutivo, del mismo modo que ocurrió con el paso de C a C++. Así, podría decirse que C# es C++++

C#, como parte de la plataforma.NET, está normalizado por ECMA desde diciembre de 2001 (C# Language Specification "Especificación del lenguaje C#"). El 7 de noviembre de 2005 salió la versión 2.0 del lenguaje, que incluía mejoras tales como tipos genéricos, métodos anónimos, iteradores, tipos parciales y tipos anulables. El 19 de noviembre de 2007 salió la versión 3.0 de C#, destacando entre las mejoras los tipos implícitos, tipos anónimos y LINQ (Language Integrated Query -consulta integrada en el lenguaje).

Aunque C# forma parte de la plataforma.NET, ésta es una interfaz de programación de aplicaciones (API), mientras que C# es un lenguaje de programación independiente diseñado para generar programas sobre dicha plataforma. Ya existe un compilador implementado que provee el marco de DotGNU - Mono que genera programas para distintas plataformas como Win32, UNIX y Linux.

Las principales características de C# que se van a explotar durante la implementación son las clases, el concepto de herencia y, estrechamente vinculadas con las posibilidades que ofrece el polimorfismo, las listas definidas en el lenguaje.

El uso de clases para caracterizar los objetos que se estudien permitirá encapsular la definición de éstos, y proporcionar un marco de programación amigable, a la par que comprensible para el lector. Así, por ejemplo, una clase *Perfil* empleada durante la implementación permitirá el acceso ordenado a la representación cartesiana de éste, a sus múltiples representaciones polares, y ofrecerá una serie de métodos asociados que permitirán realizar transformaciones sobre él garantizando su integridad en términos de estructura.

La clase *List<T>* definida de forma nativa por el lenguaje es, sin duda, la estructura de datos más flexible de las ofertadas. Al estar definida como una *plantilla*, se vale de la potencia del polimorfismo para permitir construir listas de cualquier tipo de datos (especificado en *<T>*). Así, dado que un objeto derivado de la clase *Perfil* debe tener un conjunto de representaciones polares, almacenadas en objetos derivados de la clase *Contorno*, dentro de la especificación de *Perfil* se puede definir un objeto público *polar* que derive de la clase *List<Contorno>*. De esta manera, *polar[i]* será una referencia al i-ésimo objeto *Contorno*.

Las principales ventajas de esta clase son que su tamaño crece durante la ejecución de forma dinámica sin que sea necesario llevar a cabo reservas de memoria manuales, y que proporciona un método *Sort()* que permite ordenar los valores contenidos en ella con un coste *O(n log n)*, haciendo uso de una implementación interna del conocido algoritmo *Quicksort*.

Para poder llevar a cabo ordenaciones sobre cualquier tipo de datos, la clase *List<T>* exige que la definición de la clase *T* herede de la clase *IComparable*, y que en ella se sobre-escriba el método *public int CompareTo(object o)* para establecer explícitamente el criterio de ordenación de los elementos. Así, a modo de ejemplo, en el código 4.2.1 se muestra una implementación parcial de la clase *Punto*, que será empleada para almacenar las coordenadas polares del contorno. Recuérdese que el objetivo es que la clave primaria sea la distancia al origen *r*, mientras que la clave secundaria es el ángulo α.

```
public class Punto : IComparable
{
   public double x;
   public double y;

   … (Continua) …

   public int CompareTo(object obj)
   {
      Punto p1 = (Punto)obj;
      if (this.x != p1.x)  return this.x.CompareTo(p1.x);   // Clave primaria
      else return this.y.CompareTo(p1.y);                   // Clave secundaria
   }
}
```

Código 4.2.1: Especialización del método CompareTo para permitir ordenaciones en la lista

4.3. Extracción de características

Antes de comentar los aspectos relacionados con la implementación de la búsqueda de correspondencias, es necesario detenerse brevemente en el proceso de extracción de características. Éste es independiente a la búsqueda, pero debe garantizar una serie de propiedades expuestas en el apartado 3.1.1. Así, se asume que se dispone de un conjunto de datos de partida formado por imágenes bidimensionales ya segmentadas (intensidades binarias), en las que el *fondo* está representado en color blanco, mientras que el *objeto* está representado en color negro. El objetivo es obtener un conjunto de puntos enlazados en base a su posición durante un recorrido del perímetro de la figura en sentido horario.

Para llevar esto a cabo, tal y como se ha comentado, el primer paso consiste en decidir qué punto será el primero. Así, se ha decidido que el primer punto de una forma será aquel que tenga la menor coordenada *x* y, en caso de que haya varios, será el que tenga la menor coordenada *y*. Para localizar el punto que satisface esta condición, basta con recorrer la imagen de izquierda a derecha, y de arriba abajo, hasta encontrar el primer punto que pertenezca al objeto. El código 4.3.1 muestra cómo se lleva a cabo esta búsqueda, utilizando como dato de entrada una matriz bidimensional *contorno*, cuyos valores son de tipo lógico (cero para los puntos del fondo, uno para los puntos del objeto)

```
int xActual = -1;
int yActual = -1;
for (int i = 0; i < imagenOriginal.Width; i++)
{
   for (int j = 0; j < imagenOriginal.Height; j++)
   {
      if (contorno[i][j])
      {
         xActual = i;
         yActual = j;
         break;
      }
   }
   if (xActual != -1) break;
}
```

Código 4.3.1: Búsqueda del primer punto del contorno.

El proceso de búsqueda se realiza mediante un doble bucle, en el que el índice exterior *i* se inicia a *0* y se incrementa hasta alcanzar la anchura en pixels de la imagen estudiada (*imagenOrignal.Width* proporciona este valor). Para cada valor de *i*, se estudian todos los pixels en la vertical, indexados por el índice *j* en orden ascendente. Cuando se encuentra el primer punto que pertenece al objeto (*contorno[i][j] == true*), se almacenan sus coordenadas y se finaliza la búsqueda.

Una vez obtenido el punto inicial, será necesario ir recorriendo sus 8-vecinos que también formen parte de la frontera en sentido horario, e ir almacenando sus coordenadas en un vector. Cuando se alcance de nuevo el punto inicial finalizará el proceso de búsqueda, constituyendo el vector obtenido el resultado de la extracción de características, en el que cada punto es contiguo

a su predecesor y a su antecesor con un sistema de índices cíclico (el sucesor del último punto es el primero, y el antecesor del primer punto es el último).

Para llevar a cabo este recorrido, será necesario iterar tantas veces como puntos de frontera haya (asúmase conocido el dato), y calcular el siguiente punto en base a la posición que ocupaba el anterior. Existen en la literatura muchos estudios de las tablas de desplazamiento a seguir durante la extracción de fronteras con el criterio de 8-vecindad, por lo que no se va a comentar con demasiado detalle esta parte. Las soluciones existentes están demostradas y no suponen una complejidad adicional al problema.

Así, el código 4.3.2 muestra la implementación del bucle que recorre toda la frontera construyendo el vector resultado, *perfil*, mientras que el código 4.3.3 muestra la implementación del método que permite establecer el siguiente punto en base a unas coordenadas *x*, *y*, y a la dirección de desplazamiento anterior *dir*. Nótese que las direcciones han sido codificadas según el criterio estándar de orientaciones geográficas *(N* para norte, *S* para sur…).

```
int dirActual = -1;
perfil = new int[nPixels][];
for (int i = 0; i < nPixels; i++) perfil[i] = new int[4];
for (int i = 0; i < nPixels; i++)
{
   perfil[i][0] = xActual;
   perfil[i][1] = yActual;
   contorno[xActual][yActual] = false;
   int dX, dY;
   dirActual = encuentraSiguiente(dirActual, xActual, yActual,
                   imagenOriginal.Width, imagenOriginal.Height, out dX, out dY);
   xActual += dX;
   yActual += dY;
}
```

Código 4.3.2: Recorrido de la frontera y construcción del vector resultado.

Obsérvese en el código 4.3.2 como cada punto que ha sido visitado se marca como *fondo* (*contorno[xActual][yActual] = false*) , para evitar que se produzcan "vueltas hacia atrás" y el proceso quede atrapado en un bucle infinito.

```
private int encuentraSiguiente(int dir, int x, int y, int width, int height,
                                                          out int dX, out int dY) {
   int res = -1;
   dX = 0; dY = 0;
   if (contorno[x - 1][y] && dir != E) { res = W; dX = -1; dY = 0; }
   else if (contorno[x - 1][y - 1] && dir != SE) { res = NW; dX = -1; dY = -1; }
   else if (contorno[x][y - 1] && dir != S) { res = N; dX = 0; dY = -1; }
   else if (contorno[x + 1][y - 1] && dir != SW) { res = NE; dX = 1; dY = -1; }
   else if (contorno[x + 1][y] && dir != W) { res = E; dX = 1; dY = 0; }
   else if (contorno[x + 1][y + 1] && dir != NW) { res = SE; dX = 1; dY = 1; }
   else if (contorno[x][y + 1] && dir != N) { res = S; dX = 0; dY = 1; }
   else if (contorno[x - 1][y + 1] && dir != NE) { res = SW; dX = -1; dY = 1; }
   return res;
}
```

Código 4.3.3: Búsqueda del siguiente punto en sentido de las agujas del reloj.

Una vez obtenido el vector resultado, para mantener la independencia entre la extracción de características y el algoritmo de búsqueda, se almacenarán todos los puntos obtenidos en un fichero *.XML* (acrónimo de la expresión inglesa *eXtended Markup Language*), que será el que tome como dato de partida el programa de búsqueda. La estructura con la que se almacenarán los datos queda reflejada en el código 4.3.4

```
<?xml version="1.0" encoding='UTF-8'?>
<perfil>
<punto x="11" y="18"/>
<punto x="12" y="17"/>
<punto x="13" y="16"/>
<punto x="14" y="15"/>

… (Continúa) …

<punto x="11" y="21"/>
<punto x="11" y="20"/>
<punto x="11" y="19"/>
</perfil>
```

Código 4.3.4: Formato del archivo .XML que almacenará los contornos.

Las ventajas que ofrece XML derivan de su calidad de estándar para el intercambio de datos entre programas, ya que está definido de manera formal mediante *DTDs* (acrónimo de la expresión inglesa *Document Type Definition*) y que es soportado nativamente por C#, disponiéndose de *parsers* específicos para facilitar la lectura y escritura de este tipo de ficheros.

La figura 4.3.1 ilustra el aspecto final del extractor de características desarrollado.

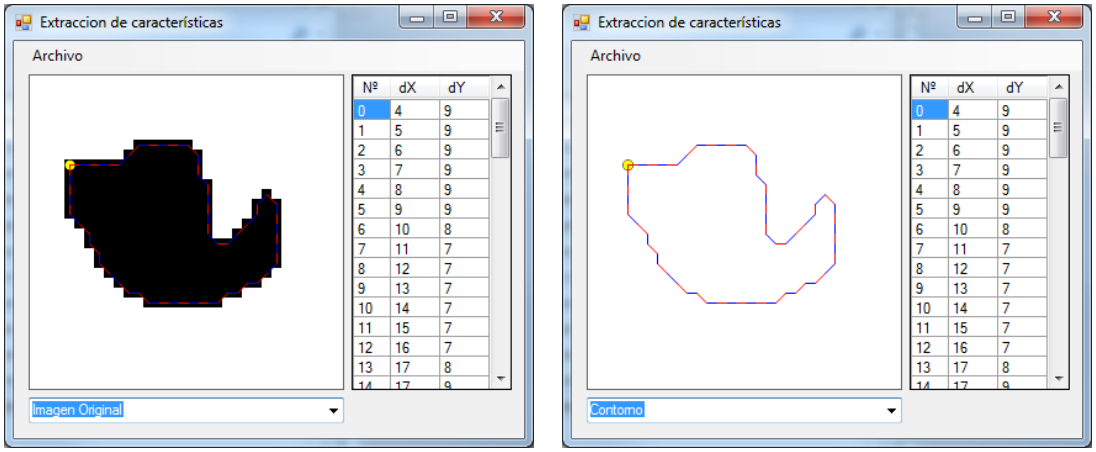

Figura 4.3.1: Aspecto final del extractor de características desarrollado.

4.4. Búsqueda

Una vez comentada la extracción de características a partir de imágenes bidimensionales segmentadas, se estudiará con mayor detalle los aspectos de la implementación de la técnica desarrollada. Se asumirá que se dispone de una serie de ficheros .XML que describan las formas a procesar y, a partir de ellas, se realizarán los cálculos que permitan obtener las posibles alineaciones entre ellas, ordenadas en base a la métrica seleccionada.

El proceso de búsqueda se estudiará en cuatro etapas: en primer lugar se introducirá el diagrama de clases completo de la aplicación desarrollada, que servirá de referencia a lo largo del resto de explicaciones. Una vez mostradas las clases que componen la aplicación, se estudiará en detalle el proceso de pre-procesado de los contornos. Posteriormente, se hablará de la implementación del algoritmo de búsqueda de alineación y, finalmente, se comentará el código responsable del cálculo de las correspondencias.

