UNIVERSIDAD DE ALMERÍA
ESCUELA SUPERIOR DE INGENIERÍA
Ingeniería Informática (Plan 1999)

Consultas Espaciales en Entornos Dinámicos

Autor: Martín García Berenguel
Director: Antonio Leopoldo Corral Liria

Septiembre de 2011

Agradecimientos

En primer lugar quiero agradecer a mi familia porque a ella le debo el poder estar aquí hoy.

También quiero agradecer a mi director de proyecto, Antonio Leopoldo Corral Liria, por la ayuda y el apoyo prestados durante la realización de este proyecto, ya que sin ellos no podría haberlo llevado a cabo.

Agradezco a Manuel Torres Gil, Alfonso José Bosch Arán y José Luis Guzmán Sánchez por haber aceptado ser los miembros del tribunal que evaluará este proyecto.

Por último agradezco a todos los compañeros y profesores que me han ayudado durante estos años en la Universidad de Almería.

Resumen

Una base de datos espacio-temporal (BDET) extiende la tecnología de las bases de datos para que cualquier tipo de entidad móvil pueda ser representada en una base de datos, y provee lenguajes de consulta que permiten realizar consultas acerca de los movimientos de dichas entidades. Las BDET se pueden contemplar desde dos perspectivas: g*estión de localización* y *datos espacio-temporales*. La perspectiva de la gestión de localización considera las posiciones actuales y futuras de un conjunto de objetos móviles. La perspectiva de los datos espacio-temporales se centra en las posiciones pasadas de dichos objetos (histórica). En este proyecto nos centraremos en la perspectiva de la gestión de localización. Aquí se considera el problema de gestionar las posiciones de un conjunto de objetos representados en una base de datos (por ejemplo, una flota de barcos en una región del océano). Se almacena el vector de movimiento de cada objeto, en lugar de almacenar su posición actual. Este vector describe la posición del objeto como una función del tiempo. El método de acceso que utilizaremos en este proyecto para reflejar el carácter dinámico de los objetos es el TPR*-tree, que pertenece a la familia de los R-trees.

Existen muchos tipos de consultas espaciales: encontrar al vecino más próximo, consultas de similitud, consultas de ventana, consultas basadas en el contenido, join espacial, etc. Las consultas espaciales convencionales carecen de sentido en entornos dinámicos, ya que los resultados devueltos pueden ser invalidados conforme los objetos se mueven. En este contexto, las *consultas predictivas* son aquellas que responden a preguntas sobre las posiciones actuales y futuras de los objetos dinámicos. Para poder responder a las consultas predictivas en estos entornos existen dos tipos de consultas: las *consultas parametrizadas en el tiempo* y las *consultas continuas*. Una *consulta parametrizada* en el tiempo devuelve el resultado actual de la consulta en el momento en que la consulta es realizada, el tiempo de expiración (es decir, el tiempo en que, debido al movimiento de los objetos o de la propia consulta, el resultado actual dejará de ser válido), y los objetos que producen la expiración (esto es, los objetos que dejan de o comienzan a pertenecer al resultado de la consulta) Un ejemplo de consulta parametrizada en tiempo podría ser *encontrar todos los barcos bajo la influencia de una tormenta en los próximos 10 minutos*. Una *consulta continua* debe ser evaluada y actualizada constantemente debido a los cambios que se producen tanto en la propia consulta como en los objetos de la base de datos, que son de naturaleza continua. El resultado de una consulta continua está compuesto de pares de la forma <*resultado, intervalo*>, donde cada resultado se devuelve junto con el intervalo de tiempo en que es válido.

En este proyecto implementaremos algoritmos para índices TPR*-tree que den respuesta a algunas de estas consultas espaciales adaptadas a entornos dinámicos, como por ejemplo la *consulta de ventana*, la *consulta de los k vecinos más próximos* o la *consulta de join espacial*. A continuación se realizarán experimentos para evaluar su comportamiento y extraer conclusiones sobre sus posibles aplicaciones.

Índice de contenidos

Capítulo 1. INTRODUCCIÓN 1
1.1 Introducción 1
1.2 Ámbito y motivación 4
1.3 Objetivos 5
1.4 Fases de desarrollo 5
1.5 Temporización 6
1.6 Aportación al alumno 6
Capítulo 2. MÉTODOS DE ACCESO ESPACIO-TEMPORALES. TPR*-TREE 7
2.1 Introducción 7
2.2 Revisión de los métodos de acceso espacio-temporales 8
2.3 La familia de los R-Tree 10
2.3.1 El MBR 11
2.4 El R-Tree 12
2.5 El R*-Tree 14
2.5.1 Principales características 14
2.5.2 Operaciones 15
2.5.2.1 Inserción 15
2.5.2.2 Eliminación 18
2.5.2.3 Búsqueda 18
2.6 El TPR-Tree 19
2.6.1 Principales características 19
2.6.2 Funciones de coste 22
2.6.3 Operaciones 24
2.6.3.1 Inserción 24
2.6.3.2 Eliminación 29
2.7 El TPR*-Tree 29
2.7.1 Principales características 29
2.7.2 Funciones de coste 30
2.7.3 Operaciones 34
2.6.3.1 Inserción 34
2.8 Linear Region Quadtrees almacenados en TPR*-Tree 37
Capítulo 3. CONSULTAS 41
3.1 Introducción 41
3.2 Consultas espaciales 42
3.3. Consultas espacio-temporales 43
3.3.1 Consultas parametrizadas en el tiempo 44
3.3.2 Consultas continuas 47
3.4 Consultas de ventana en entornos dinámicos 47
3.4.1 Consulta de ventana TP 48
3.4.1.1 Versión Depth-First 53
3.4.1.2 Versión Depth-First con ordenación 53
3.4.1.3 Versión Best-First 53
3.4.2 Consulta de ventana continua 54
3.5 Consultas de los k vecinos más próximos en entornos dinámicos 56

3.5.1 Consulta de los k vecinos más próximos TP 56
3.5.2 Consulta de los k vecinos más próximos continua 59
3.6 Consultas de join espacial en entornos dinámicos 62
3.6.1 Consulta de join espacial de intervalo 62
3.6.1.1 Versión Depth-First 62
3.6.1.2 Versión Depth-First con barrido del plano 63
3.6.1.3 Versión Breadth-First 65
3.6.2 Consulta de join espacial TP 66
3.6.2.1 Versión Depth-First 68
3.6.2.2 Versión Depth-First con ordenación 68
3.6.2.3 Versión Best-First 69
3.6.3 Consulta de join espacial continua 69
3.7 Otras consultas espaciales en entornos dinámicos 71
Capítulo 4. RESULTADOS EXPERIMENTALES 73
4.1 Introducción 73
4.2 Consultas de ventana en entornos dinámicos 75
4.2.1 Consulta de ventana TP 75
4.2.2 Consulta de ventana continua 81
4.3 Consultas de los k vecinos más próximos en entornos dinámicos 86
4.3.1 Consulta de los k vecinos más próximos TP 86
4.3.2 Consulta de los k vecinos más próximos continua 92
4.4 Consultas de join en entornos dinámicos 97
4.4.1 Consulta de join de intervalo 97
4.4.2 Consulta de join TP 108
4.4.3 Consulta de join continua 119
4.5 Consulta de semijoin espacial de los k vecinos más próximos continua 129
4.6 Conclusiones de los experimentos 135
Capítulo 5. CONCLUSIONES Y TRABAJOS FUTUROS 139
5.1 Conclusiones 139
5.2 Trabajos Futuros 141
BIBLIOGRAFÍA 143

Índice de figuras

Figura 1.1. Evolución de los métodos de acceso espacio-temporales. 2
Figura 2.1. MBRs de objetos. 11
Figura 2.2. Objetos disjuntos con MBRs solapados. 12
Figura 2.3. MBR de un conjunto de objetos. 12
Figura 2.4. R-tree. 13
Figura 2.5. R*-tree. 15
Figura 2.6. Solape de un MBR. 15
Figura 2.7. Inserción en R*-tree. 17
Figura 2.8. R*-tree tras la inserción. 18
Figura 2.9. Reajuste de un MBR unidimensional. 21
Figura 2.10. MBR de dos objetos TP en los instantes t=0 y t=2. 21
Figura 2.11. Inserción en TPR-tree. 24
Figura 2.12. Reinserción forzada en TPR-tree. 26
Figura 2.13. División en TPR-tree. 29
Figura 2.14. MBR de un rectángulo y su área de barrido. 30
Figura 2.15. Área frente a región de barrido. Las áreas integrales de R y R1 son iguales, mientras que la regiones de barrido son distintas. 32
Figura 2.16. Inserción en TPR*-tree. 35
Figura 2.17. Ejemplo de quadtree. 38
Figura 2.18. Locational code de los bloques correspondientes al quadtree de la figura 2.17. 39
Figura 2.19. FD Quadtree correspondiente a la región de la figura 2.17a mediante TPR*-tree. 39
Figura 3.1. Consulta espacio-temporal. 42
Figura 3.2. Consultas espacio-temporales en un espacio unidimensional. 44
Figura 3.3. Métrica *mindist* entre un punto y una entrada intermedia. 45
Figura 3.4. Círculo de vecindad. 46
Figura 3.5. Métrica mindist entre entradas intermedias. 46
Figura 3.6. Tiempo de influencia de un objeto con respecto a una consulta. 48
Figura 3.7. Tiempo de influencia de una entrada intermedia con respecto a una consulta. 49
Figura 3.8. Cálculo del periodo de intersección entre un objeto y una consulta. 50
Figura 3.9. Calculo del periodo de intersección para una entrada intermedia contenida en una consulta. 51
Figura 3.10. Objetos y sus dos tiempos de influencia respecto de una consulta móvil.. 55
Figura 3.11. Consulta de los k vecinos más próximos en entornos dinámicos. 56
Figura 3.12. Distancia entre una entrada intermedia y una consulta móvil. 57
Figura 3.13. Ejemplo de barrido del plano. 63
Figura 3.14. Tiempos de influencia para una consulta de Join TP. 67
Figura 3.15. Tiempos de influencia para una consulta de Join continua. 70
Figura 4.1. Cuadrante NW de la imagen raster. 74

Capítulo 1

INTRODUCCIÓN

1.1 Introducción

Las bases de datos espacio-temporales (BDET) que manejan grandes volúmenes de objetos dinámicos están cobrando importancia debido a numerosas aplicaciones emergentes (control del tráfico, meteorología, computación móvil, servicios basados en localización o LBS, etc.) y a la enorme difusión de la tecnología GPS integrada en dispositivos móviles de consumo.

Una base de datos espacio-temporal maneja objetos que cambian su posición a lo largo del tiempo. Generalmente los objetos obtienen su posición y velocidad a través de dispositivos de localización tales como GPS y los envían a un servidor de base de datos espacio-temporal que los gestiona siendo capaz de responder preguntas acerca de las posiciones pasadas, presentes y futuras de los objetos.

Las BDET soportan distintos tipos de consultas tales como consultas de ventana o de los k vecinos más próximos. Para responder a estas consultas pueden ser necesarios datos sobre el tiempo pasado, presente o futuro. Se han propuesto un gran número de estructuras de índice espacio-temporales para responder a estas consultas de forma eficiente, y aproximaciones nuevas que se basan en los puntos débiles de las anteriores. Algunas tratan de forma continua los datos pasados, presentes y futuros, otras se han desarrollado para entornos específicos tales como redes de carreteras. La figura1.1 muestra un resumen de los métodos de acceso espacio-temporales hasta 2010 [NAM10]. Las líneas indican la relación de evolución entre los distintos métodos de acceso.

Una base de datos de objetos móviles (BDOM) extiende la tecnología de las bases de datos para que cualquier tipo de entidad móvil pueda ser representada en una base de datos, y provee lenguajes de consulta que permiten realizar consultas acerca de los movimientos de dichas entidades. Las BDOM se pueden contemplar desde dos perspectivas: g*estión de localización* y *datos espacio-temporales* [GuS05].

La perspectiva de la *gestión de localización* se centra en mantener dinámicamente las posiciones de un conjunto de objetos móviles, y ser capaz de responder a consultas

sobre sus posiciones actuales y futuras, así como cualquier relación que pueda darse entre los objetos móviles a lo largo del tiempo.

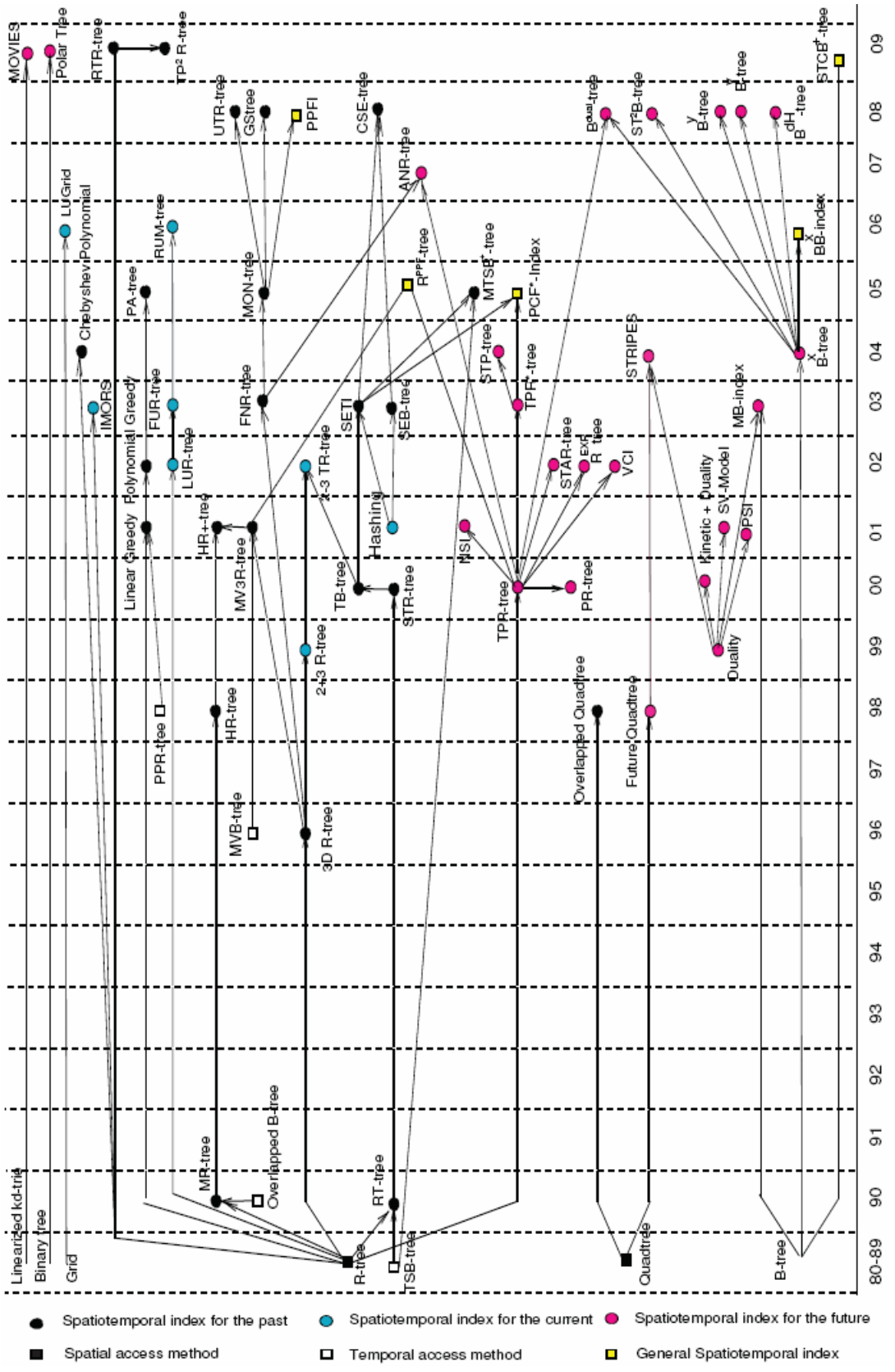


Figura 1.1. Evolución de los métodos de acceso espacio-temporales.

La perspectiva de los *datos espacio-temporales* considera los distintos tipos de objetos que se pueden almacenar en una base de datos espacial (estática) y observa como cambian a lo largo del tiempo. El objetivo es describir tanto el estado actual de los objetos espaciales, como la evolución que han sufrido (en el pasado), y poder devolver el estado de cualquier objeto en cualquier instante pasado.

En este proyecto nos centraremos en la perspectiva de la gestión de localización. Aquí se considera el problema de gestionar las posiciones de un conjunto de objetos representados en una base de datos (por ejemplo, una flota de barcos en una región del océano). Para un instante de tiempo dado no hay problema, podríamos asociar el identificador de cada objeto con los atributos x e y que indiquen su posición. Sin embargo, los objetos están en movimiento y, para mantener la información actualizada, la posición de cada objeto debe ser actualizada frecuentemente. Esto nos conduce a una situación en la que actualizaciones poco frecuentes producirán errores considerables en las posiciones de los objetos, mientras que con actualizaciones frecuentes el error será pequeño, aunque la sobrecarga producida en el sistema hará que esta opción no sea factible en la mayoría de casos prácticos.

Para superar esta dificultad se opta por almacenar el vector de movimiento de cada objeto, en lugar de almacenar su posición actual. Este vector describe la posición del objeto como una función del tiempo. De este modo, conocida la posición y velocidad de un objeto en un instante determinado, podemos calcular la posición del objeto en cualquier instante posterior. Estos vectores de movimiento también deben ser actualizados, pero mucho menos frecuentemente que las posiciones de los objetos.

El método de acceso que utilizaremos en este proyecto para representar los objetos móviles es el TPR*-tree [TPS03], que optimiza al TPR-tree [SJLL00] y como éste pertenece a la familia de los R-trees [Gut84]. El TPR-tree (R-tree parametrizado en el tiempo) es un árbol balanceado en altura similar el R-tree que es capaz de almacenar objetos móviles representando los límites de dichos objetos como funciones lineales cuyo parámetro es el tiempo. El TPR*-tree mejora al TPR-tree, teniendo en cuenta características únicas de los objetos móviles a través de un conjunto de algoritmos de construcción mejorados

Existen muchos tipos de consultas espaciales: consulta del vecino más próximo, consultas de similitud, consultas de ventana, consultas basadas en el contenido, join espacial, etc. En este proyecto, haremos uso de las consultas de ventana, de los k vecinos más próximos y del join espacial basado en el operador solape entre TPR*-trees. Por ejemplo, un TPR*-tree podría contener objetos que representen barcos moviéndose y el otro TPR*-tree indexaría los datos (de una imagen) que representan una tormenta que se desplaza en el tiempo. El usuario podría querer consultar qué barcos se encontrarán bajo la tormenta en el futuro, sabiendo que ambos conjuntos de objetos se mueven con el tiempo.

Las *consultas predictivas* son aquellas que responden a preguntas sobre las posiciones actuales y futuras de los objetos móviles. Una consulta predictiva especifica una condición espacial y un intervalo de tiempo futuro, y devuelve el conjunto de objetos que satisfacen la condición espacial en algún instante del intervalo. Por ejemplo, *encontrar todos los barcos bajo la influencia de una tormenta en los próximos 10 minutos*. Si el intervalo corresponde a un único instante futuro, entonces la consulta es llamada consulta instantánea, si no es una consulta de intervalo [Cam07, CTVM08].

Las consultas espaciales convencionales carecen de sentido en entornos dinámicos, ya que los resultados devueltos pueden ser invalidados conforme los objetos se mueven, y por tanto no pueden ser usadas como consultas predictivas. Para poder responder a las consultas predictivas en estos entornos existen dos tipos de consultas: las *consultas parametrizadas en el tiempo* y las *consultas continuas* [TP03].

Una *consulta parametrizada* en el tiempo devuelve el resultado actual de la consulta en el momento en que la consulta es realizada, el tiempo de expiración (es decir, el tiempo en que, debido al movimiento de los objetos o de la propia consulta, el resultado actual dejará de ser válido), y los objetos que producen la expiración (esto es, los objetos que dejan de o comienzan a pertenecer al resultado de la consulta). Un ejemplo de respuesta para la versión parametrizada en tiempo de la consulta *encontrar todos los barcos bajo la influencia de una tormenta en los próximos 10 minutos* podría ser de la forma <{A, B, C}, 5, A>, indicando que en el momento actual los barcos A, B y C se encuentran bajo la influencia de la tormenta y que esta situación será válida hasta dentro de 5 minutos, en que el resultado cambiará porque A dejará de estar bajo la influencia de la tormenta. Las consultas parametrizadas en el tiempo son importantes como métodos independientes y además son las componentes primitivas sobre las que se pueden construir consultas continuas más complejas.

Una *consulta continua* debe ser evaluada y actualizada constantemente debido a los cambios que se producen tanto en la propia consulta como en los objetos de la base de datos, que son de naturaleza continua. El resultado de una consulta continua está compuesto de tuplas de la forma <resultado, intervalo>, donde cada *resultado* se devuelve junto con el *intervalo* de tiempo en que es válido. Un ejemplo de respuesta para la versión continua de la consulta *encontrar todos los barcos bajo la influencia de una tormenta en los próximos 10 minutos* podría ser de la forma {<{A, B, C}, [0, 5)>, <{B, C}, [5, 7)>, <{B, D}, [7, 10)>}, indicando que los barcos A, B y C se encontrarán bajo la influencia de la tormenta en el intervalo [0, 5), los barcos B y C se encontrarán bajo la influencia de la tormenta en el intervalo [5, 7) y los barcos B y D se encontrarán bajo la influencia de la tormenta en el intervalo [7, 10).

En resumen, el ámbito de este proyecto se centra en el estudio de consultas espaciales continuas (joins y vecinos más próximos) en entornos dinámicos representados los objetos móviles mediante TPR*-trees (información raster para las imágenes y vectorial para los objetos móviles representados por puntos) sobre datos reales y sintéticos.

1.2 Ámbito y motivación

Las bases de datos de objetos móviles que manejan grandes volúmenes de objetos dinámicos están cobrando importancia debido a numerosas aplicaciones emergentes (control del tráfico, meteorología, computación móvil, servicios basados en localización o LBS, etc.). Estos sistemas se pueden clasificar en dos grandes categorías, dependiendo de si devuelven información del pasado o predicciones sobre el futuro. Este proyecto se centra en la segunda categoría.

La aplicación de los resultados de este proyecto a un SIG (Sistema de Información Geográfica) es inmediata, por las características de los datos que éste es capaz de gestionar. Un SIG es un sistema integrado compuesto por hardware, software, personal, información geográfica y algoritmos, que permite y facilita la recuperación, el análisis, gestión o representación de datos espaciales y de objetos móviles. En particular, el

apoyo que este proyecto podría aportar a un SIG en aspectos como: (a) *localización*, proporcionando información sobre las características de un lugar concreto en un instante o en un intervalo de tiempo dados, y (b) las posiciones de los objetos móviles en instantes o intervalos de tiempo futuros como resultado de una *consulta predictiva*.

1.3 Objetivos

- Estudiar en profundidad el TPR* y conocer los distintos algoritmos de consultas espaciales en entornos dinámicos [JaS07], [TP03] entre TPR*-trees [Cam07].

- Extender el TPR*-tree para almacenar regiones móviles en un contexto dinámico.

- Estudiar e implementar las consultas TP-WQ (consulta de ventana parametrizada en el tiempo) y CWQ (consulta de ventana continua) sobre TPR*-trees [TP03].

- Estudiar e implementar las consultas TP-KNNQ (k vecinos más próximos parametrizada en el tiempo) [TP02] y CNNQ (vecino más próximo continua) [TPS02] sobre TPR*-trees.

- Estudiar e implementar consultas TP-SJ (consultas de join espacial parametrizadas en el tiempo) y CSJ (consultas de join espacial continuo) entre dos TPR*-trees [ZLRB08], donde uno indexa un conjunto de objetos (puntos) móviles y el otro una región móvil.

- Extender las consultas CNNQ para obtener consultas del tipo “semi-join KNN continuo” donde se combinan dos TPR*-trees de forma similar a las CSJ del punto anterior.

- Obtener experimentalmente resultados de todas las nuevas implementaciones, extrayendo conclusiones apropiadas de los mismos.

- Obtener conclusiones a partir de los resultados experimentales y proponer trabajos futuros.

- Desarrollar una documentación que recoja el trabajo realizado durante la ejecución de este proyecto (memoria del PFC).

1.4 Fases de desarrollo

- En una primera fase se recopilará la documentación bibliográfica, para su estudio y un mejor entendimiento del núcleo de proyecto.

- Durante la segunda fase se estudiarán e implementarán en C++ el TPR*-Tree, estructura de datos base del proyecto.

- Durante la tercera fase se estudiarán e implementarán en C++ las consultas TP-WQ, CWQ, TP-KNNQ, CNNQ, TP-SJ y CSJ en TPR*-Trees.

- En una cuarta fase se diseñarán algoritmos para responder consultas del tipo "semi-join KNN continuo". A continuación, implementar en C++ dichos algoritmos y comprobar su correcto funcionamiento.

- En la quinta fase se diseñarán y realizarán una serie de experimentos con datos reales y sintéticos (siguiendo diferentes distribuciones). Con los resultados de dichos experimentos se obtendrán conclusiones que nos permitan conocer el comportamiento de los algoritmos propuestos, así como su idoneidad para los diferentes problemas prácticos a los que se puedan aplicar.

- La última fase se corresponderá con el desarrollo de la documentación.

1.5 Temporización

- Estudio de los métodos de acceso: 2 meses.

- Implementación del método de acceso TPR*-tree: 1 mes.

- Estudio e implementación de los algoritmos de consulta: 2.5 meses.

- Diseño e implementación de algoritmos del tipo semi-join KNN continuo: 0.5 meses.

- Experimentación (diseño, realización y organización de los datos): 1 mes.

- Documentación (memoria y presentación): 2 meses.

1.6 Aportación al alumno

La realización de este proyecto me ha permitido adentrarme en el estudio de la organización de datos espaciales y espaciotemporales, así como de las consultas sobre estos, una amplia e interesante área con muchas aplicaciones.

También me ha ayudado mejorar a nivel de programación, tanto por la implementación de los diferentes algoritmos como por el estudio y modificación de códigos de otros autores, con la dificultad que ello conlleva por el hecho de la falta de documentación, así como la necesidad de seguir la secuencia del código.

La realización de este proyecto monográfico me ha dado la oportuidad de acercarme al mundo de la investigación de estructuras de datos complejas y del procesamiento de consultas sobre ellas, cuya dificultad no es trivial.

Por último la realización de esta memoria me ha permitido conocer algo más el proceso de documentación de un proyecto tanto a nivel técnico como conceptual.

Capítulo 2

MÉTODOS DE ACCESO ESPACIO-TEMPORALES. TPR*-TREE

2.1 Introducción

Las bases de datos espacio-temporales que manejan grandes volúmenes de objetos dinámicos están cobrando importancia debido a numerosas aplicaciones emergentes (control del tráfico, meteorología, computación móvil, servicios basados en localización o LBS (Location-Based Services), etc.) y a la enorme difusión de la tecnología GPS integrada en dispositivos móviles de consumo.

Las bases de datos relacionales a menudo no disponen de la tecnología requerida para manejar datos complejos como los datos espacio-temporales. Al contrario que las aplicaciones de bases de datos tradicionales, las aplicaciones espacio-temporales requieren trabajar con datos complejos como puntos, líneas y polígonos en un entorno dinámico. Las operaciones con estos tipos suelen ser complejas si las comparamos con operaciones sobre tipos simples. Por tanto las estructuras de índice en las que se basan los sistemas relacionales deben ser extendidas para facilitar el almacenamiento y posterior recuperación de datos espacio-temporales.

Una base de datos de objetos móviles (BDOM) extiende la tecnología de las bases de datos para que cualquier tipo de entidad móvil pueda ser representada en una base de datos, y provee lenguajes de consulta que permiten realizar consultas acerca de los movimientos de dichas entidades. Las BDOM se pueden contemplar desde dos perspectivas: g*estión de localización* y *datos espacio-temporales* [GuS05].

Por lo tanto, para que una BDOM pueda cumplir sus requisitos es necesario definir los tipos de datos, los lenguajes de consulta y los métodos de acceso espacio-temporales. Dentro de este contexto los métodos de acceso espacio-temporales son los encargados de posibilitar la indexación espacio-temporal que permita la organización y el almacenamiento de los datos en memoria secundaria, así como el desarrollo de algoritmos para su manipulación de una manera eficiente. En este sentido el TPR*-tree (que es el método de acceso utilizado en este proyecto) permite almacenar datos espacio-temporales reflejando el carácter dinámico de estos y proporcionando una

estructura que puede ser aprovechada para mejorar el rendimiento de las operaciones realizadas sobre ellos.

El resto de este capítulo se organiza de la siguiente manera: en la sección 2.2 se realizará un repaso de los métodos de acceso espacio-temporales más relevantes, en la sección 2.3 se presenta la familia de los R-trees, en las secciones 2.4, 2.5, 2.6 y 2.7 se presentan los métodos de acceso R-tree, R*-tree, TPR-tree y TPR*-tree respectivamente. Por último, en la sección 2.8 se presenta el Linear Region Quadtree como método para almacenar una imagen raster en un TPR*-tree.

2.2 Revisión de los métodos de acceso espacio-temporales

La perspectiva de los *datos espacio-temporales* considera los distintos tipos de objetos que se pueden almacenar en una base de datos espacial (estática) y observa como cambian a lo largo del tiempo. El objetivo es describir tanto el estado actual de los objetos espaciales, como la evolución que han sufrido (en el pasado), y poder devolver el estado de cualquier objeto en cualquier instante pasado.

Una rama de la investigación sobre indexación histórica se basa en el solapamiento de índices. La idea consiste en construir un índice separado para cada instancia de tiempo, mientras que se permite que las ramas que no se modifican se comparten a lo largo del tiempo. Xu y otros propusieron el MR-Tree, que fue la primera estructura de datos solapada para indexación histórica de datos espacio-temporales [XHL90]. Nascimento y Silva continuaron posteriormente con el MR-Tree implementando detalles e introdujeron el Historical R-Tree [NS98]. Simultáneamente Tzouramanis y otros aplicaron una idea similar a su Quad-Tree y propusieron el Overlapped Quad-Tree [TVM98].

El solapamiento de índices alcanza buen rendimiento para consultas sobre intervalos de tiempo pequeños, mientras que degenera cuando se realizan consultas sobre instantes de tiempo específicos. Sin embargo, las consultas sobre intervalos de tiempo a menudo requieren consultar múltiples índices con datos duplicados, lo que causa un gran impacto sobre el rendimiento. Por otro lado, el 3D R-Tree [TVS96], que trata el tiempo como una tercera dimensión e indexa las trayectorias de los objetos como segmentos, ofrece un buen rendimiento en consultas sobre intervalos de tiempo, pero se queda atrás al manejar consultas sobre intervalos de tiempo pequeños a causa de la gran cantidad de espacio muerto en los rectángulos envolventes al indexar segmentos. El 3D R-Tree inició una rama de investigación en la que las trayectorias de los objetos son almacenadas como segmentos 3D. Pfoser y otros propusieron el STR-Tree, que es un 3D R-Tree que almacena tanto las trayectorias como la proximidad espacial [PJT00]. El STR-Tree almacena las trayectorias modificando el algoritmo de inserción del R-Tree, haciéndolo capaz de agrupar los segmentos que pertenecen a la misma trayectoria. Pfoser y otros también introdujeron el TB-Tree [PJT00], que al contrario que el STR-Tree, otorga preferencia absoluta al almacenamiento de la trayectoria. El TB-Tree solo inserta segmentos en un nodo hoja si este contiene segmentos anteriores de la misma trayectoria, y por lo tanto segmentos cercanos en el espacio deberán ser almacenados de forma separada.

En general, los índices solapados son peores para consultas sobre intervalos de tiempo largos, mientras que los índices de trayectorias de alta dimensión son peores para consultas sobre instantes de tiempo. Movido por esta observación, Tao y Papadias

propusieron una estructura híbrida, el MV3R-Tree [TP01], que usa un R-Tree multiversionado para procesar consultas sobre instantes de tiempo y un 3D R-Tree para procesar consultas sobre intervalos de tiempo mayores. MV3R-Tree es capaz de responder eficientemente a consultas sobre instantes e intervalos mayores de tiempo, pero requiere espacio de almacenamiento extra para el 3D R-Tree auxiliar y tiene un mantenimiento más costoso.

La perspectiva de la *gestión de localización* se centra en mantener dinámicamente las posiciones de un conjunto de objetos móviles, y ser capaz de responder a consultas sobre sus posiciones actuales y futuras, así como cualquier relación que pueda darse entre los objetos móviles a lo largo del tiempo.

La indexación de la posición actual y futura de objetos móviles requiere almacenar información sobre el movimiento actual del objeto. La aproximación más común consiste en representar el movimiento como función lineal del tiempo. Supongamos que $X_r = (x_1, x_2, \ldots, x_d)$ y $V_r = (x.v_1, x.v_2, \ldots, x.v_d)$ son respectivamente los vectores posición y velocidad de un objeto x en el instante t_r. La posición del objeto x en el instante t, con $t \geq t_r$, puede ser calculada como $x(t) = X_r + V_r(t - t_r)$. Esta representación de un objeto móvil coincide con la ecuación paramétrica de una recta en el dominio del tiempo y el espacio, lo que fue reflejado en trabajos previos sobre índices predictivos. Tayeb y otros propusieron el PMR Quad-Tree [TUW98] para objetos móviles, donde la idea es particionar la dimensión temporal, que es infinita, en sesiones de igual duración, e indexar los objetos móviles como segmentos que se extienden a lo largo de la sesión. Por tanto, el índice deberá ser destruido y reconstruido cuando la sesión activa expire.

El PMR Quad-Tree para objetos móviles es un índice espacial para en el dominio del tiempo y el espacio donde los objetos a indexar son segmentos. Una alternativa es transformar la trayectoria actual y futura del objeto de una línea en el dominio del tiempo y el espacio en un punto en otro espacio. Kollios y otros propusieron que la transformada de Hugh [KGT99] se puede aplicar a la función de movimiento de un objeto, transformando una línea en el dominio tiempo-espacio en un punto en un espacio dual. Los puntos en el espacio dual se pueden indexar eficientemente usando un índice espacial basado en el KD-Tree. La transformada de Hugh usada por Kollios sólo es aplicable a funciones de movimiento unidimensionales. Agarwal y otros propusieron que las funciones de movimiento bidimensionales se podrían descomponer en dos funciones unidimensionales en los planos (x, t) e (y, t) [AAE00]. La transformada de Hugh puede entonces ser aplicada a cada trayectoria.

Los métodos basados en transformadas mostraron tener buen rendimiento asintótico, sin embargo estas estructuras de datos no son prácticas debido a grandes constantes ocultas [TPS03]. Saltenis y otros propusieron el TPR-Tree [SJLL00], donde objetos móviles son almacenados eficientemente en su espacio primario. El TPR-Tree es una estructura basada en el R-Tree que adapta el algoritmo de construcción del R*-Tree para objetos móviles. La idea es expresar tanto los objetos como sus rectángulos envolventes (MBRs) como función del tiempo, y por tanto permitiendo realizar consultas sobre este índice de la misma manera que sobre un R-Tree.

El TPR-Tree es un índice predictivo muy importante. Muchas estructuras de datos han surgido basándose en este concepto. Prabhakar y otros propusieron la técnica de Indexación con Velocidad Constante (VCI) [PXKAH02], donde en lugar de almacenar la velocidad exacta del objeto, como en el TPR-Tree, se almacena la máxima velocidad

alcanzable por el objeto. VCI permite responder correctamente consultas incluso si el objeto se desvía de su trayectoria original, reduciendo el número de actualizaciones requeridas, con el coste de aumentar el número de falsos positivos. Procopiuc y otros introdujeron el STAR-Tree [PAH02], donde los MBRs se expresan como funciones por partes. STAR-Tree también es capaz de autoajustar sus MBRs cuando el rendimiento de las consultas baja de un umbral establecido. El R^{exp}-Tree introduce un nuevo tipo de rectángulos envolventes para objetos con tiempo de expiración [SJ01]. Kollios y otros [KGT99] establecen el límite inferior para el coste de una consulta de ventana predictiva (usando un espacio lineal o no lineal) y diseña varios índices casi óptimos para objetos unidimensionales. Tao y otros propusieron el TPR*-Tree [TPS03] y demostraron que su rendimiento en consultas se acerca mucho al del índice teóricamente óptimo. Finalmente, debemos destacar que en [MGA03] y [NAM10] se hace un repaso más exhaustivo de los métodos de acceso espaciotemporales surgidos en los últimos años. Además, en [CJL08] se presenta una propuesta de benchmarking para evaluar el comportamiento de distintos índices para posiciones presentes y de un futuro cercano (*gestión de localización*) que tiene en cuenta diversas méticas de rendimiento.