4.4.1. Diagrama de clases

En este apartado se pretenden introducir todos los actores que formarán parte de la aplicación final. La figura 4.4.1.1 muestra el diagrama UML de clases completo de la aplicación desarrollada. En ella, se puede apreciar una clase *Program* con perímetro discontinuo en la esquina superior izquierda. Esta clase es la que se invoca en el inicio de la ejecución del programa, y es la responsable de instanciar un objeto de la clase *Principal* que se encargará de gestionar la interfaz de usuario. Este objeto, mantendrá el control del dibujado de los elementos visuales, y capturará los eventos generados por interacciones con el usuario, para permitir que éste pueda seleccionar diversas opciones (qué figuras abrir, qué configuración mostrar…). El atributo más importante de *Principal* es el objeto *comparador*, derivado de la clase *Comparador*, que es el encargado de gestionar todas las tareas comentadas durante el capítulo anterior.

Como puede observarse, la clase *Comparador* cuenta con dos atributos de tipo *Perfil*, (*p1* y *p2*) que son las representaciones de los contornos que se están estudiando. *p1* corresponde con el contorno que se ha definido como fijo, mientras que *p2* corresponde con el contorno definido como móvil.

Cada *Perfil* está caracterizado por un atributo llamado *cartesiano*, que es un objeto de tipo *List<Punto>* que almacena la representación cartesiana del contorno, ordenada de izquierda a derecha y de abajo a arriba. Además, se dispone de un atributo *polar* que es un objeto de tipo *List<Contorno>* en el que cada elemento corresponde con una representación polar (en función del punto de inicio considerado). Finalmente, el atributo llamado *polígonos* es un objeto de tipo *list<Poligono>[]* (vector de listas de polígonos) que almacena todos los polígonos convexos definidos para cada configuración del contorno.

La clase *Contorno* cuenta con una serie de atributos que ilustran sus principales características (*anguloMax, anguloMin…)* y con un atributo *puntos* que es un objeto de tipo *List<Punto>* que almacena todos los puntos que forman la representación polar, ordenados en base a los criterios ya comentados.

La clase *Punto* cuenta con dos parámetros *x* e *y* que indican las coordenadas de éste, con dos referencias *anterior* y *siguiente* que permiten mantener el orden original de los puntos y, además, nótese como la clase deriva de *IComparable*, y dispone de un método *CompareTo* que permite establecer el criterio con el que los puntos serán ordenados dentro de la lista *polar*.

La clase *Polígono*, está formada por un vector, *puntos*, de cuatro referencias a objetos de tipo *Punto*, que servirán para almacenar los extremos de los segmentos que forman cada polígono convexo. Al igual que *Punto*, polígono deriva de *IComparable*, y dispone de un método *CompareTo*.

Finalmente, la clase *Match* será utilizada por *Comparador* para almacenar los valores de similitud calculados ante todas las alineaciones posibles. Sus atributos serán aquellos que definen una configuración (*inicio1, inicio2, angulo2*) y un cuarto valor *puntuacion* sobre el que se almacenará la similitud entre las formas. En este caso, *CompareTo* se valdrá de este valor para ordenar las correspondencias en base a su puntuación.

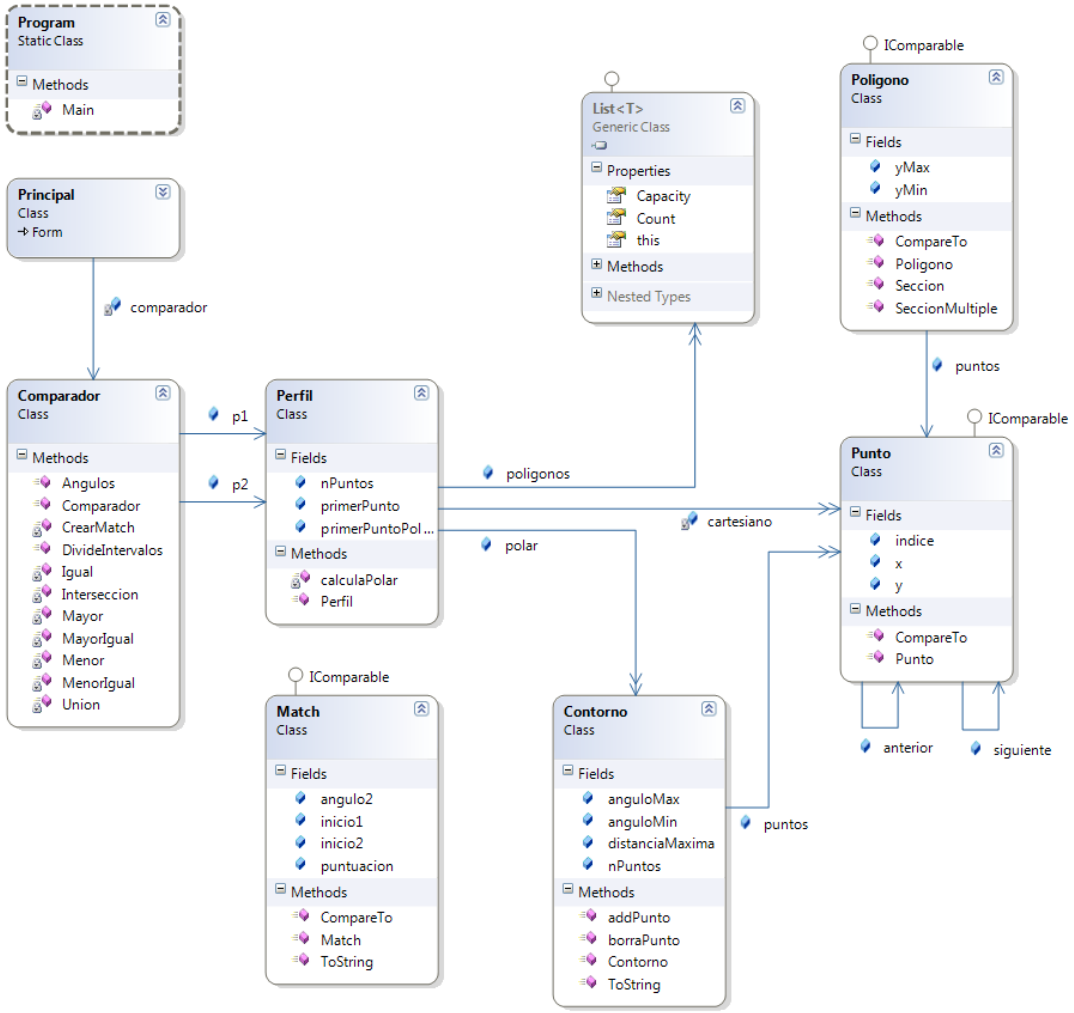

Figura 4.4.1.1: Diagrama de clases de la aplicación desarrollada.

4.4.2. Pre-procesado

Toda la fase de pre-procesamiento se lleva a cabo en la clase *Perfil*. Típicamente, cuando el usuario selecciona un archivo .XML correspondiente a un contorno, *Comparador* crea un perfil nuevo al que pasa como único parámetro la ruta del archivo. El constructor de *Perfil* se encarga de cargar los datos almacenados en el fichero, y de pre-procesar todo el contorno, antes de devolver al usuario el control de la aplicación.

La lectura del .XML carece de interés en el presente documento, por lo que la fase de pre-procesamiento comienza con una lista de puntos, *cartesiano*, en la que cada elemento es un punto del contorno y éstos están almacenados en el orden del sentido de las agujas del reloj. Dado que el criterio de ordenación de los puntos va a cambiar, y debe mantenerse registrado el orden original, el primer paso a llevar a cabo es actualizar las referencias *siguiente* y *anterior* de todos los puntos. El código 4.4.2.1 muestra este proceso, en el que se aprovecha el orden inicial de los puntos para actualizar las referencias con coste *O(n)*.

```
for (int i = 0; i < cartesiano.Count; i++)
{
   cartesiano[i].indice = i;
   cartesiano[i].siguiente = cartesiano[(i + 1) % cartesiano.Count];
   if(i > 0) cartesiano[i].anterior = cartesiano[i - 1];
   else cartesiano[i].anterior = cartesiano[cartesiano.Count - 1];
}
nPuntos = cartesiano.Count;
primerPunto = cartesiano[0];
```

Código 4.4.2.1: Actualización de referencias de los puntos

Obsérvese como la asignación es circular, el primer y último punto están debidamente referenciados, y como se almacena una primera referencia al punto inicial. Si bien es cierto que tras el re-ordenado del vector, éste estará en las primeras posiciones, no se puede garantizar que ocupe la posición 0, por lo que si se quiere reconstruir el orden original recursivamente será necesario conocer la dirección de memoria en que está almacenado el primer punto.

Una vez hecho esto, será necesario reservar espacio para almacenar las coordenadas polares en una lista de objetos derivados de la clase *Contorno*, así como reservar espacio para almacenar los índices a los primeros puntos, y para los polígonos. El número de representaciones totales será igual al número de puntos que forman el vector *cartesiano*. Iterativamente, habrá que construir la representación polar de cada posible configuración. El código 4.4.2.2 muestra cómo se lleva a cabo dicha reserva, y cómo se inicia el cálculo de cada posible representación.

```
polar = new List<Contorno>(nPuntos);
primerPuntoPolar = new List<Punto>(nPuntos);
poligonos = new List<Poligono>[nPuntos];
for (int i = 0; i < nPuntos; i++) calculaPolar(i);
```

Código 4.4.2.2: Reserva de memoria e inicio del cálculo de la representación polar.

El método privado *calculaPolar(int inicio)*, se encargará de llevar a cabo el pre-procesado para cada configuración (caracterizada por el valor *inicio*), y de almacenar los resultados en las listas previamente creadas en la posición *Lista[inicio]*, donde *Lista* puede ser *polar*, *polígonos* o

primerPuntoPolar. Así, el primer paso será trasladar las coordenadas cartesianas disponibles a sus homólogas polares, garantizando que todas las coordenadas estén comprendidas en el intervalo *[0..2π[*. Nótese que, en este paso, podrá haber discontinuidades en la representación polar, tal y como se comentó en el punto 3.2.2.

```
polar.Add(new Contorno());
polar[inicio].addPunto(new Punto(0, Math.PI * 2));
Punto actual = cartesiano[inicio];
for (int i = 1; i < cartesiano.Count; i++)
{
   Punto siguiente = cartesiano[(inicio + i) % cartesiano.Count];
   double x = Math.Sqrt(Math.Pow(siguiente.x - actual.x, 2) +
                                                Math.Pow(siguiente.y - actual.y, 2));
   double y = -Math.Atan2(siguiente.y - actual.y, siguiente.x - actual.x);
   while (y < Math.PI) y += Math.PI * 2;
   polar[inicio].addPunto(new Punto(x, y));
}
polar[inicio].puntos[0].y = polar[inicio].puntos[1].y;
polar[inicio].addPunto(
                  new Punto(0, polar[inicio].puntos[polar[inicio].nPuntos - 1].y));
```

Código 4.4.2.3: Cálculo de las coordenadas polares para los puntos cartesianos.

Recordando el apartado 3.2.2, es preciso recalcar que el primer punto tiene dos representaciones, cuyos valores no se pueden conocer hasta el final de todo el proceso. Así, el ángulo que éste forma con su sucesor/antecesor está calculado en las últimas dos líneas del código 4.4.2.3, y se obtiene consultando los valores obtenidos con sus contiguos.

Dado que los puntos que se han insertado han sido creados de nuevo (*new Punto()*), las referencias a sus vecinos deben ser calculadas de nuevo si se desea reconstruir el orden original a partir de las coordenadas polares, del mismo modo que se debe almacenar la posición del primer punto. El código 4.4.2.4 ilustra este proceso.

```
primerPuntoPolar.Add(polar[inicio].puntos[0]);
for (int i = 0; i < polar[inicio].nPuntos; i++)
{
   polar[inicio].puntos[i].siguiente = polar[inicio].puntos[(i + 1) %
                                                            polar[inicio].nPuntos];
   if (i > 0) polar[inicio].puntos[i].anterior = polar[inicio].puntos[i - 1];
   else polar[inicio].puntos[i].anterior =
                               polar[inicio].puntos[polar[inicio].puntos.Count - 1];
}
```

Código 4.4.2.4: Actualización de las referencias a los vecinos en los puntos polares.