En este proyecto nos centraremos en la perspectiva de la gestión de localización. Aquí se considera el problema de gestionar las posiciones de un conjunto de objetos representados en una base de datos (por ejemplo, una flota de barcos en una región del océano). Para un instante de tiempo dado no hay problema, podríamos asociar el identificador de cada objeto con los atributos x e y que indiquen su posición. Sin embargo, los objetos están en movimiento y, para mantener la información actualizada, la posición de cada objeto debe ser actualizada frecuentemente. Esto nos conduce a una situación en la que actualizaciones poco frecuentes producirán errores considerables en las posiciones de los objetos, mientras que con actualizaciones frecuentes el error será pequeño, aunque la sobrecarga producida en el sistema hará que esta opción no sea factible en la mayoría de casos prácticos.

Para superar esta dificultad se opta por almacenar el vector de movimiento de cada objeto, en lugar de almacenar su posición actual. Este vector describe la posición del objeto como una función del tiempo. De este modo, conocida la posición y velocidad de un objeto en un instante determinado, podemos calcular la posición del objeto en cualquier instante posterior. Estos vectores de movimiento también deben ser actualizados, pero mucho menos frecuentemente que las posiciones de los objetos.

El método de acceso que utilizaremos en este proyecto para representar los objetos móviles es el TPR*-tree, que mejora al TPR-tree y como éste pertenece a la familia de los R-trees [Gut84]. El TPR-tree (R-tree parametrizado en el tiempo) es un árbol balanceado en altura similar el R-tree que es capaz de almacenar objetos móviles representándolos como funciones lineales cuyo parámetro es el tiempo. El TPR*-tree mejora al TPR-tree, teniendo en cuenta características únicas de los objetos móviles a través de un conjunto de algoritmos de construcción mejorados.

2.3 La familia de los R-Tree

Diversos métodos de acceso han sido desarrollados para gestionar grandes volúmenes de datos multidimensionales de forma eficiente. Dentro de ellos, la familia de los R-trees son una buena opción para indexar diferentes tipos de datos espaciales tales como puntos, segmentos y polígonos y ya han sido adoptadas por sistemas comerciales como Oracle, Informix y PostgreSQL [MNP06].

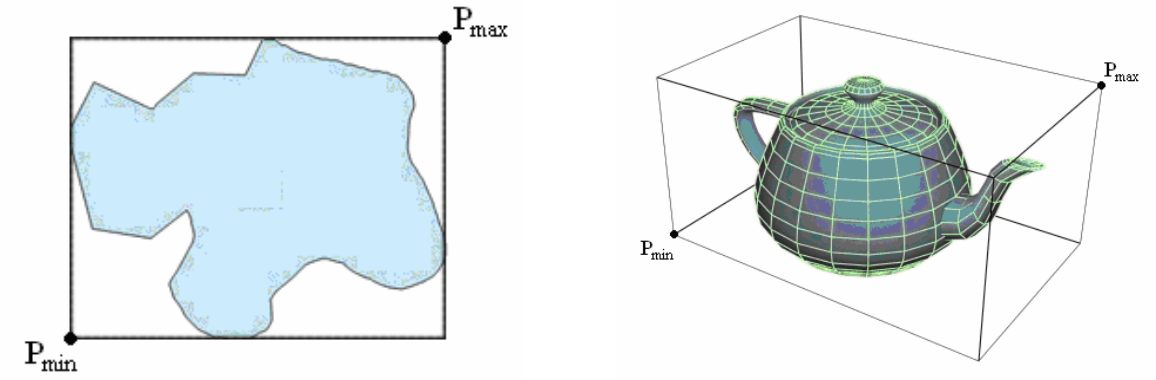
Los miembros de la familia de los R-trees son estructuras de datos arbóreas multidimensionales basadas en MBRs y balanceadas en altura. Se utilizan para organizar dinámicamente conjuntos de objetos multidimensionales (como datos espaciales) representados por rectángulos mínimos envolventes multidimensionales (MBRs). La estructura de datos se construye mediante divisiones recursivas de este espacio multidimensional, hasta obtener subespacios (mutuamente disjuntos o no) con un número de objetos tal que permita poder almacenarlas en una página de datos.

2.3.1 El MBR

Los datos espaciales tienen una estructura compleja. Un objeto espacial puede estar compuesto por un único punto o por varios miles de polígonos, distribuidos arbitrariamente en el espacio. Es por tanto necesario un método de representación que proporcione una buena aproximación de la geometría del objeto, pero también que implique un coste de memoria reducido y un procesamiento eficiente. Teniendo en cuenta estas premisas se opta por representar un objeto espacial a través del rectángulo mínimo envolvente o MBR (Minimum Bounding Rectangle) que lo cubre.

El MBR que cubre a un objeto es el menor rectángulo que contiene al objeto y cuyas caras son paralelas a los ejes de coordenadas (figura 2.1).

Más formalmente, en un espacio d-dimensional, el MBR de un objeto es un hiper-rectángulo d-dimensional delimitado por dos puntos d-dimensionales de los cuales uno está formado por las d coordenadas mínimas del objeto y el otro por las d coordenadas máximas del objeto (de manera que estos dos puntos son los extremos de la diagonal del hiper-rectángulo) [Cor02].



(a) Espacio bidimensional **(b) Espacio tridimensional**

Figura 2.1. MBRs de objetos.

El MBR aproxima la geometría del objeto que cubre. Por tanto, salvo casos particulares, la extensión del MBR es mayor que la extensión del objeto. Es entonces posible que objetos disjuntos estén cubiertos por MBRs que se solapen (figura 2.2).

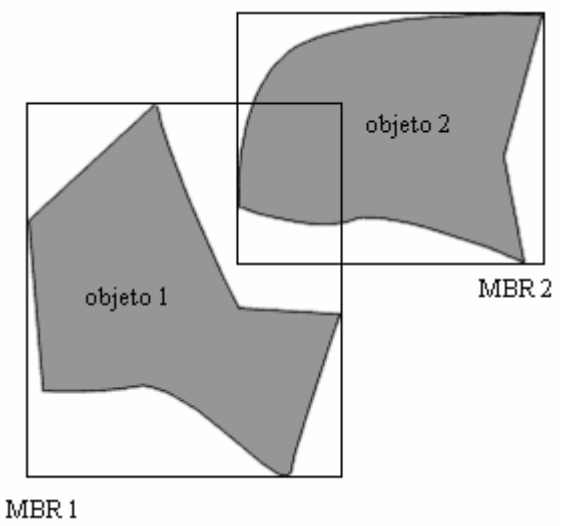


Figura 2.2. Objetos disjuntos con MBRs solapados.

El MBR descrito puede ser extendido para representar conjuntos de objetos. En este caso, los dos puntos d-dimensionales que definen al MBR estarían formados por las d coordenadas mínimas de todos los objetos del conjunto y por las d coordenadas máximas de todos los objetos del conjunto respectivamente (figura 2.3).

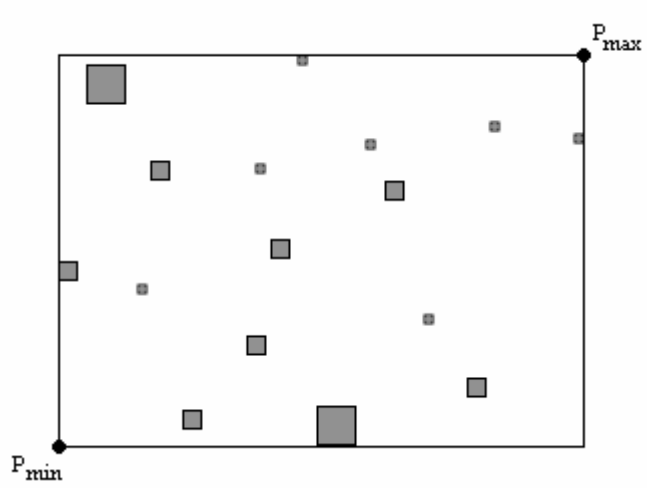


Figura 2.3. MBR de un conjunto de objetos.

2.4 El R-Tree

El R-tree [Gut84] es una estructura de datos arbórea multidimensional basada en MBRs, balanceada en altura y almacenada en disco, derivada del B^+-tree, que divide el espacio de manera recursiva en conjuntos de MBRs, los cuales se pueden solapar en un mismo nivel (figura 2.4).

El R-tree está compuesto por nodos, de los cuales existen dos tipos:

Nodos hoja. Cada entrada de un nodo hoja contiene un MBR que cubre a un objeto almacenado en la base de datos y un puntero al registro donde se encuentra la información más precisa acerca de la geometría espacial exacta del objeto. Los nodos hoja ocupan el nivel más bajo del árbol.

Nodos internos. Cada entrada de un nodo interno contiene un MBR que cubre el conjunto de MBRs de un nodo hijo y un puntero hacia dicho nodo.

Un R-tree cumple las propiedades siguientes:

- Todos los nodos, a excepción del nodo raíz, deben contener entre m y M entradas. El limite inferior m evita la degradación del árbol. El limite superior M se puede elegir de manera que el tamaño de un nodo coincida con el tamaño de una página de datos.

- El nodo raíz debe tener al menos 2 entradas, a menos que sea un nodo hoja.

- Todos los nodos hoja se encuentran al mismo nivel (balanceado en altura).

Nótese que en un R-tree los MBRs que están en nodos internos pueden solaparse, es decir, MBRs en diferentes entradas de un nodo pueden tener un área de solape común (figura 2.4). Éste es un inconveniente importante del R-tree, puesto que si hay un elevado índice de solape entre los MBRs que forman el R-tree puede producir un bajo rendimiento en el procesamiento de consultas, ya que es necesario visitar varios nodos para determinar si un objeto determinado está en el árbol.

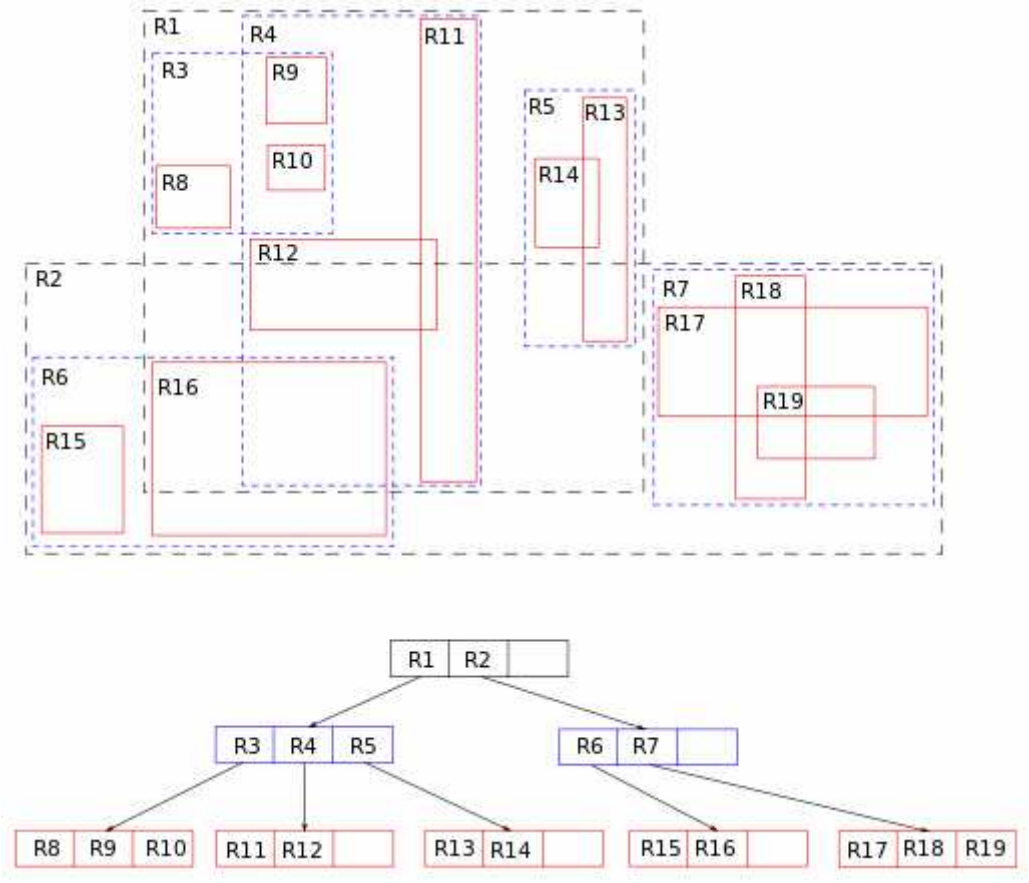


Figura 2.4. R-tree.

Inserción en el R-tree. La inserción en un R-tree se realiza a nivel de hojas y es parecida a la del B^+-tree. Para insertar un objeto (su MBR) en un R-tree se recorre el árbol desde el nodo raíz hasta el nivel de hojas buscando en cada paso el nodo cuyo

MBR requiera un menor incremento de su extensión. Una vez localizado el nodo hoja en el que se debe insertar el nuevo objeto se debe comprobar si la inserción producirá un desbordamiento (overflow), es decir, si el nodo ya contenía el número máximo de entradas M permitido. En caso de que no se produzca desbordamiento se inserta el objeto en el nodo y se reajusta su MBR de modo que siga cubriendo a todos los objetos que contiene. Este reajuste se propaga hacia los niveles superiores hasta el nodo raíz si es necesario. En caso de que se produzca desbordamiento se procede a dividir el nodo de manera que las M+1 entradas queden repartidas entre dos nodos. El criterio de selección de ambos conjuntos de nodos tratará de reducir la probabilidad de que en búsquedas futuras se visiten estos nuevos nodos.

Eliminación en el R-tree. La eliminación de un objeto comienza con la búsqueda del mismo. Una vez localizado el nodo que lo contiene se procede a comprobar si la eliminación producirá una insuficiencia (underflow), es decir, si el nodo ya contenía el mínimo número de entradas permitido m. En caso de no producirse insuficiencia se elimina la entrada correspondiente al objeto y se reajusta el MBR del nodo si es posible, propagando el reajuste hacia niveles superiores mientras sea necesario. En caso de producirse insuficiencia se procede a eliminar el nodo y reinsertar todas sus entradas en el índice.

Búsqueda en el R-tree. La búsqueda en el R-tree es similar a la del B^+-tree. Comenzando en el nivel más alto se desciende hasta el nivel de hojas explorando los subárboles cuyos MBRs solapen con el MBR del objeto buscado. El principal inconveniente es que al estar permitido el solape entre MBRs de nodos de un mismo nivel es posible tener que explorar más de un camino que no nos conducirán a la solución. Por tanto cuanto mayor sea el solape entre los MBRs menor será el rendimiento para consultas del índice.

2.5 El R*-Tree

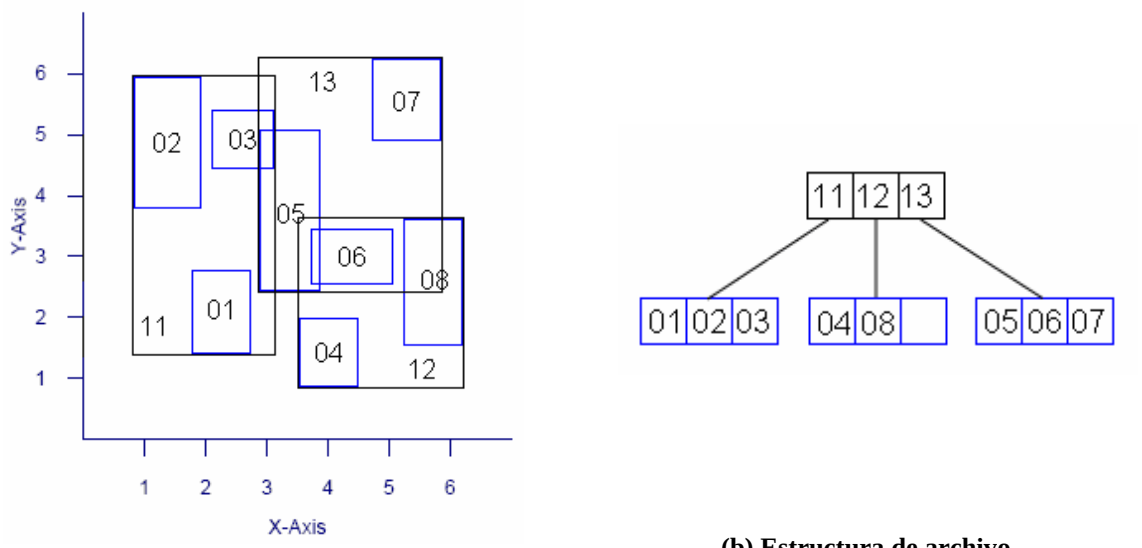
Los sucesivos miembros de la familia de los R-trees han surgido como mejoras o extensiones de los índices anteriores. Una de estas mejoras es el R*-tree.

2.5.1 Principales características

El R*-tree [BKSS90] es un índice derivado del R-tree y como tal mantiene su estructura, es decir, es un índice arbóreo multidimensional balanceado en altura y basado en MBRs. Al igual que el R-tree, está formado por nodos, que pueden ser nodos hoja o nodos internos. Los nodos tienen entre m y M entradas y el MBR asociado a un nodo cubre los MBRs de todos sus descendientes. También cumple el resto de propiedades del R-tree. En la figura 2.5 podemos ver un ejemplo de R*-tree.

Como se indica en la sección anterior, la distribución espacial de los MBRs y en especial el solape entre MBRs de un mismo nivel tiene una influencia directa en el rendimiento del R-tree para operaciones de consulta.

El solape de un MBR R asociado a una entrada de un nodo se define como la suma de las áreas solapadas de R con los MBRs asociados as las demás entradas del nodo (figura 2.6).



(a) Estructura de MBRs **(b) Estructura de archivo**

Figura 2.5. R*-tree.

Cuanto mayor es el solape, mayor es la probabilidad de que en una búsqueda se deban explorar caminos que no conducen a la solución. Por tanto los diseñadores del R*-tree tuvieron como objetivo reducir el solape entre los MBRs de un mismo nivel, evitando la exploración de estos caminos. Para ello diseñaron un nuevo algoritmo de inserción para el R*-tree.

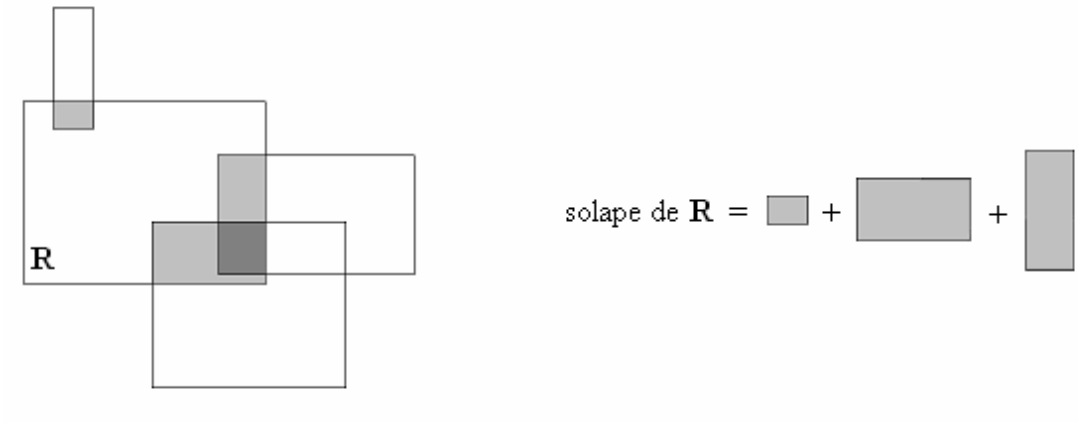


Figura 2.6. Solape de un MBR.

2.5.2 Operaciones

2.5.2.1 Inserción

El algoritmo de inserción del R*-tree trata de minimizar las siguientes métricas de coste: (i) el área y (ii) el perímetro de cada MBR, (iii) el solapamiento entre dos MBRs en el mismo nodo y (iv) la distancia entre el centroide de un MBR y el del nodo que lo contiene. La minimización de estas métricas produce un árbol más compacto y reduce la probabilidad de que un MBR interseque con la región de consulta. Además introduce un mecanismo para que la estructura del árbol sea menos dependiente del orden de

inserción de los objetos. A continuación se describe de forma general el algoritmo de inserción del R*-tree.

Dado un nuevo objeto a insertar, el algoritmo de inserción decide, en cada nivel del árbol, la rama por la que continuar. Partiendo del nivel de nodo raíz y aplicándose de forma recursiva hasta alcanzar el nivel 1 (las hojas están en el nivel 0).

Si el nivel del nodo es superior a 1, entonces se selecciona la entrada cuyo MBR necesita un menor incremento en su área para contener al nuevo objeto. En caso de empate se elige el MBR con menor área.

Si el nivel del nodo es 1, entonces se escoge la entrada cuyo MBR necesite un menor incremento de solape para contener el nuevo objeto. En caso de empate seleccionar el MBR que necesite menor incremento de su área para contener al nuevo objeto. Nótese que se consideran diferentes métricas entre el nivel 1 y los nodos superiores.

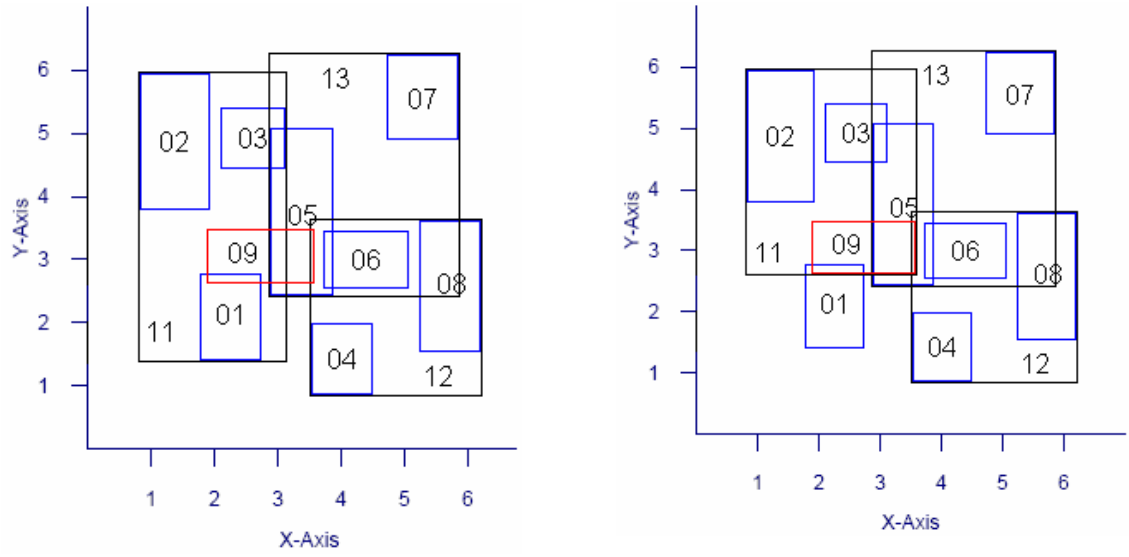
Una vez seleccionado el nodo que contendrá al nuevo objeto insertado, se procede a comprobar si se produce un desbordamiento (el nodo hoja seleccionado ya tiene el máximo de entradas permitidas M). En caso de no haber desbordamiento se inserta el objeto en el nodo y se ajustan los MBRs propagando el reajuste hacia los noveles superiores. En caso de producirse desbordamiento se comprueba si es la primera vez que se produce para este nodo.

Si es la primera vez que ocurre un desbordamiento para el nodo se procederá a reinsertar parte de sus entradas en el árbol. Si no es la primera vez se procede a la división del nodo.

Reinserción forzada. Éste es el mecanismo que ayuda a la construcción de una estructura más compacta, haciendo que esta sea menos dependiente del orden de inserción de los objetos. Se intenta eliminar y reinsertar una parte de las entradas del nodo, evitando una división si alguna entrada pudiera ser asignada a otros nodos. Las M+1 entradas (las M del nodo más la nueva) se ordenan de acuerdo a las distancias entre los centros de sus MBRs y el centro del MBR que cubre al nodo. El conjunto de entradas que se reinsertan son aquellas cuyas distancias al centro están en el 30% mayor, en orden creciente de distancia.

División. La división de un nodo se realiza si el desbordamiento persiste tras la reinserción, dando como resultado dos nuevos nodos con al menos m entradas cada uno. El primer paso elige un eje de división, el que tenga el menor perímetro general, que calcula como se indica a continuación. En cada eje, el algoritmo ordena las entradas por el valor de la coordenada de su límite inferior. Entonces considera cada agrupación de las entradas así ordenadas que asegura que cada nodo tenga al menos m entradas, en total se consideran M - 2m + 2 agrupaciones distintas. Finalmente el perímetro general del eje es igual a la suma de todos los perímetros de las agrupaciones obtenidas.

Tras decidir el eje de división (el de menor perímetro general), el algoritmo de división ordena las entradas (de acuerdo a los límites inferiores o superiores) en la dimensión seleccionada y, de nuevo, examina todas las posibles agrupaciones. La agrupación seleccionada es la que tiene el menor solapamiento entre los MBRs de los nodos resultantes. En caso de empate se seleccionará la agrupación cuya suma de áreas sea la menor. Finalmente se divide el nodo de acuerdo a la agrupación seleccionada.



(a) Antes de la inserción del objeto 09 (en rojo)

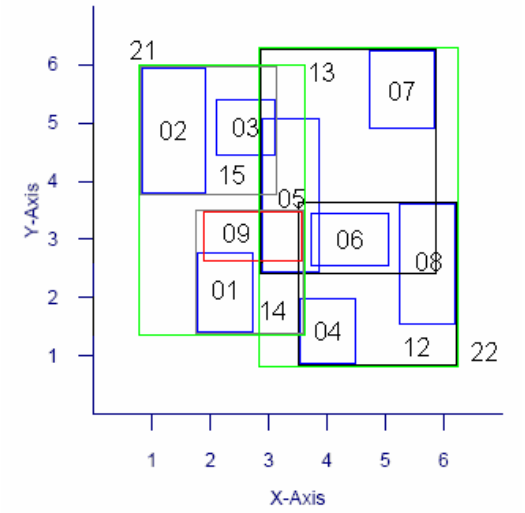
(b) El MBR del nodo 11 se ajusta a los objetos 02, 03 y 09. El objeto 01 ha sido extraido para su reinserción

Figura 2.7. Inserción en R*-tree.

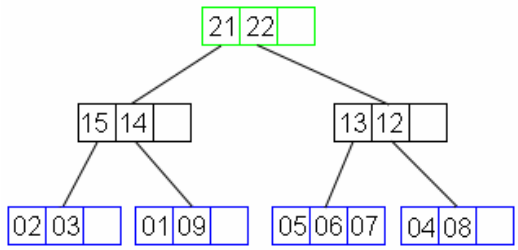
Veamos un ejemplo de inserción para el R*-tree de la figura 2.5. Insertamos el objeto 09 cuyo MBR podemos ver en la figura 2.7a. Comenzamos por el nodo raíz. Al ser este de nivel 1 (nivel inmediatamente superior al nivel hoja) optamos por elegir la entrada que implique un menor aumento del área de solape para contener a 09. Tras estudiar las dos entradas del nodo raíz obtenemos que el nodo que necesita menor incremento del área de solape para acomodar a 09 es el nodo 11 (apuntado por la entrada homónima). El nodo 11 contiene 3 entradas (01, 02 y 03) que almacenan los MBRs de otros tantos objetos. Al contar el nodo 11 con el número máximo de entradas (M=3) se produce un desbordamiento. Supongamos que es el primer desbordamiento de este nodo, por tanto, debemos recurrir al mecanismo de reinserción forzada. Elegimos una de las M+1 entradas (las tres que tenía y la nueva) para reinsertarla en el árbol. La entrada elegida es la que presente una mayor distancia entre su centro y el centro del MBR que cubre al nodo, que en este caso es la 01. Una vez seleccionada la entrada se reajusta el MBR del nodo 11 figura 2.7b. Cuando procedemos a reinsertar al nodo 01 de nuevo obtenemos como mejor destino al nodo 11 y por tanto se produce un nuevo desbordamiento. Al no ser este el primer desbordamiento acudimos al mecanismo de división para repartir las M+1 entradas en dos nuevos nodos.

Comenzamos ordenando las entradas en el eje X, obteniendo la secuencia {02, 01, 09, 03} y en el eje Y, quedando la secuencia {01, 09, 02, 03}. Puesto que m=2 y M=3, tenemos que $M - 2m + 2 = 1$, que son las posibles agrupaciones en cada eje. Por tanto, las únicas agrupaciones posibles manteniendo el número mínimo de entradas por nodo son: {(02, 01), (09, 03)} en el eje X y {(01, 09), (02, 03)} en el eje Y. Podemos comprobar que la suma de los perímetros de (01, 09) y (02, 03) es menor que la de (02, 01) y (09, 03). De esta manera obtenemos dos nuevos nodos, uno que contiene a {01, 09}, llamémosle 14, y otro que contiene a {02, 03}, llamémosle 15. Estos dos nodos sustituyen al nodo 11, por tanto debemos eliminar la entrada 11 del nodo raíz y sustituirla por dos nuevas entradas 14 y 15 que apunten a sus nodos correspondientes. Debido a que el nodo raíz no puede albergar las cuatro entradas 14, 15, 12 y 13 se

produce un desbordamiento. Al ser un desbordamiento en el nodo raíz no se da el mecanismo de reinserción, si no que directamente pasamos a la división del nodo.



(a) Estructura de MBRs



(b) Estructura de archivo. El árbol ha aumentado su altura en un nivel tras la división del nodo raíz

Figura 2.8. R*-tree tras la inserción.

Aplicando el mismo método que antes, obtenemos una partición a lo largo del eje X que divide a las entradas en dos grupos (15, 14) cubierto por el MBR 21 y (13, 12) cubierto por el MBR 22 (verde en la figura). Estas nuevas entradas serán insertadas en un nuevo nodo raíz, aumentando en un nivel la altura del árbol, cuyo estado final podemos ver en la figura 2.8b.

2.5.2.2 Eliminación

La eliminación en el R*-tree es similar a la del R-tree. Una vez encontrado el nodo que contiene el objeto a eliminar se comprueba si se producirá una insuficiencia. En caso negativo se elimina el objeto y se reajusta el MBR del nodo, propagando el reajuste hacia niveles superiores si es necesario. En caso afirmativo se elimina el nodo y se insertan todas sus entradas en el árbol siguiendo el nuevo algoritmo de inserción. Notar que la eliminación de un nodo por insuficiencia puede provocar a su vez una insuficiencia en el nodo padre y que se puede propagar hacia niveles superiores. Por lo que el algoritmo de eliminación es aplicable a cualquier nivel.

2.5.2.3 Búsqueda

La búsqueda en el R*-tree es similar a la del R-tree, pero se beneficia de la mejor estructura del índice alcanzada gracias al nuevo algoritmo de inserción (más compacta y con menor solape). En [BKS90] se realiza una comparativa entre el R*-tree y el R-tree. De los resultados se deduce que el R*-tree mejora el rendimiento del R-tree, por lo que parece ser la alternativa más eficiente para la indexación de datos espaciales.

2.6 El TPR-Tree

El R-Tree parametrizado en el tiempo (Time Parametrized R-tree) [SJLL00] surge como una evolución del R*-tree desarrollada con el objetivo de almacenar objetos con un comportamiento dinámico. Para reflejar este comportamiento hace uso de funciones cuya variable es el tiempo.

2.6.1 Principales características

El TPR-tree es un árbol balanceado en altura similar el R-Tree. Todos los nodos menos al nodo raíz contienen entre m y M entradas, con m = M/2, mientras que el nodo raíz contiene entre 2 y M entradas. Los nodos hoja contienen entradas con registros del índice de la forma (TPMBR, IDobjeto) donde TPMBR es el MBR parametrizado en el tiempo que cubre al objeto móvil apuntado por IDobjeto. Los nodos internos contienen entradas de la forma (TPMBR, nodoHijo) donde TPMBR es el MBR parametrizado en el tiempo que cubre todos los TPMBRs de las entradas del nodo hijo, y nodoHijo apunta al nodo hijo. A continuación definiremos objeto parametrizado en el tiempo (en adelante TP) y MBR TP. Después estudiaremos las diferentes funciones de coste usadas por el TPR-Tree. Por último presentaremos los algoritmos de inserción y eliminación del TPR-Tree con un ejemplo.

El TPR-Tree anticipa las posiciones futuras de los objetos expresándolas como función del tiempo. Sean X_{tL} y V_{tL} (X_{tR} y V_{tR}) vectores d-dimensionales que representan la posición y la velocidad respectivamente del punto inferior o izquierdo (superior o derecho) del MBR que cubre al objeto X en el instante t. Las ecuaciones TP para el MBR que cubre al objeto X se definen como:

$$X_L(t) = X_L(t_r) + V_L(t - t_r)$$

$$X_R(t) = X_R(t_r) + V_R(t - t_r) \qquad (1)$$

Donde t_r es un instante de referencia, $X_L(t_r)$ ($X_R(t)$) es la posición del punto inferior (superior) del MBR que cubre al objeto X en el instante t_r y V_L (V_R) es su velocidad. Notar que las ecuaciones (1) son las ecuaciones paramétricas de una recta.

Supongamos que la posición y la velocidad de X son observadas en un instante t_{obs}. X se puede expresar como un objeto TP con respecto a (1) de la siguiente manera:

1. Seleccionar un instante arbitrario t_r.
2. Obtener $X_L(t_{obs})$ observando la posición inferior de X en algún instante t_{obs}. t_{obs} puede ser o no igual a t_r.
3. Obtener V_L observando la velocidad inferior de X en el instante t_{obs}.
4. Resolver $X_L(t_r)$ en la fórmula:

$$X_L(t_r) = X_L(t_{obs}) + V_L(t_r - t_{obs})$$

$X_R(t_r)$ se obtiene de forma similar.

Ahora el objeto X está completamente especificado por su MBR ya que conocemos los componentes de su función de movimiento $X_L(t_r)$, $X_R(t_r)$, t_r y V_L y V_R.

Si el objeto X presenta un movimiento de traslación pura (es decir, no se deforma) entonces $V_L = V_R$. Si además es un objeto puntual, $X_L = X_R$, quedando su ecuación TP de esta manera:

$$X(t) = X(t_r) + V(t - t_r)$$

El R-Tree define un MBR para los objetos $\{o_1, \ldots, o_k\}$ como el menor rectángulo que contiene a $o_1, \ldots, o_k$. Es posible construir y mantener un MBR para el instante presente y el futuro, pero es computacionalmente muy costoso. En lugar de eso, TPR-Tree emplea MBRs que son mínimos en algún instante, pero probablemente no en los instantes posteriores.

Un MBR TP d-dimensional R se define como:

$$R_L(t) = R_L(t_r) + V_L(t - t_r)$$

$$R_R(t) = R_R(t_r) + V_R(t - t_r)$$

(2)

Donde t_r es un instante de referencia, $R_L(t_r)$ y $R_R(t_r)$ son vectores d-dimensionales que representan los bordes espaciales inferior y superior en el instante t_r respectivamente, y V_L y V_R son los vectores de velocidad para los bordes inferior y superior de R respectivamente.

Supongamos un MBR TP R que cubre los objetos $\{o_1, \ldots, o_k\}$ en algún instante t_{obs} y posteriores. R puede expresarse con respecto a las ecuaciones (2) de la siguiente manera:

1. Seleccionar un instante arbitrario t_r.

2. Obtener $R_L(t_{obs})$ observando las posiciones inferiores de$\{o_1(t_{obs}), \ldots, o_k(t_{obs})\}$ en algún instante t_{obs}, $R_L(t_{obs})$ es igual a $\min\{o_{1L}(t_{obs}), \ldots, o_{kL}(t_{obs})\}$. t_{obs} puede ser o no igual a t_r. De manera análoga $R_R(t_{obs})$ es igual al máximo de las posiciones superiores de los objetos, $\max\{o_{1R}(t_{obs}), \ldots, o_{kR}(t_{obs})\}$.

3. Obtener V_L observando la velocidad inferior de $\{o_1, \ldots, o_k\}$ en algún instante t_{obs}, V_L es igual a min{velocidades{$o.V_{1L}, \ldots, o.V_{kL}$}}. De manera análoga V_R es el máximo de las velocidades superiores max{velocidades{$o.V_{1R}, \ldots, o.V_{kR}$}}.

4. Resolver $R_L(t_r)$ en la fórmula:

$$R_L(t_r) = R_L(t_{obs}) + V_L(t_r - t_{obs})$$

$R_R(t_r)$ se obtiene de forma similar.

Ahora el MBR TP R está completamente especificado ya que conocemos los componentes de su función de movimiento $R_L(t_r)$, $R_R(t_r)$, t_r, V_L y V_R.

Un MBR TP (en adelante MBR hará referencia a un MBR TP, salvo que se indique lo contrario) nunca encoge, y como mínimo crece lo suficiente para seguir siendo válido. En [SJLL00] se sugiere que cada vez que un objeto sea actualizado su MBR (y los MBRs de todos sus antecesores) deberían ajustarse para ser mínimos. Es decir, todos los MBRs a lo largo del camino de actualización son (re)construidos con $t_{obs} = t_{act}$, donde t_{act} es el tiempo de la actualización. Los MBRs construidos con $t_{obs} = t_{act}$ son llamados *MBR de tiempo de actualización*.

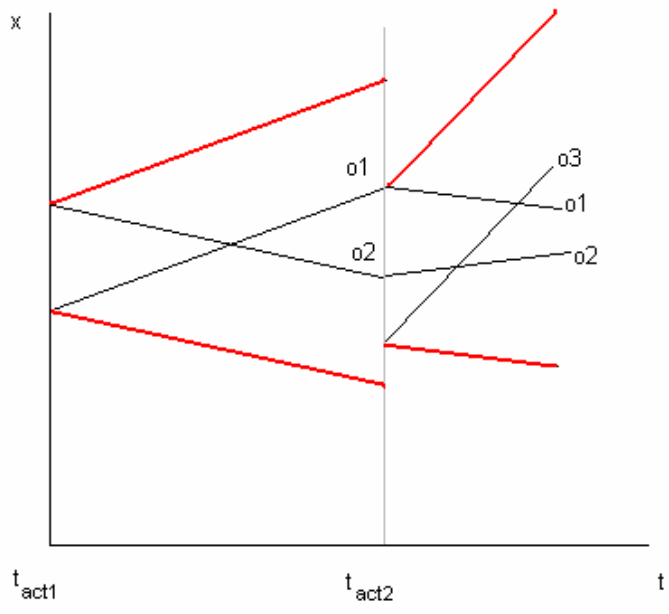


Figura 2.9. Reajuste de un MBR unidimensional.

La figura 2.9 muestra los MBRs de tiempo de actualización en acción. Un MBR de tiempo de actualización unidimensional se muestra en rojo. En el instante t_{act1} los objetos o_1 y o_2 son insertados. En t_{act2} el objeto o_3 es insertado y las velocidades de o_1 y de o_2 son actualizadas. Notar que el MBR es mínimo en t_{act1} y en t_{act2}.

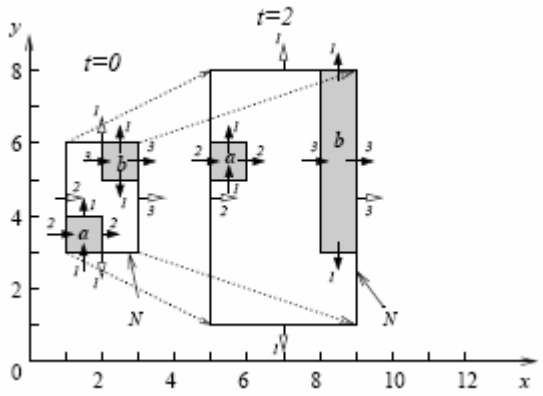


Figura 2.10. MBR de dos objetos TP en los instantes t=0 y t=2.

En la figura 2.10 podemos ver un par de objetos parametrizados en el tiempo y su MBR. Tomando el subíndice 1 para indicar la dimensión X y el subíndice 2 para la dimensión Y, el objeto *a* queda definido por los parámetros $a_{1L} = 1$, $a_{1R} = 2$, $a_{2L} = 3$, $a_{2R} = 4$ para su posición de referencia y $a.V_{1L} = 2$, $a.V_{1R} = 2$, $a.V_{2L} = 1$, $a.V_{2R} = 1$ para su

velocidad. El objeto *b* queda definido por los parámetros $b_{1L} = 2$, $b_{1R} = 3$, $b_{2L} = 5$, $b_{2R} = 6$ para su posición de referencia y $b.V_{1L} = 3$, $b.V_{1R} = 3$, $b.V_{2L} = -1$, $b.V_{2R} = 1$ para su velocidad. Por tanto, tomando t = 0 como tiempo de actualización, el MBR que cubre a los objetos *a* y *b* como parámetros $MBR_{1L} = \min \{a_{1L}, b_{1L}\} = 1$, $MBR_{1R} = \max \{a_{1R}, b_{1R}\} = 3$, $MBR_{2L} = \min \{a_{2L}, b_{2L}\} = 3$, $MBR_{2R} = \max \{a_{2R}, b_{2R}\} = 6$ para su posición de referencia y $MBR.V_{1L} = \min \{a.V_{1L}, b.V_{1L}\} = 2$, $MBR.V_{1R} = \max \{a.V_{1R}, b.V_{1R}\} = 3$, $MBR.V_{2L} = \min \{a.V_{2L}, b.V_{2L}\} = -1$, $MBR.V_{2R} = \max \{a.V_{2R}, b.V_{2R}\} = 1$ para su velocidad. En el instante t = 0 el MBR es mínimo. En el instante t = 2 vemos que ya no lo es.

2.6.2 Funciones de coste

Como el TPR-Tree tiene la estructura de un R*-Tree, la inserción y la eliminación aplicables al R*-Tree lo son también al TPR-Tree con unas pocas modificaciones.

Los algoritmos de inserción y eliminación del R*-Tree tratan de minimizar ciertas funciones de coste [BKSS90]. El TPR-Tree adopta los algoritmos de inserción y eliminación del R*-Tree, modificando las funciones de coste para adaptarlas al dominio espacio-temporal.

La tabla 1 describe las funciones de coste usadas por el R*-Tree y por el TPR-Tree. El TPR-Tree adapta esas funciones al dominio espacio-temporal expresándolas como función del tiempo e integrando sobre [CT, CT + h], donde CT es el instante de actualización y h es el horizonte temporal. Insistir en que el TPR-Tree es un índice espacio-temporal que se mantiene funcional indefinidamente, pero se optimiza para un horizonte temporal específico. Es decir, un MBR se mantiene óptimo desde el instante de su última actualización t_{act}, hasta el instante $t_{act} + h$, para algún horizonte h.

Función de coste	R*-Tree	TPR-Tree
Área	$Area(R)$, donde R es un rectángulo y *Area()* es la función que calcula el área del rectángulo	$\int_{CT}^{CT+h} Area(R(t))$, donde *R(t)* es un rectángulo parametrizado en el tiempo, *Area()* es la función de área para un rectángulo parametrizado en el tiempo y *CT* es el último instante de actualización
Margen	$Margen(R)$, donde R es un rectángulo y *Margen()* es la función que calcula el margen o perímetro del rectángulo	$\int_{CT}^{CT+h} Margen(R(t))$, donde *R(t)* es un rectángulo parametrizado en el tiempo, *Margen()* es la función de margen para un rectángulo parametrizado en el tiempo y *CT* es el último instante de actualización
Área de solape entre dos rectángulos	$Area(R)$, donde R el solape rectangular entre dos rectángulos	$\int_{CT}^{CT+h} Area(R(t))$, donde *R(t)* es un rectángulo parametrizado en el tiempo que representa el solape entre dos rectángulos parametrizados en el tiempo, *Area()* es la función de área para un rectángulo parametrizado en el tiempo y *CT* es el último instante de actualización
Distancia entre los centroides de dos rectángulos	$Dist(c1, c2)$, donde *c1* y *c2* son los centroides de dos rectángulos	$\int_{CT}^{CT+h} Dist(c1(t),c2(t))$, donde *c1(t)* y *c2(t)* son los centroides de dos rectángulos parametrizados en el tiempo expresados como puntos parametrizados en el tiempo, *Dist(c1, c2)* es la función de distancia para dos puntos parametrizados en el tiempo y *CT* es el último instante de actualización

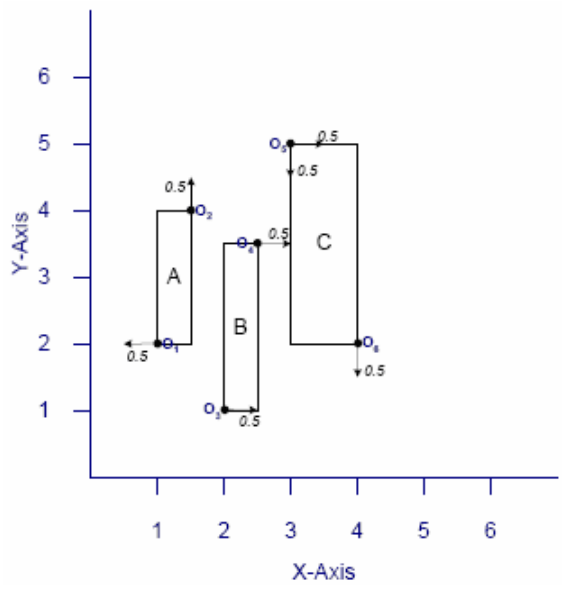
Tabla 1. Funciones de coste para el TPR-tree.

2.6.3 Operaciones

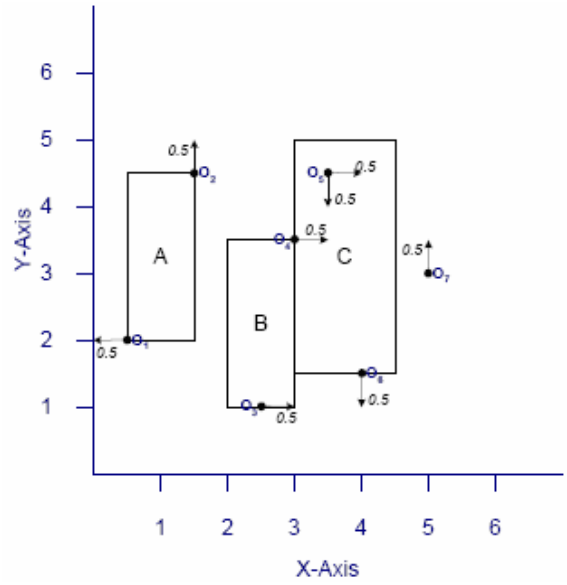
2.6.3.1 Inserción

Detallaremos el algoritmo de inserción del TPR-Tree mediante pseudocódigo y siguiendo un ejemplo.

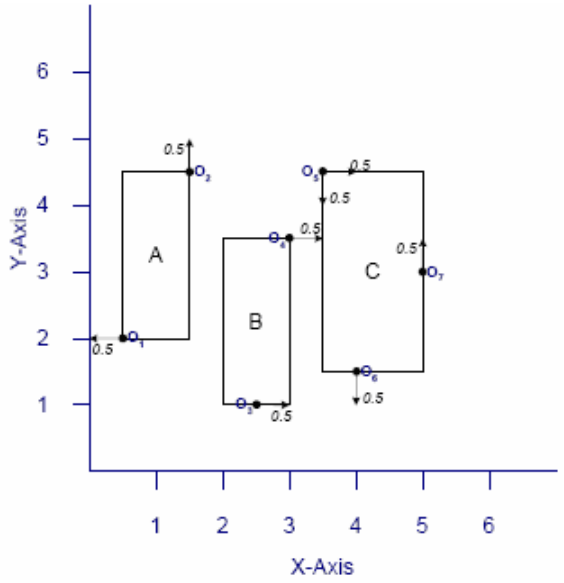
El algoritmo *ElegirSubarbol* es usado por el TPR-Tree y que se deriva del R*-Tree. Notar que usa las funciones de coste adaptadas que se definen en la tabla 1.



(a) Nodos hoja A, B y C en t=0



(b) Nodos hoja A, B y C antes de la inserción de o_7 en t=0.5



(c) Nodos hoja A, B y C la inserción de o_7 en t=0.5

Figura 2.11. Inserción en TPR-tree.

Considerar la figura 2.11a, donde se muestran tres nodos hoja A, B y C en el instante t = 0 con m = 2 y M = 3. Ahora supongamos que queremos insertar el objeto o_7 en el instante t = 0.5 como se muestra en la figura 2.11b. Se selecciona el nodo C porque es el que necesita el menor incremento en el área de solapamiento integral. En la figura 2.11c vemos el estado del índice tras la inserción de o_7. Notar que el MBR de C está más ajustado a causa de esta actualización.

El algoritmo *ElegirSubarbo*l devuelve el nodo N en el que mejor se acomoda la entrada E, y en el que se insertará E si N no está lleno (su número de entradas es menor que M). Sin embargo, si N no tiene espacio suficiente, la reinserción forzada tiene lugar y varias entradas de N son reinsertadas.

Algoritmo ElegirSubarbol(nodo N, entrada NE, nivel L)
/* Al principio N es la raíz del TPR-tree, NE es la entrada a insertar y L el nivel en el que se insertará. El algoritmo devuelve el nodo de nivel L en el que mejor acomodo encuentra NE */
1. **si** N es de nivel L
2. devolver N
3. **si** N.nivel + 1 es el nivel hoja /* N apunta a nodos hoja, se selecciona la hoja que implique un menor incremento en el área integral de solape */
4. **para todas** las entradas $E_i \in \{E_1 \ldots E_p\} \in N$
5. $solape(E_i) = \sum_{k=1, k\neq i}^{p} \int_{CT}^{CT+h} Area(E_i.mbr \cap E_k.mbr)$, $1 \leq i \leq p$
6. **para todas** las entradas $E_i \in \{E_1 \ldots E_p\} \in N$
7. $E_i.mbr' = E_i.mbr \cup NE.mbr$
8. $aumentoSolape(E'_i) = \sum_{k=1, k\neq i}^{p} \int_{CT}^{CT+h} Area(E_i.mbr' \cap E_k.mbr)$, $1 \leq i \leq p$
9. **devolver** el nodo hoja apuntado por E_k
   donde $\forall\, i,\ min\{ aumentoSolape(E'_i) - solape(E_i)\} = aumentoSolape(E_k)$
10. **si_no** /* N apunta a nodos internos, se selecciona el nodo que implique menor incremento en el área integral */
11. **para todas** las entradas $E_i \in \{E_1 \ldots E_p\} \in N$
12. $E_i.mbr' = E_i.mbr \cup NE.mbr$
13. $aumentoSolape(E_i) = \int_{CT}^{CT+h} Area(E_i.mbr') - \int_{CT}^{CT+h} Area(E_i.mbr)$
    /* avanzar en la rama que ofrece el menor incremento en el área integral */
14. **ElegirSubarbol**(E_k,NE,L) donde $\forall\, i,\ min\{ aumentoArea(E_i)\} = aumentoArea(E_k)$

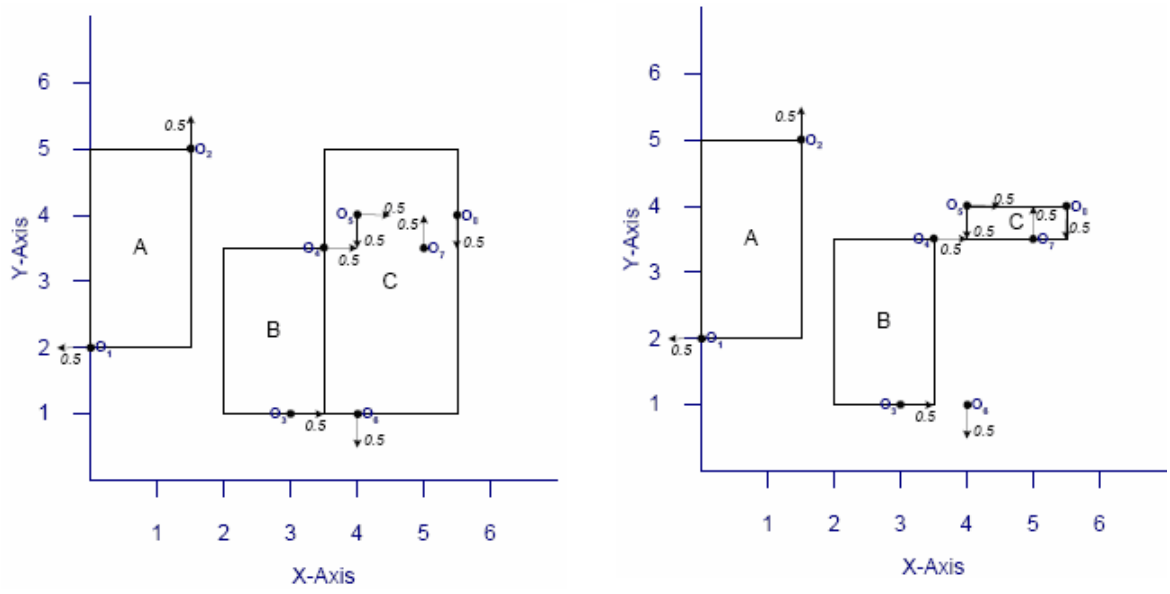
fin ElegirSubarbol

Algoritmo ReinsercionForzada(nodo N)
/* Recibe un (sub)árbol TPR desbordado con raíz N. El centroide de un nodo es el centroide de su MBR */
1. $centroide(N) = centroide(N.mbr)$
2. **para todas** las entradas $E_i \in \{E_1 \ldots E_p\} \in N$
3. $centroide(E_i) = centroide(E_i.mbr)$
4. $centroideDist(E_i) = \int_{CT}^{CT+h} Dist(centroide(N), centroide(E_i))$

5. **para todas** las entradas $E_i \in \{E_1 \ldots E_p\} \in N$, donde *centroideDist*($E_i$) esté en los primeros 30 percentiles
6. insertar E_i en el nivel $E_i.nivel$

fin ReinsercionForzada

El algoritmo *ReinsercionForzada* describe el procedimiento de reinserción forzada usada por el TPR-Tree. Notar su similitud con el del R*-Tree. La figura 2.12 continúa con el ejemplo de la figura 2.11, donde insertamos el objeto o_8 en la posición (5.5, 4) en el instante t = 1. Esto causa que el nodo C se desborde, y como resultado el objeto o_6 se selecciona para ser reinsertado, ya que tiene la mayor distancia al centroide de C.



(a) Nodos hoja A, B y C antes de la reinserción forzada

(b) Nodos hoja A, B y C tras eliminar o_6 para su reinserción. C es el nodo que requiere un menor aumento del área de solape integral para albergar a o_6

Figura 2.12. Reinserción forzada en TPR-tree.

Algoritmo DividirNodo(nodo N)

/* Recibe un nodo desbordado, N. M y m son el número máximo y mínimo de entradas en un nodo respectivamente. El resultado consiste en dos conjuntos de entradas en las que queda dividido el nodo */

1. ejeDivision = ElegirEjeDivision(N)
2. **ordenar** las entradas de N a lo largo de ejeDivision según sus MBRs en el instante $t_{actualización}$ en $\{E_1, \ldots, E_M\}$
3. /* calcular el área integral de solape para cada una de las $M-2m+2$ agrupaciones */
4. **para** $k = 1, \ldots, M-2m+2$
5. $Grupo_{k1} = \{E_1, \ldots, E_{m-1+k}\}$
6. $Grupo_{k2} = \{E_{m+k}, \ldots, E_M\}$
7. $areaSolape_k = \int_{CT}^{CT+h} Area(Grupo_{k1}.mbr \cap Grupo_{k2}.mbr)$, $1 \le i \le M-2m+2$
8. **devolver** la agrupación $\{E_1, \ldots, E_{m-1+k}\}$, $\{E_{m+k}, \ldots, E_M\}$, $1 \le i \le M-2m+2$ donde $\forall k = 1, \ldots, M-2m+2$, $min\{areaSolape_k\} = areaSolape_i$

fin DividirNodo

Algoritmo ElegirEjeDivision(nodo N)
/* Recibe un nodo desbordado, N. M y m son el número máximo y mínimo de entradas en un nodo respectivamente. D es el número de dimensiones espaciales. El resultado consiste en una de las 2D dimensiones, en la que dividir las entradas del nodo */
1. **para** d_i, $1 \leq i \leq 2D$
2. **ordenar** las entradas de N a lo largo de d_i, según sus MBRs en el instante $t_{actualización}$ en $\{E_1, \ldots, E_M\}$
3. **para** $k = 1, \ldots, M-2m+2$
4. $Grupo_{k1} = \{E_1, \ldots, E_{m-1+k}\}$
5. $Grupo_{k2} = \{E_{m+k}, \ldots, E_M\}$
6. $margenTotal_i = \int_{CT}^{CT+h} Margen(Grupo_{k1}.mbr) + Margen(Grupo_{k2}.mbr)$
7. **devolver** d_i, $1 \leq i \leq 2D$, *donde* $\forall\, k = 1, \ldots, 2D$, $min\{margenTotal_k\} = margenTotal_i$

fin ElegirEjeDivision

La reinserción básicamente trata de retrasar la costosa operación de dividir un nodo. La división se realiza cuando un nodo seleccionado para la reinserción ya está lleno y se produjo un desbordamiento previo en el mismo nivel. En la figura 2.12b el nodo C se vuelve a seleccionar para insertar el objeto o_6 ya que causa el menor incremento del área integral de solapamiento para los nodos A, B y C. Como el nodo C está lleno y se produjo un desbordamiento previo en el mismo nivel (en el propio nodo C), el TPR-tree dividirá al nodo C en dos nodos.

El proceso de división comienza seleccionando un eje de división, como en el R*-Tree. Para esto se sigue el procedimiento mostrado en el algoritmo SeleccionarEjeDivision. Los cálculos necesarios para elegir el eje de división del nodo C de la figura 2.12b son los siguientes:

1. Ordenar los objetos a lo largo del eje X: $\{o_5, o_6, o_7, o_8\}$

   $Grupo_1 = \{o_5, o_6\}$

   $Grupo_2 = \{o_7, o_8\}$

   $Grupo_1.mbr_L(t) = (4, 1) + (0, -0.5)t$

   $Grupo_1.mbr_R(t) = (4, 4) + (0{,}5\ -0.5)t$

   $Margen(Grupo_1.mbr) = 2\times(0.5t + 3) = 6 + t$

   $\int_1^3 Margen(Grupo_1.mbr) = \int_1^3 6 + t = 16$

   $Grupo_2.mbr_L(t) = (5, 3.5) + (0, -0.5)t$

   $Grupo_2.mbr_R(t) = (5.5, 4) + (0, -0.5)t$

   $Margen(Grupo_2.mbr) = 2\times(0.5 + 0.5 + t) = 2 + 2t$

$$\int_1^3 \text{Margen}(\text{Grupo}_2.\text{mbr}) = \int_1^3 2 + 2t \; = 12$$

Margen total para la agrupación $\{o_5, o_6, o_7, o_8\} = 28$

2. Ordenar los objetos a lo largo del eje Y: $\{o_6, o_7, o_5, o_8\}$

   $\text{Grupo}_1 = \{o_6, o_7\}$

   $\text{Grupo}_2 = \{o_5, o_8\}$

   $\text{Grupo}_1.\text{mbr}_L(t) = (4, 1) + (0, -0.5)t$

   $\text{Grupo}_1.\text{mbr}_R(t) = (5, 3.5) + (0, 0.5)t$

   $\text{Margen}(\text{Grupo}_1.\text{mbr}) = 2\times(1 + 2.5 + t) = 7 + 2t$

   $$\int_1^3 \text{Margen}(\text{Grupo}_1.\text{mbr}) = \int_1^3 7 + 2t \; = 22$$

   $\text{Grupo}_2.\text{mbr}_L(t) = (4, 4) + (0, -0.5)t$

   $\text{Grupo}_2.\text{mbr}_R(t) = (5.5, 4) + (0.5, -0.5)t$

   $\text{Margen}(\text{Grupo}_2.\text{mbr}) = 2\times(1.5 + 0.5t) = 3 + t$

   $$\int_1^3 \text{Margen}(\text{Grupo}_2.\text{mbr}) = \int_1^3 3 + t \; = 10$$

   Margen total para la agrupación $\{o_6, o_7, o_5, o_8\} = 32$

3. Ordenar los objetos según su velocidad a lo largo del eje X (V-X). Notar que hay tres posibles agrupaciones porque los objetos o_6, o_7 y o_8 tienen la misma velocidad en el eje X. TPR-Tree no especifica como actuar cuando esto ocurre, así que ordenamos según los subíndices de los objetos del ejemplo. $\{o_6, o_7, o_8, o_5\}$.

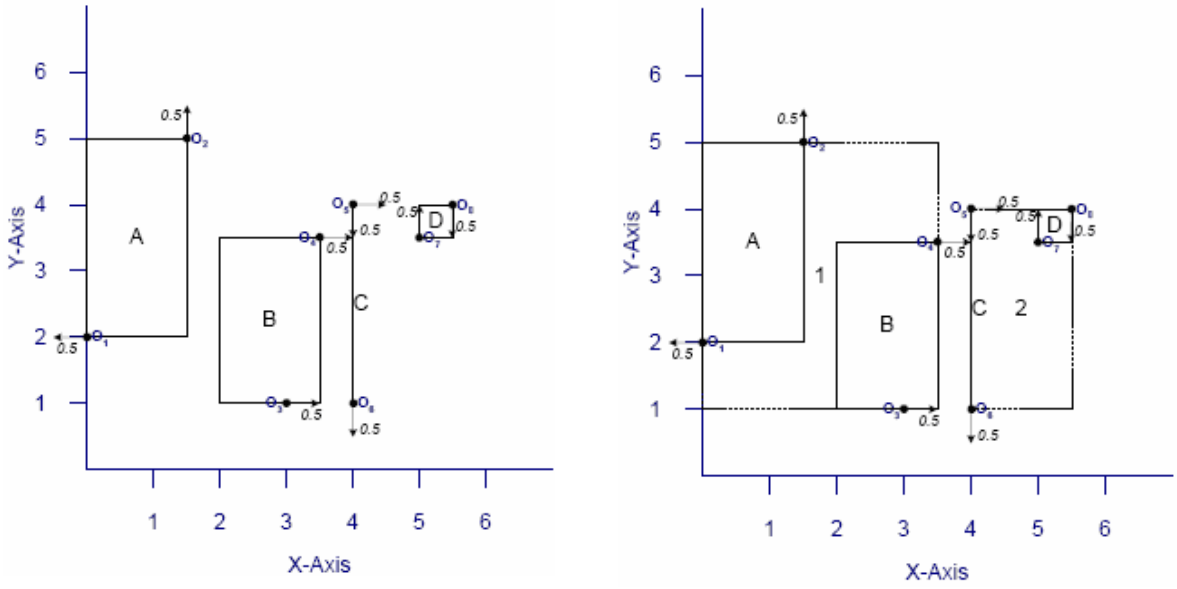
   Siguiendo un proceso análogo al anterior obtenemos que el margen total para la agrupación $\{o_6, o_7, o_8, o_5\}$ es 32

4. Ordenar los objetos según su velocidad a lo largo del eje Y (V-Y). Notar que hay tres posibles agrupaciones porque los objetos o_5, o_6 y o_8 tienen la misma velocidad en el eje Y. TPR-Tree no especifica como actuar cuando esto ocurre, así que ordenamos según los subíndices de los objetos del ejemplo. $\{o_5, o_6, o_8, o_7\}$.

   Siguiendo un proceso análogo al anterior obtenemos que el margen total para la agrupación $\{o_5, o_6, o_8, o_7\}$ es 28

Una vez calculados todos los márgenes o perímetros para las cuatro ordenaciones posibles (X → 28, Y → 32, V-X → 32 y V-Y → 28), la división se producirá a lo largo del eje X (o el V-Y), ya que tiene el menor margen total, cuyo valor es 28. Ahora el algoritmo *DividirNodo* ordenará las entradas del nodo C a lo largo del eje X y determinará la agrupación óptima que minimiza el área de intersección integral. Sin embargo, como sólo hay cuatro entradas en el nodo la única agrupación posible es {(5, 6), (7, 8)}. La figura 2.13a muestra el índice tras dividir el nodo C en los nuevos nodos C y D.

La división del nodo C en dos nuevos nodos C y D causa el desbordamiento del nodo raíz, que ahora tendría cuatro entradas. Por tanto que hay que dividir el nodo raíz. La figura 2.13b muestra el estado final del ejemplo.



(a) Nodos hoja A, B y C tras la división de C

(b) Estructura final del índice tras la inserción de o_8. Los nodos 1 y 2 aparecen como resultado de la división del nodo de nivel superior que contenía a A, B y C

Figura 2.13. División en TPR-tree.

2.6.3.2 Eliminación

La eliminación de un objeto en el TPR-Tree es igual que la del R*-Tree. Primero se busca el nodo hoja que contiene al objeto que se debe borrar mediante una consulta puntual. El objeto se elimina de la hoja. Si el número de entradas que queda en el nodo es menor que el mínimo necesario, tiene lugar la reinserción en el índice de todas las entradas del nodo de la forma descrita anteriormente.

2.7 El TPR*-Tree

2.7.1 Principales características

El TPR*-Tree [TPS03] mejora al TPR-Tree introduciendo un nuevo conjunto de funciones de coste basadas en un modelo de coste de consultas revisado. Tao y otros demostraron que el rendimiento del TPR*-Tree casi alcanza el óptimo teórico basado en

el modelo de coste propuesto por ellos. Además, el TPR*-Tree utiliza un algoritmo de inserción revisado y diferentes criterios para seleccionar las entradas a las que se les aplicará la reinserción forzada, todo lo cual ayuda a mejorar la selectividad en consultas. En esta sección presentaremos las funciones de coste revisadas y el nuevo modelo de coste. Después mostraremos los algoritmos revisados de inserción y reinserción forzada del TPR*-Tree, ayudándonos de un ejemplo.

2.7.2 Funciones de coste

Las funciones de coste del TPR*-Tree se basan en un modelo de coste para consultas que mide con precisión el número de accesos para una consulta espacio-temporal dada, q, donde la consulta q es un rectángulo TP especificado mediante la ecuación (2). Recordar que, para el TPR-Tree, un nodo es visitado si su MBR interseca con el rectángulo de consulta q en el intervalo de tiempo [q_{inicio}, q_{fin}].

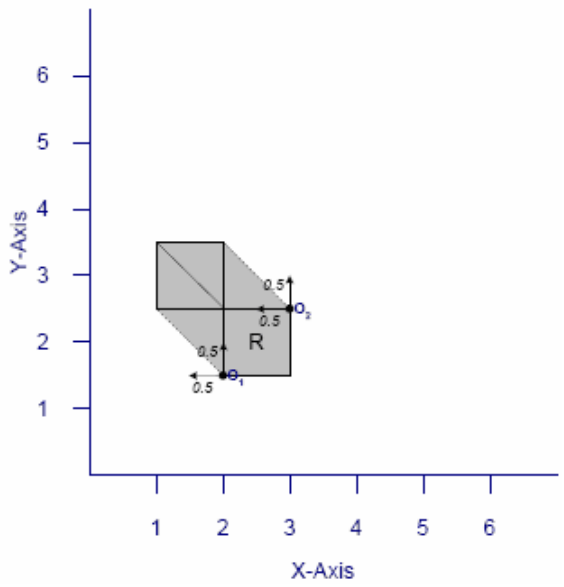


Figura 2.14. MBR de un rectángulo y su área de barrido.

La probabilidad de que q visite un nodo n es por tanto la probabilidad de que q interseque al MBR R del nodo n. Supongamos que el conjunto de consultas {q} son consultas sobre puntos estáticos (es decir, no tienen ni tamaño ni velocidad) que se extienden en el intervalo [q_{inicio}, q_{fin}], y están distribuidas uniformemente en el espacio. La probabilidad de que un nodo satisfaga estas consultas es entonces proporcional al área de la envoltura convexa de los vértices de R(q_{inicio}) y R(q_{fin}) [TSP03]. Se llama a esta envoltura región de barrido de R sobre [q_{inicio}, q_{fin}], y en adelante la denotaremos como SR(R, [q_{inicio}, q_{fin}]). La figura 2.14 muestra un MBR TP y su región de barrido. El coste medio de responder consultas de longitud [q_{inicio}, q_{fin}] sobre puntos estáticos se puede estimar mediante la siguiente ecuación:

$$\forall\, n \in \{Nodos\},\ \sum_n Area(SR(n.MBR, [q_{inicio}, q_{fin}])) \qquad (3)$$

Donde definimos SR(n.MBR, [q_{inicio}, q_{fin}]) como la región de barrido del MBR de n durante el intervalo [q_{inicio}, q_{fin}] y {Nodos} como el conjunto de los nodos del árbol.

Podemos optimizar el índice para consultas sobre puntos estáticos minimizando la ecuación (3) en los algoritmos de construcción. En [TPS03] se propone un método para estimar el coste de consultas sobre regiones móviles definidas mediante la ecuación (2). La idea es reducir el problema de la estimación de la selectividad para consultas sobre regiones móviles a la estimación de la selectividad para consultas sobre puntos estáticos (que si sabemos resolver). Esta reducción se consigue aplicando una transformación al MBR TP de cada nodo con respecto a q. Sean R y q un MBR TP y una consulta respectivamente definidos de la siguiente manera:

$$R_L(t) = R_L(t_r) + R.V_L(t - t_r)$$

$$R_R(t) = R_R(t_r) + R.V_R(t - t_r)$$

$$q_L(t) = q_L(t_r) + q.V_L(t - t_r)$$

$$q_R(t) = q_R(t_r) + q.V_R(t - t_r) \qquad (4)$$

El MBR TP transformado R' respecto a la consulta q se define entonces como:

$$R'_L(t) = R_L(t_r) - |q_L(t_r) - q_R(t_r)| / 2 + (R.V_L - q.V_R)(t - t_r) \qquad (5)$$

$$R'_R(t) = R_R(t_r) - |q_L(t_r) - q_R(t_r)| / 2 + (R.V_R - q.V_L)(t - t_r) \qquad (6)$$

Lema 2.1. *Dada una consulta q y un MBR TP R, se obtiene R' aplicando la transformación definida en la ecuación (5). La consulta q interseca con R sii el centroide de q(q_{inicio}), un punto estático, interseca con R'.*

De acuerdo al Lema 1, cuya demostración se presenta en [TSP03], podemos usar la transformación definida en la ecuación (5) para hallar la intersección entre dos MBRs TP (q y R) a partir de un punto estático y un MBR TP (centroide de q y R'). Por tanto podemos transformar una consulta sobre una región móvil lanzada al índice en una consulta sobre puntos estáticos aplicando la transformación a todos los MBRs de todos los nodos. Es importante mencionar que para un MBR R particular, el rectángulo transformado R' es el mismo para el conjunto de consultas {q} donde para todas las consultas q_L, q_R, $q.V_L$, $q.V_R$, t_r y $[q_{inicio}, q_{fin}]$ son los mismos. Nos referiremos a estas consultas que comparten los mismos parámetros como consultas pertenecientes a la misma "clase".

La transformación definida en las ecuaciones (5) y (6) junto con el Lema 1 nos permite estimar el coste de cualquier consulta de ventana q. Sea n.MBR' el MBR TP del nodo n obtenido aplicando la transformación respecto a la clase de consultas $\{q_c\}$ a su MBR original. El coste medio de responder a las consultas de la clase $\{q_c\}$ se define como:

$$\forall\, n \in \{Nodos\},\ \sum_n Area(SR(n.MBR', [q_{inicio}, q_{fin}]))$$

(7)

Obviamente es ideal optimizar un índice para una clase de consultas particular a partir de la ecuación 7. Sin embargo, el índice sólo se puede optimizar para una clase de consultas cada vez y, además, el calculo de la ecuación 7 es normalmente muy costoso, ya que se necesita un paso extra para transformar el MBR de cada nodo. En [TPS03] se prueba que un índice optimizado para una clase de consultas específica sólo alcanza un 2% más de rendimiento que uno optimizado para consultas de puntos estáticos. Por tanto decidieron optimizar el TPR*-Tree minimizando la ecuación (3) en las funciones de coste.

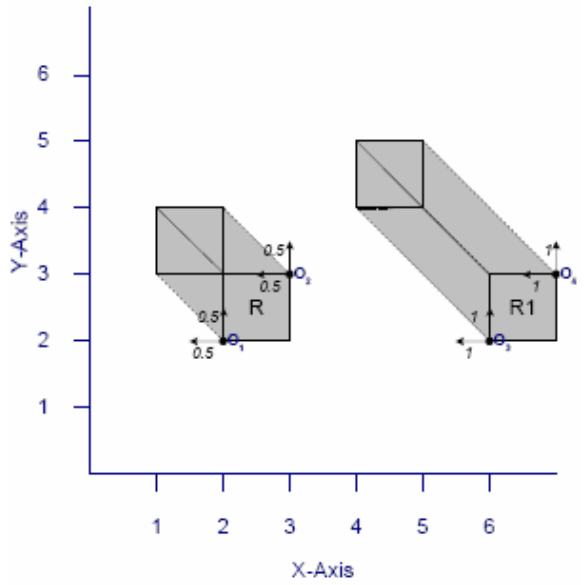


Figura 2.15. Área frente a región de barrido. Las áreas integrales de R y R1 son iguales, mientras que la regiones de barrido son distintas.

Dado el modelo de coste definido en la ecuación 3, vemos que las métricas integrales usadas por el TPR-Tree no pueden minimizar la selectividad de las consultas con precisión. En particular, es posible que las áreas integrales coincidan, mientras que las áreas de barrido sean diferentes. Consideremos la figura 2.15 donde se muestran dos MBRs R y R1 en el instante t = 0. Si establecemos el horizonte temporal h como h = 2, vemos que la métrica integral de R tanto como la de R1 son 1. Esto sugiere que tanto R como R1 tienen la misma selectividad, mientras que claramente esto no es cierto, ya que Area(SR(R, [0, 2])) ≠ Area(SR(R1 [0, 2])). Las funciones de coste revisadas se muestran el la tabla 2.