Para simplificar la resolución de la búsqueda resulta conveniente eliminar los puntos que no aportan ninguna información útil. Para poder establecer cuáles son estos puntos, es preciso considerar a sus vecinos. Si el predecesor y antecesor están a la misma distancia r o ángulo α respecto al origen local de coordenadas, la información que aporta el punto actual no aporta ningún significado a la representación, por lo que éste puede ser eliminado. Nótese que, para llevar a cabo este cálculo iterativamente, antes de realizar un borrado será preciso consultar el resto de puntos, por lo que se opta por crear una lista temporal de los puntos a borrar

List<Punto> borrar, y almacenar en ella las referencias de los puntos seleccionados. El proceso de borrado se llevará a cabo en una segunda iteración, tal y como muestra el código 4.4.2.5.

```
List<Punto> borrar = new List<Punto>();
for (int i = 0; i < polar[inicio].nPuntos; i++)
{
   actual = polar[inicio].puntos[i];
   if (actual.x == actual.anterior.x && actual.x == actual.siguiente.x)
      borrar.Add(actual);
   if (actual.y == actual.anterior.y && actual.y == actual.siguiente.y)
      borrar.Add(actual);
}
foreach (Punto i in borrar) polar[inicio].borraPunto(i);
```

Código 4.4.2.5: Eliminación de puntos innecesarios.

Hecho esto, llega el momento de evitar las discontinuidades. Para ello, se debe verificar que la distancia entre un punto y su sucesor nunca sea superior a π radianes. De ser así, si el signo de ésta es negativo, se deberá rotar el punto siguiente en 2π radianes y, si el signo es positivo, se deberá rotar en -2π radianes. La consecuencia inmediata de esta transformación es que se eliminan todas las discontinuidades. El problema, que el intervalo angular en que pueden estar representados los puntos se incrementa considerablemente.

```
for (int i = 0; i < polar[inicio].nPuntos - 1; i++)
{
   actual = polar[inicio].puntos[i];
   Punto siguiente = actual.siguiente;
   if (siguiente.y - actual.y >= Math.PI) siguiente.y -= Math.PI * 2;
   if (siguiente.y - actual.y <= -Math.PI) siguiente.y += Math.PI * 2;
}
```

Código 4.4.2.6: Eliminación de discontinuidades.

Para solventar esta contrariedad, se deberán obtener los ángulos máximo y mínimo del conjunto de puntos, y garantizar que el mínimo esté comprendido en el intervalo $[0, 2\pi[$, pudiendo estar la totalidad de los puntos representados en el intervalo $[0, 4\pi[$. De esta forma, es posible que haya que rotar todos los puntos $+/-2\pi$, en función del ángulo mínimo.

```
for (int i = 0; i < polar[inicio].nPuntos; i++)
{
   actual = polar[inicio].puntos[i];
   if (actual.y < polar[inicio].anguloMin) polar[inicio].anguloMin = actual.y;
   if (actual.y > polar[inicio].anguloMax) polar[inicio].anguloMax = actual.y;
}

double desp = 0;
if (polar[inicio].anguloMin >= Math.PI * 2) desp = -Math.PI * 2;
if (polar[inicio].anguloMin < 0) desp = Math.PI * 2;
for (int i = 0; i < polar[inicio].nPuntos; i++) polar[inicio].puntos[i].y += desp;
polar[inicio].anguloMin += desp;
polar[inicio].anguloMax += desp;
```

Código 4.4.2.7: Representación de los puntos en el intervalo $[0, 4\pi[$.

Finalmente, se hará uso de la función *Sort()* propia de la lista para ordenar los puntos en base al criterio establecido (de izquierda a derecha y de abajo a arriba), y se obtendrá la distancia máxima para poder usar este valor durante la búsqueda de la alineación. Nótese que, una vez ordenados los puntos, ésta corresponderá con la coordenada *r* del último punto del vector

```
polar[inicio].puntos.Sort();
polar[inicio].distanciaMaxima = polar[inicio].puntos[polar[inicio].nPuntos - 1].x;
```

Código 4.4.2.8: Ordenación de la lista de puntos polares y cálculo de la distancia máxima.

El último paso restante de la fase de pre-procesado consiste en la obtención de los polígonos convexos polares que caractericen la configuración que se está estudiando. Recuérdese que, para ello, debía cumplirse la premisa de que cualquier punto debía tener sobre/bajo sí otros puntos, y no segmentos, por lo que será necesario calcular e insertar nuevos puntos interpolados que garanticen esta restricción. Así, para conseguir un coste lineal *O(n)*, aprovechando el orden establecido, será interesante crear en una primera pasada una lista en la que se almacenen sin duplicados todas las distancias *r* en las que hay puntos definidos.

```
List<double> coordenadasX = new List<double>();
for (int i = 0; i < polar[inicio].nPuntos; i++)
   if (!coordenadasX.Contains(polar[inicio].puntos[i].x))
      coordenadasX.Add(polar[inicio].puntos[i].x);
```

Código 4.4.2.9: Creación de la lista con todas las coordenadas r para las que hay puntos definidos en la representación polar.

Hecho esto, se recorrerán todos los puntos estudiando aquellos segmentos en los que el punto actual ocupe la posición de menor distancia *r*. Como se ha comentado en el apartado 3.2.3, se pueden dar cuatro casos posibles: que el punto estudiado esté a mayor distancia que sus vecinos, por lo que no se considerará, que sólo su sucesor esté a mayor distancia que él, por lo que se considerará el segmento que ambos definen, que sólo su predecesor esté a mayor distancia que él, por lo que se considerará este segmento, o que ambos vecinos estén a mayor distancia que él, por lo que habrá que considerar dos segmentos. El código 4.4.2.10 ilustra este proceso.

```
List<Punto> puntosNuevos = new List<Punto>();
int indiceCoordenadas = 0;
foreach (Punto pInicio in polar[inicio].puntos) {
   while (coordenadasX[indiceCoordenadas] < pInicio.x) indiceCoordenadas++;
   if (pInicio.siguiente.x > pInicio.x) {
      Punto pFin = pInicio.siguiente;
      Punto anterior = pInicio;
      for (int i = indiceCoordenadas + 1; coordenadasX[i] < pFin.x; i++) {
         double x = coordenadasX[i];
         double y = pInicio.y +
                    ((x - pInicio.x) / (pFin.x - pInicio.x) * (pFin.y - pInicio.y));
         Punto nuevo = new Punto(x, y);
         nuevo.anterior = anterior;
         puntosNuevos.Add(nuevo);
         anterior = nuevo;
      }
   }
```

```
    if (pInicio.anterior.x > pInicio.x)
    {
       Punto pFin = pInicio.anterior;
       Punto siguiente = pInicio;
       for (int i = indiceCoordenadas + 1; coordenadasX[i] < pFin.x; i++)
       {
          double x = coordenadasX[i];
          double y = pInicio.y +
                     ((x - pInicio.x) / (pFin.x - pInicio.x) * (pFin.y - pInicio.y));
          Punto nuevo = new Punto(x, y);
          nuevo.siguiente = siguiente;
          puntosNuevos.Add(nuevo);
          siguiente = nuevo;
       }
    }
}
```

Código 4.4.2.10: Creación de puntos intermedios que garantizan que ningún punto tiene sobre/bajo él aristas.

Del código anterior hay que destacar dos aspectos: nótese como *indiceCoordenadas* nunca decrementa su valor. Inicialmente vale cero y, a medida que los puntos estudiados están más a la derecha, su valor va incrementándose. Esto puede hacerse únicamente debido a que los puntos se estudian ordenadamente, de izquierda a derecha, por lo que se garantiza que nunca es necesario llevar a cabo una búsqueda sobre el vector *coordenadasX*, obteniéndose un coste *O(n)* en un proceso que, potencialmente, debía ser *O(n log n)*. El segundo aspecto a destacar es la referenciación parcial de los puntos nuevos que se crean. Cuando se interpola un segmento no es posible conocer, a priori, cuántos puntos se insertarán, por lo que no se puede establecer inmediatamente la vecindad de un punto. Lo que sí se puede asegurar es que si se da el segundo caso contemplado en el código, por ejemplo, en el que *pInicio.anterior.x* > *pInicio.x* (el segmento lo forman el punto estudiado y su predecesor) tras la primera inserción, el punto siguiente del recién creado será el punto de inicio, el punto siguiente del segundo que se cree será el primer punto que se había creado, y así sucesivamente. De esta manera, cuando se da este caso es posible pronunciarse a cerca del sucesor, mientras que cuando se da el otro caso *pInicio.siguiente.x* > *pInicio.x* es posible pronunciarse únicamente a cerca del antecesor.

Como sobre un segmento nunca sucederán los dos casos no hay posibilidad de que aparezcan informaciones contradictorias por lo que, una vez definidos los puntos nuevos, será necesaria una segunda pasada en la que éstos se inserten ordenadamente en la lista original, y en la que se reconstruyan las referencias parcialmente definidas que hayan quedado. El código 4.4.2.11 muestra cómo se lleva a cabo este proceso.

```
foreach (Punto p in puntosNuevos)
{
   if (p.anterior != null)
   {
      p.siguiente = p.anterior.siguiente;
      p.siguiente.anterior = p;
      p.anterior.siguiente = p;
      polar[inicio].addPunto(p);
   }
```

```
    else if (p.siguiente != null)
    {
        p.anterior = p.siguiente.anterior;
        p.anterior.siguiente = p;
        p.siguiente.anterior = p;
        polar[inicio].addPunto(p);
    }
}
polar[inicio].puntos.Sort();
```

Código 4.4.2.11: Inserción de los nuevos puntos, ordenación del vector polar y actualización de las referencias parcialmente definidas.

Una vez obtenida la representación que cumple todas las restricciones necesarias para el cálculo de polígonos convexos, es el momento de abordar este problema. Recuérdese del apartado 3.2.3 el procedimiento: para cada punto deben tomarse los vecinos cuya coordenada *r* sea mayor a la del punto estudiado. En base a éstos, debe insertarse sobre un polígono vacío los puntos *actual* y *vecino* hasta que el polígono activo tenga cuatro puntos. En este momento, tal y como se ha demostrado, el polígono estará totalmente definido, y se añadirá a la lista de polígonos, creando un nuevo polígono activo vacío, y prosiguiendo con el resto de vecinos y puntos del contorno. El resultado tras iterar sobre todos los puntos será una lista de polígonos definidos en los intervalos angulares *[0, 4π[*. El código 4.4.2.12 ilustra este proceso.

```
poligonos[inicio] = new List<Poligono>();
List<Poligono> intervalosTemp = new List<Poligono>();
List<Punto> puntosIntervalo = new List<Punto>();
for (int i = 0; i < polar[inicio].nPuntos; i++)
{
    List<Punto> vecinos = new List<Punto>();
    actual = polar[inicio].puntos[i];
    if (actual.x == polar[inicio].distanciaMaxima) break;
    if (actual.siguiente.x > actual.x) vecinos.Add(actual.siguiente);
    if (actual.anterior.x > actual.x) vecinos.Add(actual.anterior);
    vecinos.Sort();

    foreach (Punto vecino in vecinos)
    {
        puntosIntervalo.Add(actual);
        puntosIntervalo.Add(vecino);
        if (puntosIntervalo.Count == 4)
        {
            intervalosTemp.Add(new Poligono(puntosIntervalo[0], puntosIntervalo[1],
                                            puntosIntervalo[4], puntosIntervalo[5]));
            puntosIntervalo.Clear();
        }
    }
}
```

Código 4.4.2.12: Creación de los polígonos convexos que definen el contorno.

Para solucionar las ambigüedades que aparecen en la representación angular de los polígonos, será necesario corregir sus coordenadas de manera que se garantice que cada polígono esté definido en el intervalo *[0, 2π]*, siendo el ángulo *2π* válido únicamente en los polígonos que hayan tenido que ser divididos. De este modo, recuérdese que pueden darse tres casos: el

polígono puede estar definido en un intervalo válido, por lo que no será necesario actuar sobre él. Alternativamente, la totalidad del polígono puede estar en un intervalo no válido, por lo que todos sus puntos deben desplazarse *-2π* radianes. En último lugar, es posible que el polígono esté definido parcialmente en el intervalo *[0, 2π[*, por lo que será necesario dividirlo en dos: un primer polígono definido por el segmento inferior original y un nuevo segmento superior cuyas coordenadas *r* coincidan con los del segmento inferior y cuyas coordenadas *α* sean *2π,* y un segundo polígono cuyo segmento superior sea el original desplazado *-2π* radianes y un nuevo segmento inferior cuyas coordenadas *r* coincidan con los del segmento superior y cuyas coordenadas *α* sean 0. El siguiente fragmento de código ilustra este procedimiento de normalización, tras el que se garantiza que todos los polígonos están representados en el intervalo *[0, 2π]*.

```
foreach (Poligono inter in intervalosTemp)
{
   // Totalmente encima de los 2π radianes
   if (inter.yMin >= Math.PI * 2)
   {
      Poligono nuevo = new Poligono(
               new Punto(inter.puntos[0].x, inter.puntos[0].y - (Math.PI * 2)),
               new Punto(inter.puntos[1].x, inter.puntos[1].y - (Math.PI * 2)),
               new Punto(inter.puntos[4].x, inter.puntos[4].y - (Math.PI * 2)),
               new Punto(inter.puntos[5].x, inter.puntos[5].y - (Math.PI * 2)));
      poligonos[inicio].Add(nuevo);
   }

   // El intervalo debe dividirse en otros dos
   else if (inter.yMax > Math.PI * 2)
   {
      Poligono nuevo = new Poligono(
                  new Punto(inter.puntos[0].x, inter.puntos[0].y),
                  new Punto(inter.puntos[1].x, inter.puntos[1].y),
                  new Punto(inter.puntos[0].x, Math.PI * 2),
                  new Punto(inter.puntos[1].x, Math.PI * 2));
      poligonos[inicio].Add(nuevo);
      nuevo = new Poligono(
                  new Punto(inter.puntos[0].x, 0),
                  new Punto(inter.puntos[1].x, 0),
                  new Punto(inter.puntos[0].x, inter.puntos[4].y - (Math.PI * 2)),
                  new Punto(inter.puntos[1].x, inter.puntos[5].y - (Math.PI * 2)));
      poligonos[inicio].Add(nuevo);
   }

   // El intervalo ya está dentro de [0.. 2π[
   else poligonos[inicio].Add(inter);
}
poligonos[inicio].Sort();
```

Código 4.4.2.13: Corrección de ambigüedades en la representación de los polígonos.