Función de coste	TPR-Tree	TPR*-Tree
Área	$\int_{CT}^{CT+h} Area(R(t))$, donde *R(t)* es un rectángulo parametrizado en el tiempo, *Area()* es la función de área para un rectángulo parametrizado en el tiempo y *CT* es el último instante de actualización	$Area(SR(R(t),[CT,CT+h]))$, donde *R(t)* es un rectángulo parametrizado en el tiempo y *Area(SR(R(t),[t_1,t_2]))* es el área de la región de barrido de *R(t)* entre los instantes t_1 y t_2
Margen	$\int_{CT}^{CT+h} Margen(R(t))$, donde *R(t)* es un rectángulo parametrizado en el tiempo, *Margen()* es la función de margen o perímetro para un rectángulo parametrizado en el tiempo y *CT* es el último instante de actualización	$Margen(SR(R(t),[CT,CT+h]))$, donde *R(t)* es un rectángulo parametrizado en el tiempo y *Margen(SR(R(t),[t_1,t_2]))* es el margen de la región de barrido de *R(t)* entre los instantes t_1 y t_2
Área de solape entre dos rectángulos	$\int_{CT}^{CT+h} Area(R(t))$, donde *R(t)* es un rectángulo parametrizado en el tiempo que representa el solape entre dos rectángulos parametrizados en el tiempo, *Area()* es la función de área para un rectángulo parametrizado en el tiempo y *CT* es el último instante de actualización	No aplicable. El TPR*-tree trata de minimizar la función *Area(SR(R(t),[CT,CT+h]))* cuando selecciona un nodo hoja para realizar una inserción
Distancia entre los centroides de dos rectángulos	$\int_{CT}^{CT+h} Dist(c1(t),c2(t))$, donde *c1(t)* y *c2(t)* son los centroides de dos rectángulos parametrizados en el tiempo expresados como puntos parametrizados en el tiempo, *Dist(c1, c2)* es la función de distancia para dos puntos parametrizados en el tiempo y *CT* es el último instante de actualización	No aplicable. El TPR*-tree selecciona entradas que minimizarán la función *Area(SR(R(t),[CT,CT+h]))* cuando considera entradas para la reinserción forzada

Tabla 2. Funciones de coste para el TPR*-tree.

2.7.3 Operaciones

2.6.3.1 Inserción

El algoritmo *SeleccionarSubArbol* del TRP-Tree usa una *aproximación greedy*, que selecciona la rama con el menor incremento en el área integral para cada nivel intermedio. En caso de que dos o más ramas tengan el mismo incremento (por ejemplo 0), la *aproximación greedy* selecciona una de ellas aleatoriamente, haciendo posible que la rama más idónea sea descartada. Ilustramos esta situación en la figura 2.16a, donde se muestran dos nodos intermedios 1 y 2 con sus respectivos nodos hoja a, b, c y d.

Si insertamos el objeto o_1, ningún nodo intermedio incrementa su área integral. En este caso, el *algoritmo greedy* selecciona aleatoriamente una rama para continuar. Si no tenemos suerte, es posible que se seleccione el nodo 1, cuando claramente la mejor elección es el nodo 2.

El problema con la *aproximación greedy* se amplifica en el TPR-Tree porque los MBRs TP crecen con el tiempo, aumentando el área de solapamiento. Motivados por esta observación, se propone una modificación del algoritmo *SeleccionarSubArbol* que garantiza la selección de la mejor ruta de inserción. El algoritmo *SeleccionarSubArbolRevisado* almacena las rutas candidatas exploradas en una cola de prioridad. Los elementos en la cola de prioridad se ordenan según su *coste de degradación*, donde el coste de degradación de una ruta *p* se define como el incremento acumulado del área de barrido si el objeto se inserta a través de *p*. El algoritmo termina cuando la primera ruta candidata en la cola alcanza el nivel del árbol requerido.

Notar que el algoritmo de inserción revisado del TPR*-Tree no usa el incremento del solapamiento para calcular el coste de degradación para los nodos hoja, por lo que es un algoritmo lineal en el tiempo.

En el ejemplo de la figura 2.16a, el algoritmo *SeleccionarSubArbolRevisado* inicializa la cola de prioridad PQ con {((1), 0), ((2), 0)}, donde se indica el coste de degradación para cada ruta con un horizonte temporal h = 1. El coste de degradación de cada uno de los nodos 1 y 2 es 0, porque ninguno necesita incrementar su área de barrido para incluir o_1. El algoritmo selecciona el nodo 1 e introduce en PQ las rutas (a, 1) y (b, 1) junto con sus costes de degradación respectivos. Ahora el estado de PQ es {((2), 0), ((a, 1), 2.55), ((b, 1), 3)}. El algoritmo continúa entonces explorando el nodo 2 e inserta en PQ las rutas (d, 2) y (c, 2) con sus costes de degradación. El estado de PQ ahora es {((c, 2), 0),((a, 1), 2.25), ((b, 1), 3), ((d, 2), 5.25)}. El algoritmo termina ya que la ruta con el menor coste de degradación (es decir (c, 2)) alcanza el nivel de inserción necesario.

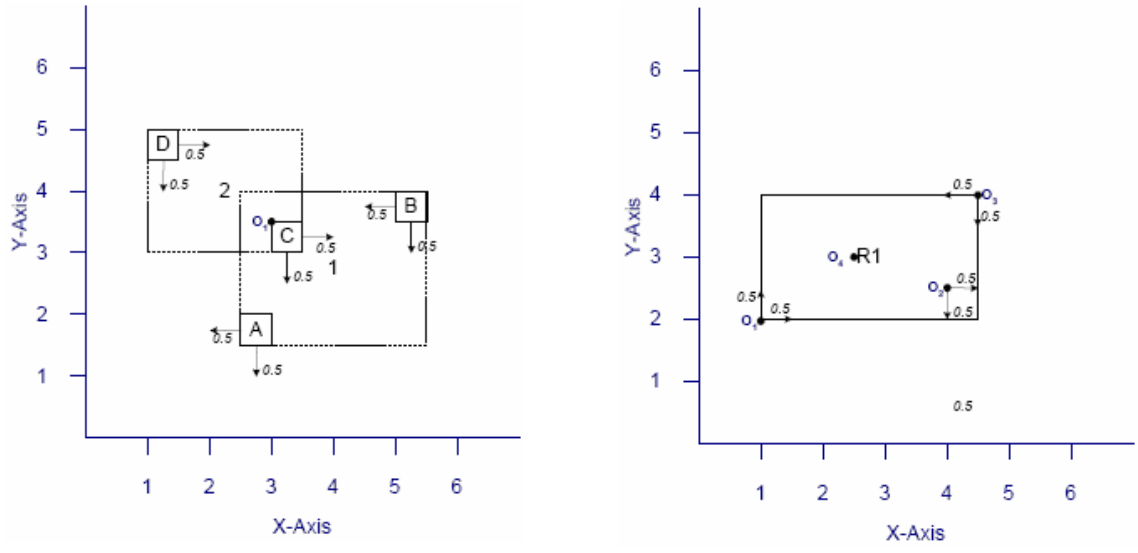
Algoritmo ElegirSubarbolRevisado(cola_prioiridad PQ, nodo N, entrada NE, nivel L)
/* Al principio N es la raíz del TPR-tree, NE es la entrada a insertar y L el nivel en el que se insertará. La cola de prioridad almacena los caminos explorados hasta el momento, con el que tiene el menor coste de degradación acumulativo en la cima. El algoritmo devuelve el nodo de nivel L en el que mejor acomodo encuentra NE. */
1. **si** *N* es el nodo raíz
2. *PQ.insertar(N, 0)* /* inicializar la cola */
3. **si** *N.nivel = L*

4. devolver *N*
5. *N' = PQ.cima*
   /* para todas las entradas E_i de *N'* calcular el coste de degradación para el camino $N' \cup E_i$ e insertarlo en la cola */
6. **para todas** las entradas $E_i \in \{E_1 \ldots E_p\} \in N'$
7. $E_i.costeDegradacion = SR(Area(E_i.mbr \cup NE.mbr),[CT, CT+h]) -$
8. $SR(Area(E_i.mbr),[CT, CT+h])$
9. $PQ.insertar(E_i, N'.costeDegradacion + E_i.costeDegradacion)$
10. **ElegirSubarbolRevisado**(*PQ,PQ.cima,NE,L*) /* explorar el camino con menor coste de degradación */

fin ElegirSubarbolRevisado

Aunque la versión revisada de *SeleccionarSubArbol* incurre en un gran número de accesos a nodos comparado con la aproximación greedy, en promedio una solución óptima se puede encontrar explorando de 2 a 3 rutas completas. En [TPS03] también señalan que el incremento en el rendimiento de consulta alcanzado gracias a una mejor estructura de árbol ayudará en la fase de eliminación de una operación de actualización.

El uso de MBRs conservativos introduce un interesante problema al seleccionar qué entradas de un nodo desbordado reinsertar. Mientras el TPR-Tree selecciona las entradas con la mayor distancia integral al centroide del MBR R (como se muestra en el algoritmo *ReinsercionForzada*), es posible que la extensión espacial y de velocidad de R no decrezca. Consecuentemente, las entradas seleccionadas para la reinserción forzada vuelvan a ser insertadas en R, causando una división del nodo. Esta situación se muestra en la figura 2.16b, donde un nodo R desbordado con cuatro objetos o_1, o_2, o_3 y o_4 necesita seleccionar un objeto para la reinserción forzada. Si establecemos el horizonte temporal a h = 2, o_2 no decrece la extensión espacial o de velocidad de R. Una opción mucho mejor podrían ser o_1 u o_3, que reducirían la extensión espacial y de velocidad de R.



(a) Los nodos intermedios 1 y 2 y los nodos hoja a, b c y d. El objeto o_1 se va a insertar

(b) Objetos o_1, o_2, o_3 y o_4 con su MBR. La eliminación de o_2 no reduce al MBR de ninguna manera

Figura 2.16. Inserción en TPR*-tree.

Motivados por esta observación, las entradas seleccionadas para la reinserción forzada serán aquellas que reduzcan el área de barrido del MBR del nodo desbordado. La aproximación más sencilla es ordenar todas las entradas a lo largo de todas las dimensiones y direcciones, y hallar el beneficio de cada ordenación calculando el decrecimiento en el área de barrido si los primeros λ percentiles fueran eliminados para su reinserción. Las entradas serían entonces ordenadas inversamente al orden con el mayor beneficio, y las primeras λ entradas serían seleccionadas para la reinserción. Sin embargo, esto implica ordenar cada entrada 4×d veces, donde d es la dimensión del espacio de datos. Si asumimos que los datos en los nodos hoja están distribuidos uniformemente, la extensión de un nodo hoja R en la i-ésima dimensión tras eliminar las primeras λ entradas se puede estimar como $[R_{iL}, R_{iR} - \lambda(R_{iR} - R_{iL})]$, donde las extensiones en las otras dimensiones resultan ser aproximadamente las mismas. En el caso de nodos intermedios, las entradas probablemente no estén distribuidas uniformemente y por tanto están ordenadas a lo largo de cada dimensión espacio-temporal en ambas direcciones. Asumiendo esto, podemos estimar la extensión del MBR TP de un nodo hoja tras eliminar las primeras λ entradas sin ordenar las entradas. El algoritmo *ReinsercionForzadaRevisada* se muestra a continuación.

Algoritmo ReinsercionForzadaRevisado(nodo N)
/* Recibe un (sub)árbol TPR* desbordado cuya raíz es N */
1. **si** *N* es nodo hoja /* si es hoja estimamos el decremento del área de barrido para cada ordenación asumiendo que los datos están uniformemente distribuidos en los nodos */
2. **para** $i = 1, \ldots, 2d$
   /* mbrReducido es un rectángulo parametrizado en el tiempo d-dimensional que almacena las estimaciones de los MBRs de N si eliminamos las primeras λ entradas en la *i*-ésima dimensión */
3. $mbrReducido_i = [N.mbr_{iL}, N.mbr_{iR} - (\lambda(N.mbr_{iR}, N.mbr_{iL}))]$
4. $mbrReducido_k = [N.mbr_{kL}, N.mbr_{kR}], \forall k \neq i$
   /* el beneficio de eliminar las primeras λ entradas en la dimensión *i* se define como la cantidad de decremento en el área de barrido entre el MBR de *N* y el MBR de mbrReducido */
5. $beneficio_i = Area(SR(N.mbr),[CT, CT+h]) - Area(SR(mbrReducido),[CT, CT+h])$
   /* *D* es la dimensión con mayor beneficio */
6. $D = k$, donde $\forall i = 1, \ldots, 2d$, $max\{beneficio_i\} = beneficio_k$
7. $\{E\}$= entradas de *N* ordenadas en la dimensión *D*
8. insertar las primeras λ entradas de $\{E\}$ en el nivel *N.nivel*
9. **si_no** /* si no es nodo hoja no podemos estimar el tamaño de mbrReducido, y tenemos que ordenar las entradas en el nodo en orden ascendente y en orden descendente para cada una de las 2d dimensiones */
10. **para** $i = 1, \ldots, 2d$
11. $\{E_{ia}\}$= entradas de *N* en orden ascendente en la dimensión i, eliminando las primeras λ entradas
12. $\{E_{id}\}$= entradas de *N* en orden descendente en la dimensión i, eliminando las primeras λ entradas
    /* el beneficio se calcula para las dos ordenaciones */
13. $beneficio_{ia} = Area(SR(N.mbr),[CT, CT+h]) - Area(SR(E_{ia}.mbr),[CT, CT+h])$
14. $beneficio_{id} = Area(SR(N.mbr),[CT, CT+h]) -$

Area(SR(E_{id}.mbr),[CT, CT+h])
/* buscar el mayor beneficio para las dos ordenaciones */
15. $D_a = k$, donde $\forall\, i = 1, \ldots, 2d$, $max\{beneficio_{ia}\} = beneficio_{ka}$
16. $D_d = k$, donde $\forall\, i = 1, \ldots, 2d$, $max\{beneficio_{id}\} = beneficio_{kd}$
17. **si** $D_a > D_d$
18. $\{E\}$= entradas de N ordenadas en la dimensión D en orden ascendente
19. **eliminar** las primeras λ entradas de $\{E\}$ e insertarlas en el nivel *N.nivel*
20. si_no
21. $\{E\}$= entradas de N ordenadas en la dimensión D en orden descendente
22. **eliminar** las primeras λ entradas de $\{E\}$ e insertarlas en el nivel *N.nivel*

fin ReinsercionForzadaRevisado

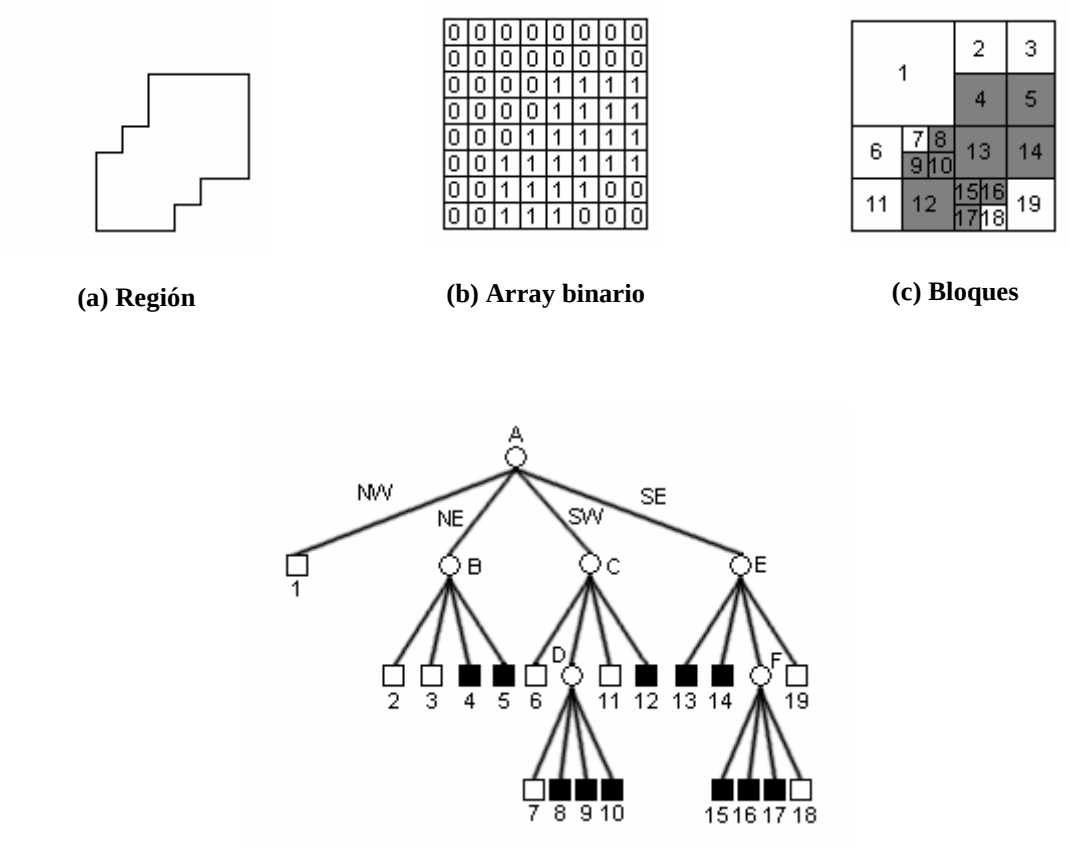
2.8 Linear Region Quadtrees almacenados en TPR*-Tree

Como hemos visto, existen diversas estructuras para organizar datos espaciales. Un quadtree [Sam90] es una estructura jerárquica que se basa en una descomposición recursiva del espacio. La resolución de la descomposición puede estar determinada de antemano o por la naturaleza de los datos de entrada.

Un quadtree característico es el llamado *region quadtree*, o simplemente *quadtree*, usado para representar regiones bidimensionales binarias. Este divide sucesivamente al array que contiene a la imagen en cuatro cuadrantes de igual tamaño. Si todo el array no está compuesto sólo por 0 o solo por 1, entonces se divide en cuatro cuadrantes. A su vez, si cada uno de estos cuadrantes no está compuesto sólo por 0 o solo por 1, entonces se dividen en 4 subcuadrantes. El proceso se repite hasta que todas las regiones están formadas totalmente por 1 (pertenecen completamente a la región) o por 0 (están completamente fuera de la región). Podemos decir que la resolución del quadtree está determinada por los datos de entrada.

Como ejemplo podemos ver la región en mostrada en la figura 2.17a representada por el array de $2^3 \times 2^3$ de la figura 2.17b. Los píxeles que están dentro de la región tienen valor 1 mientras que los píxeles que están fuera tiene valor 0.

El nodo raíz se corresponde con el array completo y cada hijo de un nodo representa un cuadrante (designados en orden NW, NE, SW y SE) de la región representada por ese nodo. Los nodos hoja se corresponden con las regiones que no necesitan más subdivisiones para ser representadas. Si un nodo hoja tiene todos sus píxeles con valor 1 (por tanto pertenece a la región) diremos que es un nodo negro, si tiene todos sus píxeles con valor 0 (por tanto están fuera de la región) diremos que es un nodo blanco. Los nodos no hoja contiene tanto 1 como 0 y se denominan nodos grises. Para una imagen de $2^n \times 2^n$ el nodo raíz se encuentra en el nivel n mientras que un nodos de nivel 0 se corresponde con un píxel en la imagen.



(a) Región **(b) Array binario** **(c) Bloques**

(d) Quadtree

Figura 2.17. Ejemplo de quadtree.

Representamos el quadtree mediante un *linear quadtree* [TVM98], que consiste en una lista formada por los nodos hoja negros, es decir, solo los nodos que se corresponden con la región. Cada nodo se codifica con un número en base 4 o 5 que llamamos *locational code* (existen distintas versiones) y que representa la secuencia de dígitos que determina el camino para llegar desde el nodo raíz hasta el nodo hoja a través del árbol.

El *FD linear quadtree* representa una imagen mediante la colección de los *FD codes* de los nodos hoja del quadtree que representa la imagen. Partiendo de una imagen cuadrada de lado 2^n, el FD code de un nodo hoja de lado 2^k se representa mediante dos partes. La primera, el *locational code*, se compone de *n* dígitos en base 4, donde los dígitos 0, 1, 2 y 3 representan las direcciones NW, NE, SW y SE (o los cuadrantes superior izquierdo, superior derecho, inferior izquierdo e inferior derecho) respectivamente. Como un nodo de lado 2^k tiene altura *k*, con $0 \leq k \leq n$, entonces los primeros $n - k$ dígitos representan el camino desde la raíz del árbol hasta el nodo. Los restantes k dígitos son iguales a 0. El número obtenido es el mismo que se obtiene si se entrelazan los bits que forman las coordenadas *x* e *y* del píxel de la esquina superior izquierda del nodo (Z-order). La segunda parte tiene $log_2(n+1)$ dígitos que representan la altura del nodo, es decir, el valor de *k*.

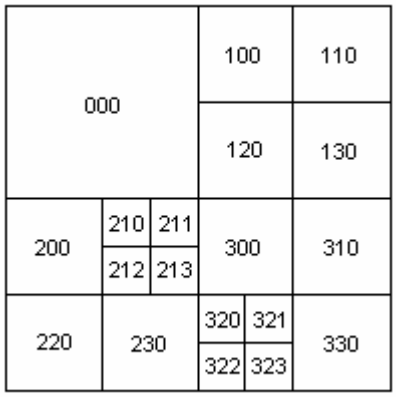


Figura 2.18. Locational code de los bloques correspondientes al quadtree de la figura 2.17.

Por ejemplo, tomemos el locational code 130 en la figura 2.18 correspondiente al bloque 5 en la figura 2.17c. Es un nodo de lado 2^1, por tanto su altura es $k = 1$. Como la imagen es de tamaño $2^3 \times 2^3$, $n = 3$. Así pues, los primeros $n - k$ dígitos, esto es, los primeros $3 - 1 = 2$ dígitos, cuyos valores son 1 y 3, indican el camino a recorrer desde la raíz del árbol hasta el nodo 5 y los k restantes tienen valor 0. Además los dígitos 130 expresados en base binaria son 01 11 00, que desentrelazados quedan como 010 y 100, los cuales corresponden a 2 y 6, que son las coordenadas para el punto superior izquierdo del bloque 5 en los ejes y y x.

A partir de los FD codes de los nodos hoja negros (que forman la región que queremos representar) obtenemos los MBRs que los cubren, puesto que tenemos la información necesaria para determinar tanto su posición (la esquina superior izquierda o NW) como su tamaño (directamente relacionado con la altura del nodo en el árbol). Entonces podemos insertar estos MBRs en un TPR*-tree junto con su velocidad para reasentar una imagen móvil.

Para realizar esta operación se sigue el mismo proceso que en [Cam07], con la diferencia de que una vez obtenido el MBR de un bloque correspondiente con un nodo hoja negro (una parte de la región representada), este se inserta en un TPR*-tree junto con su velocidad. Puesto que estamos considerando imágenes con movimiento de traslación pura (sin deformación) todos los nodos se mueven con la misma velocidad y además, para cada dimensión, las velocidades de los extremos superior e inferior de cada nodo también son iguales. El valor de la velocidad debe ser suministrado al sistema junto con la imagen.

Tras procesar la imagen de la figura 2.17 e insertar los FD locational codes de sus nodos negros e insertarlos en un TPR*-tree con m = 2 y M = 3 obtenemos el árbol de la figura 2.19.

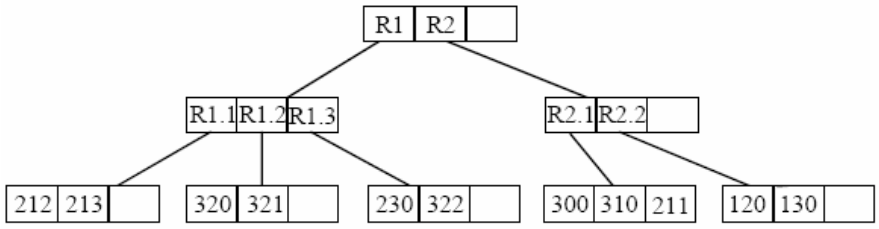


Figura 2.19. FD Quadtree correspondiente a la región de la figura 2.17a mediante TPR*-tree.

Capítulo 3

CONSULTAS

3.1 Introducción

Existen muchos tipos de consultas espaciales: encontrar al vecino más próximo, consultas de similitud, consultas de rango, consultas basadas en el contenido, join espacial, join basado en distancia, etc.

Las *consultas predictivas* son aquellas que responden a preguntas sobre las posiciones actuales y futuras de los objetos móviles. Una consulta predictiva especifica una condición espacial y un intervalo de tiempo futuro, y devuelve el conjunto de objetos que satisfacen la condición espacial en algún instante del intervalo (por ejemplo, *encontrar todos los barcos bajo la influencia de una tormenta en los próximos 10 minutos*). Si el intervalo corresponde a un único instante futuro, entonces la consulta es llamada consulta instantánea, si no es una consulta de intervalo [Cam07, CTVM08].

Las consultas espaciales convencionales carecen de sentido en entornos dinámicos, ya que los resultados devueltos pueden ser invalidados conforme los objetos se mueven, y por tanto no pueden ser usadas como consultas predictivas. Para poder responder a las consultas predictivas en estos entornos existen dos tipos de consultas: las *consultas parametrizadas en el tiempo* y las *consultas continuas* [TP03].

Una *consulta parametrizada en el tiempo* devuelve el resultado actual de la consulta en el momento en que la consulta es realizada, el tiempo de expiración (es decir, el tiempo en que, debido al movimiento de los objetos o de la propia consulta, el resultado actual dejará de ser válido), y los objetos que producen la expiración (esto es, los objetos que dejan de o comienzan a pertenecer al resultado de la consulta). Un ejemplo de respuesta para la versión parametrizada en el tiempo de la consulta *encontrar todos los barcos bajo la influencia de una tormenta en los próximos 10 minutos* podría ser de la forma <{A, B, C}, 5, A>, indicando que en el momento actual los barcos A, B y C se encuentran bajo la influencia de la tormenta y que esta situación será válida hasta dentro de 5 minutos, en que el resultado cambiará porque A dejará de estar bajo la influencia de la tormenta. Las consultas parametrizadas en el tiempo son importantes como métodos independientes y además son las componentes primitivas sobre las que se pueden construir consultas continuas más complejas.

Una *consulta continua* debe ser evaluada y actualizada constantemente debido a los cambios que se producen tanto en la propia consulta como en los objetos de la base de datos, que son de naturaleza continua. El resultado de una consulta continua está compuesto de tuplas de la forma <resultado, intervalo>, donde cada resultado se devuelve junto con el intervalo de tiempo en que es válido. Un ejemplo de respuesta para la versión continua de la consulta *encontrar todos los barcos bajo la influencia de una tormenta en los próximos 10 minutos* podría ser de la forma {<{A, B, C}, [0, 5)>, <{B, C}, [5, 7)>, <{B, D}, [7, 10)>}, indicando que los barcos A, B y C se encontrarán bajo la influencia de la tormenta en el intervalo [0, 5), los barcos B y C se encontrarán bajo la influencia de la tormenta en el intervalo [5, 7) y los barcos B y D se encontrarán bajo la influencia de la tormenta en el intervalo [7, 10).

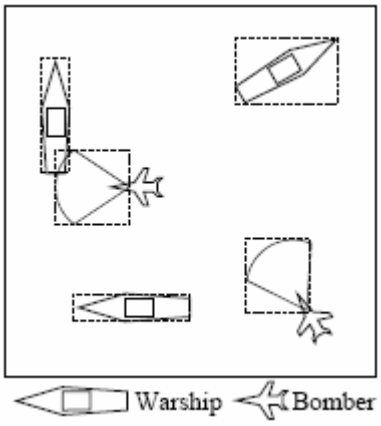


Figura 3.1. Consulta espacio-temporal.

En la figura 3.1 podemos ver un posible escenario de consulta. El primer conjunto de datos contiene las posiciones de una flota de barcos, representada por sus MBRs. El segundo conjunto contiene las posiciones de un escuadrón de bombarderos, representados por el MBR del área sobre la que cada avión puede abrir fuego. El objetivo de la consulta es determinar qué barco se encuentra a tiro para cada bombardero en cada instante (un join continuo).

El resto de este capítulo se organiza de la siguiente manera: en la sección 3.2 se definen las consultas espaciales y se describen algunas de ellas, en la sección 3.3 se describen las consultas espacio-temporales, en la sección 3.4 se describen las consultas de ventana en entornos dinámicos y se desarrollan distintos algoritmos para responder a ellas, en la sección 3.5 se describen las consultas de los k vecinos más próximos en entornos dinámicos y se desarrollan distintos algoritmos para responder a ellas, en la sección 3.6 se describen las consultas de join espacial en entornos dinámicos y se desarrollan distintos algoritmos para responder a ellas, por último, en la sección 3.7 se describen otras consultas en entornos dinámicos y se plantean algoritmos para responder a algunas de ellas.

3.2 Consultas espaciales

Una *consulta espacial* es un conjunto de objetos que satisfacen una condición espacial. Una condición espacial está determinada por las distintas relaciones espaciales que pueden darse entre objetos espaciales, como las relaciones topológicas de adyacencia, solape, inclusión, etc; y las relaciones métricas como la distancia [Cor02].

Consulta exacta. Dado un conjunto de objetos espaciales P y un objeto O' definidos por sus MBRs, encontrar todos los objetos O de P con el mismo MBR que O'.

Consulta para la localización de un punto. Dado un conjunto de objetos espaciales P y un punto p, encontrar todos los objetos O de P que se solapan con p.

Consulta de ventana. Dado un conjunto de objetos espaciales P y una región W definidos por sus MBRs, encontrar todos los objetos O de P que tienen al menos un punto en común con W.

Consulta de solape. Dado un conjunto de objetos espaciales P y un objeto O' definidos por sus MBRs, encontrar todos los objetos O de P que tienen al menos un punto en común con O'.

Consulta del vecino más próximo. Dado un conjunto de objetos espaciales P y un objeto O' definidos por sus MBRs, encontrar el objeto O de P con una menor distancia desde O'.

Join espacial. Dados dos conjuntos de objetos espaciales P y Q, y una condición espacial, encontrar todos los pares de objetos (O, O') pertenecientes al producto cartesiano de P × Q que cumplan la condición espacial.

Las consultas descritas anteriormente están definidas en entornos estáticos. Esto quiere decir que el transcurso del tiempo no tiene influencia en el resultado de las consultas, ya que las características espaciales de los objetos (sus MBRs), así como las relaciones entre estas permanecen constantes en todo momento.

3.3. Consultas espacio-temporales

En entornos dinámicos las características espaciales de los objetos (sus MBRs) varían con el tiempo, y por tanto las relaciones entre ellas también lo hacen. Las *consultas predictivas* son aquellas que responden a preguntas sobre las posiciones actuales y futuras de los objetos móviles. Una consulta predictiva especifica una condición espacial y un intervalo de tiempo futuro, y devuelve el conjunto de objetos que satisfacen la condición espacial en algún instante del intervalo. Ahora debemos tener en cuenta no solo las posiciones de los objetos, si no también los momentos en que se realizan las consultas, ya que las unas dependen de los otros.

Así surge una primera distinción entre consultas espacio-temporales predictivas: las consultas instantáneas y las consultas de intervalo [SJLL00].

Una *consulta instantánea* se realiza para un instante de tiempo t, de manera que el resultado buscado consiste en el conjunto de puntos que cumplen la consulta en t.

Una *consulta de intervalo* se realiza para un intervalo de tiempo $[t_1, t_2]$, de manera que el resultado buscado consiste en el conjunto de puntos que cumplen la consulta en algún instante del intervalo.

Una consulta instantánea es un caso particular de consulta de intervalo en la que $t_1 = t_2$.

En la figura 3.2 podemos ver ejemplos de consultas instantáneas y de intervalo (consultas de ventana en este caso) en un espacio unidimensional.

Dado t_l, el tiempo de lanzamiento de la consulta, las características de los objetos (posición y velocidad) dependen de t_l, ya que varían conforme el tiempo avanza. Por ejemplo, la trayectoria del objeto o_1 es una para $t_l < 1$, otra para $1 \leq t_l < 3$ y otra para $3 \leq t_l$; así que la respuesta a Q1 será o_1 si $t_l(Q1) < 1$ y ninguno si $1 \leq t_l(Q1)$. Por este motivo las consultas realizadas para un futuro más o menos lejano carecen de valor, ya que las posiciones predichas en t_l son cada vez menos precisas conforme avanza el tiempo.

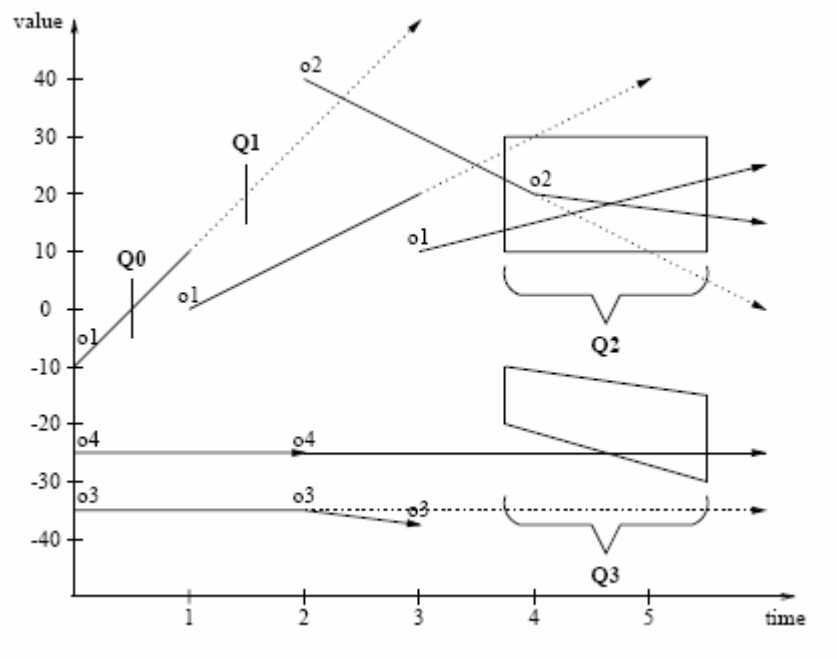


Figura 3.2. Consultas espacio-temporales en un espacio unidimensional.

Las consultas de intervalo devuelven como resultado el conjunto de objetos que cumplen la condición de la consulta en algún instante del intervalo de consulta, de modo que en la figura 3.2 el resultado de Q3 será o_4, aunque no solape con la consulta durante todo el intervalo, sino que sólo en una parte de éste. Este resultado, aún siendo correcto, carece de exactitud, ya que realmente no sabemos a partir de qué instante comienza el solape ni cuando termina. Esta falta de exactitud es una de las causas por las que aparecen las consultas parametrizadas en el tiempo y las consultas continuas, que requieren una evaluación continua debido a los cambios en la condición de la consulta o en la base de datos.

3.3.1 Consultas parametrizadas en el tiempo

Las *consultas parametrizadas en el tiempo* (en adelante consultas TP) son las primitivas básicas sobre las que se construyen las consultas continuas. Las consultas TP son también importantes como métodos independientes, ya que devuelven los resultados con un periodo de validez, lo que las hace útiles en la práctica [TP02].

Una consulta TP devuelve un resultado de la forma **<R,T,C>**, donde **R** es el resultado de la consulta en el instante de inicio de la consulta, **T** es el tiempo de expiración (o validez) de R y **C** es el conjunto de objetos que afecta a R en T, es decir, el cambio que

causa la expiración del resultado. A R se le llama componente convencional de la consulta, mientras que a (T, C) se le llama componente parametrizada en el tiempo. Un ejemplo de consulta TP sería *obtener los barcos de una flota que se encuentran sobre un banco de peces en los próximos 10 minutos*. El resultado a esta consulta podría ser <{A, B}, 4, {A}>, indicando que los barcos A y B se encuentran sobre el banco y que a partir del minuto 4 el barco A dejará de estar sobre el banco.

La componente convencional de una consulta TP cambia porque hay objetos que *influyen* en su validez. Denotamos el *tiempo de influencia* de un objeto o respecto a una consulta q como $\mathbf{T_{inf}(o,q)}$. El tiempo de expiración de una consulta TP es el menor tiempo de influencia de todos los objetos. Por tanto, obtener la componente parametrizada en el tiempo de una consulta consiste en buscar el objeto con menor tiempo de influencia respecto a la consulta. Esto es equivalente a realizar una búsqueda del vecino más próximo usando como métrica de distancia el tiempo de influencia $T_{inf}(o,q)$. El objetivo es encontrar los objetos (C) con menor tiempo de influencia (T) que son los que generarán el cambio añadiéndose o sustrayéndose del resultado (R).

La estrategia seguida para resolver el problema de la búsqueda del vecino más próximo (es decir, para obtener los objetos con menor T_{inf}) ha sido de tipo *branch-and-bound* (BaB). La estrategia BaB (ramificación y poda) se aplica al recorrido de las estructuras de índice arbóreas en las que se basan los métodos de acceso y consiste en recorrer el índice descartando los caminos (o podando las ramas) que sabemos de antemano que no nos conducen a la solución óptima.

La primera aplicación de la estrategia BaB a la búsqueda del vecino más próximo utilizando R-trees fue propuesta por Roussopoulos [RKV95]. El algoritmo define la métrica *mindist* que devuelve la menor distancia entre un punto y cualquiera de los objetos que puedan estar en un subárbol de una de sus entradas (figura 3.3).

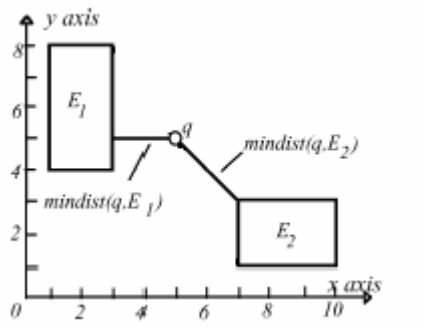


Figura 3.3. Métrica *mindist* entre un punto y una entrada intermedia.

Comenzando desde la raíz, se realiza un recorrido en profundidad (depth first o DF) ordenando las entradas de cada nodo según su *mindist*. Este proceso se repite recursivamente hasta el nivel de nodos hoja, donde están los candidatos a vecino más próximo. En la fase de retroceso (backtracking) se podan las entradas con una *mindist* mayor a la del vecino más próximo encontrado hasta el momento, puesto que sabemos que ninguno de sus objetos podrá tener una distancia menor a la de éste.

Papadopoulos and Manolopoulos [PM97] demostraron que la aproximación DF es subóptima, y que para realizar la consulta del vecino más próximo sólo es necesario visitar las entradas que estén dentro del *círculo de vecindad* (con centro en el punto de

consulta y radio igual a la distancia entre el punto de consulta y su vecino más próximo).

Hjaltason and Samet [HS99] propusieron un nuevo algoritmo para encontrar el vecino más próximo usando R-trees. Éste realiza un recorrido primero el mejor (best-first o BF) y mantiene un heap en el que se almacenan las entradas en orden creciente de su *mindist*. Es un algoritmo no recursivo que comienza por introducir en el heap las entradas del nodo raíz junto con su *mindist*. En cada iteración toma el nodo apuntado por la entrada en la raíz del heap e introduce sus entradas en el heap junto con su *mindist*. La primera entrada a nivel de nodos hoja apunta al primer candidato a vecino más próximo. El algoritmo termina cuando la entrada en la raíz del heap tiene una *mindist* mayor a la del vecino más próximo encontrado hasta ese momento o el heap está vacío. Éste es un algoritmo óptimo, ya que solamente visita los nodos necesarios para obtener al vecino más próximo.

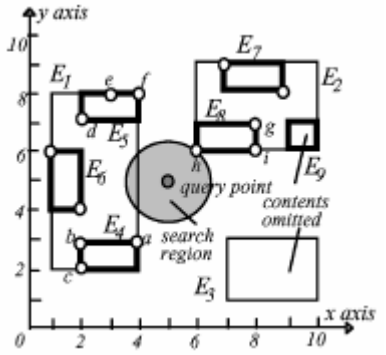


Figura 3.4. Círculo de vecindad.

La estrategia BaB se aplica también a la búsqueda de los pares más cercanos para dos conjuntos de datos. Existen varios algoritmos basados en recorridos DF y BF. La principal diferencia es que ahora *mindist* se define también para un par de entradas como la mínima distancia entre dos objetos que pertenezcan a los subárboles de estas.

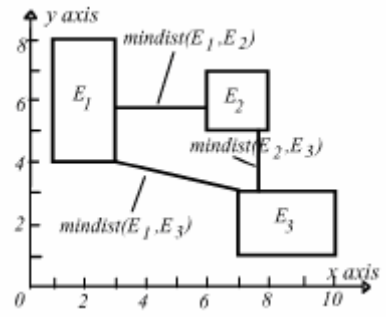


Figura 3.5. Métrica mindist entre entradas intermedias.

Si la *mindist* de un par de entradas es mayor que la del par más próximo encontrado hasta el momento entonces ningún par de objetos de estas entradas podrá tener menor distancia que el par más próximo actual.

3.3.2 Consultas continuas

Una *consulta continua* devuelve resultados de la forma $\{<R_1,T_1>,<R_2,T_2>,...,<R_m,T_m>\}$ donde $<R_i,T_i>$ es el resultado i-ésimo (R_i) con su tiempo de expiración (T_i) y m es el número de cambios en el resultado [TP03]. Siguiendo con el mismo ejemplo usado para la consulta TP, *obtener los barcos de una flota que se encuentran sobre un banco de peces en los próximos 10 minutos*. El resultado a esta consulta podría ser {<{A,B},4>,<{B},5>,<{A,B,C},1>}, indicando que sobre el banco se encuentran los barcos A y B los próximos 4 minutos, después sobre el banco se encontrará el barco B durante los siguientes 5 minutos y por último sobre el banco se encontrarán los barcos A, B y C durante 1 minuto.

Una consulta continua puede tener diversas condiciones de terminación: número de cambios en el resultado, un intervalo de tiempo (como en el ejemplo y que es la implementada en este proyecto) o hasta que la ventana de consulta alcance una determinada posición, entre otras.

Como se ha indicado anteriormente, una consulta continua se puede responder realizando las consultas TP de forma repetitiva hasta alcanzar la condición de terminación. Esta aproximación repetitiva tiene un coste de ejecución directamente relacionado con el resultado de la consulta, ya que por cada cambio será necesaria la ejecución de una consulta TP. Sin embargo, a excepción de la primera consulta TP, para el resto no es necesario calcular la componente convencional R. Por ejemplo, una vez alcanzado el resultado de la primera consulta TP $<R_1,T_1,C_1>$, para la segunda solamente necesitaríamos obtener T_2 y C_2, ya que R_2 se obtiene aplicando C_1 sobre R_1. Es decir, las consultas TP siguientes solamente necesitan calcular T y C, ya que R se obtiene aplicando C de manera incremental. Esto representa un ahorro significativo de cálculos y ha motivado al desarrollo de algoritmos de una pasada (*single-pass*) que responden a consultas continuas con un único recorrido de la estructura de índice.

Siguiendo con la analogía que se estableció anteriormente entre la consulta del vecino más próximo y la consulta TP, podemos deducir que responder a una consulta continua es equivalente a obtener los k vecinos más próximos tomando como métrica el tiempo de influencia. Para ciertas consultas (consulta de ventana y join) el tiempo de influencia de los objetos no depende del resultado, por tanto puede ser calculado independientemente de este. De esta forma obtener los primeros k cambios en el resultado $\{<R_1,T_1>,<R_2,T_2>,...,<R_m,T_m>\}$ será equivalente a calcular los k vecinos más próximos tomando el tiempo de influencia como métrica de distancia y k = m.

Para otras consultas, como los k vecinos más próximos, el tiempo de influencia de los objetos depende de los resultados intermedios y no puede ser calculado de antemano, por lo que no es posible aplicar este método y se ha de tomar un nuevo enfoque para desarrollar algoritmos single-pass que respondan a las consultas. Esto se describirá más adelante.

3.4 Consultas de ventana en entornos dinámicos

La consulta de ventana es una de las consultas espaciales más utilizadas ya que puede ser aplicada para resolver un gran número de problemas. Por tanto el estudio de su uso en entornos dinámicos es de gran interés.

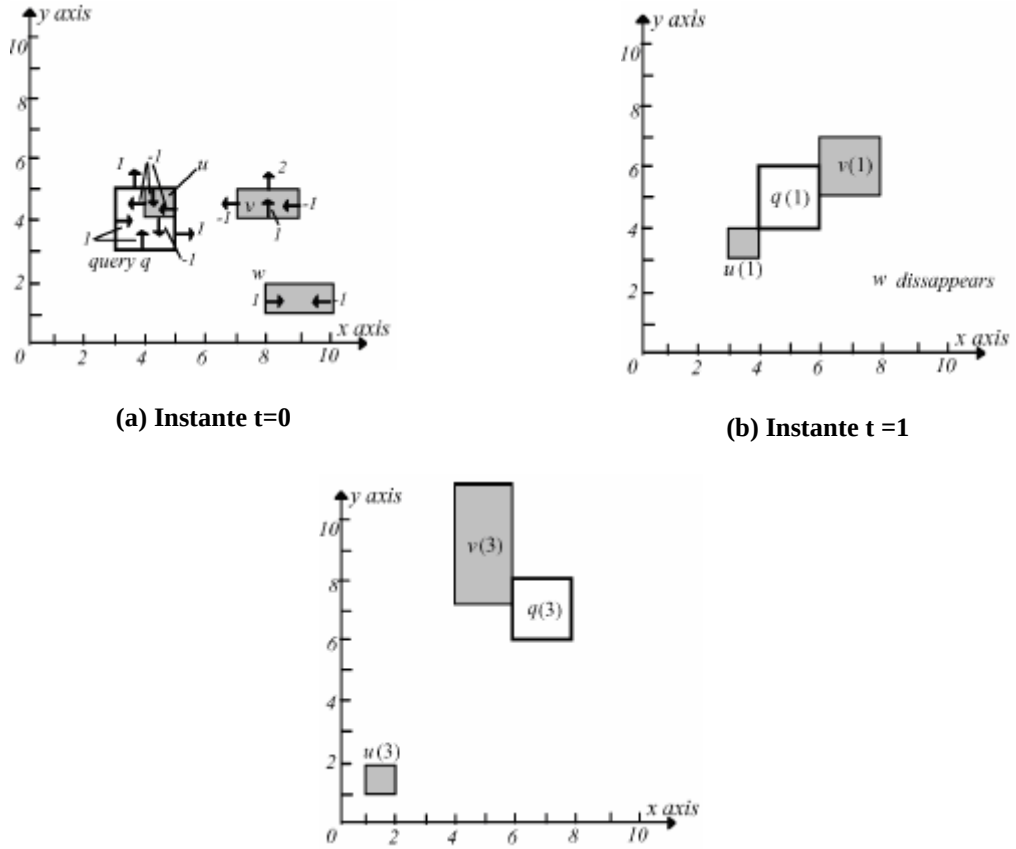
Una consulta de ventana recibe como parámetros una ventana de consulta, un conjunto de objetos y un intervalo de tiempo, devolviendo como resultado los objetos del conjunto que solapen con la ventana en algún momento del intervalo.

3.4.1 Consulta de ventana TP

Una consulta de ventana TP recibe como parámetros una ventana de consulta, un conjunto de objetos y un intervalo de tiempo y devuelve como resultado los objetos del conjunto que solapen con la ventana al inicio del intervalo (R), el tiempo de expiración de este resultado (T) y los objetos que causan la expiración del resultado (C).

Siguiendo con la filosofía propuesta, responder a la consulta de ventana TP consiste en obtener el objeto con menor tiempo de influencia (T_{inf}). Por tanto debemos definir la métrica *tiempo de influencia* para una consulta de ventana y entonces aplicar la estrategia BaB para obtener el objeto que la minimice.

Para encontrar el tiempo de influencia de un objeto o respecto de una consulta q $T_{inf}(o,q)$ debemos considerar el periodo de intersección $[T_s, T_e)$, que es el intervalo de tiempo durante el que o y q intersecan. Se pueden presentar dos casos: si o y q intersecan actualmente entonces $T_{inf}(o,q)$ será el instante en que dejen de intersecar, es decir, el fin del periodo de intersección T_e; si o y q no intersecan actualmente entonces $T_{inf}(o,q)$ será el instante en que comiencen a intersecar, es decir, el inicio del periodo de intersección T_s [TP02].



(a) Instante t=0

(b) Instante t =1

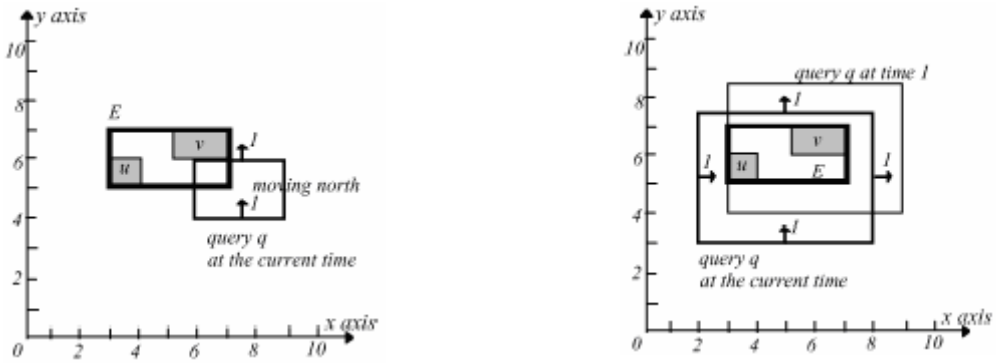
(c) Instante t=3

Figura 3.6. Tiempo de influencia de un objeto con respecto a una consulta.

En la figura3.6 podemos ver una ventana de consulta *q* y tres objetos móviles *u*, *v* y *w* en los instantes 0, 1 y 3. Vemos que $T_{inf}(u,q) = T_{inf}(v,q) = 1$. El objeto *w* desaparece en el instante 1, hecho que debe ser tenido en cuenta en el cálculo del periodo de intersección.

También debemos definir el tiempo de influencia de una entrada de la estructura de índice (en este caso TPR*-tree) E con respecto a una consulta q para poder aplicar la estrategia BaB y podar las ramas que no contengan candidatos a ser la solución. Éste representa una cota inferior para el tiempo de influencia de todos los objetos en el subárbol de E. De nuevo se presentan dos situaciones. Si E y q no intersecan $T_{inf}(E,q)$ será T_s, es decir, el momento en el que empiezan a intersecar. Si E y q intersecan entonces debemos distinguir a su vez dos casos: si E incluido en q y si E interseca parcialmente con q.

En el caso de la intersección parcial $T_{inf}(E,q) = 0$, ya que en cualquier momento un objeto en el subárbol de E puede comenzar o dejar de intersecar con q (figura 3.7a). En el caso de la inclusión $T_{inf}(E,q)$ será el momento en que E y q comiencen a intersecar parcialmente, es decir, cuando E deje de estar incluido en q, ya que hasta ese momento todos los objetos en el subárbol de E solapan con q y no pueden influenciar el resultado (figura 3.7b).



(a) E interseca parcialmente con q

(b) E está contenido en q

Figura 3.7. Tiempo de influencia de una entrada intermedia con respecto a una consulta.

Sea o un objeto en un espacio n-dimensional. Denotamos su MBR (VBR) por $\{[o_{1L}, o_{1R}],\ldots, [o_{nL}, o_{nR}]\}$ ($\{[o.V_{1L}, o.V_{1R}],\ldots, [o.V_{nL}, o.V_{nR}]\}$) y su proyección en la dimensión i-ésima por $[o_{iL}, o_{iR}]$ ($[o.V_{iL}, o.V_{iR}]$).

Para ciertos valores de las velocidades puede darse el caso de que un objeto desaparezca. Esto debe ser tenido en cuenta para el cálculo de $[T_s, T_e)$, ya que el tiempo de influencia de un objeto no puede ser mayor que su tiempo de desaparición (el tiempo de desaparición seria una cota superior para el tiempo de influencia). El tiempo de desaparición de un objeto $o.T_{DSP}$ será el menor instante en el que una de las proyecciones del objeto colapse, esto es, el extremo izquierdo o_{iL} se encuentre con el derecho o_{iR}. Para la proyección i-ésima si $o.V_{iR} \geq o.V_{iL}$ entonces T_{iDSP} será infinito, ya que nunca se encontrarán; si no $T_{iDSP} = (o_{iR} - o_{iL})/(o.V_{iL} - o.V_{iR})$. Finalmente $o.T_{DSP} = \min(o.T_{iDSP})$, para i = 1, …, n.

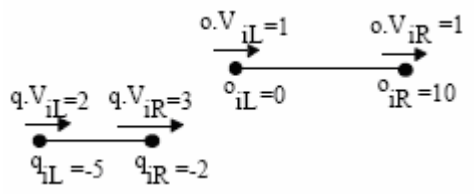
Un objeto o y una consulta q intersecan si lo hacen en todas las dimensiones. Este método calcula primero el periodo de intersección $[T_{is}, T_{ie})$, con i = 1, …, n para cada una de las n dimensiones y a partir de aquí obtiene el periodo de intersección de o y q.

A continuación calculamos el periodo de intersección en la dimensión i-ésima $[T_{is}, T_{ie})$. Primero comenzamos con el cálculo de T_{is}. T_{is} es el comienzo del periodo de intersección, por lo que partimos de que o y q no intersecan actualmente. Pueden darse dos situaciones posibles:

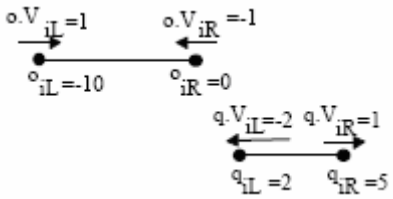
a) o se encuentra a la derecha de q (figura 3.8a). Si o está a la derecha de q, o y q se encontrarán en T_{iLR}, el momento en que el extremo izquierdo de o (o_{iL}) alcance al extremo derecho de q (q_{iR}). El valor de T_{iLR} será infinito si $o.V_{iL} \geq q.V_{iR}$ (ya que nunca se encontrarán) o $(o_{iL} - q_{iR})/(q.V_{iR} - o.V_{iL})$ en caso contrario.

b) o se encuentra a la izquierda de q (figura 3.8b). Este es un caso análogo al anterior. Ahora consideramos T_{iRL}, el instante en que el extremo derecho de o (o_{iR}) se encuentra con el extremo izquierdo de q (q_{iL}). El valor de T_{iRL} será infinito si $o.V_{iR} \leq q.V_{iL}$ (ya que nunca se encontrarán) o $(o_{iR} - q_{iL})/(q.V_{iL} - o.V_{iR})$ en caso contrario. En general, $T_{is} = \min(T_{iRL}, T_{iLR})$ si no desaparecen antes ni o ni q, en cuyo caso T_{is} será infinito.

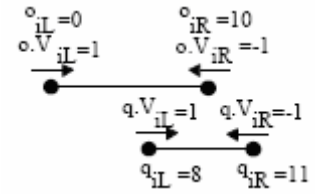
En caso de que o y q intersequen en el instante actual (figura 3.8c) T_{is} será igual al tiempo actual en todas las dimensiones.



(a) *o* a la derecha de *q*



(b) *o* a la izquierda de *q*



(c) *o* interseca con *q*

Figura 3.8. Cálculo del periodo de intersección entre un objeto y una consulta.

En segundo lugar calculamos T_{ie}. Para calcular el fin del periodo de intersección, partimos de que o y q intersecan, y T_{ie} será el momento en que:

a) o pase a estar por completo a la derecha de q. Éste será el momento en que el extremo izquierdo de o alcance al extremo derecho de q, T_{iLR}, que ya hemos obtenido.

b) o pase a estar por completo a la izquierda de q. Éste será el momento en que el extremo derecho de o alcance al extremo izquierdo de q, T_{iRL}, que ya hemos obtenido. En general $T_{ie} = \min(\max(T_{iLR}, T_{iRL}), o.T_{DSP}, q.T_{DSP})$.

Finalmente el periodo de intersección $[T_s, T_e)$ será la intersección de los periodos de intersección para cada dimensión $[T_{is}, T_{ie})$, con i = 1, …, n. Éste es el pseudocódigo del algoritmo:

Calcular_Periodo_Intersección (o, q)
1. $[T_s, T_e) = [0, \infty]$
2. **para cada** dimensión i
3. calcular tiempo de desaparición $o.T_{iDSP}$, $q.T_{iDSP}$
4. $T_{DSP} = \min(o.T_{DSP}, q.T_{DSP})$
5. $T_{iLR} = (o_{iL} - q_{iR})/(q.V_{iR} - o.V_{iL})$
6. **si** $T_{iLR} < 0$ entonces $T_{iLR} = \infty$ /*nunca se encontrarán*/
7. $T_{iRL} = (o_{iR} - q_{iL})/(q.V_{iL} - o.V_{iR})$
8. **si** $T_{iRL} < 0$ entonces $T_{iRL} = \infty$ /*nunca se encontrarán*/
9. **si** $[o_{.iL}, o_{.iR}]$ no interseca con $[q_{.iL}, q_{.iR}]$
10. **si** $\max(T_{iLR}, T_{iRL}) \leq T_{DSP}$
11. $T_{is} = \min(T_{iLR}, T_{iRL})$; $T_{ie} = \max(T_{iLR}, T_{iRL})$
12. **si_no_si** $\min(T_{iLR}, T_{iRL}) \leq T_{DSP} \leq \max(T_{iLR}, T_{iRL})$
13. $T_{is} = \min(T_{iLR}, T_{iRL})$; $T_{ie} = T_{DSP}$
14. **si_no** $T_{is} = T_{ie} = \infty$
15. **si_no** /* $[o_{.iL}, o_{.iR}]$ interseca con $[q_{.iL}, q_{.iR}]$ */
16. $T_{is} = 0$
17. $T_{ie} = \min(T_{iLR}, T_{iRL}, T_{DSP})$
18. $[T_s, T_e) = [T_s, T_e) \cap [T_{is}, T_{ie})$
19. **devolver** $[T_s, T_e)$

fin Calcular_Periodo_Intersección

Continuando con la notación anterior el tiempo de intersección parcial T_{PI} es el menor tiempo de intersección T_{iPI}, es decir, el menor tiempo en el que la proyección i-ésima de la entrada intermedia E comienza a intersecar parcialmente con la proyección i-ésima de la consulta q, para cualquier dimensión i = 1, …, n. Teniendo como situación de partida que E está contenido completamente en q (figura 3.9), este instante puede ser T_{iLL}, el momento en que el extremo izquierdo de E (E_{iL}) alcanza al extremo izquierdo de q (q_{iL}), o T_{iRR}, el momento en que el extremo derecho de E (E_{iR}) alcanza al extremo derecho de q (q_{iR}).

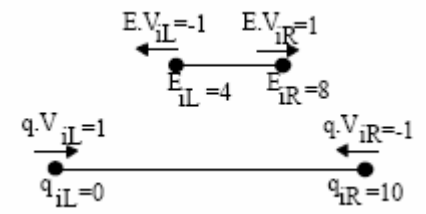


Figura 3.9. Calculo del periodo de intersección para una entrada intermedia contenida en una consulta.

El valor de T_{iLL} será infinito si $E.V_{iL} \geq q.V_{iL}$ (ya que nunca se alcanzarán) o $(E_{iL} - q_{iL})/(q.V_{iL} - E.V_{iL})$ en caso contrario. El valor de T_{iRR} será infinito si $E.V_{iR} \leq q.V_{iR}$ (ya que nunca se alcanzarán) o $(E_{iR} - q_{iR})/(q.V_{iR} - E.V_{iR})$ en caso contrario. También se deben tener en cuenta los tiempos de desaparición de E y q, $E.T_{DSP}$ y $q.T_{DSP}$.

Calcular_T_{PI} (E, q, [T_s, T_e))
1. $T_{PI} = \infty$
2. **para cada** dimensión i
3. $T_{iLL} = (E_{iL} - q_{iL})/(q.V_{iL} - E.V_{iL})$
4. $T_{iRR} = (E_{iR} - q_{iR})/(q.V_{iR} - E.V_{iR})$
5. **si** $T_{iLL} \in [T_s, T_e)$ y $T_{iRR} \in [T_s, T_e)$
6. $T_{iPI} = \min(T_{iLL}, T_{iRR})$
7. **si** $T_{iLL} \in [T_s, T_e)$ y $T_{iRR} \notin [T_s, T_e)$
8. $T_{iPI} = T_{iLL}$
9. **si** $T_{iLL} \notin [T_s, T_e)$ y $T_{iRR} \in [T_s, T_e)$
10. $T_{iPI} = T_{iRR}$
11. **si** $T_{iLL} \notin [T_s, T_e)$ y $T_{iRR} \notin [T_s, T_e)$
12. $T_{iPI} = \infty$
13. $T_{PI} = \min(T_{PI}, T_{iPI})$
14. **devolver** T_{PI}

fin Calcular_T_{PI}

En resumen, la métrica *tiempo de influencia* queda descrita de la siguiente manera:

- $T_{inf}(o,q) = T_s$ si o y q no intersecan.
- $T_{inf}(o,q) = T_e$ si o y q intersecan.
- $T_{inf}(E,q) = T_s$ si E y q no intersecan.
- $T_{inf}(E,q) = 0$ si E y q intersecan parcialmente.
- $T_{inf}(E,q) = T_{PI}$ si E está incluido en q.

Ahora se puede aplicar una estrategia BaB para obtener el vecino más cercano. Gracias a la estructura del TPR*-tree y a la métrica definida sobre el tiempo de influencia, cuando para una entrada intermedia E su $T_{inf}(E,q)$ es mayor que el encontrado hasta ese momento podemos afirmar que también lo serán todos los tiempos de influencia de los objetos en el subárbol de E, y por tanto podemos descartar E y "podarla" del recorrido del árbol.

Pero nuestro objetivo no es únicamente obtener la componente TP de la consulta (T y C), además se debe buscar la componente convencional (R), por lo que el algoritmo también debe visitar las entradas que intersequen con la ventana de consulta.

Por último, debemos distinguir entre los objetos con T_{inf} igual al menor encontrado hasta el momento y los objetos con T_{inf} menor al menor encontrado hasta el momento. Sea T el menor tiempo de influencia encontrado hasta el momento (y por tanto también el T de la componente TC de la consulta). En el primer caso si $T_{inf}(o,q) = T$, entonces o pasaría a formar parte de C y T permanecería inalterado. En el segundo caso, si $T_{inf}(o,q) < T$, entonces C estaría compuesto únicamente por o y T se debería actualizar a $T_{inf}(o,q)$.

3.4.1.1 Versión Depth-First

A continuación mostramos el algoritmo recursivo que recorre el árbol de manera DF. Éste es su pseudocódigo:

Algoritmo TP_WQ_DF (q, nodo actual N)
1. /* en la primera llamada se le pasa el nodo raíz, además
2. inicialmente T=∞, R=Ø, C= Ø */
3. **si** N es nodo hoja
4. **para cada** objeto o en N
5. **si** $T_{inf}(o,q) < T$
6. $C = \{o\}$; $T = T_{inf}(o,q)$
7. **si_no si** $T_{inf}(o,q) = T$
8. $C = C \cup \{o\}$
9. **si** o interseca con q entonces $R = R \cup \{o\}$
10. **si_no** /* N es un nodo intermedio */
11. **para cada** entrada E en N
12. **si** ($T_{inf}(E,q) \leq T$) o (E interseca con q)
13. **TP_WQ_DF**(q,E.hijo)

fin TP_WQ_DF

3.4.1.2 Versión Depth-First con ordenación

Es también un algoritmo recursivo que recorre el árbol de manera DF. La diferencia con el anterior radica en que en los niveles intermedios procesa las entradas de los nodos en orden creciente de sus tiempos de influencia. De esta manera es posible que se evite recorrer algunas ramas ya que comenzaremos por la de menor tiempo de influencia. En el peor de los casos se recorrerán las mismas ramas. Éste es su pseudocódigo:

Algoritmo TP_WQ_DF_ordenacion(q, nodo actual N)
1. /* en la primera llamada se le pasa el nodo raíz, además
2. inicialmente T=∞, R=Ø, C= Ø */
3. **si** N es nodo hoja
4. **para cada** objeto o en N
5. **si** $T_{inf}(o,q) < T$
6. $C = \{o\}$; $T = T_{inf}(o,q)$
7. **si_no**_si $T_{inf}(o,q) = T$
8. $C = C \cup \{o\}$
9. **si** o interseca con q entonces $R = R \cup \{o\}$
10. **si_no** /* N es un nodo intermedio */
11. ordenar las entradas E de N según su $T_{inf}(E,q)$
12. **para cada** entrada E en N
13. **si** ($T_{inf}(E,q) \leq T$) o (E interseca con q)
14. **TP_WQ_DF_ordenacion**(q,E.hijo)

fin TP_WQ_DF_ordenacion

3.4.1.3 Versión Best-First

Éste es un algoritmo iterativo que sigue un recorrido BF del árbol. Como se indicó anteriormente hace uso de un heap. En el heap se almacenan las entradas visitadas junto

con su tiempo de influencia. Cuando la entrada en la raíz del heap tiene un tiempo de influencia mayor al menor encontrado hasta el momento el algoritmo termina. Éste es su pseudocódigo:

Algoritmo TP_WQ_BF(q)
1. inicializar un heap H que acepta pares <clave,entrada>
2. tomar el nodo raíz R
3. **para cada** entrada E en R insertar < $T_{inf}(E,q)$,E> en H
4. **mientras** (H no vacío)
5. extraer raiz <clave,E> de H /* E tiene la menor clave hasta este momento */
6. **si** E apunta a un nodo hoja
7. **para cada** objeto o en E.hijo
8. **si** $T_{inf}(o,q) < T$
9. $C = \{o\}$; $T = T_{inf}(o,q)$
10. **si_no_si** $T_{inf}(o,q) = T$
11. $C = C \cup \{o\}$
12. **si** o interseca con q entonces $R = R \cup \{o\}$
13. **si_no** /* E apunta a un nodo no hoja */
14. **si** ($T_{inf}(E,q) \leq T$) o (E interseca con q)
15. **para cada** entrada E' en E.hijo
16. insertar < $T_{inf}(E',q)$,E'> en H

fin TP_WQ_BF

3.4.2 Consulta de ventana continua

Una consulta de ventana continua recibe como parámetros una ventana de consulta, un conjunto de objetos y un intervalo de tiempo y devuelve como resultado una lista de la forma $\{<R_1,T_1>,<R_2,T_2>,...,<R_m,T_m>\}$ donde R_i es el conjunto de los objetos que solapan con la ventana, T_i el tiempo de validez de R_i y m el número total de cambios. Como se indicó anteriormente el número total de cambios depende de la condición de terminación que se dé a la consulta. En este caso se ha elegido de manera que T_m coincida con el final del intervalo de consulta.

La respuesta a la consulta de ventana continua consiste en obtener los k vecinos más próximos tomando como métrica de distancia el tiempo de influencia. En este caso se deben considerar dos tiempos de influencia: T_{infs} (inicio del periodo de intersección) y T_{infe} (final del periodo de intersección). Esto se debe a que los objetos pueden influenciar el resultado dos veces, cuando comienza el solapamiento con la ventana de consulta y cuando termina. Sea $[T_s, T_e)$ el periodo de intersección del objeto o y la consulta q. Si o y q no intersecan actualmente T_{infs} será T_s, el momento en que comiencen a intersecar y T_{infe} será T_e, el momento en que dejen de intersecar. Si o y q intersecan actualmente T_{infs} será T_e, el momento en que dejen de intersecar y T_{infe} será infinito, ya que el objeto ya no volverá a influenciar el resultado de la consulta. El tiempo de influencia de las entradas intermedias se define igual que en la consulta TP, puesto que su función sigue siendo la de cota inferior para la poda durante el recorrido del árbol.

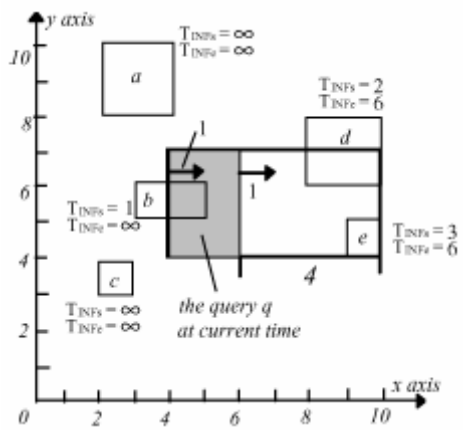


Figura 3.10. Objetos y sus dos tiempos de influencia respecto de una consulta móvil.

En la figura 3.10 vemos un conjunto de objetos junto a sus tiempos de influencia T_{infs} y T_{infe}. Los algoritmos devuelven los objetos en orden creciente de sus tiempos de influencia, que corresponde con la secuencia de cambios en el resultado de la consulta.

Este algoritmo recorre el árbol de manera BF y se ha desarrollado mediante una aproximación *single-pass*. Por tanto, en lugar de obtener todos los conjuntos R_i, lo que hará es calcular el primer conjunto de resultados, R_1, y todos los cambios C_i, de manera que el cálculo de los R_i es inmediato. Para poder obtener R_1 el algoritmo debe visitar todas las entradas que intersequen con la ventana de consulta q. Para el cálculo de C_i el algoritmo funciona de forma muy parecida a los algoritmos de consulta de ventana TP. Cuando un objeto tiene tiempo de influencia T_i se inserta en el conjunto C_i. Si su tiempo de influencia es mayor se crea un nuevo conjunto C_{i+1} y se inserta el objeto en él. Como se ha indicado antes el algoritmo termina cuando el tiempo de influencia de la raíz del heap es mayor que el final del intervalo de consulta. Su pseudocódigo es el siguiente:

Algoritmo CWQ

1. inicializar un heap H que acepta pares <clave,entrada>
2. tomar el nodo raíz R
3. **para cada** entrada E en R insertar < T_{inf}(E,q),E> en H
4. i = -1 /*contador de resultados devueltos*/
5. ultimo_T = -1
6. **mientras** (H no vacío y H.raiz.clave $\leq t_2$) /* t_2 es el fin del intervalo de consulta */
7. extraer raiz <clave,E> de H /* E tiene la menor clave hasta este momento */
8. **si** E apunta aun objeto /* E es una entrada de un nodo hoja */
9. **si** clave > ultimo_T
10. i = i + 1; C_i = {E}; T_i = clave; ultimo_T = clave;
11. **si_no** /* clave es igual a ultimo_T */
12. $C_i = C_i \cup \{E\}$
13. **si** E interseca con q entonces R = R $\cup$ {o}
14. **si_no** /* E apunta a un nodo */
15. **para cada** entrada E' en E.hijo
16. **si** E' apunta aun objeto /* E' es una entrada de un nodo hoja */
17. insertar < T_{infs}(E',q),E'> en H,
18. insertar < T_{infe}(E',q),E'> en H
19. **si_no** /* E apunta a un nodo */
20. insertar < T_{inf}(E',q),E'> en H

fin CWQ

3.5 Consultas de los k vecinos más próximos en entornos dinámicos

La consulta de los k vecinos más próximos es la más estudiada en el contexto del procesamiento de consultas, y por ello es interesante prestar interes a su aplicación en entornos dinámicos.

Una consulta de los k vecinos más próximos en entornos dinámicos recibe como parámetros un objeto de consulta, un conjunto de objetos y un intervalo de tiempo, devolviendo como resultado los k objetos del conjunto que tengan una menor distancia desde el objeto de consulta durante el intervalo.

3.5.1 Consulta de los k vecinos más próximos TP

Una consulta de los k vecinos más próximos recibe como parámetros un objeto de consulta, un conjunto de objetos y un intervalo de tiempo y devuelve como resultado los k objetos del conjunto que tengan una menor distancia desde el objeto de consulta al inicio del intervalo (R), el tiempo de expiración de este resultado (T) y los objetos que causan la expiración del resultado (C).

De nuevo, responder a la consulta de los k vecinos más próximos TP consiste en obtener el objeto con menor tiempo de influencia. Por tanto debemos definir la métrica *tiempo de influencia* para una consulta de los k vecinos más próximos y entonces aplicar la estrategia BaB para obtener el objeto que la minimice [TP03].

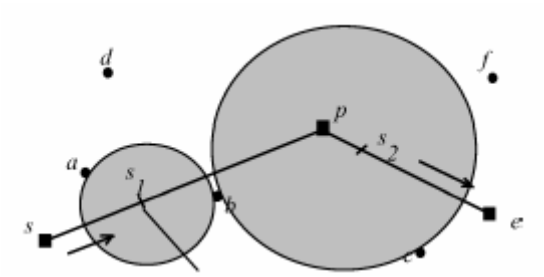


Figura 3.11. Consulta de los k vecinos más próximos en entornos dinámicos.

En la figura 3.11 podemos ver un ejemplo de consulta de los k vecinos más próximos. Para responder a la versión TP de esta consulta observamos que el punto de consulta *q* parte de la posición *s*, momento en el cual su vecino más próximo es el punto *a*. En el instante en que *q* alcanza la posición s_1 el punto *b* comienza a estar más cerca de él que *a*, que es el tiempo de expiración del resultado inicial.

Primero discutiremos el caso más sencillo de un único vecino, es decir, k = 1 y a continuación extenderemos el razonamiento para un valor arbitrario de $k \geq 1$.

Sea q un objeto de consulta puntual y sea q.NN el vecino más próximo al objeto de consulta q en el instante actual. El tiempo de influencia de un objeto o, $T_{inf}(o,q)$, será el menor instante t posterior al instante actual para el que o esté más próximo a q que q.NN. Si llamamos o(t), q(t) y q.NN(t) a las funciones que nos dan las posiciones de o, q y q.NN respectivamente en el instante t, entonces podemos sustituir el párrafo

anterior por la ecuación $||o(t), q(t)|| \leq ||q.NN(t), q(t)||$ y $t \geq t_{act}$ (el operador $|| \ ||$ indica la métrica de distancia, que en este caso es el cuadrado de la distancia euclídea). Si no existe ningún t que cumpla las condiciones, entonces $T_{inf}(o,q)$ será infinito (∞), indicando que o nunca estará más próximo a q que q.NN.