En este punto, el pre-procesado del contorno ha concluido y se dispone de una lista ordenada de polígonos convexos polares que lo caracterizan totalmente. Pese a que se puede liberar espacio borrando las listas de puntos polares, en la implementación presentada se han mantenido por no suponer un coste considerable, y por reutilizar estos datos ante posibles ampliaciones futuras.

4.4.3. Búsqueda de alineación

Una vez el usuario ha seleccionado dos archivos que contienen la descripción de los dos contornos a estudiar, el objeto que deriva de la clase *Comparador* ha creado dos instancias de la clase *Perfil*, *p1* y *p2*, y ha invocado a su constructor encargado de llevar a cabo el pre-procesado de ambos.

Para poder obtener las correspondencias entre todas las configuraciones posibles, será necesario calcular primero la alineación de cada una de ellas y, posteriormente, aplicar la métrica seleccionada para cuantificar la calidad del match. Este apartado detalla el cálculo de la alineación, dada una configuración, y el siguiente apartado se centra en el cálculo de la correspondencia, dada una alineación.

Así, el método encargado de obtener todas las alineaciones posibles, dada una configuración, *List<Match> Angulos(int inicio1, int inicio2)*, toma como únicos parámetros los puntos iniciales para cada contorno, y devuelve una lista con todas las alineaciones posibles, y la puntuación que cada una de ellas ha obtenido durante el cálculo de las correspondencias.

Tal y como se comentaba en el apartado 3.3.1, el primer paso a llevar a cabo en la búsqueda de alineaciones consiste en subdividir los polígonos convexos de ambos perfiles para garantizar que todos los puntos que los forman están alineados entre ellos. Con el ánimo de mantener el coste *O(n)*, se procederá en dos pasos: en primer lugar se obtendrá una lista con todas las coordenadas *r* de los puntos de ambos contornos sin duplicados (y ordenada ascendentemente) y, posteriormente, en base a esta lista se recorrerán los polígonos de cada contorno, subdividiendo aquellos que no cumplan el criterio de alineación. El código 4.4.3.1 muestra el cálculo de esta lista y la invocación a un método privado *DivideIntervalos* que se encarga de procesar cada contorno, aplicando las subdivisiones necesarias sobre sus polígonos.

```
List<double> coordenadasX = new List<double>();
for (int i = 0; i < p1.polar[inicio1].nPuntos; i++)
   if (!coordenadasX.Contains(p1.polar[inicio1].puntos[i].x))
      coordenadasX.Add(p1.polar[inicio1].puntos[i].x);
for (int i = 0; i < p2.polar[inicio2].nPuntos; i++)
   if (!coordenadasX.Contains(p2.polar[inicio2].puntos[i].x))
      coordenadasX.Add(p2.polar[inicio2].puntos[i].x);
coordenadasX.Sort();

double distMax = Math.Min(p1.polar[inicio1].distanciaMaxima,
                                                p2.polar[inicio2].distanciaMaxima);
List<Poligono> intervalos1 = DivideIntervalos(
                                    p1.poligonos[inicio1], coordenadasX, distMax);
List<Poligono> intervalos2 = DivideIntervalos(
                                    p2.poligonos[inicio2], coordenadasX, distMax);
```

Código 4.4.3.1: División de los polígonos convexos de ambas figuras para garantizar la alineación de sus segmentos.

Nótese en el código anterior como se ha obtenido la distancia mínima de las distancias máximas de ambos contornos para que, durante la búsqueda, se pueda detener el proceso cuando lo puntos restantes no aporten nueva información.

El método *DivideIntervalos*, toma como parámetros de entrada los intervalos de un polígono, el vector calculado de coordenadas *r* sin duplicados, y la distancia máxima que será la que le permita detener la ejecución antes de comprobar todos los puntos. Como resultado proporciona una nueva lista de polígonos, cuya unión es igual a la unión de la lista que se pasó, pero en la que puede que varios polígonos hayan sido subdivididos. El código 4.4.3.2 muestra la implementación de este método.

```
public List<Poligono> DivideIntervalos(List<Poligono> intervalos,
                                        List<double> coordenadasX, double distMax)
{
   List<Poligono> res = new List<Poligono>();
   int inicioIntervalos = 0;
   int inicioCoordenadas = 0;

   while (coordenadasX[inicioCoordenadas] < distMax)
   {
      // Obtenemos la posición del último intervalo del mismo tramo
      int finIntervalos;
      for (finIntervalos = inicioIntervalos; finIntervalos < intervalos.Count;
                                                                  finIntervalos++)
         if (intervalos[finIntervalos].puntos[0].x ==
                                    intervalos[inicioIntervalos].puntos[1].x) break;

      // Obtenemos la posición de la última coordenada del mismo tramo
      int finCoordenadas;
      for (finCoordenadas = inicioCoordenadas;
                             finCoordenadas < coordenadasX.Count; finCoordenadas++)
         if (coordenadasX[finCoordenadas] ==
                                    intervalos[inicioIntervalos].puntos[1].x) break;

      // Si no hay ninguna coordenada X intermedia, los intervalos no se dividen
      if (finCoordenadas - inicioCoordenadas == 1)
         for (int i = inicioIntervalos; i < finIntervalos; i++)
            res.Add(intervalos[i]);

      // Sino, dividimos los intervalos del tramo con todas las coordenadas
      else
      {
         for (int i = inicioIntervalos; i < finIntervalos; i++)
         {
            List<Poligono> temp = intervalos[i].SeccionMultiple(
                               coordenadasX, inicioCoordenadas + 1, finCoordenadas);
            foreach (Poligono inter in temp) res.Add(inter);
         }
      }

      // Actualizamos los inicios para la siguiente iteración
      inicioIntervalos = finIntervalos;
      inicioCoordenadas = finCoordenadas;
   }
   res.Sort();
   return res;
}
```

Código 4.4.3.2: Implementación del método de subdivisión de intervalos

Básicamente, durante la división de intervalos, se recorre el vector de coordenadas *r* de izquierda a derecha, seleccionando para cada contorno los polígonos que están definidos en la distancia actual (lista *intervalos*). En caso de que los polígonos seleccionados tengan su punto p_0 definido en la distancia actual, y su punto p_1 definido en la siguiente distancia de la lista de distancias, no será necesario dividirlos, y se adjuntarán automáticamente a la lista de resultados. En caso contrario, se invocará a un método definido en *Poligono* llamado *SeccionMultiple* que, tomando como parámetros de entrada las coordenadas *r* intermedias y un polígono devuelve una lista de sub-polígonos cuya unión corresponde al polígono original, y cuyos segmentos están alineados a las coordenadas *r* proporcionadas.

Para garantizar que el coste es lineal y que no se procesan más puntos de los necesarios, nótese como la guarda del bucle exterior falla en el momento que la distancia *r* estudiada supera el mínimo de las distancias máximas de ambos contornos. Durante todas las iteraciones, los índices que apuntan sobre las listas de coordenadas *r* y polígonos únicamente se incrementan, por lo que no se llevan a cabo búsquedas que incrementen la complejidad del método.

El método *SeccionMultiple*, cuya implementación se muestra en el código 4.4.3.3, debe calcular los ángulos α que correspondan a las coordenadas *r* proporcionadas, mediante la interpolación lineal de los segmentos superior e inferior del polígono que se esté estudiando. Así, salta a la vista que las primeras coordenadas α deberán corresponderse con las de los puntos iniciales de ambos segmentos p_0, p_2. Las coordenadas intermedias se corresponderán con la interpolación lineal de los segmentos (obtenida mediante el método *Seccion*), y las últimas coordenadas se corresponderán con los puntos finales de ambos segmentos p_1, p_3. Finalmente, deberán emparejarse las distancias del vector de coordenadas *r* con los ángulos obtenidos para ellas, formando polígonos convexos cuya definición se corresponda con el criterio dado en el apartado 3.2.3.

```
public List<Poligono> SeccionMultiple(List<double> x, int inicio, int fin)
{
   List<double[]> segmentos = new List<double[]>();
   segmentos.Add(new double[] { puntos[0].y, puntos[4].y });
   for (int i = inicio; i < fin; i++) segmentos.Add(Seccion(x[i]));
   segmentos.Add(new double[] { puntos[1].y, puntos[5].y });

   List<Poligono> res = new List<Poligono>();
   for (int i = 0; i < segmentos.Count - 1; i++)
   {
      Punto p1 = new Punto(x[inicio - 1 + i], segmentos[i][0]);
      Punto p2 = new Punto(x[inicio + i], segmentos[i + 1][0]);
      Punto p3 = new Punto(x[inicio - 1 + i], segmentos[i][1]);
      Punto p4 = new Punto(x[inicio + i], segmentos[i + 1][1]);
      res.Add(new Poligono(p1, p2, p3, p4));
   }
   return res;
}
```

Código 4.4.3.3: Implementación del método SeccionMultiple.

Una vez divididos todos los intervalos, puede comenzar el proceso de búsqueda de la alineación de los contornos. Dado que éste es un tanto complejo, su explicación se dividirá en dos partes: en primer lugar se estudiará el bucle externo, que es el encargado de iterar a lo largo

de todas las coordenadas *r*, intersectando las restricciones de cada conjunto de intervalos con el espacio libre. Aquí, se estudiará la guarda del bucle, la evolución de la búsqueda y la selección de intervalos que definen el problema local, dada una distancia *r*. En segundo lugar se estudiará el bucle interno que, tomando como datos de partida los intervalos seleccionados en el bucle externo, calculará las restricciones locales que impone cada polígono del contorno móvil y las intersectará con el resto de polígonos definidos para la distancia *r*, con el ánimo de obtener la restricción global que existe en *r* y que posteriormente debe ser intersectada con el espacio libre.

Así, el código 4.4.3.4 muestra la implementación del bucle externo de la búsqueda, en la que se ha indicado (en color rojo), la posición en la que habrá que insertar el bucle interno, que más adelante se comentará.

```
List<double[]> libre = new List<double[]>();
libre.Add(new double[4] { 0, Math.PI * 2 });

double xActual = 0;
int indice1 = 0;
int indice2 = 0;
while (xActual < distMax)
{
   List<double[]> segmentos1 = new List<double[]>();
   while(indice1 < intervalos1.Count)
   {
      if (intervalos1[indice1].puntos[0].x > xActual) break;
      segmentos1.Add(new double[]
           { intervalos1[indice1].puntos[0].y, intervalos1[indice1].puntos[4].y });
      indice1++;
   }

   List<double[]> segmentos2 = new List<double[]>();
   while (indice2 < intervalos2.Count)
   {
      if (intervalos2[indice2].puntos[0].x > xActual) break;
      segmentos2.Add(new double[]
           { intervalos2[indice2].puntos[0].y, intervalos2[indice2].puntos[4].y });
      indice2++;
   }

   // Obtenemos las rotaciones posibles
   List<double[]> posiblesXActual = new List<double[]>();
   posiblesXActual.Add(new double[4] { 0, Math.PI * 2 });

   … (Bucle interno: Cálculo de los ángulos posibles en la coordenada r actual) …

   // Intersectamos los ángulos posibles para la X actual con el espacio libre
   libre = Interseccion(libre, posiblesXActual);
   if (libre.Count == 0) break;

   // Actualizamos para la siguiente iteracion
   if (indice1 == intervalos1.Count) break;
   xActual = intervalos1[indice1].puntos[0].x;
}
```

Código 4.4.3.4: Bucle externo de la búsqueda de alineaciones.