Para el caso de una entrada intermedia E el tiempo de influencia $T_{inf}(E,q)$ indica el menor instante en el que un objeto en el subárbol de E puede estar más cerca de q que q.NN (figura 3.12a). Es una cota inferior que nos permitirá podar el subárbol de E si su tiempo de influencia es mayor al menor encontrado hasta ese momento. En este caso el sistema de ecuaciones a plantear sería $||E(t), q(t)|| \leq ||q.NN(t), q(t)||$ y $t \geq t_{act}$.

Tanto los objetos del conjunto de datos como las entradas intermedias que estamos considerando tienen extensión espacial, es decir, no son puntuales. Así que las funciones de distancia $||o(t), q(t)||$ y $||E(t), q(t)||$ requieren un estudio de las distintas situaciones que se pueden presentar. Por ejemplo, en la figura 3.12b la distancia entre el punto de consulta q y la entrada E definida por los puntos a, b, c y d se debería calcular: (1) respecto de a para los puntos de la trayectoria de q anteriores al punto e, (2) respecto del segmento a-b para los puntos de la trayectoria de q entre los puntos e y f, (3) respecto de b para los puntos de la trayectoria de q entre los puntos f y g, (4) respecto del segmento b-c para los puntos de la trayectoria de q entre los puntos g y h y (5) respecto de c para los puntos de la trayectoria de q posteriores al punto h. A continuación se describe el método utilizado para este cálculo, presentado en [BJKS02].

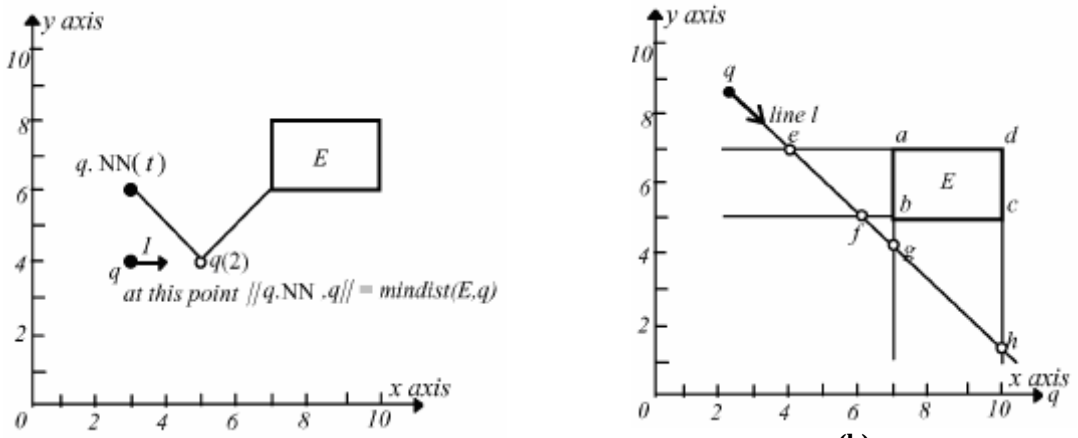


Figura 3.12. Distancia entre una entrada intermedia y una consulta móvil.

Ésta es la fórmula de la distancia cuadrada $d_q(p,t)$ entre dos puntos móviles d-dimensionales $q = (x_1, x_2, \ldots, x_d, v_1, v_2, \ldots, v_d)$ y $p = (y_1, y_2, \ldots, y_d, w_1, w_2, \ldots, w_d)$. En la fórmula x_i e y_i representan las coordenadas en el instante t = 0 cuando no son usados como funciones.

$$d_q(p, t) = \sum_{i=1..d}(x_i(t) - y_i(t))^2 = \sum_{i=1..d}(x_i + v_i t - y_i - w_i t)^2$$

$$= t^2 \sum_{i=1..d}(v_i - w_i)^2 + 2t \sum_{i=1..d}(x_i - y_i)(v_i - w_i) + \sum_{i=1..d}(x_i - y_i)^2$$

Sea $R = ([x_{1L}, x_{1R}], [x_{2L}, x_{2R}], \ldots, [x_{dL}, x_{dR}], [v_{1L}, v_{1R}], [v_{2L}, v_{2R}], \ldots, [v_{dL}, v_{dR}])$ un rectángulo parametrizado en el tiempo. El siguiente algoritmo obtiene una función por

partes que calcula la menor distancia cuadrada $d_q(R, t)$ entre un punto móvil q y un rectángulo R durante el intervalo de tiempo $[t_1, t_2]$.

Algoritmo Distancia(q, R, t_-, t_+)
1. inicializar un conjunto de instantes E
2. E = Ø
3. **para cada** dimensión i = 1, …, d
4. **si** $v_i \neq v_{iL}$ y $t_{iL} = (x_i - x_{iL})/(v_{iL} - v_i) \in [t_1, t_2]$ añadir t_{iL} a E
5. **si** $v_i \neq v_{iR}$ y $t_{iR} = (x_i - x_{iR})/(v_{iR} - v_i) \in [t_1, t_2]$ añadir t_{iR} a E
6. **ordenar** E. Los elementos de E dividen a $[t_1, t_2]$ en como mucho 2d + 1 intervalos.
7. **para cada** uno de los intervalos T_j:
8. $d_q(R, t) = \sum_{i=1..d} d_{q,i}(R, t)$

fin Distancia

donde

$$d_{q,i}(R, t) = \begin{cases} t^2(v_{iL} - v_i)^2 + 2t(x_{iL} - x_i)(v_{iL} - v_i) + (x_{iL} - x_i)^2 & \text{si } \forall t \in T_j (x_i + v_i t \leq x_{iL} + v_{iL} t) \\ t^2(v_{iR} - v_i)^2 + 2t(x_{iR} - x_i)(v_{iR} - v_i) + (x_{iR} - x_i)^2 & \text{si } \forall t \in T_j (x_i + v_i t \geq x_{iR} + v_{iR} t) \\ 0 & \text{en cualquier otro caso} \end{cases}$$

El algoritmo obtiene durante el paso 2 los instantes en los que el punto q cruza con los hiperplanos móviles $x_i = x_{iL}(t)$ y $x_i = x_{iR}(t)$, es decir, las extensiones de los lados opuestos del rectángulo en la dimensión x_i. En la figura 3.12b estos instantes serían aquellos en los que q estuviera en e, f, g y h. Notar que t_{iL} no tiene que ser necesariamente menor que t_{iL}. Durante cada periodo T_j q no cruza con ningún hiperplano. Cuando q está dentro de R la distancia es $d_q(R, t) = 0$.

Para cada periodo T_j tenemos ahora los coeficientes a_j, b_j y c_j, dependientes de las posiciones y velocidades de los objetos y la consulta, para expresar la distancia $d_q(R, t)$ en la forma $a_j t^2 + b_j t + c_j$. Por tanto, podemos expresar el sistema de inecuaciones $||o(t), q(t)|| \leq ||q.NN(t), q(t)||$, $t \geq t_{act}$ (o $||E(t), q(t)|| \leq ||q.NN(t), q(t)||$, $t \geq t_{act}$) de la forma $At^2 + Bt + C \leq 0$, $t \geq t_{act}$. La solución de este sistema nos da un intervalo (que puede ser vacío) durante el que o (ó E) está más cerca de q que q.NN para cada T_j. El menor de estos tiempos será el tiempo de influencia de o (ó de E) respecto de q.

Supongamos ahora que en lugar de un único vecino más próximo tenemos k vecinos q.1NN, q.2NN, …, q.kNN. El tiempo de influencia de un objeto o $T_{inf}(o,q)$ será el menor instante en que o comienza a estar más cerca de q que cualquiera de sus k vecinos más próximos. Para obtener $T_{inf}(o,q)$ calculamos primero el tiempo de influencia T_{inf_i} de o con respecto a cada uno de los vecinos q.iNN (i = 1, …, k) como se ha descrito anteriormente. El tiempo de influencia $T_{inf}(o,q)$ será el menor de los T_{inf_i}. Este mismo método se aplica a las entradas intermedias.

Definido el tiempo de influencia para objetos y entradas intermedias, aplicamos la estrategia BaB para obtener la componente TP del resultado de la consulta. A diferencia de la consulta de ventana TP, en la que en una sola pasada se obtienen las componentes convencional (R) y TP (T, C) del resultado, este algoritmo necesita obtener previamente la componente convencional, ya que los tiempos de influencia de los objetos dependen de los resultados actuales. Para esto se plica un algoritmo normal de cálculo de los k

vecinos más próximos. El es el pseudocódigo del algoritmo BF se muestra a continuación:

Algoritmo k_NN(q)
/* Se le pasan la raíz del árbol */
1. inicializar un heap H que acepta pares <clave,entrada>
2. inicializar una lista NN con k lugares que acepte pares <clave,objeto>
3. **para cada** elemento NN_i de NN
4. NN_i.clave = ∞
5. tomar el nodo raíz R
6. **para cada** entrada E en R
7. insertar < distancia(E,q),E> en H
8. **mientras** (H no vacío y H.raiz.clave < NN_k.clave)
9. extraer raiz <clave,E> de H
10. **si** E es una entrada de nivel hoja
11. **si** clave < NN_k.clave
12. NN_k.clave = clave; NN_k.entrada = E;
13. ordenar NN
14. **si_no** /* E es una entrada de nivel intermedio */
15. **si** clave < NN_k.clave
16. **para cada** entrada E' en E.hijo
17. insertar < distancia(E',q),E'> en H
18. **devolver** NN

fin k_NN

Algoritmo TP_k_NN_BF(q,R)
/* Se le pasa la raíz del árbol. Inicialmente T=∞, C= Ø. R se ha obtenido mediante k_NN(q) */
1. inicializar un heap H que acepta pares <clave,entrada>
2. tomar el nodo raíz R
3. **para cada** entrada E en R
4. insertar < T_{inf}(E,q),E> en H
5. **mientras** (H no vacío y H.raiz.clave < T)
6. extraer raiz <clave,E> de H
7. **si** E es una entrada de nivel hoja
8. **si** clave < T
9. C = {E}; T = T_{inf}(E,q)
10. **si_no_si** clave = T
11. C = C ∪ {E}
12. **si_no** /* E es una entrada de nivel intermedio */
13. para cada entrada E' en E.hijo
14. insertar < T_{inf}(E',q),E'> en H
15. **devolver** T, C

fin TP_k_NN_BF

3.5.2 Consulta de los k vecinos más próximos continua

Una consulta de los k vecinos más próximos continua recibe como parámetros un objeto de consulta, un conjunto de objetos y un intervalo de tiempo, devolviendo como resultado una lista de la forma {<R_1,T_1>,<R_2,T_2>,...,<R_m,T_m>} donde R_i es el conjunto

i-ésimo de los k objetos que tengan una menor distancia desde el objeto de consulta, T_i el tiempo de validez de R_i y m el número total de cambios. Como se indicó anteriormente el número total de cambios depende de la condición de terminación que se dé a la consulta. En este caso se ha elegido de manera que T_m coincida con el final del intervalo de consulta.

Continuando con el ejemplo de la figura 3.11, el siguiente cambio se dará cuando el punto de consulta alcance la posición s_2, en que *c* comienza a estar más cerca de *q* que *b*. El proceso continúa hasta que el punto de consulta alcance su destino.

En este caso no es posible responder a la consulta continua mediante un algoritmo *single-pass* que obtenga los sucesivos resultados y sus tiempos de validez mediante una consulta de los k vecinos más próximos, tomando el tiempo de influencia como distancia métrica. Esto se debe a que, como se dijo antes, los tiempos de influencia se deben calcular a partir de los resultados inmediatamente anteriores, así que no podemos obtener todos los tiempos de influencia de los objetos de antemano. Por este motivo se ha optado por implementar una versión repetitiva del algoritmo que ejecuta sucesivas consultas TP para ir obteniendo los conjuntos R_i y sus tiempos de validez T_i. Éste es el pseudocódigo:

Algoritmo CkNNQ_Repetitivo(q)
/* Se le pasa la raíz del árbol */
1. i = 0 /*contador de resultados devueltos*/
2. ultimo_T = -1
3. R_i = k_NN(q)
4. **mientras** (ultimo_T < t_2)
5. T_i, C_i = TP_K_NN_BF(R_i,q)
6. R_{i+1} = Aplicar(C_i ,R_i)
7. ultimo_T = T_i;
8. i = i+1;

fin CkNNQ_Repetitivo

Algoritmo Aplicar(R,C)
1. **para cada** elemento o en C
2. **si** o ∈ R
3. R = R − {o}
4. **si_no**
5. R = R ∪ {o}
6. **devolver** R

fin Aplicar

No obstante se ha desarrollado un método para responder a la consulta de los k vecinos más próximos mediante un algoritmo *single-pass* [TP03], [TPS02] que no se ha implementado en este proyecto.

La aplicación de este método implica ciertas restricciones. La primera es que los objetos considerados en el conjunto de datos no tienen extensión espacial, es decir, son puntuales. La segunda es que la condición de terminación es temporal, es decir, se establece un tiempo límite TL y se considera el recorrido del punto de consulta q desde su posición en el instante inicial 0 hasta su posición en el instante final TL. Partiendo

del intervalo de tiempo [0, TL] se va actualizando una lista de instantes SL que contiene los instantes T_i en que los conjuntos de vecinos más próximos R_i cambian.

Comenzaremos exponiendo el funcionamiento de este método para un único vecino y a continuación lo extenderemos a un valor de k arbitrario.

La lista SL contiene todos los instantes t_i, $0 \leq i \leq |SL|$ en los que el vecino más próximo de q cambia, además de los instantes inicial y final, 0 y TL. Dados dos instantes consecutivos de la lista, t_i y t_{i+1}, el vecino más próximo de q en el instante t_i y durante el intervalo $[t_i, t_{i+1}]$ será t_i.NN, y diremos que t_i.NN cubre a t_i y a $[t_i, t_{i+1}]$.

Al inicio, la lista SL contiene solamente a 0 y TL (SL = {0,TL}) y el vecino más próximo (es decir, el punto que cubre a 0 y a [0, TL]) es desconocido. De manera incremental se actualizará la lista que al final contendrá todos los instantes de cambio (T_i) y sus vecinos más próximos (R_i).

Para procesar un objeto o debemos comprobar si existe algún instante en el que o esté más cerca de q que el vecino más próximo encontrado hasta el momento. O dicho de otra forma, debemos buscar un instante t perteneciente a $[t_i, t_{i+1}]$ $0 \leq i \leq |SL| - 1$ para el que $||q(t), o(t)|| < ||q(t), t_i.NN(t)||$ (el operador $|| \ ||$ expresa la distancia cuadrática). No podemos comprobar exhaustivamente todos los instantes de un intervalo, puesto que son infinitos. Para comprobar la condición anterior haremos uso del siguiente lema:

Lema 3.1. *Dada una lista de instantes SL = {t_0, t_1, …, $t_{|SL|-1}$} y un punto o, o cubre algún instante en el intervalo de consulta sii o cubre algún instante de la lista SL.*

Es decir, para saber si o cubre algún instante en [0, TL] sólo necesitamos saber si o cubre algún instante de SL. Por tanto debemos resolver la inecuación $||q(t), o(t)|| < ||q(t), t_i.NN(t)||$ para cada uno de los t_i en SL, que tiene la forma $At^2 + Bt + C < 0$, con A, B y C constantes dependientes de las posiciones y velocidades de o, q y t_i.NN. Obtendremos un periodo de influencia para cada t_i en SL durante el que o está más cerca de q que t_i.NN. Si el periodo es vacío entonces dejamos el punto o y continuamos. Si el periodo no es vacío debemos actualizar SL, que consiste en calcular el nuevo vecino más próximo en los instantes de inicio y fin del periodo de influencia.

Aprovechando la estructura de datos subyacente (TPR*-tree) podemos evitar el procesamiento de innecesario de objetos. En concreto las entradas intermedias que nunca estarán más cerca de q que su vecino más próximo se podarán.

Para extender este método a un valor arbitrario de k debemos tener en cuenta que ahora los conjuntos de vecinos más próximos están formados por k elementos, es decir, $R_i = \{t_i.1NN, t_i.2NN, \ldots, t_i.kNN\}$ $(0 \leq i \leq |SL| - 1)$. Ahora debemos comprobar si $||q(t), o(t)|| < ||q(t), t_i.jNN(t)||$ para cada t_i en SL y para cada j = 1, …, k. Obtendremos k periodos de influencia para cada t_i y continuaremos igual que en el caso de k = 1, es decir, los objetos que produzcan todos los periodos de influencia vacíos serán ignorados y los que no deberán actualizar la lista SL. Seguiremos el mismo procedimiento para las entradas intermedias. El algoritmo completo y sus detalles se pueden encontrar en [TP03].

3.6 Consultas de join espacial en entornos dinámicos

Una consulta de join espacial en entornos dinámicos recibe como parámetros dos conjuntos de objetos, una condición espacial y un intervalo de tiempo, devolviendo como resultado los pares de objetos del producto cartesiano de los conjuntos de objetos que cumplan la condición espacial en algún momento del intervalo.

En nuestro caso se ha tomado como condición espacial para el join el *solape* de los objetos. Se ha implementado la consulta de join en sus versiones de intervalo, TP y continua. Además, los algoritmos mostrados son para árboles de alturas iguales. Para árboles de alturas diferentes se aplica la técnica fijar las hojas [Cor00]. El algoritmo recorre las estructuras de índice normalmente hasta que en una de las dos alcanza el nivel de hoja. En ese momento se detiene la propagación del algoritmo en el índice en el que hemos alcanzado el nivel hoja, mientras que en el otro continuamos normalmente.

3.6.1 Consulta de join espacial de intervalo

La consulta de join espacial de intervalo parte de dos conjuntos de datos y un intervalo de tiempo y tiene por objetivo devolver todos pares de objetos de ambos conjuntos que se solapen en algún momento del intervalo. Para saber si un par de objetos se solapan hacemos uso de su periodo de intersección que nos dirá el intervalo de tiempo $[T_s, T_e)$ durante el cual los objetos se solapan. Si este periodo interseca total o parcialmente con el intervalo de consulta, entonces el par de objetos forma parte del resultado de la consulta. Una posible solución sería comparar todos los pares posibles de objetos. Para evitar tener que comparar todos los pares de objetos posibles hacemos uso de las características de las estructuras de índice subyacentes a los conjuntos de datos (TPR*-tree). En concreto, dado un par de entradas intermedias E_1 y E_2 pertenecientes a cada conjunto de datos y su periodo de intersección $[T_{Es}, T_{Ee})$, para cualquier par de objetos $(o_1, o_2) \in E_1 \times E_2$, su periodo de intersección $[T_{os}, T_{oe})$ está incluido en $[T_{Es}, T_{Ee})$. Por tanto, si el periodo de intersección de E_1 y E_2 no interseca con el intervalo de consulta podemos asegurar que el periodo de intersección de cualquier par de objetos en los subárboles de E_1 y E_2 tampoco lo hará y podremos descartarlos de la búsqueda.

3.6.1.1 Versión Depth-First

Éste es un algoritmo recursivo que recorre los dos índices en profundidad (DF) de manera simultánea. Empezando por los nodos raíz de ambos índices, toma cada par de entradas de los nodos y comprueba si existe solape espacio-temporal. En caso afirmativo aplica la recursión sobre el par de nodos apuntados por las entradas. Al alcanzar el nivel de nodos hoja si las entradas se solapan se añaden al conjunto de resultados. Éste es su pseudocódigo:

Algoritmo Join_DF(N_A, N_B, R, t_1, t_2)
/* en la primera llamada se le pasan los nodos raíz de los índices de ambos conjuntos de datos, R_A y R_B junto con R = Ø. Suponemos árboles de igual altura* /
1. **si** N_A es nodo hoja /* y por tanto también N_B */
2. **para cada** par de entradas (E_A, E_B) en $N_A \times N_B$
3. $[T_s, T_e)$ = **Calcular_Periodo_Intersección** (E_A, E_B)
4. **si** $[T_s, T_e) \cap [t_1, t_2] \neq Ø$ /* si E_A y E_B intersecan */
5. $R = R \cup \{(E_A, E_B)\}$

6. **si_no** /* N_A y N_B son internos */
7. **para cada** par de entradas (E_A, E_B) en $N_A \times N_B$
8. $[T_s, T_e)$ = **Calcular_Periodo_Intersección** (E_A, E_B)
9. **si** $[T_s, T_e) \cap [t_1, t_2] \neq \emptyset$ /* si E_A y E_B intersecan */
10. **Join_DF**(E_A.hijo, E_B.hijo, R, t_1, t_2)

fin Join_DF

3.6.1.2 Versión Depth-First con barrido del plano

Éste es un algoritmo recursivo que recorre los dos índices en profundidad (DF) de manera simultánea. Para evitar comparar todos los pares de entradas de un par de nodos (y por tanto evitar operaciones innecesarias y disminuir el tiempo de respuesta) aplica un método basado en el barrido del plano (*plane-sweep*) propuesto por [BKS93] para índices R*-tree.

El objetivo es ordenar las entradas del par de nodos de acuerdo a las posiciones de sus MBRs para poder acceder a ellos en ese orden. Pero no es posible mapear rectángulos bidimensionales a una secuencia unidimensional sin pérdida de localidad. Se opta por la siguiente solución. Consideremos una secuencia de rectángulos R_{sec} = <r_1,…,r_n definidos por sus esquinas inferior (r_i.xl, r_i.yl) y superior (r_i.xu, r_i.yu). Además llamamos $\pi_x(r_i)$ a la proyección de r_i en el eje X. La secuencia R_{sec} = <r_1, …, r_n> se ordena respecto al eje X según el valor creciente de r_i.xl. Desplazamos la línea de barrido (perpendicular a uno de los ejes, el eje x en este caso) en sentido creciente. Para dos secuencias de rectángulos R_{seq} = <r_1, …, r_n> y S_{seq} = <s_1, …, s_n>, las ordenamos como se indicó. Desplazamos la línea de barrido hasta el primer rectángulo t $\in$ $R_{sec} \cup S_{sec}$ (el que tenga menor valor de xl). Si t pertenece a R_{sec} recorremos S_{sec} en orden hasta encontrar un rectángulo s_h tal que s_h.xl > t.xu. Entonces para todos los rectángulos de S_{sec} s_i con $1 \leq i < h$ se cumple que $\pi_x(s_i)$ interseca con $\pi_x(t)$. Por tanto, para cada s_i debemos comprobar si $\pi_y(s_i)$ interseca con $\pi_y(t)$. En ese caso los rectángulos s_i y t intersecan. Entonces t se marca como procesado y se avanza la línea de barrido al siguiente rectángulo de $R_{sec} \cup S_{sec}$ sin marcar. Este proceso se repite hasta procesar todos los rectángulos de R_{sec} y S_{sec}. Es reseñable que la implementación de este método puede llevarse a cabo sin la necesidad de usar ninguna estructura de datos auxiliar.

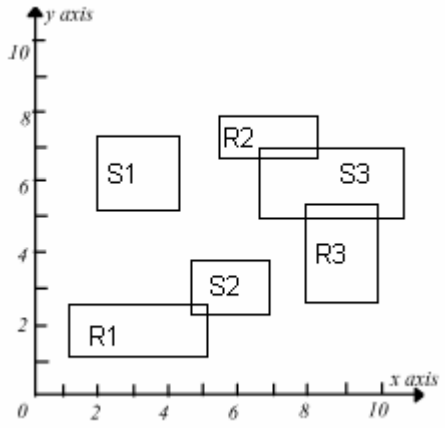


Figura 3.13. Ejemplo de barrido del plano.

En la figura 3.13 podemos ver un ejemplo del procesamiento de los rectángulos por plane-sweep sobre dos conjuntos R = {R1, R2, R3} y S = {S1, S2, S3}. Para una línea

de barrido paralela al eje Y, tomamos el primer rectángulo en R ∪ S, que es R1. R1 es comprobado con S1 y S2, puesto que su proyección en el eje X interseca con las de estos, dando como resultado que R1 interseca con S2. A continuación se avanza hasta el siguiente rectángulo en R ∪ S, S1. Este no se comprueba con ninguno, ya que su proyección en el eje X no interseca con ningún elemento de R. La secuencia continua con el procesamiento de S2 (comparado con R2, resultado negativo), R2 (comparado con S3, resultado afirmativo) y S3 (comparado con R3, resultado afirmativo).

Plane-sweep fue pensado para R*-tree, pero nosotros lo usamos con TPR*-tree, por lo que en lugar de aplicarlo directamente a los MBRs, que varian con el tiempo, debemos hacerlo sobre los rectángulos envolventes que encierran a los MBRs durante todo el intervalo de consulta. A nivel hoja la comprobación del solape se sigue realizando mediante el cálculo del periodo de intersección.

Algoritmo Join_DF_PS(N_A, N_B, R, t_1, t_2)
/* en la primera llamada se le pasan los nodos raíz de los índices de ambos conjuntos de datos, R_A y R_B junto con R = Ø. Suponemos árboles de igual altura* /
1. **si** N_A es nodo hoja /* y por tanto también N_B */
2. **para cada** par de entradas (E_A, E_B) en $N_A \times N_B$
3. $[T_s, T_e)$ = **Calcular_Periodo_Intersección** (E_A, E_B)
4. **si** $[T_s, T_e) \cap [t_1, t_2] \neq \emptyset$ /* si E_A y E_B intersecan */
5. $R = R \cup \{(E_A, E_B)\}$
6. **si_no** /* N_A y N_B son internos */
7. N_{envA} = **Envolvente**(N_A, t_1, t_2)
8. ordenar N_{envA} según el valor de la coordenada xl
9. N_{envB} = **Envolvente**(N_B, t_1, t_2)
10. ordenar N_{envB} según el valor de la coordenada xl
11. i = j = 0;
12. **mientras** (i < N_{envA}.num_entradas) y (j < N_{envB}.num_entradas)
13. **si** (N_{envA}.entrada$_i$.xl < N_{envB}.entrada$_j$.xl)
14. k = j;
15. **mientras** (k < N_{envB}.num_entradas)
y (N_{envB}.entrada$_j$.xl ≤ N_{envA}.entrada$_i$.xu)
16. **si** (N_{envA}.entrada$_i$.yl ≤ N_{envB}.entrada$_j$.yu)
y (N_{envA}.entrada$_i$.yu ≥ N_{envB}.entrada$_j$.yl)
17. **Join_DF_PS**(N_{envA}.entrada$_i$.hijo,
N_{envB}.entrada$_k$.hijo, R, t_1, t_2)
18. k = k + 1;
19. i = i + 1;
20. **si_no**
21. k = i;
22. **mientras** (k < N_{envA}.num_entradas)
y (N_{envA}.entrada$_i$.xl ≤ N_{envB}.entrada$_j$.xu)
23. **si** (N_{envB}.entrada$_j$.yl ≤ N_{envA}.entrada$_i$.yu)
y (N_{envB}.entrada$_j$.yu ≥ N_{envA}.entrada$_i$.yl)
24. **Join_DF_PS**(N_{envA}.entrada$_k$.hijo,
N_{envB}.entrada$_j$.hijo, R, t_1, t_2)
25. k = k + 1;
26. j = j + 1;
fin Join_DF_PS

Para obtener los rectángulos envolventes correspondientes las entradas de un nodo durante un intervalo de tiempo$[t_1, t_2]$ usamos el siguiente algoritmo.

Algoritmo Envolvente(N_A, t_1, t_2)
1. declarar un nodo N_{env} para las entradas envolventes
2. **para cada** entrada E en N_A /* crear una entrada envolvente */
3. declarar una entrada E'
4. **para cada** dimensión d de E
5. $E'_{dL} = \min(E_{dL} + E.V_{dL}(t_1), E_{dL} + E.V_{dL}(t_2))$
6. $E'_{dR} = \max(E_{dR} + E.V_{dR}(t_1), E_{dR} + E.V_{dR}(t_2))$
7. insertar E' en N_{env}
8. **devolver** N_{env}

fin Envolvente

3.6.1.3 Versión Breadth-First

Estos algoritmos realizan un recorrido en anchura simultáneo (breadth first o BF) y no recursivo de los índices [HJR97]. Mantiene una estructura llamada *índice join intermedio* que para cada nivel contiene todos los pares de nodos que solapan. De modo que para construir el IJI[i+1] se basa en el contenido de IJI[i]. El algoritmo comienza construyendo el IJI[0] a nivel raíz y al terminar el IJI a nivel de hojas contiene los pares de objetos que cumplen el join. En cada iteración el algoritmo toma cada uno de los pares de nodos que hay en el IJI de nivel superior y comprueba para cada par de entradas de cada nodo si solapan. En caso de ser así los nodos apuntados por las entradas pasan a formar parte del IJI en ese nivel. La comprobación del solape entre dos entradas se realiza mediante el cálculo de su periodo de intersección.

El hecho de que en cada nivel se disponga de información global de ambos índices (no como en los algoritmos DF, que manejan información local, de caminos concretos dentro de cada árbol) hace posible plantearse estrategias de mejora global que pueden tener un mayor impacto en la reducción del tiempo de respuesta. Se observa que en un nivel dado, un nodo puede aparecer en múltiples pares dentro del IJI. Puede darse el caso de que, dependiendo del orden de estas parejas en el IJI, el nodo deba ser leído de disco tantas veces como el número de parejas en que participa. Una estrategia encaminada a reducir el número de accesos a disco (y por tanto el tiempo de respuesta) sería ordenar los pares dentro del IJI para minimizar la necesidad de cargar más de una vez un mismo nodo. Se usan dos criterios:

1. Join BF ordsum. Se ordenan los pares según el valor de la suma de los centros de sus MBRs en el eje X. Al ser MBRs dinámicos tomaremos como rectángulos la envolvente de cada MBR a lo largo del intervalo de consulta $[t_1, t_2]$, es decir, para un nodo con MBR $[o_L, o_R]$ y VBR $[o.V_L, o.V_R]$ en el eje X, tomaremos como rectángulo envolvente en el eje X

$$R = [R_L, R_R] = [\min(o_L + o.V_L(t_1), o_L + o.V_L(t_2)), \max(o_R + o.V_R(t_1), o_R + o.V_R(t_2))]$$

Por tanto, para el nodo A el centro sería $centro_A = (A_L+A_R)/2$, para el nodo B el centro sería $centro_B = (B_L+B_R)/2$ y el valor de ordenación sería $centro_A + centro_B$.

2. Join BF ordcen. Se ordenan los pares según el valor del punto medio del rectángulo envolvente que encierra a los MBRs de ambos nodos durante el intervalo de

consulta $[t_1, t_2]$ en el eje X. Para un par de nodos A y B con MBRs $[a_L, a_R]$ y $[b_L, b_R]$ y VBRs $[a.V_L, a.V_R]$ y $[b.V_L, b.V_R]$ en el eje X, tomaremos como rectángulo envolvente en el eje X

$$R = [R_L, R_R] =$$

$$[\min(a_L + a.V_L(t_1), a_L + a.V_L(t_2), b_L + b.V_L(t_1), b_L + b.V_L(t_2)),$$

$$\max(a_R + a.V_R(t_1), a_R + a.V_R(t_2), b_R + b.V_R(t_1), b_R + b.V_R(t_2))]$$

Por tanto el valor de ordenación sería $(R_L+R_R) / 2$.

Algoritmo Join_BF(R_A, R_B, t_1, t_2)
1. inicializar una matriz IJI que acepte ternas <nodo,nodo,clave>
2. **Procesar_Nodos**(R_A,R_B,t_1,t_2,IJI_0) /* comienza procesando los nodos raíz de cada índice*/
3. i = 0
4. **mientras** (i < altura de TPR_A) /* y por tanto i < altura de TPR_B */
5. ordenar IJI_i según el valor de la clave de ordenación
6. **para cada** par de nodos (N_A, N_B) en IJI_i
7. **Procesar_Nodos**(N_A,N_B,t_1,t_2,IJI_{i+1})
8. **devolver** IJI

fin Join_BF

Algoritmo **Procesar_Nodos(N_A, N_B, t_1, t_2, L)**
1. **para cada** par de entradas (E_A, E_B) en $N_A \times N_B$
2. $[T_s, T_e)$ = **Calcular_Periodo_Intersección** (E_A, E_B)
3. **si** $[T_s, T_e) \cap [t_1, t_2] \neq \emptyset$ /* si E_A y E_B intersecan */
4. obtener clave de ordenación
5. insertar <E_A.hijo,E_B.hijo,clave> en L

fin Procesar_Nodos

3.6.2 Consulta de join espacial TP

Una consulta de join espacial TP recibe como parámetros dos conjuntos de objetos, una condición espacial y un intervalo de tiempo, devolviendo como resultado los pares de objetos del producto cartesiano de los conjuntos de objetos que cumplan la condición espacial al inicio del intervalo (R), el tiempo de expiración de este resultado (T) y los pares de objetos que causan la expiración del resultado (C).

Responder a la consulta de join TP consiste en obtener el par de objetos con menor tiempo de influencia. Por tanto debemos definir la métrica *tiempo de influencia* para una consulta de join y aplicar la estrategia BaB para obtener el par que la minimice [TP03].

El resultado del join (R) cambiará en un instante futuro (T) porque pares de objetos que ya solapan dejarán de hacerlo o porque pares de objetos que aun no solapan comenzarán a hacerlo (C). El tiempo de influencia en esta consulta se define por tanto para pares de objetos, cada uno de ellos perteneciente a cada uno de los conjuntos de datos que se están procesando, $T_{inf}(o_1,o_2)$. El objetivo es encontrar el par de objetos (o_1, o_2) (C) con el menor tiempo de influencia (T), para lo que debemos realizar una

búsqueda de los pares más próximos tomando como métrica de distancia el tiempo de influencia $T_{inf}(o_1,o_2)$.

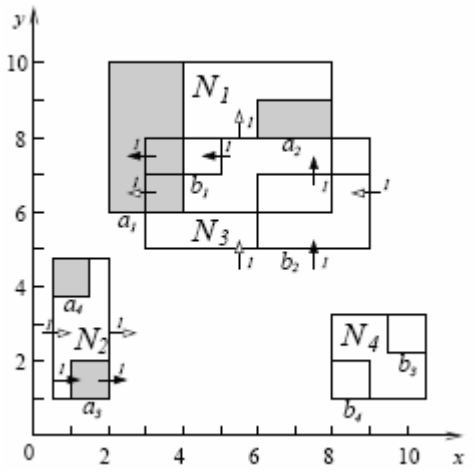
Sean P y Q dos conjuntos de datos, sea (o_1, o_2) un par de objetos pertenecientes a $P \times Q$, y sea $[T_s, T_e)$ el periodo de intersección de o_1 y o_2, el tiempo de influencia $T_{inf}(o_1,o_2)$ será T_e si T_s es igual al instante actual (es decir, si ya intersecan) o T_s si T_s es mayor al instante actual (es decir, aun no intersecan). El tiempo de influencia puede ser infinito, indicando que el par de objetos no cambiará nunca su estado (juntos o disjuntos) y por tanto nunca influirá en el resultado.

Para un par de entradas intermedias (E_1, E_2) pertenecientes a $P \times Q$ definimos el tiempo de influencia $T_{inf}(E_1,E_2)$ como una cota inferior del tiempo de influencia para cualquier par de objetos (o_1, o_2) tales que o_1 se encuentre en el subárbol delimitado por E_1 y o_2 se encuentre en el subárbol delimitado por E_2. Si el periodo de intersección de E_1 y E_2 es $[T_s, T_e)$ entonces $T_{inf}(E_1,E_2)$ será T_s. Esto se debe a que una vez que los MBRs de E_1 y E_2 comienzan a intersecar, en cualquier momento un par de objetos perteneciente a $E_1 \times E_2$, pueden comenzar a intersecar o a dejar de intersecar, generando un cambio en el resultado y por tanto un tiempo de influencia.

En resumen, la métrica *tiempo de influencia* queda descrita de la siguiente manera:

- $T_{inf}(o_1,o_2) = T_s$ si T_s es mayor al instante actual.
- $T_{inf}(o_1,o_2) = T_e$ si T_s es igual al instante actual.
- $T_{inf}(E_1,E_2) = T_s$.

En la figura 3.14 podemos ver dos conjuntos de datos con sus tiempos de influencia respectivos. Un valor infinito indica que el par de objetos nunca influenciará el resultado. Según los valores de la tabla, la componente TP de la consulta de join estará formada por $C = (a_2, b_2)$ y $T = 1$, que es el par con menor tiempo de influencia.



(a) Join entre dos conjuntos de datos

	a_1	a_2	a_3	a_4
b_1	3	∞	∞	∞
b_2	∞	1	∞	∞
b_3	∞	∞	∞	∞
b_4	∞	∞	6	∞

(b) Tiempos de influencia para cada par de objetos

Figura 3.14. Tiempos de influencia para una consulta de Join TP.