En vista del código mostrado, nótese como la guarda del bucle falla cuando la coordenada *r* que se está estudiando supera la distancia mínima de las distancias máximas de ambos contornos.

Las primeras líneas de código dentro del bucle tienen el objetivo de seleccionar los datos sobre los que actuará en el bucle interno. Recordando la figura 3.3.2.4, y su explicación, el primer paso a llevar a cabo consistía en una reducción de la dimensionalidad del problema. Así, es necesario obtener los intervalos angulares en los que los polígonos están definidos para una distancia *r*, expresada en el código como *xActual*. Dado que los segmentos de ambos contornos están alineados, este cálculo puede realizarse de manera trivial.

Centremos la atención, por ejemplo, en el contorno fijo, y asumamos que el índice que selecciona el polígono actual de éste, *indice1*, apunta al primer polígono cuyo punto $p_0 = r$. En una lista llamada *segmentos1*, se deben introducir todos los intervalos angulares en los que el contorno fijo esté definido. Así, recorriendo los polígonos sucesores al indicado por *indice1* cuyo punto $p_0 = r$, se obtendrán fácilmente estos valores. Cuando el polígono que se esté estudiando tenga $p_0 > r$ la guarda del bucle fallará, quedando *indice1* actualizado para la siguiente iteración. La figura 4.4.3.1 muestra gráficamente el resultado de este proceso, en el que están representados en negro *b)* los intervalos pertenecientes a *segmentos1*, y en verde los intervalos pertenecientes a *segmentos2*. El apartado *c)* de la imagen muestra cómo quedan los índices después de la selección de intervalos. Nótese como ya están ubicados en las posiciones en que deberán comenzar en la siguiente iteración.

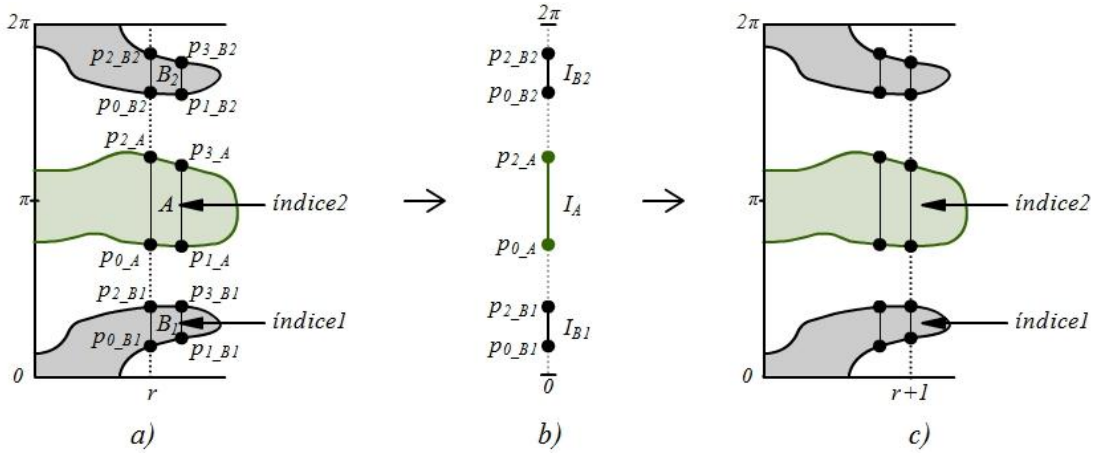

Figura 4.4.3.1: Selección de intervalos y actualización de índices para la siguiente iteración.

Una vez reducida la dimensionalidad del problema, el código mostrado inicializa una lista *posiblesXActual* con un único intervalo *[0, 2π[*. El objetivo de esta lista consiste en recoger todos los ángulos en los que el perfil móvil puede encontrarse sin incurrir en una penetración con el perfil fijo. Su cálculo se comentará más adelante cuando se detalle el bucle interno, pero si resulta conveniente adelantar que, tras éste, *posiblesXActual* contendrá una serie de intervalos definidos entre *[0, 2π[* que deberán intersectarse con el espacio libre.

Nótese como las dos primeras líneas de código han definido la lista *libre*, que ha sido inicializada con un único intervalo *[0, 2π[*. Tras estudiar las restricciones impuestas en cada *r*, esta lista se intersecta con *posiblesXActual*, para reflejar qué intervalos angulares van quedando libres a medida que se avanza sobre el eje *r*. Si en algún momento la intersección es vacía, no tiene sentido continuar la búsqueda, ya que la alineación no es posible. En este caso, se sale del bucle.

Dado que la intersección entre intervalos no supone una complejidad considerable no se comentará la implementación de esta operación. Únicamente resulta interesante precisar que, debido a errores de redondeo consecuencia de la representación interna de las variables de tipo *double*, no ha sido posible hacer uso de los operadores definidos por C# ($>$, $<$, $=$, $\geq$, $\leq$). En su lugar, ha sido necesario implementar operaciones homólogas que consideren una tolerancia máxima para pronunciarse acerca de la relación entre los valores estudiados. A continuación se muestra el código desarrollado.

```
private static bool Igual(double A, double B) { return Math.Abs(A - B) < 0.00001; }
private static bool Menor(double A, double B) { return A - B < -0.00001; }
private static bool Mayor(double A, double B) { return A - B > 0.00001; }
private static bool MenorIgual(double A, double B) {return Menor(A,B)||Igual(A,B);}
private static bool MayorIgual(double A, double B) {return Mayor(A,B)||Igual(A,B);}
```

Código 4.4.3.5: Implementación de los operadores de comparación.

Finalmente el bucle interno de la búsqueda obtendrá, para cada segmento del perfil móvil extraído, los intervalos en los que éste no penetra en los segmentos del perfil fijo extraídos, *posiblesSegmento*. Este resultado se intersectará con los obtenidos para el resto de segmentos del perfil móvil quedando, finalmente, en *posiblesXActual* reflejadas todas las restricciones impuestas por los contornos para una distancia *r*. El código 4.4.3.6 muestra la implementación de este bucle, cuya ubicación el bucle externo está indicada en color rojo en el código 4.4.3.4.

```
foreach (double[] segmento2 in segmentos2)
{
   List<double[]> posiblesSegmento = new List<double[]>();
   for (int i = 0; i < segmentos1.Count; i++)
   {
      double altura2 = segmento2[1] - segmento2[0];
      double altura1 = segmentos1[(i+1) % segmentos1.Count][0] - segmentos1[i][1];
      if (altura1 < 0) altura1 += Math.PI * 2;
      if (MayorIgual(altura1, altura2))
      {
         double desplazamiento = segmentos1[i][1] - segmento2[0];
         double[] nuevo = new double[4]
                  { desplazamiento, desplazamiento + Math.Abs(altura2 - altura1) };
         while (nuevo[0] < 0) { nuevo[0] += Math.PI * 2; nuevo[1] += Math.PI * 2; }
         posiblesSegmento.Add(nuevo);
      }
   }
   posiblesXActual = Interseccion(posiblesXActual, posiblesSegmento);
}
```

Código 4.4.3.6: Bucle interno de la búsqueda de la alineación.

4.4.4. Cálculo de correspondencias

Una vez se han obtenido las rotaciones que alinean dos contornos, es necesario apoyarse en alguna métrica de similitud que establezca cuán buena es la correspondencia obtenida. La métrica a escoger dependerá de las características del problema real que se pretenda solucionar, y su implantación sobre los resultados obtenidos es inmediata: todas ellas calculan distancias en base a las coordenadas de los puntos.

Con el objetivo de disponer de una aplicación de prueba real, y continuar ilustrando las bondades de la representación polar escogida, se ha optado por hacer uso de la métrica LCP ampliada que se propuso en el apartado 3.4. Ésta consistía en obtener el número de puntos que hay en contacto entre las formas, teniendo en cuenta un grado de tolerancia introducido por el usuario. La mejora que introduce la representación polar es que, además de los puntos originales de los contornos, se dispone de una serie de puntos interpolados que garantizan que, dada una coordenada polar *r*, un punto tiene sobre/bajo sí otros puntos, no aristas. Así, se considerará que el punto homólogo de uno dado será aquel del otro contorno cuya coordenada *r* sea igual, y cuya distancia angular sea mínima. En base a este criterio, si la distancia geométrica entre ellos es menor que la tolerancia, se considerará que están en correspondencia y se incrementará la puntuación del match.

Nótese que, considerando los puntos polares interpolados, se incrementa considerablemente el poder descriptivo de la métrica. A modo de ejemplo, se plantea el caso ilustrado en la figura 4.4.4.1, en el que se ha definido una tolerancia $\epsilon = 0$. De usar LCP tradicional *a)*, únicamente los puntos de inicio de ambos contornos estarían en contacto, mientras que es evidente que existen múltiples puntos sobre los que se ha establecido una correspondencia. Mediante LCP polar *b)*, utilizando los puntos interpolados, puede observarse como el número de puntos en contacto se corresponde con la realidad. Este tipo de errores en LCP tradicional están muy relacionados con los errores introducidos durante la discretización de los contornos.

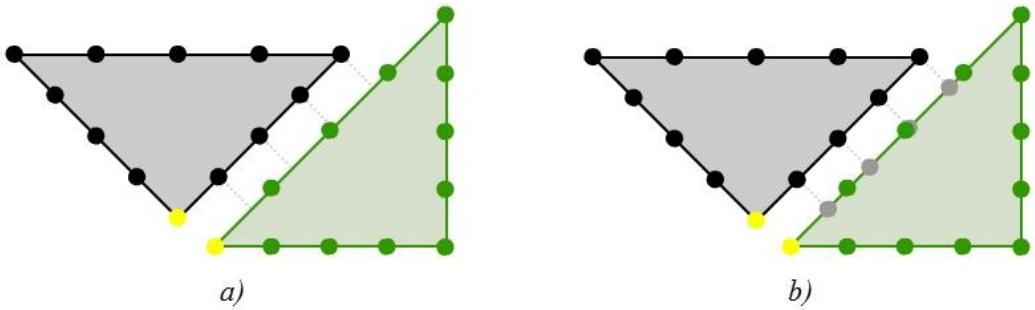

Figura 4.4.4.1: Diferencias entre LCP tradicional y LCP polar

Para implementar LCP polar, se partirá de la lista *libre* obtenida durante la búsqueda de la alineación, cuyos miembros son intervalos de rotación libres de penetración, de los que sus extremos constituirán el conjunto de rotaciones que alinean las formas (tal y como se explicó en el apartado 3.3.2). Así, para cada intervalo perteneciente a la lista *libre*, deberán estudiarse dos correspondencias, una para cada extremo. El resultado final será una lista de objetos derivados de la clase *Match*, que contendrán la alineación que se está estudiando (i_p, i_q, α_q) y la puntuación obtenida durante el cálculo de correspondencias, que será la clave de ordenación de la lista.

El siguiente código muestra cómo se genera esta lista de correspondencias, que constituye las últimas líneas del método de búsqueda de alineaciones:

```
List<Match> res = new List<Match>();
foreach (double[] l in libre)
{
   res.Add(CrearMatchLCP(inicio1, inicio2, l[0], intervalos1, intervalos2));
   res.Add(CrearMatchLCP(inicio1, inicio2, l[1], intervalos1, intervalos2));
}
return res;
```

Código 4.4.4.1: Creación de la lista de correspondencias para una configuración dada.

La implementación del método *CrearMatchLCP* debe recorrer todos los puntos *originales* de uno de los contornos, comprobando si sus homólogos originales o interpolados satisfacen la condición de distancia fijada por el usuario. Para cada emparejamiento válido, la puntuación de la correspondencia se incrementará en una unidad. El código 4.4.4.2 muestra la implementación del método *CrearMatchLCP*.

```
double puntuacion = 0;
double distMin = 0.5;
int indice1 = 0;
int indice2 = 0;

while (p1.polarOriginal[inicio1].puntos[indice1].x == 0) indice1++;

while (indice1 < p1.polarOriginal[inicio1].puntos.Count)
{
   Punto actual = p1.polarOriginal[inicio1].puntos[indice1];
   while (indice2 < intervalos2.Count)
   {
      if (intervalos2[indice2].puntos[0].x == actual.x) break;
      indice2++;
   }
   for (int i = indice2; i < intervalos2.Count; i++)
   {
      if (intervalos2[i].puntos[0].x > actual.x) break;
      double distancia1 = Math.Abs(Math.Sin((actual.y -
                     (intervalos2[i].puntos[0].y + angulo2)) / 2) * 2 * actual.x);
      if (distancia1 < distMin) { puntuacion++; break; }

      double distancia2 = Math.Abs(Math.Sin((actual.y -
                     (intervalos2[i].puntos[4].y + angulo2)) / 2) * 2 * actual.x);
      if (distancia2 < distMin) { puntuacion++; break; }
   }
   indice1++;
}
Match res = new Match(inicio1, inicio2, angulo2);
res.puntuacion = puntuacion;
return res;
```

Código 4.4.4.2: Creación de la lista de correspondencias para una configuración dada.

En este código, la lista *p1.polarOriginal[inicio1]*, contiene todos los puntos no interpolados del contorno fijo original. El método recorrerá cada uno de ellos en busca de la distancia con su

homólogo en el otro contorno. No obstante, nótese como los puntos de inicio no aportan ninguna información (siempre están en contacto), por lo que *indice1*, que es la variable que apunta al punto activo, se desplaza hacia la derecha hasta encontrar el primer punto cuya coordenada *r* sea mayor que cero.

La variable *indice2* es la que apunta al polígono actual del contorno móvil que ha sido empleado durante el proceso de alineación, siendo *intervalos2[indice2].puntos[0]* el punto de inicio del segmento inferior del polígono actual e *intervalos2[indice2].puntos[4]* el punto de inicio del segmento superior. Nótese que el valor de *indice2* nunca se decrementa, por lo que se garantiza el coste lineal *O(n)*.

El punto actual del contorno fijo definirá dos distancias con cada polígono del contorno móvil. Si alguna de ellas es inferior al umbral establecido, se pasará a estudiar el siguiente punto del contorno fijo, incrementando en uno la puntuación del match.

4.5. Resultados

Tras el proceso de implementación, se ha obtenido una aplicación que ha permitido comprobar la validez de los resultados, a la par que ha sido muy útil durante el desarrollo de la técnica, ya que disponer de una representación gráfica del problema sobre ambos espacios de coordenadas simultáneamente ha permitido comprender mejor las peculiaridades del problema a solucionar. La figura 4.5.1 muestra el aspecto de la interfaz durante la ejecución

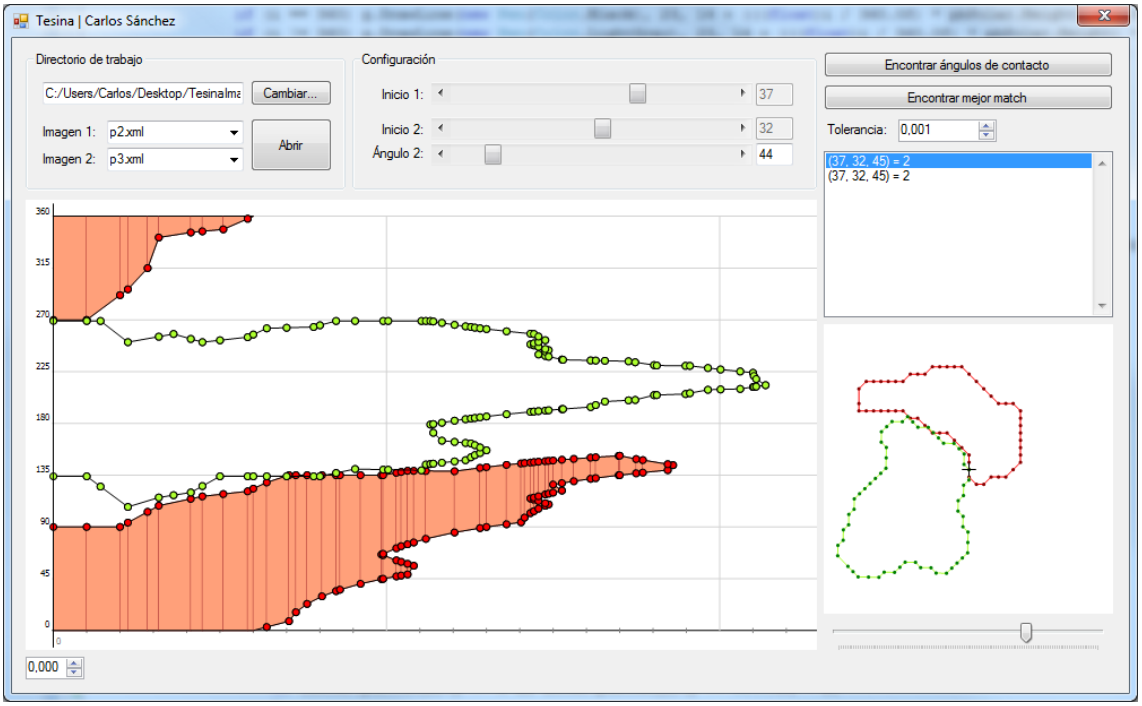

Figura 4.5.1: Aspecto de la aplicación desarrollada

Como puede observarse, la aplicación permite al usuario seleccionar los contornos a comparar (esquina superior izquierda), y le permite elegir una configuración específica mediante barras de deslizamiento (zona superior central). Cada vez que se modifica algún valor, las previas polar y cartesiana se actualizan mostrando la representación de ambos contornos superpuesta: en el espacio polar, el perfil móvil está representado en verde por sus puntos, mientras que el perfil fijo está representado en rojo por sus polígonos. En el espacio cartesiano, se mantienen los colores, y los puntos de contacto se encuentran directamente superpuestos bajo una cruz negra.

En la esquina superior derecha, un botón cuya etiqueta reza "*Encontrar ángulos de contacto"*, permite que, dada una configuración elegida por el usuario, se obtenga una lista de alineaciones puntuada que se muestra en el área de texto ubicada algo más abajo. Cada vez que el usuario pulsa sobre alguna alineación, la ventana de la aplicación se redibuja para mostrar las representaciones polar y cartesiana correspondientes.

Inmediatamente debajo aparece un segundo botón cuya etiqueta reza "*Encontrar mejor match"* que, cuando es pulsado, inicia el proceso de búsqueda ingenuo entre todas las configuraciones posibles. Los resultados obtenidos, debidamente puntuados, se muestran en el área de texto inferior de manera ordenada.

El campo numérico llamado *tolerancia*, permite establecer el umbral con el que LCP polar considerará que dos puntos homólogos están en correspondencia.

4.5.1. Costes

Al disponer de una implementación, es posible analizar los costes en base a datos reales. Así, será interesante confirmar que las predicciones se cumplen para la fase de pre-procesado y para la fase de búsqueda.

La imagen 4.5.1.1 muestra los resultados obtenidos durante el pre-procesado, en la que se comparan los tiempos de cómputo respecto al número de puntos que forman el contorno. Para evitar errores, cada contorno se ha cargado 10 veces, y el tiempo mostrado en la gráfica se ha obtenido como la media de los ensayos.

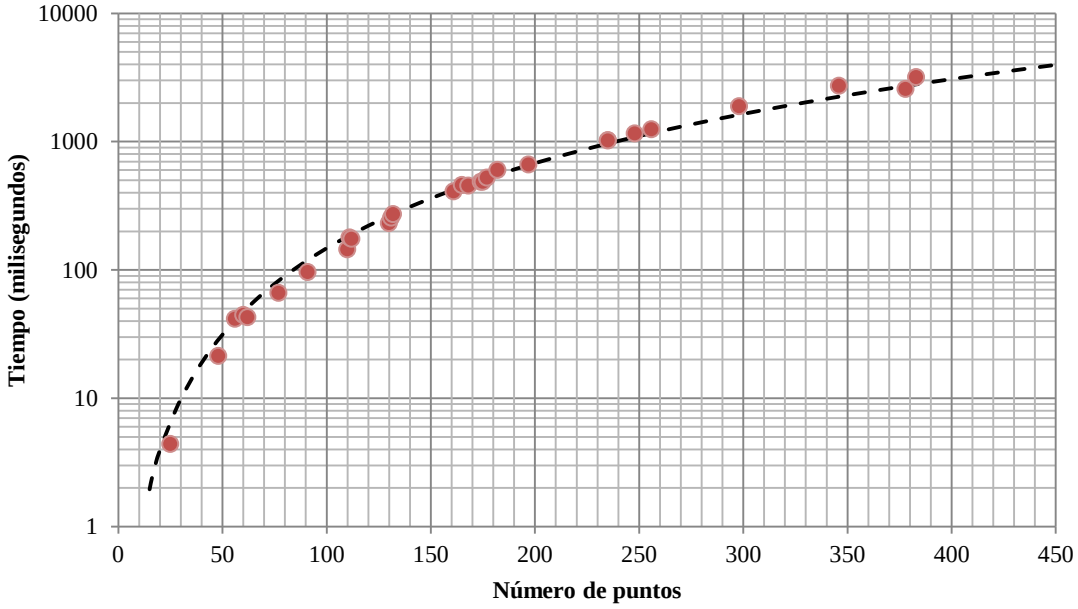

Figura 4.5.1.1: Tiempo de pre-procesado por número de puntos del contorno

En base a los resultados mostrados, considerando que la línea discontinua representa la función de coste $f(x)=n^2log(n)$, a la que se ha aplicado un coeficiente constante para poder comparar los datos, se puede apreciar cómo se confirma claramente el análisis de complejidad llevado a cabo en el apartado 3.5. Todas las muestras obtenidas se alinean en torno a la función de coste estimada.

La figura 4.5.1.3 muestra los resultados obtenidos durante el proceso de búsqueda. Nótese que, para no tener que representar una gráfica tridimensional, el número de puntos mostrado se ha obtenido como la suma del número de puntos de ambos contornos. Al igual que en la gráfica anterior, cada medición se ha tomado 10 veces, y el valor mostrado es la media de todas ellas.

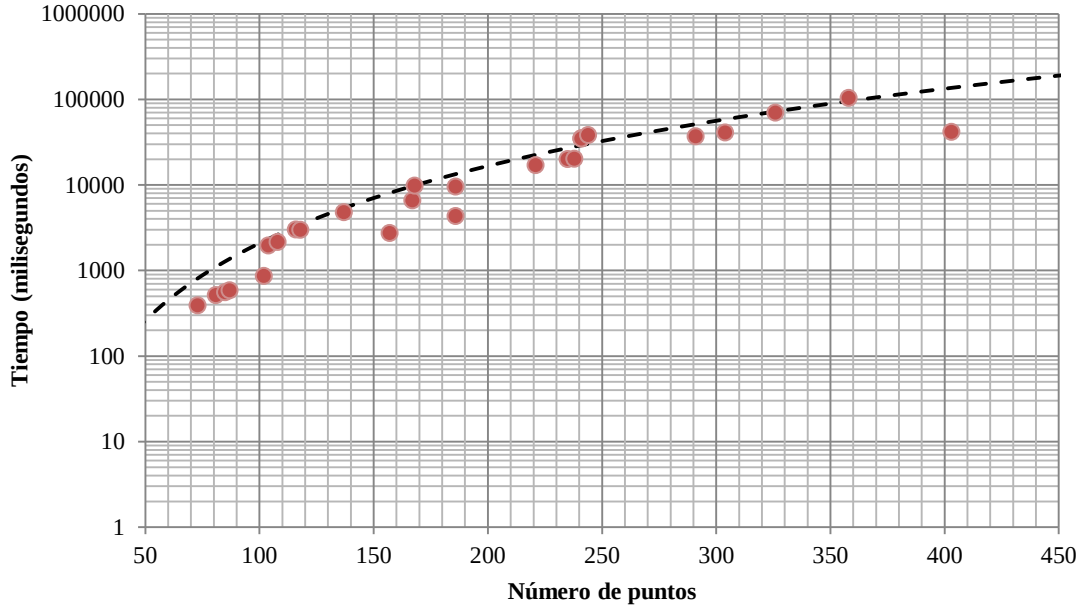

Figura 4.5.1.1: Tiempo de pre-procesado por número de puntos del contorno

En base a los resultados obtenidos, y considerando que la línea discontinua representa la función de coste $f(x)=n^3$, se puede apreciar cómo se confirma claramente el análisis de complejidad llevado a cabo en el apartado 3.5. No obstante, es preciso destacar que la variabilidad en los tiempos de cómputo es mayor en este caso que en el anterior. El motivo de este fenómeno se debe a que el algoritmo de búsqueda cuenta con condiciones de terminación precoz. Así, cuanto mayor sea la diferencia en tamaño de las dos muestras, menor será el tiempo de búsqueda, ya que debe recordarse que ésta se interrumpía cuando se alcanzaba la menor distancia máxima de ambos contornos. Además, cuanto mayor sea la irregularidad de las formas mayor será la posibilidad de que se den configuraciones con cero alineaciones posibles, que es la segunda condición de terminación adelantada.

Obsérvese como se dispone de dos combinaciones diferentes de contornos con 186 puntos en total, y cómo el tiempo de ejecución de una de ellas dobla al otro (9’5176 segundos frente a 4’321.2 segundos). Probablemente, la primera muestra se haya obtenido en base a dos contornos de tamaño similar, mientras que en la segunda los tamaños hayan sido mucho más dispares.