Ahora se puede emplear un algoritmo BaB que realice la búsqueda de los pares más cercanos tomando como métrica de distancia el tiempo de influencia. Además también

se obtendrá la componente convencional de la consulta en una sola pasada. El algoritmo explorará los subárboles de un par de entradas E_1 y E_2 si estas solapan (por lo que podrían contener un par perteneciente a R) o si el tiempo de influencia T_{inf} (E_1,E_2) es más pequeño o igual que el menor tiempo de influencia encontrado hasta el momento (indicando que podrían contener un par candidato a ser el par más cercano, C). El caso contrario los subárboles de estas entradas intermedias serán podadas de la búsqueda. En cuanto a los pares de objetos el algoritmo realiza varias acciones. Si o_1 y o_2 solapan entonces el par se añade a R. Si su tiempo de influencia es más pequeño que el menor encontrado hasta el momento entonces actualizamos T a $T_{inf}(o_1,o_2)$ y actualizamos el conjunto C a (o_1, o_2). Si su tiempo de influencia es igual al menor encontrado hasta el momento entonces añadimos (o_1, o_2) al conjunto C.

3.6.2.1 Versión Depth-First

Es un algoritmo recursivo que recorre los árboles de manera DF. Éste es su pseudocódigo:

Algoritmo TP_Join_DF(nodo N_1, nodo N_2)
/* N_1 y N_2 son nodos de cada uno de los árboles.
1. En la primera llamada se le pasan los nodos raíz de cada uno */
2. **si** N_1 es hoja /* N_2 también pues son de la misma altura */
3. **para cada** par de objetos (o_1, o_2)
4. **si** $T_{inf}(o_1,o_2) < T$
5. $C = \{(o_1, o_2)\}$; $T = T_{inf}(o_1,o_2)$
6. **si_no_si** $T_{inf}(o_1,o_2) = T$
7. $C = C \cup \{(o_1, o_2)\}$
8. **si** o_1 interseca con o_2 entonces $R = R \cup \{(o_1, o_2)\}$
9. **si_no** /* N_1 y N_2 son nodos intermedios */
10. **para cada** par de entradas (E_1,E_2)
11. **si** ($T_{inf}(E_1,E_2)) \leq T$ o (E_1 interseca con E_2)
12. **TP_Join_DF**(E_1.hijo,E_2.hijo)

fin TP_Join_DF

3.6.2.2 Versión Depth-First con ordenación

Es también un algoritmo recursivo que recorre los árboles de manera DF. La diferencia con el anterior radica en que en los niveles intermedios procesa los pares de entradas de los nodos en orden creciente de sus tiempos de influencia. De esta manera es posible que se evite recorrer algunas ramas que el algoritmo anterior no evitaría. Este es su pseudocódigo:

Algoritmo TP_Join_DF_Ordenacion(nodo N_1, nodo N_2)
/* N_1 y N_2 son nodos de cada uno de los árboles. En la primera
1. llamada se le pasan los nodos raíz de cada uno */
2. **si** N_1 es hoja /* N_2 también pues son de la misma altura */
3. **para cada** par de objetos (o_1, o_2)
4. **si** $T_{inf}(o_1,o_2) < T$
5. $C = \{(o_1, o_2)\}$; $T = T_{inf}(o_1,o_2)$
6. **si_no_si** $T_{inf}(o_1,o_2) = T$
7. $C = C \cup \{(o_1, o_2)\}$
8. **si** o_1 interseca con o_2 entonces $R = R \cup \{(o_1, o_2)\}$

9. **si_no** /* N_1 y N_2 son nodos intermedios */
10. ordenar los pares de entradas (E_1, E_2) según su $T_{inf}(E_1,E_2)$
11. **para cada** par de entradas (E_1, E_2)
12. **si** ($T_{inf}(E_1,E_2)$) ≤ T o (E_1 interseca con E_2)
13. **TP_Join_DF_Ordenacion** (E_1.hijo,E_2.hijo)

fin TP_Join_DF_Ordenacion

3.6.2.3 Versión Best-First

Es un algoritmo iterativo que sigue un recorrido BF de los árboles. Como se indicó anteriormente hace uso de un heap. En el heap se almacenan los pares de entradas o pares de objetos visitados junto con su tiempo de influencia. El algoritmo termina cuando no quedan elementos en el heap. Este es su pseudocódigo:

Algoritmo TP_Join_BF(nodo N_1, nodo N_2)
/* Se le pasan las raíces de ambos árboles */
1. inicializar un heap H que acepta ternas <clave,entrada,entrada>
2. tomar los nodos raíz R_1, R_2
3. **para cada** par de entradas (E_1, E_2) en $R_1 \times R_2$
4. insertar < $T_{inf}(E_1,E_2)$, E_1, E_2> en H
5. **mientras** (H no vacío)
6. extraer raiz <clave,E_1,E_2> de H
7. **si** E_1, E_2 son entradas de nivel hoja
8. **si** clave < T
9. C = {(E_1, E_2)}; T = clave;
10. **si_no** /* clave es igual a T */
11. C = C ∪ {(E_1, E_2)}
12. **si** E_1 interseca con E_2 entonces R = R ∪ {(E_1, E_2)}
13. **si_no** /* E_1, E_2 son entradas de nivel intermedio */
14. **si** (clave ≤ T) o (E_1 interseca con E_2)
15. **para cada** par de entradas (E_1', E_2') en E_1.hijo × E_2.hijo
16. insertar < $T_{inf}(E_1',E_2')$, E_1', E_2'> en H

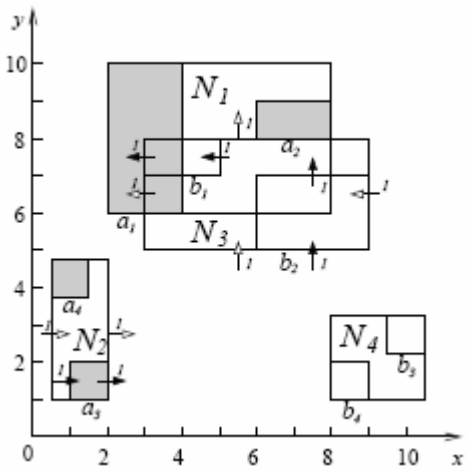
fin TP_Join_BF

3.6.3 Consulta de join espacial continua

Una consulta de join continuo recibe como parámetros dos conjuntos de objetos, una condición espacial y un intervalo de tiempo, devolviendo como resultado una lista de la forma {<R_1,T_1>,<R_2,T_2>,...,<R_m,T_m>} donde R_i es el conjunto i-ésimo de los pares de objetos que cumplan la condición espacial, T_i el tiempo de validez de R_i y m el número total de cambios. Como se indicó anteriormente el número total de cambios depende de la condición de terminación que se dé a la consulta. En este caso se ha elegido de manera que T_m coincida con el final del intervalo de consulta.

La respuesta a la consulta de join continuo consiste en obtener los k pares más próximos tomando como métrica de distancia el tiempo de influencia. Ahora cada par de objetos puede generar dos tiempos de influencia, T_{infs} y T_{infe}, que son los instantes en que comienzan y dejan de solaparse. Si denotamos el periodo de intersección del par (o_1, o_2) por [T_s, T_e), se nos presentan dos posibilidades: (i) si o_1 y o_2 intersecan actualmente entonces T_{infs} será T_e, el instante en que dejen de intersecar, y T_{infe} será infinito, mientras que (ii) si o_1 y o_2 no intersecan actualmente T_{infs} será T_s y T_{infe} será T_e.

El tiempo de influencia de las entradas intermedias se define igual que en la versión TP, puesto que su función sigue siendo la de cota inferior para la poda durante el recorrido de los índices [TP03].



(a) Join entre dos conjuntos de datos

	a_1	a_2	a_3	a_4
b_1	3/ ∞	∞ / ∞	∞ / ∞	∞ / ∞
b_2	∞ / ∞	1/4	∞ / ∞	∞ / ∞
b_3	∞ / ∞	∞ / ∞	∞ / ∞	∞ / ∞
b_4	∞ / ∞	∞ / ∞	6/8	∞ / ∞

(b) Tiempos de influencia para cada par de objetos

Figura 3.15. Tiempos de influencia para una consulta de Join continua.

Continuando con el ejemplo de la figura 3.14, en la figura 3.15b podemos ver los dos tiempos de influencia para cada par de objetos a partir de las velocidades indicadas en 3.15a. El objetivo de los algoritmos es ir devolviendo los pares en orden creciente de sus tiempos de influencia. En este caso el resultado sería (sin especificar condición de terminación): {<(a_2, b_2), 1>,< (a_1, b_1), 3>,< (a_2, b_2), 4>,< (a_3, b_4), 6>,< (a_3, b_4), 8>}.

Habiendo descrito la métrica tiempo de influencia podemos emplear un algoritmo BaB para realizar la búsqueda de los pares más próximos usando a esta como distancia.

El algoritmo que se ha implementado realiza un recorrido BF simultáneo de los dos índices y mantiene un heap donde inserta pares de objetos o entradas junto con su tiempo de influencia. En cada iteración toma el par de entradas en la raíz del heap y lo procesa como se indica a continuación (asumiendo árboles de igual altura). Si son entradas de nivel hoja, entonces comprueba si intersecan. En caso afirmativo añade el par al conjunto R. A continuación comprueba si su tiempo de influencia es igual al último tiempo de influencia encontrado, si es así inserta el par en el conjunto C_i, si no es así entonces el tiempo de influencia será mayor y se debe generar un nuevo conjunto C_{i+1} que contenga al par junto con un nuevo T_{i+1} con el tiempo de influencia del par. Si el par de entradas son de nivel intermedio entonces el algoritmo inserta todos los pares formados por los hijos de las entradas junto con sus tiempos de influencia en el heap. El algoritmo termina cuando el heap queda vacío o cuando se da la condición de terminación, que en este caso es que el tiempo de influencia en la raíz del heap sea mayor que el fin del intervalo de consulta. Su pseudocódigo es el siguiente:

Algoritmo CSJ

/* Se le pasan las raíces de ambos árboles */

1. inicializar un heap H que acepta ternas <clave,entrada,entrada>
2. tomar los nodos raíz R_1, R_2

3. **para cada** par de entradas (E_1, E_2) en $R_1 \times R_2$
4. insertar < $T_{inf}(E_1,E_2)$, E_1, E_2> en H
5. i = -1 /*contador de resultados devueltos*/
6. ultimo_T = -1
7. **mientras** (H no vacío y H.cima.clave ≤ t_2) /* t_2 es el fin del intervalo de consulta */
8. extraer raiz <clave,E_1,E_2> de H
9. **si** E_1, E_2 son entradas de nivel hoja
10. **si** clave > ultimo_T
11. i = i + 1;
12. C_i = {(E_1, E_2)}; T_i = clave; ultimo_T = clave;
13. **si_no** /* clave es igual a ultimo_T */
14. C_i = C_i ∪ {(E_1, E_2)}
15. **si** E_1 interseca con E_2 entonces R = R ∪ {(E_1, E_2)}
16. **si_no** /* E_1, E_2 son entradas de nivel intermedio */
17. **para cada** par de entradas (E_1', E_2') en E_1.hijo × E_2.hijo
18. **si** E_1', E_2' son entradas de nivel hoja
19. insertar < $T_{infs}(E_1',E_2')$, E_1', E_2'> en H,
20. insertar < $T_{infe}(E_1',E_2')$, E_1', E_2'> en H
21. **si_no** /* E_1', E_2' son entradas de nivel intermedio */
22. insertar < $T_{inf}(E_1',E_2')$, E_1', E_2'> en H

fin CSJ

3.7 Otras consultas espaciales en entornos dinámicos

Existen muchas otras consultas espaciales que podemos plantearnos implementar en entornos dinámicos. Entre las más destacadas podemos citar las siguientes:

- Consulta de inclusión.
- Consulta de adyacencia.
- Consulta del vecino más próximo inversa.
- Consulta de join basado en distancia o join de similitud.
- Consulta de semijoin de los k vecinos más próximos.
- Consulta de semijoin basado en distancia.

En este proyecto vamos a implementar la consulta de semijoin de los k vecinos más próximos continua.

Esta consulta recibe dos conjuntos de datos P y Q y un intervalo de consulta, devolviendo una lista de la forma {<R_1,T_1>,<R_2,T_2>,...,<R_m,T_m>} para cada objeto en P, donde la lista es el resultado de realiza una consulta de los k vecinos más próximos continua sobre el conjunto Q durante el intervalo de consulta. Para ello realiza un recorrido DF del índice del conjunto P y para cada objeto de P realiza una consulta de los k vecinos más próximos continua en el conjunto Q usando el algoritmo desarrollado en la sección 3.5.2.

Se ha implementado un algoritmo repetitivo y recursivo que no aprovecha la estructura de índice subyacente (TPR*-tree).

Algoritmo SJCkNNQ(q,R)
/* Se le pasa el nodo raíz del árbol que indexa el conjunto de datos A, R_A y el conjunto B */
1. **si** R_A es un nodo hoja
2. **para cada** entrada E en R_A
3. **CkNNQ_Repetitivo**(E.hijo)
   /* realiza una consulta CkNN con E.hijo como punto de consulta sobre el conjunto de datos B */
4. **si_no** /* R_A es un nodo intermedio */
5. **para cada** entrada E en R_A
6. **SJCkNNQ**(q,E.hijo) /* recursión */

fin SJCkNNQ

Capítulo 4

RESULTADOS EXPERIMENTALES

4.1 Introducción

A lo largo de este proyecto se han implementado diferentes algoritmos para responder a una serie de consultas espaciotemporales. En esta sección se realizarán experimentos para estudiar y comparar el comportamiento de los diferentes algoritmos bajo distintos parámetros.

Para la realización de los experimentos se han seguido las líneas marcadas en [Tao03]. Para las consultas de join se ha utilizado una imagen binaria de 2048×2048 píxeles (que es la misma que la usada en [Cam07]) a la que se le asigna una velocidad (figura 4.1). También se han usado conjuntos de datos sintéticos de distinta cardinalidad que se describe a continuación. Las posiciones de los objetos siguen una distribución uniforme en el universo de la imagen $[0, 2047]^2$, las velocidades para cada dimensión (sólo traslación) se distribuyen uniformemente en [-v, v] siendo v el 0.5% del tamaño del universo de datos por unidad de tiempo, y el tamaño de los objetos es el 0.005% del tamaño del universo de datos en cada dimensión. Además se ha utilizado un conjunto de datos reales llamado ST [http://www.rtreeportal.org] que contiene 131461 MBRs de las calles de Los Ángeles. Este conjunto de datos se ha escalado al universo de la imagen y se la han añadido velocidades (sólo traslación) distribuidas uniformemente en [-v, v] siendo v el 0.5% del tamaño del universo de datos por unidad de tiempo.

Para las consultas de ventana se ha usado la imagen anterior y se han generado ventanas de consulta que siguen una distribución uniforme en el universo $[0, 2047]^2$ con velocidades distribuidas uniformemente en [-v, v] siendo v el 0.5% del tamaño del universo de datos por unidad de tiempo.

Para las consultas de los k vecinos más próximos se han usado los conjuntos de puntos sintéticos y se han generado puntos de consulta que siguen una distribución uniforme en el universo $[0, 2047]^2$ con velocidades distribuidas uniformemente en [-v, v] siendo v el 0.5% del tamaño del universo de datos por unidad de tiempo.

Para las consultas de semijoin de los vecinos más próximos se usan los conjuntos de puntos reales y sintéticos. Además se ha generado una imagen de 2048×2048 con 100

puntos cuyas posiciones que siguen una distribución uniforme en $[0, 2047]^2$. La razón de esto es que no tiene sentido plantear una consulta de los vecinos más próximos cuando en uno de los conjuntos de datos se almacena una imagen que tiene regiones extensas y que al ser procesada para indexarse en un TPR*-tree se verá sometida a una división arbitraria.

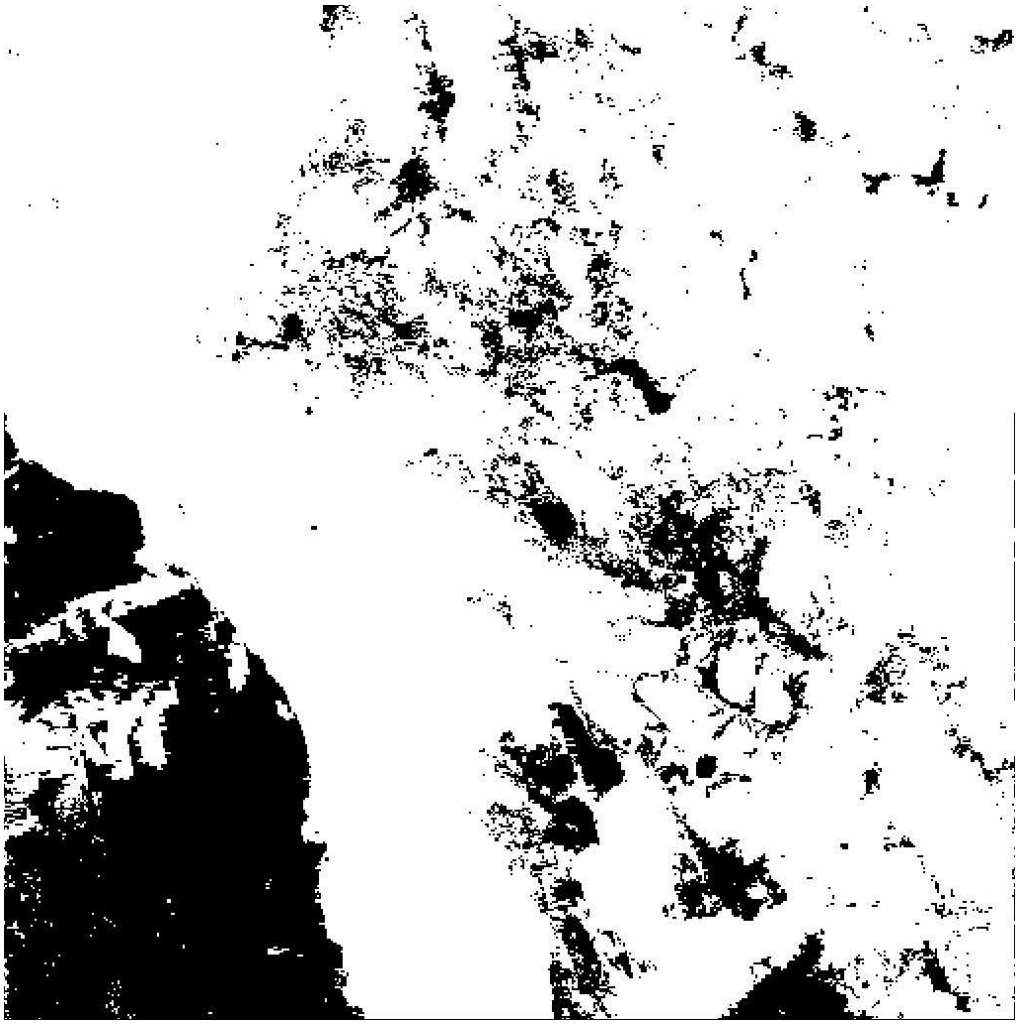


Figura 4.1. Cuadrante NW de la imagen raster.

TPR*-tree. El tamaño del nodo del TPR*-tree se hace coincidir con el tamaño de una página de disco [TP03], 1 KB, para poder leerlo o escribirlo en una única operación. En un nodo del TPR*-tree se almacena el identificador del bloque (int, 4 bytes), el nivel del nodo en el árbol (char, 1 byte), el número de entradas contenidas (int, 4 bytes) y las propias entradas. Cada entrada contiene un puntero al nodo al que hace referencia (int, 4 bytes), el MBR del nodo al que hace referencia y el VBR del nodo al que hace referencia. En un espacio d-dimensional, un rectángulo se determina mediante dos puntos d-dimensionales (sus esquinas superior e inferior), por tanto, necesitamos $2 \times d$ valores para determinar un rectángulo. Si estos valores los almacenamos en punto flotante (4 bytes) y nos encontramos en un espacio bidimensional, tendremos que utilizar $2 \times 2 \times 4 = 16$ bytes para un MBR y otros tantos para un VBR. Así pues, una entrada ocupa un espacio de 4 bytes (hijo) + 16 bytes (MBR) + 16 bytes (VBR) = 36 bytes. Por tanto, el número máximo de entradas que podemos almacenar en un nodo, el fanout, viene dado por la expresión

$$Tam\_pagina =$$

$$4\ B\ (id) + 1\ B\ (nivel) + 4\ B\ (n\_entradas\_contenidas) + fanout \times 36\ B\ (entradas)$$

lo que nos da un fanout de *28*.

Cada índice TPR*-tree (tanto el que contiene a la imagen como el que contiene a los objetos) cuenta con una caché LRU, de modo que no será necesario cargar de disco una página que ya se encuentre en la caché, evitando generar un fallo de página. En sistema sin caché los tiempos de respuesta estarían fuertemente relacionados con los fallos de página. Sin embargo, a causa de la caché del sistema operativo, que se interpone en una capa entre las cachés LRU de los índices y la memoria secundaria, en los resultados de los experimentos esta correlación no es tan fuerte.

Las pruebas se han realizado en un equipo con las siguientes características: CPU AMD Athlon 3700+ 2.2GHz, 1 GB de memoria RAM y Sistema Operativo Windows XP sp3.

Para cada experimento se han fijado una serie de parámetros y se ha tomado otro como variable. Se ha medido el tempo de respuesta y el número de fallos de página en función del valor asignado a este último. Se presentan los resultados para datos sintéticos y para datos reales.

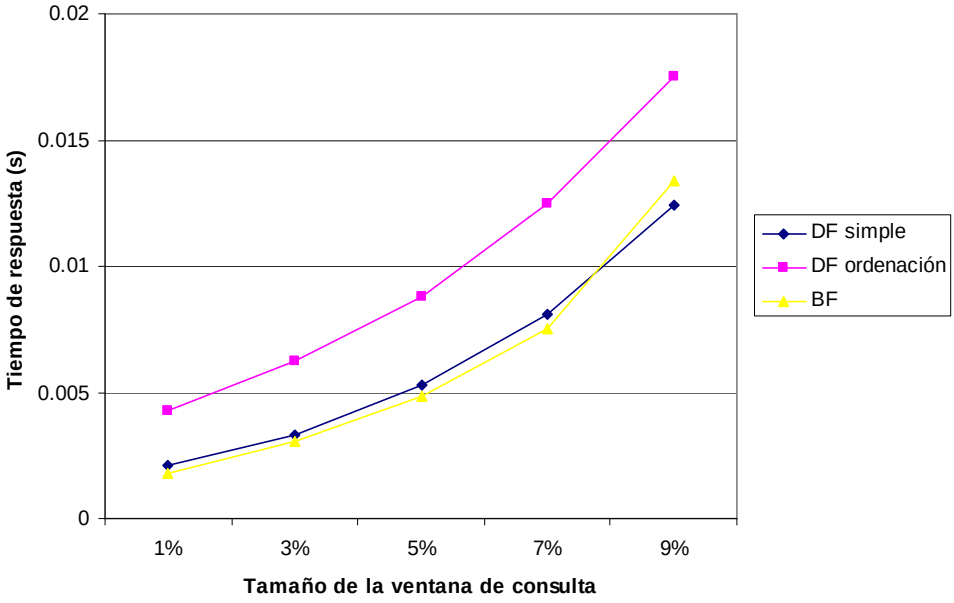
4.2 Consultas de ventana en entornos dinámicos

4.2.1 Consulta de ventana TP

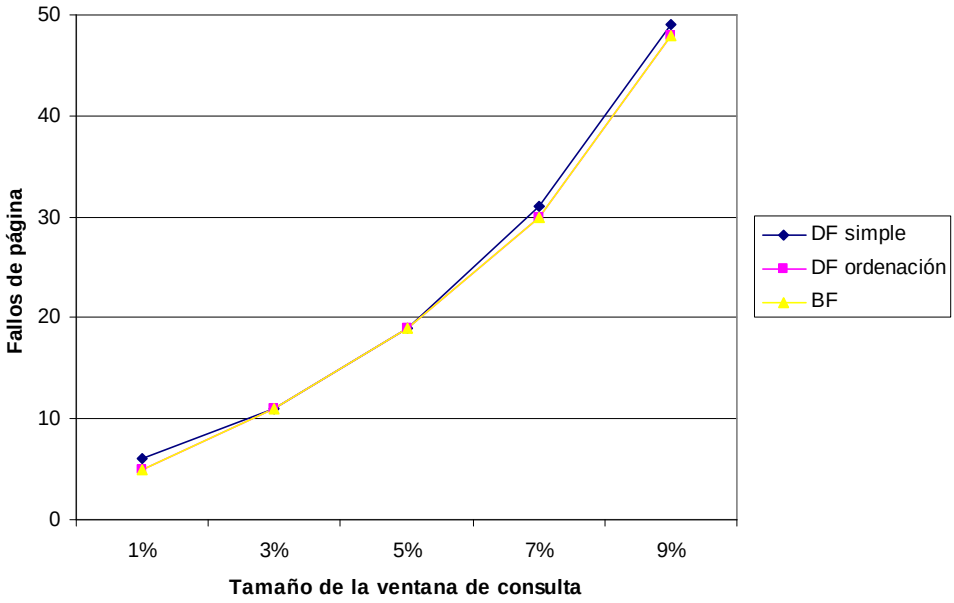
Estos son los resultados de los experimentos realizados con los algoritmos implementados para responder a la consulta de ventana TP: WQ TP DF simple (DF simple), WQ TP DF ordenación (DF ordenaxión) y WQ TP BF (BF).

Para la serie de experimentos realizados con estos algoritmos se han tenido en cuenta las siguientes variables: tamaño de la ventana de consulta (expresado en % del "universo" de datos en cada dimensión), tamaño de caché, tamaño de intervalo de consulta, instante de consulta, y velocidad de la imagen. En este caso no hay conjuntos de datos reales, puesto que todas las ventanas de consulta son generadas de manera aleatoria, como se indica en la introducción de este capítulo.

Experimento 1. Variación del tamaño de la ventana. Se han fijado las siguientes variables: tamaño de caché 25, intervalo [0, 5], velocidad de la imagen (1, 1). El tamaño de la ventana de consulta toma los siguientes valores: 1%, 3%, 5%, 7%, 9%.



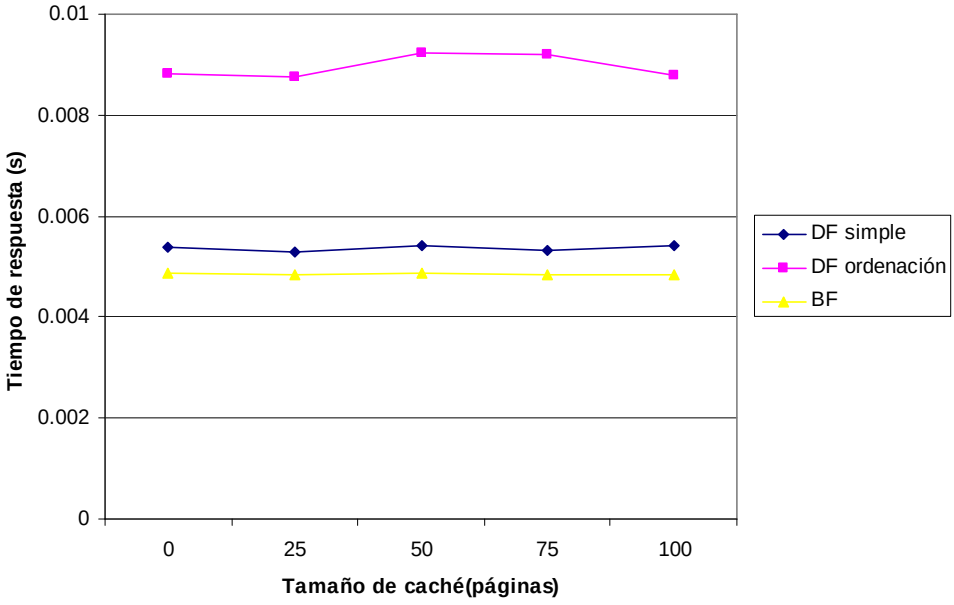
Gráfica 1. Tiempo de respuesta frente a tamaño de ventana para la consulta de ventana TP.



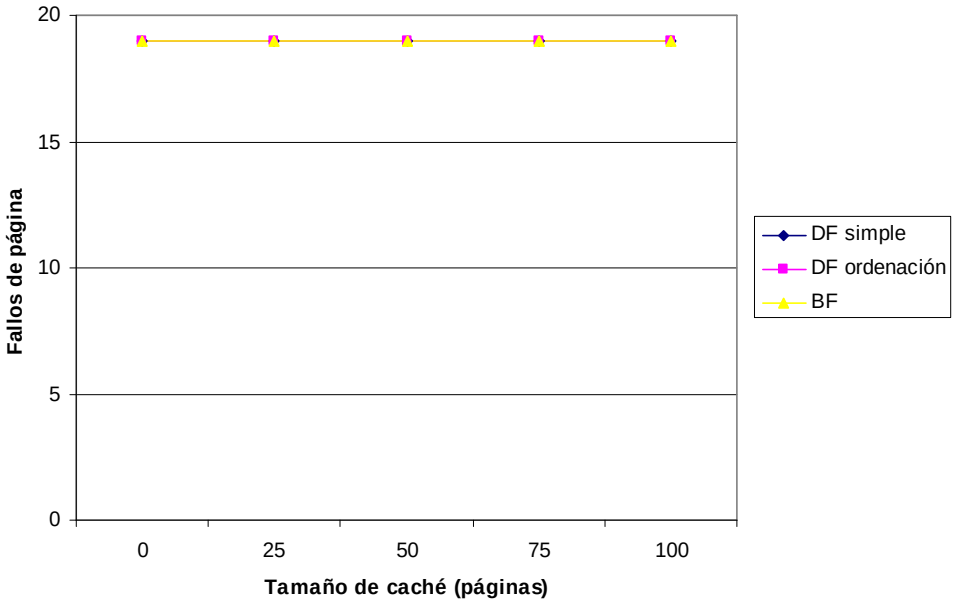
Gráfica 2. Fallos de página frente a tamaño de ventana para la consulta de ventana TP.

Los resultados del experimento muestran que, como es de esperar, tanto el tiempo de respuesta como los fallos de página (que dan una idea del número de accesos a disco) crecen con el tamaño de la consulta. Lo más importante es que comprobamos que *BF* supera a los dos algoritmos *DF* en tiempo de ejecución (salvo para tamaño de 9%, donde *DF simple* es mejor, aunque para experimentos con tamaños mayores *BF* lo vuelve a superar) y que la sobrecarga de ordenar las entradas por su tiempo de influencia penaliza a *DF ordenación*. En cuanto a fallos de página *DF simple*, como era de esperar, tiene peores resultados, aunque muy cerca de los otros dos algoritmos.

Experimento 2. Variación del tamaño de caché. Se han fijado las siguientes variables: tamaño de la ventana 5%, intervalo [0, 5], velocidad de la imagen (1, 1). El tamaño de caché toma los siguientes valores: 0, 25, 50, 75, 100.



Gráfica 3. Tiempo de respuesta frente a tamaño de caché para la consulta de ventana TP.

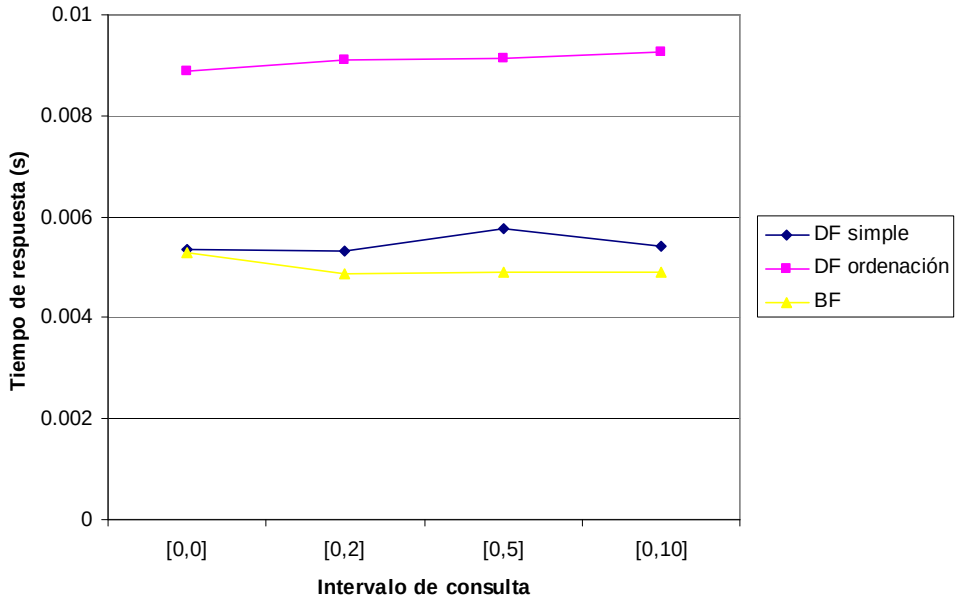


Gráfica 4. Fallos de página frente a tamaño de caché para la consulta de ventana TP.

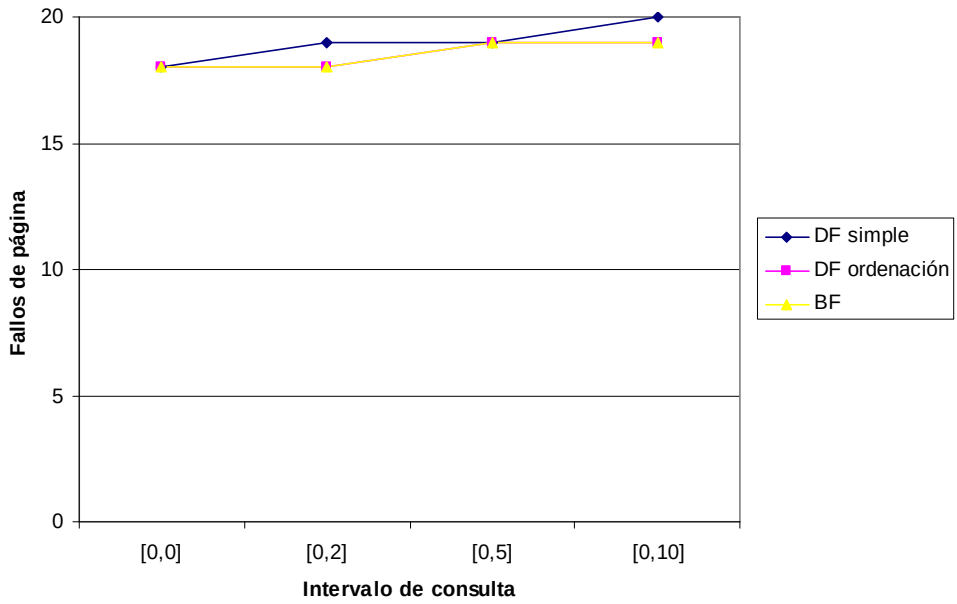
El tamaño de caché no tiene relevancia en el rendimiento de este algoritmo debido a los pocos accesos a nodos que requiere. Podemos pensar que las páginas sólo se cargan una vez y por tanto cada acceso es un fallo, a la vista de que los resultados para caché 0 son iguales que para cachés con más páginas. De nuevo *DF simple* mejora a *DF*

ordenación en tiempo de respuesta gracias a evitar la ordenación de las entradas y *BF* (no recursivo) supera a ambos.

Experimento 3. Variación del intervalo de consulta. Se han fijado las siguientes variables: tamaño de caché 25, tamaño de la ventana 5%, velocidad de la imagen (1, 1). El intervalo de consulta toma los siguientes valores: [0, 0], [0, 2.5], [0, 5], [0, 10].



Gráfica 5. Tiempo de respuesta frente a tamaño de intervalo para la consulta de ventana TP.

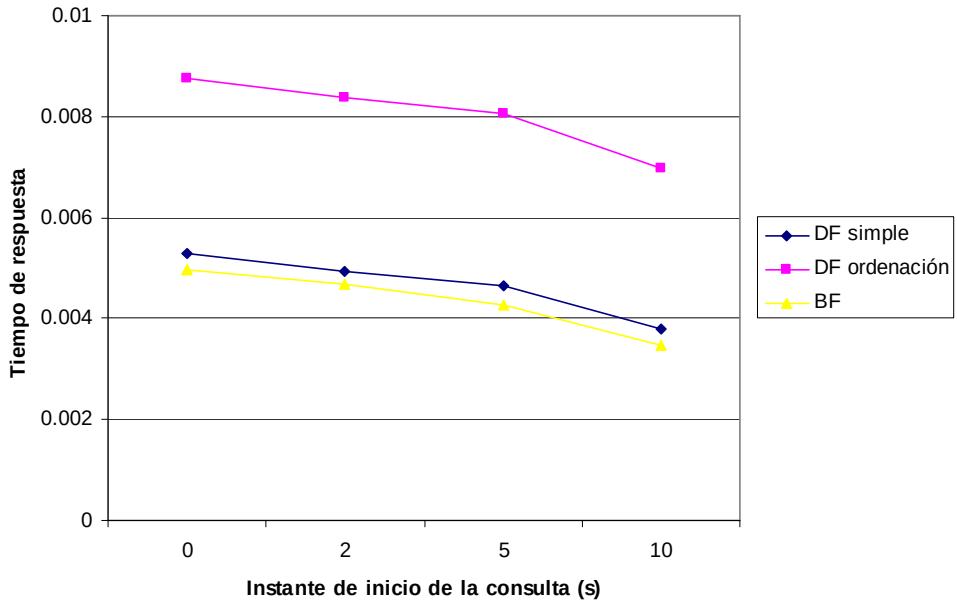


Gráfica 6. Fallos de página frente a tamaño de intervalo para la consulta de ventana TP.