Capítulo 5

Conclusiones

A lo largo de este documento se ha presentado una técnica que permite calcular alineaciones y correspondencias para dos contornos bidimensionales cualesquiera, permitiendo obtener resultados exactos en base a la métrica aplicada.

Mediante una representación polar de los patrones que definen los contornos ha sido posible realizar la búsqueda de manera eficiente. Así, gracias al pre-procesamiento desarrollado, todo el cálculo de alineación y correspondencia se ha llevado a cabo mediante operaciones sencillas y, al definirse un criterio de ordenación para la información, ha sido posible que durante el proceso de búsqueda no haya habido necesidad de ejecutar bucles internos que permitan obtener los puntos homólogos a los puntos estudiados.

La conjunción de estas dos características ha dotado al algoritmo presentado de un coste de $O(n^3)$ que, pese a que en otras áreas supondría un mal resultado, en las técnicas de reconocimiento de patrones supone una mejora a los resultados hasta ahora obtenidos. Los estudios realizados sobre la implementación real del algoritmo confirman los datos obtenidos teóricamente.

De hecho, considerando que la técnica desarrollada es ingenua (debe comprobar todas las alineaciones posibles $O(n^2)$, utilizando todos los puntos para la verificación $O(n)$), se ha alcanzado la cota mínima de complejidad, por lo que para introducir mejoras en el rendimiento será necesario un planteamiento radicalmente distinto.

Recordando la tabla 2.5.1.1, en la tabla 5.1 se han marcado con fondo verde las técnicas comparables a la desarrollada. Nótese que las técnicas 3D han sido excluidas (por razones obvias), y que las técnicas que introducen aleatorización también se han descartado por no proporcionar resultados exactos. Finalmente, el método de *cuatro puntos coplanares* no se ha considerado por precisar de una correspondencia exacta entre los puntos escogidos como referencia para la alineación (la solución buscada debe ser robusta ante degradaciones de los contornos, bien introducidas por errores de discretizado durante la adquisición de imágenes, o bien por la propia erosión de los fragmentos originales).

Ante las técnicas de *alineación* y de *pose clustering*, la mejora es de $O(n^2 \log n)$ para la primera, y de $O(m)$ para la segunda, pese a que los requisitos espaciales son superiores.

Respecto a las técnicas de *hashing geométrico 2D*, nótese como sólo se ha mejorado la complejidad de las consultas en $O(\log n)$, aunque los requerimientos de pre-procesado se han contraído en $O(n)$, y el espacio necesario en $O(m)$.

Técnica	Método	2D		3D	
		Tiempo	Espacio	Tiempo	Espacio
Alineación	Algoritmo ingenuo	$O(m^3 n^2 \log n)$	-	$O(m^4 n^3 \log n)$	-
	Aleatorizado	$O(mn^2 \log n)$	-	$O(mn^3 \log n)$	-
	Verificación aleatoria	$O(n^2 \log n)$	-	$O(n^3 \log n)$	-
	4 puntos coplanares	$O(n^2 + k)$	$O(n)$	$O(n^2 + k)$	$O(n)$
Pose clustering	Algoritmo ingenuo	$O(m^2 n^2 + h)$	$O(h)$	$O(m^3 n^3 + h)$	$O(h)$
	Aleatorizado	$O(m n^2 + h)$	$O(h)$	$O(m n^3 + h)$	$O(h)$

Técnica	Método	Pre-procesado	Espacio	Consulta
Hashing geométrico 2D	Algoritmo original	$O(m^3 \log n)$	$O(m^3)$	$O(n^3 \log n)$
	Aleatorizado	$O(r^3 \log r)$	$O(r^3)$	$O(n^3 \log n)$

Tabla 5.1: Complejidad de las técnicas comparables a la técnica desarrollada.

Pese a que el problema de la correspondencia se haya solucionado en un ámbito general, de vuelta al problema arqueológico original, la implementación desarrollada supone una herramienta de gran valor para el proceso de reconstrucción a partir de fragmentos, ya que su aplicación es inmediata.

No obstante, nótese que en un problema con datos reales, el grado de erosión que pueden haber sufrido las figuras a comparar puede invalidar parcialmente la métrica que se emplee para cuantificar la similitud. En ese caso, la aplicación servirá como herramienta de ayuda en la toma de decisiones, sugiriendo a un usuario final las n configuraciones que mejor puntuación hayan obtenido, siendo éste quien se encargue de tomar la decisión final.

5.1. Ampliaciones futuras

Pese a haber obtenido una buena solución al problema original, durante la etapa de desarrollo y documentación del proyecto han aparecido nuevos retos que dejan abierto el tema. Así, la técnica desarrollada debe concebirse más como una base formal para nuevos proyectos que como una solución cerrada.

En este sentido, apoyándose en los logros obtenidos en cuanto al sistema de representación de los patrones, se distinguen dos líneas paralelas de ampliación: la aplicación de las técnicas desarrolladas a casos reales y la continuación en el estudio teórico del problema de la correspondencia.

Respecto a la aplicación real del problema, la arqueología sigue siendo un contexto ideal sobre el que continuar trabajando: los requisitos se corresponden exactamente con el problema a solucionar, existe gran cantidad de información susceptible de ser analizada y se trata de un tema de interés social e investigador.

En una aplicación arqueológica real se debería enfatizar más en el estudio de las métricas, ya que la erosión que sufren los fragmentos analizados supone un nuevo reto. En esta dirección, sería interesante considerar que el grado de similitud entre alineaciones no depende exclusivamente de la correspondencia entre los perfiles de los contornos. Enriquecer la medida de distancia con información relacionada con el color, las texturas de los contornos, las características de sus anversos, o la ubicación en la que fueron obtenidos ayudaría, sin duda, a clasificar mucho mejor la bondad de las correspondencias. Asimismo, afrontar el problema de la similitud como un problema de optimización multi-objetivo aportaría ventajas considerables. El procedimiento consistiría en añadir a la correspondencia únicamente aquellos puntos que incrementen la superficie en contacto, minimizando el error cometido. La relación entre estas dos características permitiría comparar de forma mucho más precisa piezas erosionadas por el paso del tiempo.

Relacionando la aplicación práctica de la técnica con el estudio teórico del problema está la extensión más inmediata: ampliar de la dimensionalidad de los datos. En este sentido, existe un gran número de restos arqueológicos para los que sólo se puede estudiar la correspondencia a partir de sus representaciones 3D. Nótese además que los requisitos del problema real imponen la necesidad de obtener una solución genérica: los restos arqueológicos presentan topologías muy variadas, con un gran número de irregularidades. Centrar el estudio sobre volúmenes sintéticos no permitiría aplicar los resultados teóricos al ámbito de trabajo.

En esta posible ampliación, los resultados obtenidos sobre el problema 2D son de gran ayuda, ya que las conclusiones alcanzadas son extrapolables a problemas de mayor dimensionalidad. No obstante, para que la aplicación sea inmediata, será necesario disponer de datos de entrada expresados como superficies cerradas, no como nubes de puntos. Recuérdese que la técnica desarrollada se basa en la relación *anterior*/*siguiente* de los puntos, y que el objetivo del pre-procesado es obtener un conjunto de polígonos convexos que definan un espacio ocupado (o libre). Así, abordar el problema de las 3D implicaría asumir que se dispone de datos de entrada que cumplan estos requisitos, o desarrollar técnicas que permitan generar objetos sólidos a partir de nubes de puntos (que es la información que proporcionan los escáneres 3D). No obstante, existen múltiples técnicas en la actualidad que permiten llevar a cabo este proceso.

El problema 3D añadiría nuevas dificultades en cuanto a la representación de los datos originales, puesto que sería necesario emplear un sistema de representación formal de los objetos que facilitase las búsquedas. En principio, una buena apuesta serían los modelos B-Rep (acrónimo de la expresión inglesa *Boundary Representation*) que, apoyada en la representación de aristas aladas, permitiría establecer fácilmente la vecindad de los puntos.

Respecto a la búsqueda, asumiendo que el planteamiento de las alineaciones se mantuviera (anulando las traslaciones), el espacio de soluciones se ampliaría en una dimensión, ya que sería necesario contemplar dos ángulos, θ y φ, tal y como ilustra la figura 5.1.1.

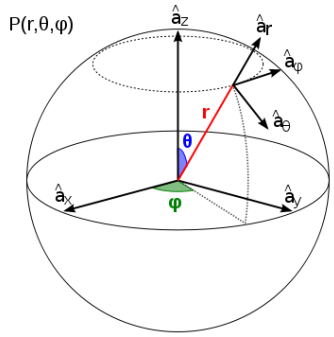

Figura 5.1.1: Sistema de coordenadas esférico (imagen obtenida de Wikipedia)

Dado que incluir una nueva dimensión en los datos de entrada implica una explosión combinatoria en el espacio de soluciones que incrementaría considerablemente los costes, sería necesario encontrar algún método de aceleración que atenuase estas consecuencias. En este plano, puramente teórico, el estudio de la relación entre las correspondencias parciales frente a las globales podría aportar grandes ventajas. Así, un planteamiento alternativo para las alineaciones estudiadas sería tratar jerárquicamente las representaciones de las formas de entrada. Si se generan versiones de diferente resolución de los datos originales, y se estudia la correspondencia primero de forma burda, refinando únicamente las configuraciones más prometedoras se podrían obtener algoritmos de búsqueda cuya complejidad teórica sería de $O(n \log n)$ para el estudio de la alineación + correspondencia en el caso 2D (frente al $O(n^3)$ obtenido). Tal y como ilustra la figura 5.1.2, sería necesario considerar los errores de discretización cometidos, para garantizar que la búsqueda no se detiene ante mínimos locales.

Figura 5.1.2: Comparación jerárquica de formas.

Como ampliación referida a la implementación, sería interesante portar el algoritmo original a código de bajo nivel, ejecutado sobre unidades especializadas en cálculo numérico en coma flotante.

Así, la nueva generación de GPUs (acrónimo de la expresión inglesa *Graphic Processor Units*) proporciona rendimientos muy superiores a las CPUs en este tipo de operaciones (del orden de las seis veces). La figura 5.1.3 muestra una comparativa en GFLOPS por segundo entre procesadores Intel y GPUs de nVidia.

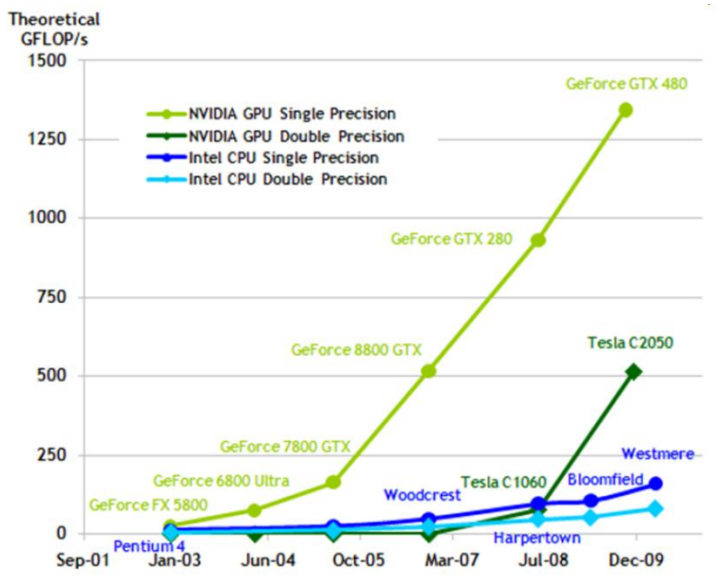

Figura 5.1.3: Comparativa entre GPUs y CPUs publicada por nVidia.

Además, dado que la naturaleza de las operaciones que se llevan a cabo permiten la distribución inmediata de los cálculos, una implementación paralela que aprovechase la potencia de múltiples unidades de procesamiento permitiría reducir los tiempos de forma lineal al número de unidades que se empleasen. En este sentido, las GPUs también proporcionan grandes ventajas, por contar con múltiples núcleos de procesamiento independientes (hasta 480 por tarjeta) que permiten obtener una potencia muy elevada, a un coste muy asequible.

Bibliografía

[1] KOLLER D., TRIMBLE J, NAJBJERG T., GELFAND N., LEVOY M., 2005: *Fragments of the City: Stanford's Digital Forma Urbis Romae Project*. Journal of Roman Archaeology (Proc .of the Third Williams Symposium on Classical Architecture).

[2] BROWN B. J., TOLER-FRANKLIN C., NEHAB D., BURNS M., DOBKIN D, VLACHOPOULOS A., DOUMAS C, RUSINKIEWICZ S., WEYRICH T., 2008: *A System for High-Volume Acquisition and Matching of Fresco Fragments: Reassembling Theran Wall Paintings.* ACM Transactions on Graphics (Proc. SIGGRAPH).

[3] NIHSHANKA D., 2007:*Project Report – A Survey of Recent Advances in Shape Matching.* Duke University, Durham, USA.

[4] VELKTKAMP R. C., HAGEDOORN M., 2001: *State-of-the-Art in Shape Matching*. International Conference on Shape Modeling and Applications, pp. 188-197.

[5] VAN KAICK O., ZHAG H., HAMARNEH G., COHEN-OR, D., 2010: *A Survey on Shape Correspondence*. Eurographics STAR Report (2010)

[6] KAZHDAN M., FUNKHOUSER T., RUSINKIEWICZ S., 2004: *Shape matching and anisotropy*. Proc. SIGGRAPH (2004), pp. 623–629.

[7] ZHANG H., SHEFFER A., COHEN-OR D., ZHOU, Q., VAN KAICK O., TAGLIASACCHI A., 2008:*Deformation-drivenshape correspondence*. Computer Graphics Forum (Proceedings SGP), pp. 1431–1439.

[8] ALEXA M., 2002: *Recent advances in mesh morphing*. ComputerGraphics Forum, pp. 173–198.

[9] WARD A. D., HMARNEH G., 2009: *The groupwise medial axis transform for fuzzy skeletonization and pruning.* IEEE PAMI (Aceptado para una publicación futura)

[10] DAVIES R. H., TWINING C. J., COOTES T. F., WATERTON J. C., TAYLOR C. J., 2002: *A minimum description length approach to statistical shape modeling*. IEEE Transactions on Medical Imaging, pp. 525–537.

[11] CASTELLANI U., CRISTANI M., FANTONI S., MURINO V., 2008: *Sparse points matching by combining 3D mesh saliency with statistical descriptors*. Computer Graphics Forum (Proc. EUROGRAPHICS),pp. 643–652.

[12] BIASOTTI S., MARINI S., SPAGNUOLO M., FALCIDIENO B, 2006: *Sub-part correspondence by structural descriptors of 3D shapes.* Computer-Aided, pp. 1002–1019.

[13] RUSINKIEWICZ S., LEVOY M., 2001: *Efficient variants of the ICP algorithm*. Proc. 3rd International Conference on 3D Digital Imaging andModeling, pp. 145–152.

[14] GELFAND N., MITRA N. J., GUIBAS L. J., POTTMANN H., 2005: *Robust global registration*. Proc. Symposium OnGeometric Processing (SGP), pp. 197–206.

[15] AIGER D., MITRA N. J., COHEN-OR D., 2008*: 4-points congruent sets for robust surface registration*. ACM Transactions OnGraphics (Proc. SIGGRAPH), pp. 1–10.

[16] CHANG W., ZWICKER M., 2008: *Automatic registration for articulated shapes. Computer Graphics Forum* (Proc. SGP), pp. 1459–1468.

[17] AU O. K.-C., COHEN-OR D., TAI C.-L., FU H., ZHENG Y., 2010: *Electors voting for fast automatic shape correspondence*.Computer Graphics Forum (Proceedings EUROGRAPHICS).

[18] TURK G., LEVOY M., 1994: *Zippered polygon meshes from range images*. Proc. ACM SIGGRAPH, pp. 311–318.

[19] IRANI S., RAGHAVAN P., 1996: *Combinatorial and experimental results for randomized point matching algorithms*. Proc.Symposium on Computational Geometry, pp. 68–77.

[20] ELAD A., KIMMEL R., 2003: On bending invariant signaturesfor surfaces. IEEE PAMI, pp. 1285–1295.

[21] JAIN V., ZHANG H., VAN KAICK O., 2007:*Non-rigid spectral correspondence of triangle meshes.* International Journal on Shape Modeling, pp. 101–124.

[22] HUANG Q.-X., ADAMS B., WICKE M., GUIBAS L. J., 2008: *Non-rigid registration under isometric deformations*. ComputerGraphics Forum (Proc. SGP), pp. 1449–1457.

[23] ANGUELOV D., SRINIVASAN P., PANG H.-C., KOLLER D., THRUN S., DAVIS J., 2004: *The correlated correspondence algorithm for unsupervised registration of nonrigid surfaces*. NIPS.

[24] PAULY M., MITRA N. J., GIESEN J., GROSS M., 2005: *GUIBAS L. J.: Example-based 3D scan completion*. Proc.Symposiumon Geometric Processing (SGP).

[25] MITRA N. J., FLORY S., OVSJANIKOV M., GELFAND N., GUIBAS L., POTTMANN H., 2007: *Dynamic geometry registration*. Proc. Symposium on Geometric Processing (SGP),pp. 173–182.

[26] WAND M., JENKE P., HUANG Q.-X., BOKELOH M., GUIBAS L., SCHILLING A., 2007: *Reconstruction of deforming geometry from time-varying point clouds*. Proc. Symposium on Geom. Processing (SGP), pp. 49–58.

[27] SHARF A., ALCANTARA D. A., LEWINER T., GREIF C., SHEFFER A., AMENTA N., COHEN-OR D., 2008: *Space-time surface reconstruction using incompressible flow*. ACM Trans. OnGraphics (Proc. SIGGRAPH Asia).

[28] PEKELNY Y., GOTSMAN C., 2008: *Articulated object reconstruction and markerless motion capture from depth video*. ComputerGraphics Forum (Proc. EUROGRAPHICS), pp. 399–408.

[29] DE AGUIAR E., STOLL C., THEOBALT C., AHMED N., SEIDEL H.-P., THRUN S., 2008: *Performance capture from sparse multi-view video*. ACM Trans. on Graphics (Proc. SIGGRAPH).

[30] LI H., ADAMS B., GUIBAS L. J., PAULY M., 2009: *Robust single-view geometry and motion reconstruction*. ACM Trans. OnGraphics (Proc. SIGGRAPH Asia).

[31] GALL J., STOLL C., DE AGUIAR E., THEOBALT C., ROSENHAHN B., SEIDEL H.-P., 2009: *Motion capture using joint skeleton tracking and surface estimation*. Proc. IEEE Conf. on CVPR.

[32] CHANG W., ZWICKER M., 2009: *Range scan registration using reduced deformable models*. Computer Graphics Forum (Proc. EUROGRAPHICS).

[33] TEVS A., BOKELOH M., WAND M., SCHILLING A., SEIDEL H.-P., 2009: *Isometric registration of ambiguous and partial data*. In Proc. IEEE Conf. on CVPR.

[34] ZHENG Q., SHARF A., TAGLIASACCHI A., CHEN B., ZHANG H., SHEFFER A., COHEN-OR D., 2010:*Consensus skeleton for non-rigid space-time registration*. Computer Graphics Forum (Proc. EUROGRAPHICS).

[35] FISCHLER M. A., BOLLES R. C., 1981: *Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography*. Commun ACM, pp. 381–395.

[36] WOLFSON H. J., RIGOUTSOS I., 1997: *Geometric hashing: an overview*. IEEE Computational Science & Engineering, pp. 10–21.

[37] OLSON C. F., 1997: *Efficient pose clustering using a randomized algorithm*. Int. J. Comput. Vision, pp. 131–147.

[38] HUTTENLOCHER D. P., ULLMAN S., 1990: *Recognizing solid objects by alignment with an image*. Int. J. of Computer Vision, pp. 195–212.

[39] ALLEN B., CURLESS B., POPOVI´C Z., 2003: *The space of human body shapes: reconstruction and parameterization from range scans*. ACM Trans. on Graphics (Proc. SIGGRAPH), pp. 587–594.

[40] CORNEA N. D., SILVER D., MIN P., 2007: *Curve-skeleton properties, applications, and algorithms*. IEEE Trans. on Visualizationand Computer Graphics (TVCG), pp. 530–548.

[41] TANG L., HAMARNEH G., 2008: *SMRFI: Shape matching via registration of vector-valued feature images*. In Proc. ComputerVision and Pattern Recognition (CVPR), pp. 1–8.

[42] BUSTOS B., KEIM D. A., SAUPE D., SCHRECK T., VRANI´C D. V., 2005: *Feature-based similarity search in 3D object databases*. ACM Comput. Surv., pp. 345–387.

[43] TANGELDER J. W. H., VELTKAMP R. C., 2008: *A survey of content based 3D shape retrieval methods*. Multimedia Tools andApplications, pp. 441–471.

[44] MIKOLAJCZYK K., SCHMID C., 2005: *A performance evaluation of local descriptors*. IEEE PAMI, pp. 1615–1630.

[45] MACIEL J., COSTEIRA J. P., 2003: *A global solution to sparse correspondence problems*. IEEE PAMI, pp. 187–199.

[46] BERG A. C., BERG T. L., MALIK J., 2005: *Shape matching and object recognition using low distortion correspondences*. InProc. IEEE Conf. on CVPR, pp. 26–33.

[47] LEORDEANU M., HEBERT M., 2005: *A spectral technique for correspondence problems using pairwise constraints*. In Proc. International Conference on Computer Visision (ICCV), vol. 2, pp. 1482–1489.

[48] GUYON I., ELISSEEFF A., 2003: *An introduction to variable and feature selection.* The Journal of Machine Learning Research 3, pp. 1157–1182.

[49] WARD A. D., HAMARNEH G., 2007: *Statistical shape modeling using MDL incorporating shape, appearance, and expert knowledge*. Lecture Notes in Computer Science (Proc. MICCAI), pp. 278–285.

[50] BELONGIE S., MALIK J., PUZICHA J., 2000: *Shape context: A new descriptor for shape matching and object recognition*. InNIPS, pp. 831–837.

[51] KÖRTGEN M., PARK G.-J., NOVOTNI M., KLEIN R., 2003: *3D shape matching with 3D shape contexts*. In Proc. 7thCentral European Seminar on Computer Graphics

[52] JOHNSON A., HEBERT M., 1999: *Using spin-images for efficient multiple model recognition in cluttered 3D scenes*. IEEEPAMI, pp. 433–449.

[53] LI X., GUSKOV I., 2005:*Multi-scale features for approximate alignment of point-based surfaces.* In Proc. Symposium on Geometry Processing (SGP).

[54] FUNKHOUSER T., SHILANE P., 2006: *Partial matching of 3D shapes with priority-driven search.* In Proc. Symposium on Geometric Processing, pp. 131–142.

[55] GATZKE T., GRIMM C., GARLAND M., ZELINKAS., 2005: *Curvature maps for local shape comparison.* In Proc. Conference on Shape Modeling and Applications, pp. 244–253.

[56] MANAY S., CREMERS D., HONG B.-W., YEZZIA. J., SOATTO S., 2006: *Integral invariants for shape matching*. IEEEPAMI, pp. 1602–1618.

[57] GAL R., COHEN-OR D., 2006: *Salient geometric features for partial shape matching and similarity.* ACM Trans. on Graphics, pp. 130–150.

[58] CAETANO T. S., MCAULEY J. J., CHENG L., LE Q. V., SMOLA A. J., 2009: *Learning graph matching*. IEEE PAMI, pp. 1048–1058.

[59] SUMNER R. W., POPOVI´C J., 2004: *Deformation transfer for triangle meshes.* ACM Trans. on Graphics (Proc. SIGGRAPH), pp. 399–405.

[60] HEIMANN T., MEINZER H.-P., 2009: *Statistical shape models for 3D medical image segmentation: A review*. Medical ImageAnalysis, pp. 543–563.

[61] FISCHLER M. A., BOLLES R. C., 1981: *Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography*. Commun. ACM, pp. 381–395.

[62] HUTTENLOCHER D. P., 1981: *Fast affine point matching: an output-sensitive method*. In Proc. IEEE Conference on CVPR,pp. 263–268.

[63] SCHÖLKOPF B., STEINKE F., BLANZ V., 2005: *Object correspondence as a machine learning problem*. In Proc. 22nd International Conference on Machine Learning, pp. 776–783.

[64] LI H., SUMNER R. W., PAULY M., 2008: *Global correspondence optimization for non-rigid registration of depth scans*.Computer Graphics Forum (Proc. SGP) .

[65] ZHENG Y., DOERMANN D., 2006: *Robust point matching for nonrigid shapes by preserving local neighborhood structures.*IEEE PAMI, pp. 643–649.

[66] ZASS R., SHASHUA A., 2008: *Probabilistic graph and hypergraph matching*. In Proc. IEEE Conference on CVPR.

[67] CHUI H., RANGARAJAN A., 2003: *A new point matching algorithm for non-rigid registration*. Computer Vision and Image Understanding, pp. 114–141.

[68] BROWN B. J., RUSINKIEWICZ S., 2007: *Global non-rigid alignment of 3-D scans.* ACM Trans. on Graphics (Proc. SIGGRAPH).

[69] SEBASTIAN T. B., KLEIN P. N., KIMIA B. B., 2004: *Recognition of shapes by editing their shock graphs*. IEEE PAMI, pp. 550–571.

[70] D. COHEN, I. SHUTTERLAND 1973: *Principles of Interactive Computer Graphics.* McGraw-Hill Education, pp. 124-252