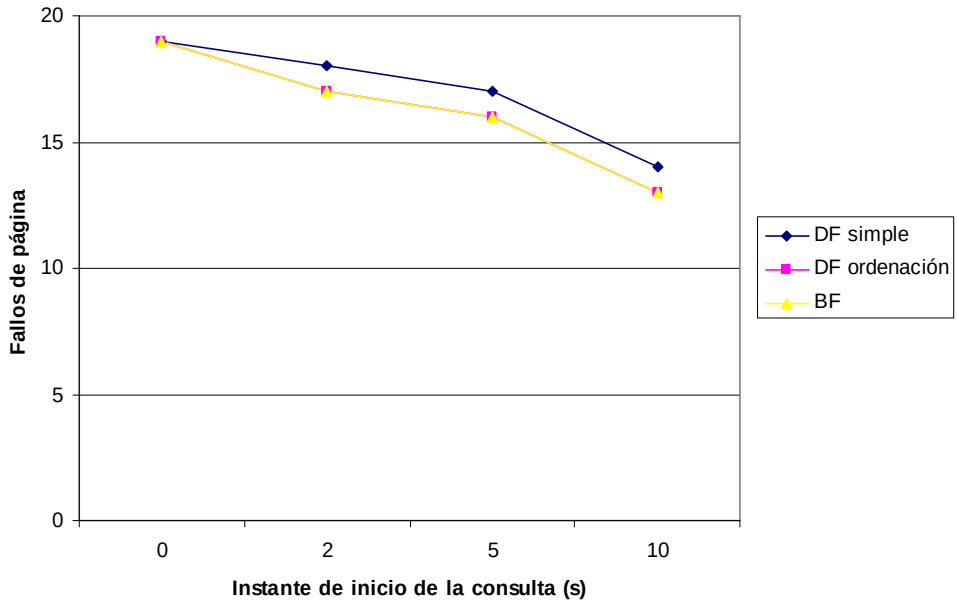
La combinación de dos hechos: el algoritmo busca el primer cambio (componente TP) y la imagen se traslada pero no se deforma (sus MBRs no crecen con el tiempo) hace que la influencia del tamaño del intervalo de consulta sea pequeña. Para el tiempo de ejecución (valores en milésimas de segundo) vemos que no hay una tendencia clara. Para fallos de página la tendencia es de crecimiento ligero. De nuevo *BF* tiene los

mejores resultados en tiempo de ejecución y fallos de página. *DF ordenación* se ve penalizado en tiempo de ejecución por la sobrecarga de ordenar las entradas y *DF simple* incurre en más fallos de página por su menor selectividad.

Experimento 4. Variación del instante de consulta. Se han fijado las siguientes variables: tamaño de caché 25, tamaño de la ventana 5%, velocidad de la imagen (1, 1). El intervalo de consulta toma los siguientes valores: [0, 5], [2, 7], [5, 10], [10, 15].



Gráfica 7. Tiempo de respuesta frente a inicio de intervalo para la consulta de ventana TP.

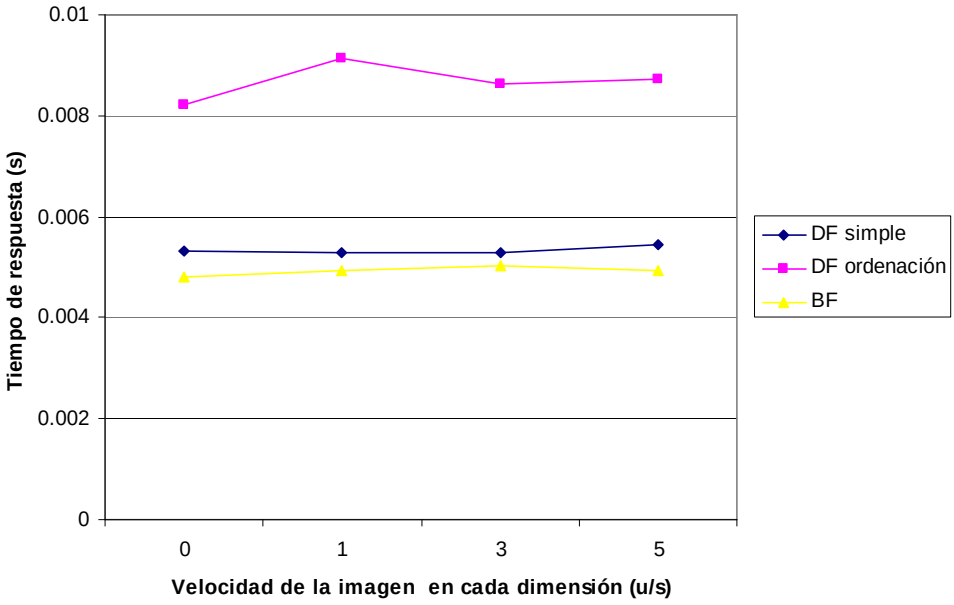


Gráfica 8. Fallos de página frente a inicio de intervalo para la consulta de ventana TP.

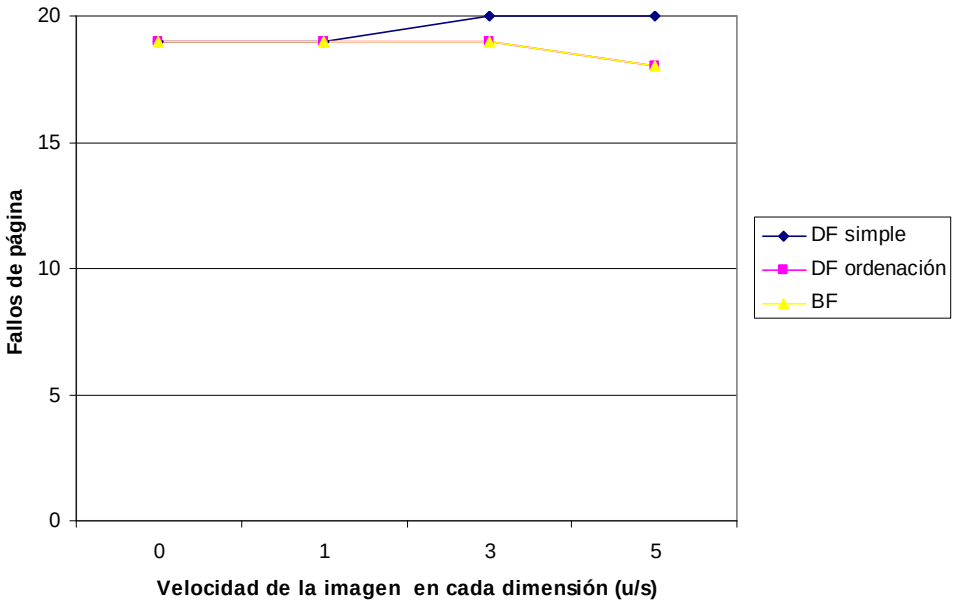
Debido a los dos factores mencionados en el experimento anterior, retrasar el inicio del intervalo no hace que los MBRs sean mayores, ni que la ventana de consulta solape con más regiones de la imagen. De hecho cuanto más tiempo pasa más distancia las

separa y por tanto menor probabilidad de solape. La comparativa entre los tres algoritmos sigue la tendencia de todos los experimentos.

Experimento 5. Variación de la velocidad de la imagen. Se han fijado las siguientes variables: tamaño de caché 25, tamaño de la ventana 5%, intervalo [0, 5]. La velocidad de la imagen toma los siguientes valores: (0, 0), (1, 1), (3, 3), (5, 5).



Gráfica 9. Tiempo de respuesta frente a velocidad de la imagen para la consulta de ventana TP.



Gráfica 10. Fallos de página frente a velocidad de la imagen para la consulta de ventana TP.

La velocidad de desplazamiento de la imagen no tiene una influencia clara en los resultados. Podemos pensar que el aumento de la velocidad "comprime" los tiempos de influencia, haciéndolos menores, pero eso no evita el tener que procesarlos (y por tanto

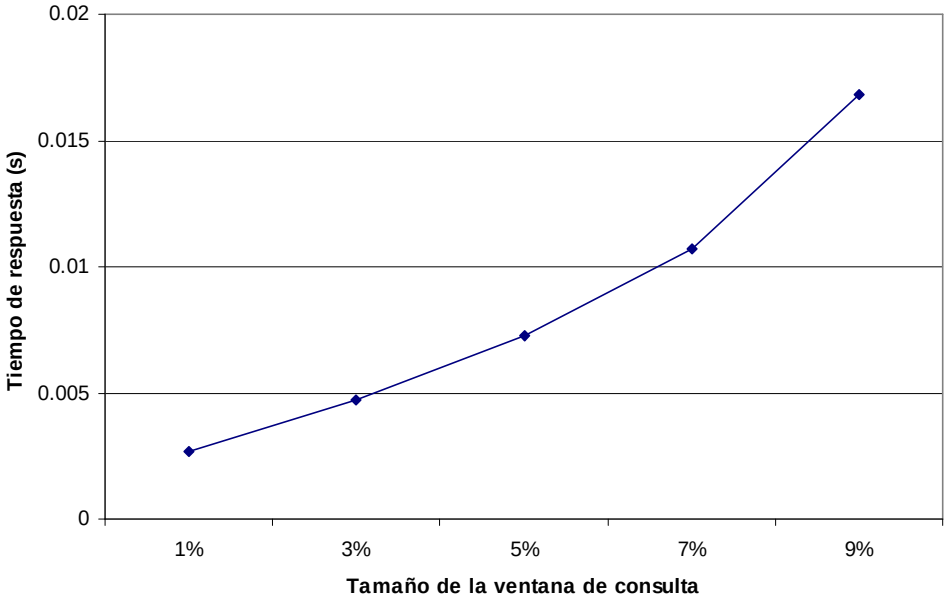
no reduce el tiempo de ejecución). Parece que a mayores velocidades la diferencia entre fallos de página de *DF simple* por un lado y *DF ordenación* y *BF* por otro aumenta. Se mantiene la tendencia de los demás experimentos entre los tres algoritmos.

4.2.2 Consulta de ventana continua

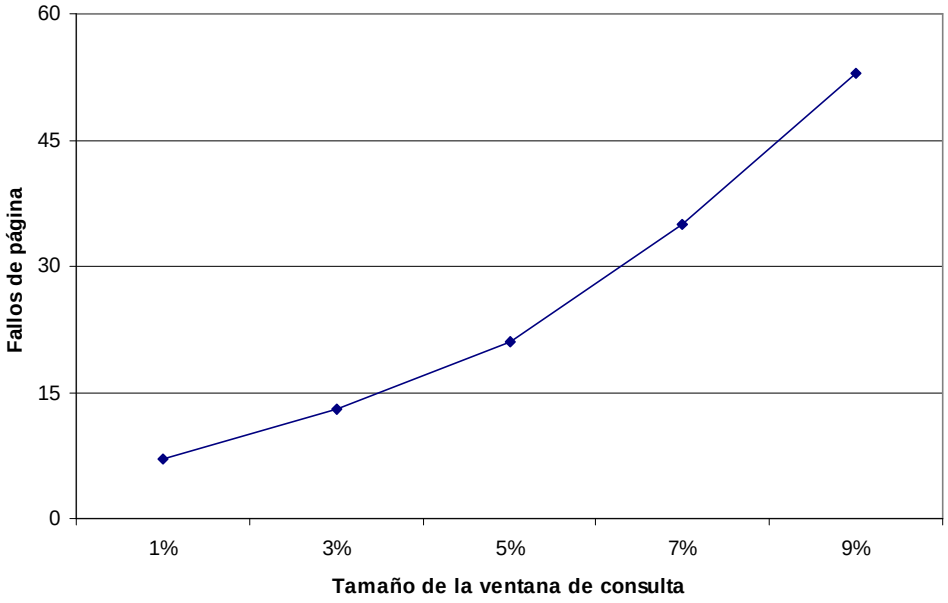
Estos son los resultados de los experimentos realizados con el algoritmo implementado para responder a la consulta de ventana continua: CWQ.

Para la serie de experimentos realizados con este algoritmo se han tenido en cuenta las siguientes variables: tamaño de la ventana de consulta (expresado en % del "universo" de datos en cada dimensión), tamaño de caché, tamaño de intervalo de consulta, instante de consulta, y velocidad de la imagen. En este caso no hay conjuntos de datos reales, puesto que todas las ventanas de consulta son generadas de manera aleatoria.

Experimento 1. Variación del tamaño de la ventana. Se han fijado las siguientes variables: tamaño de caché 25, intervalo [0, 5], velocidad de la imagen (1, 1). El tamaño de la ventana de consulta toma los valores: 1%, 3%, 5%, 7%, 9%.



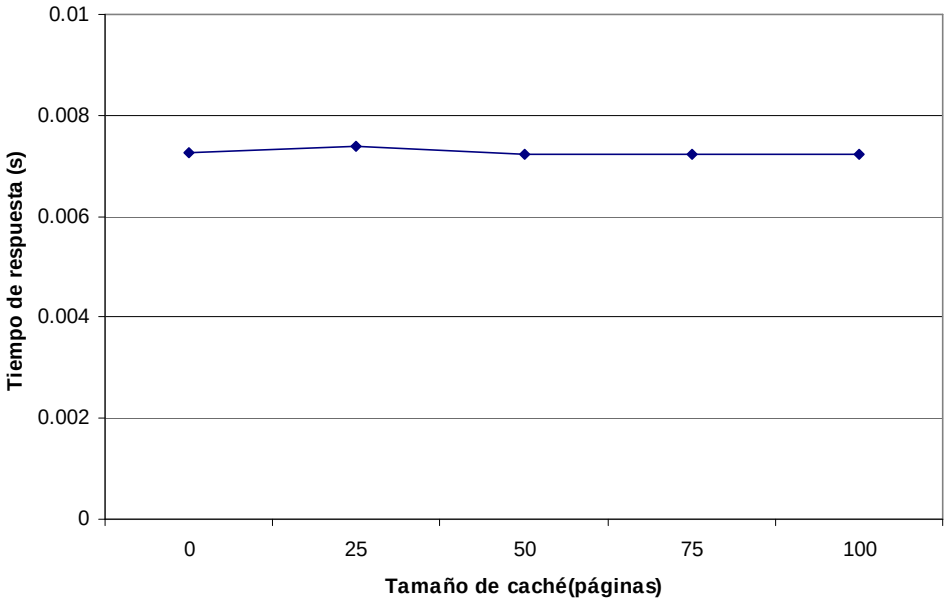
Gráfica 11. Tiempo de respuesta frente a tamaño de ventana para la consulta de ventana continua.



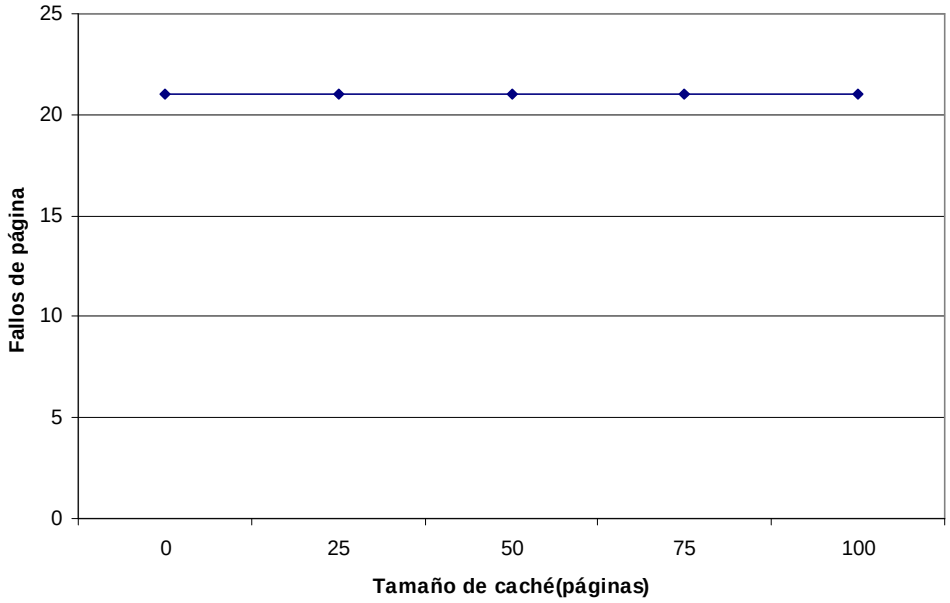
Gráfica 12. Fallos de página frente a tamaño de ventana para la consulta de ventana continua.

Tanto el tiempo de respuesta como el número de fallos crecen de manera lineal con el tamaño de la ventana (recordemos que los valores en el eje de abscisas indica la proporción de tamaño entre la ventana y el espacio de datos en cada dimensión). De esta manera coincide con el modelo de coste propuesto para TPR*-trees [TPS03] del apartado 2.7.2 (ecuación (3)).

Experimento 2. Variación del tamaño de caché. Se han fijado las siguientes variables: tamaño de la ventana 5%, intervalo [0, 5], velocidad de la imagen (1, 1). El tamaño de caché toma los valores: 0, 25, 50, 75, 100.



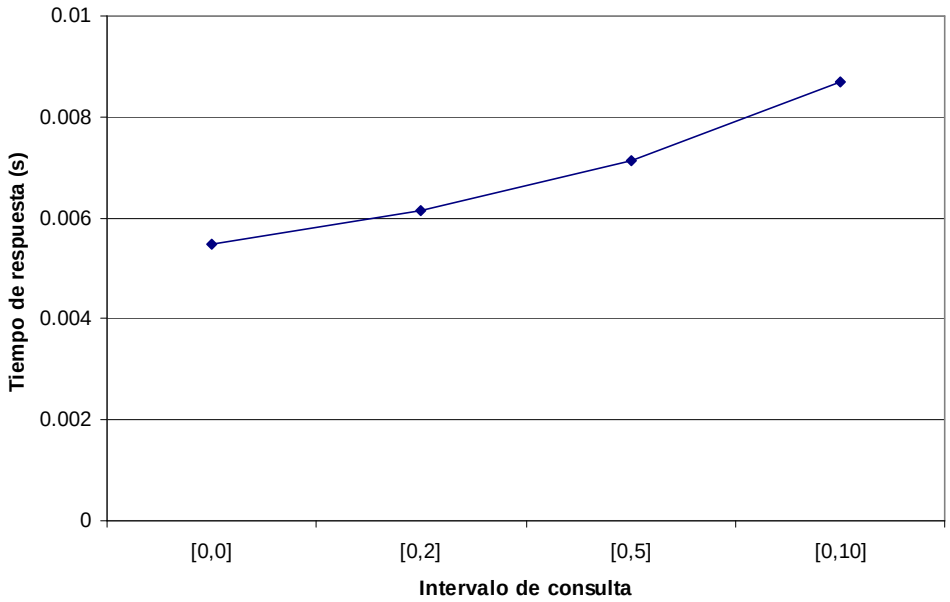
Gráfica 13. Tiempo de respuesta frente a tamaño de caché para la consulta de ventana continua.



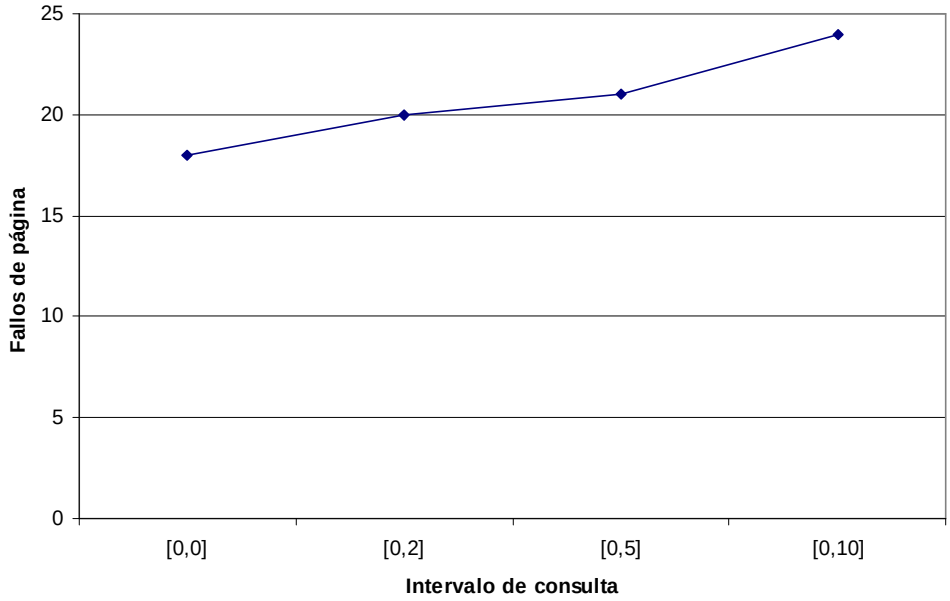
Gráfica 14. Fallos de página frente a tamaño de caché para la consulta de ventana continua.

El tamaño de la caché no afecta al comportamiento del algoritmo a causa de los pocos accesos a disco necesarios para responder a la consulta.

Experimento 3. Variación del intervalo de consulta. Se han fijado las siguientes variables: tamaño de caché 25, tamaño de la ventana 5%, velocidad de la imagen (1, 1). El intervalo de consulta toma los valores: [0, 0], [0, 2], [0, 5], [0, 10].



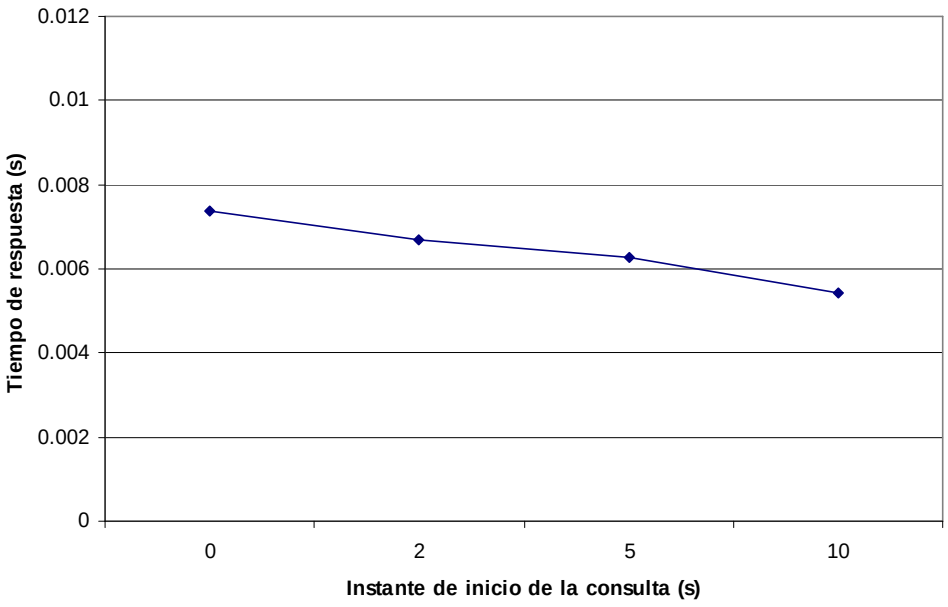
Gráfica 15. Tiempo de respuesta frente a tamaño de intervalo para la consulta de ventana continua.



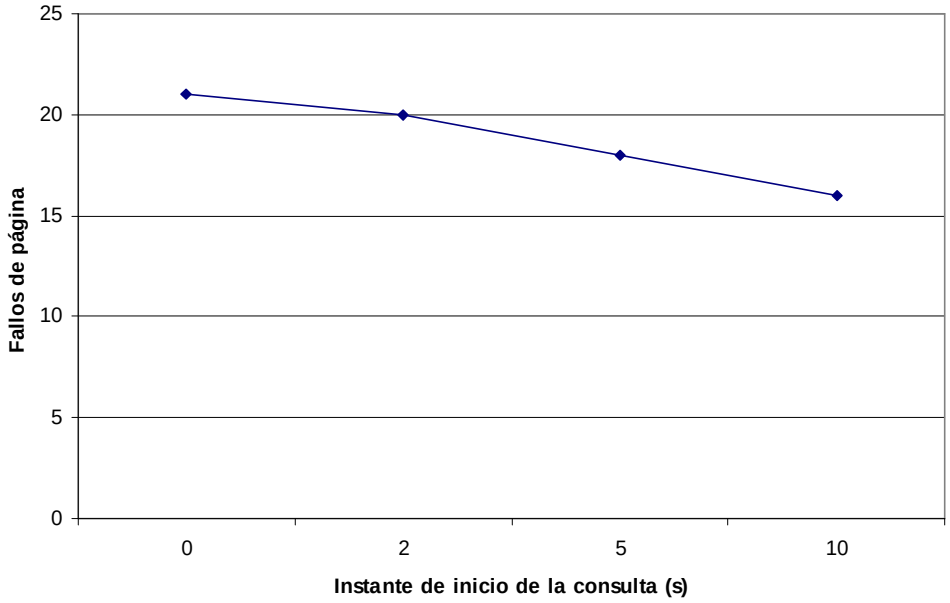
Gráfica 16. Fallos de página frente a tamaño de intervalo para la consulta de ventana continua.

Este algoritmo devuelve cambios durante el intervalo, por tanto, cuanto mayor sea el tamaño del intervalo mayor número de resultados a devolver, lo que aumenta el tiempo de respuesta y el número de fallos. Este comportamiento se refleja en las gráficas. También vemos como la característica incremental del algoritmo se muestra en la poca pendiente de la línea.

Experimento 4. Variación del instante de consulta. Se han fijado las siguientes variables: tamaño de caché 25, tamaño de la ventana 5%, velocidad de la imagen (1, 1). El intervalo de consulta toma los siguientes valores: [0, 5], [2, 7], [5, 10], [10, 15].



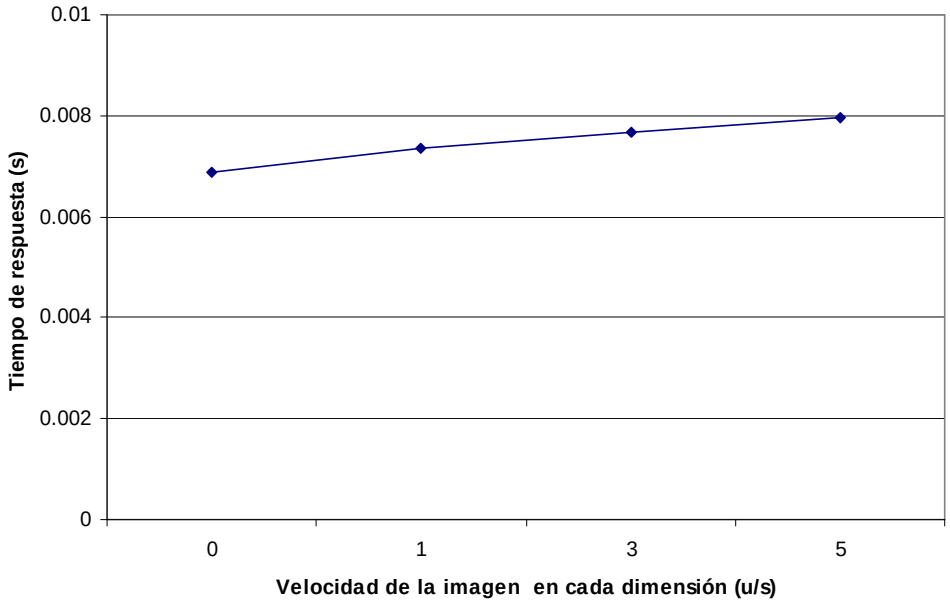
Gráfica 17. Tiempo de respuesta frente a inicio de intervalo para la consulta de ventana continua.



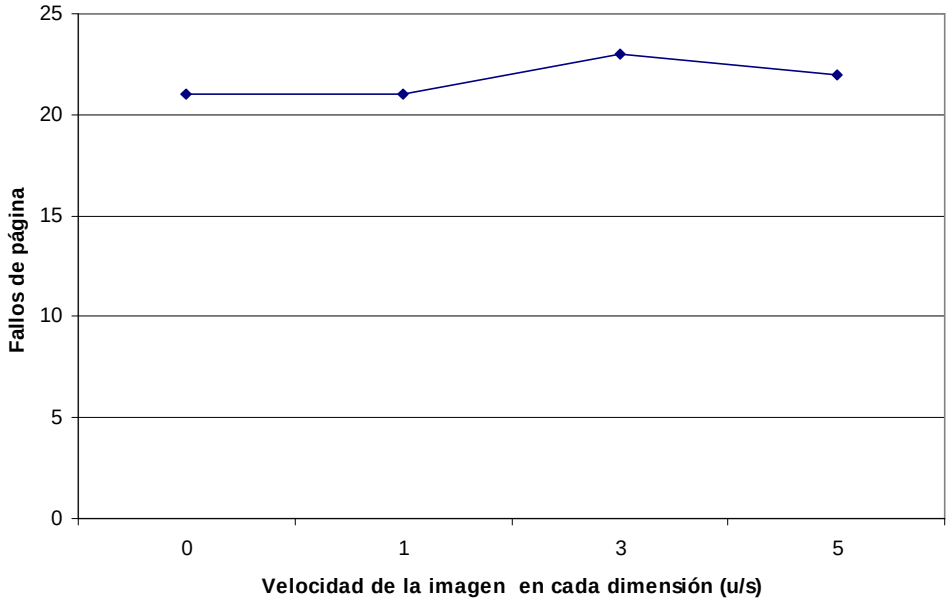
Gráfica 18. Fallos de página frente a inicio de intervalo para la consulta de ventana continua.

Retrasar el inicio de la consulta supone que la ventana de consulta tenga una mayor distancia a la imagen, reduciendo la probabilidad de solape, por tanto disminuyen los fallos de página y el tiempo de respuesta.

Experimento 5. Variación de la velocidad de la imagen. Se han fijado las siguientes variables: tamaño de caché 25, tamaño de la ventana 5%, intervalo [0, 5]. La velocidad de la imagen toma los siguientes valores: (0, 0), (1, 1), (3, 3), (5, 5).



Gráfica 19. Tiempo de respuesta frente a velocidad de la imagen para la consulta de ventana continua.



Gráfica 20. Fallos de página frente a velocidad de la imagen para la consulta de ventana continua.

Aumentar la velocidad de la imagen puede acelerar los cambios y por tanto para un intervalo dado aumentar el número de resultados a devolver. Esto puede repercutir en el tiempo de ejecución y en los accesos a disco (que no son directamente proporcionales a los fallos de página).

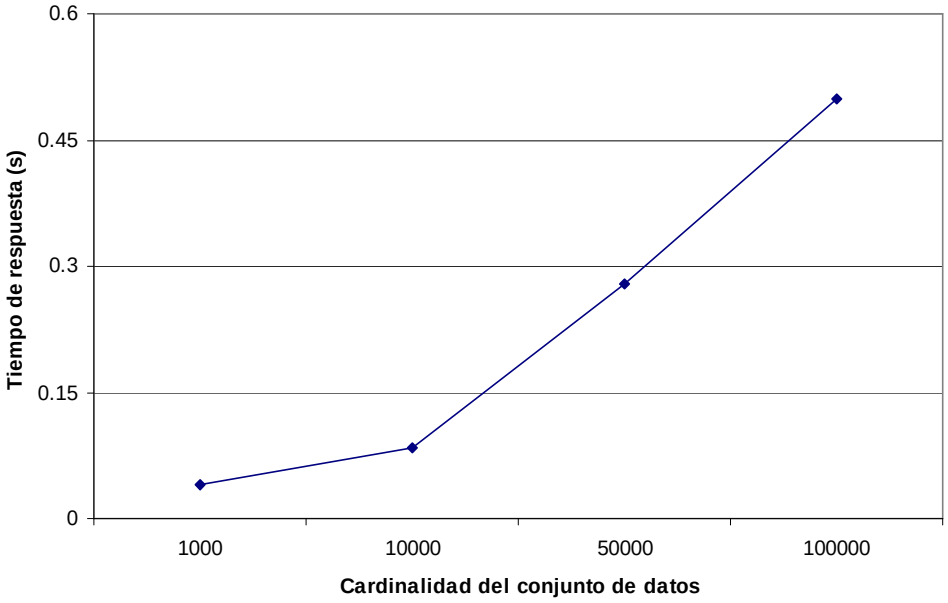
4.3 Consultas de los k vecinos más próximos en entornos dinámicos

4.3.1 Consulta de los k vecinos más próximos TP

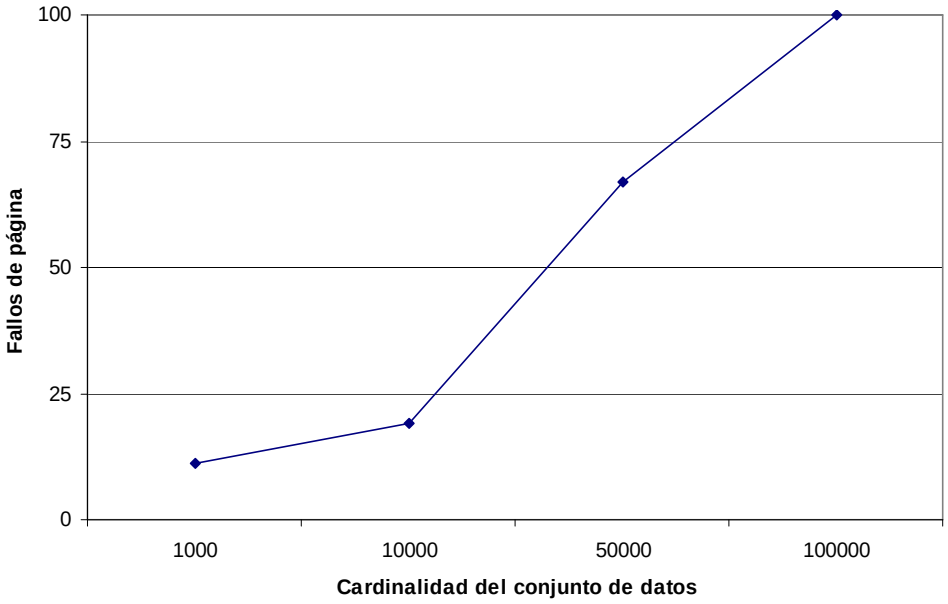
Estos son los resultados de los experimentos realizados con el algoritmo implementado para responder a la consulta de los k vecinos más próximos TP: TP_k_NN_BF.

Para la serie de experimentos realizados con este algoritmo se han tenido en cuenta las siguientes variables: cardinalidad del conjunto de datos, tamaño de caché, tamaño de intervalo de consulta, instante de consulta y k.

Experimento 1. Variación de la cardinalidad. Se han fijado las siguientes variables: tamaño de caché 25, intervalo [0, 5], k = 5. La cardinalidad del conjunto de datos toma los siguientes valores: 1.000, 10.000, 50.000, 100.000.



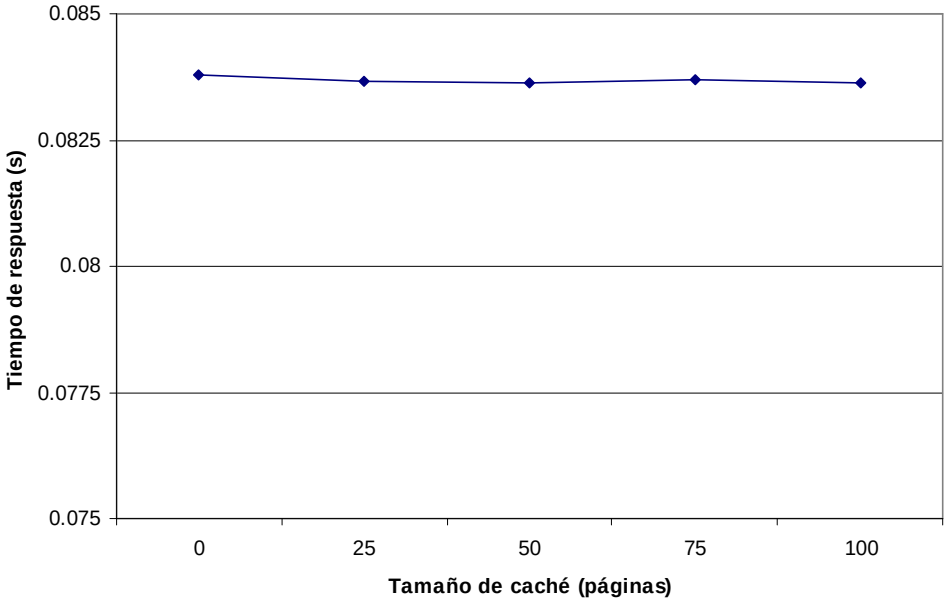
Gráfica 21. Tiempo de respuesta frente a cardinalidad para la consulta kNN TP.



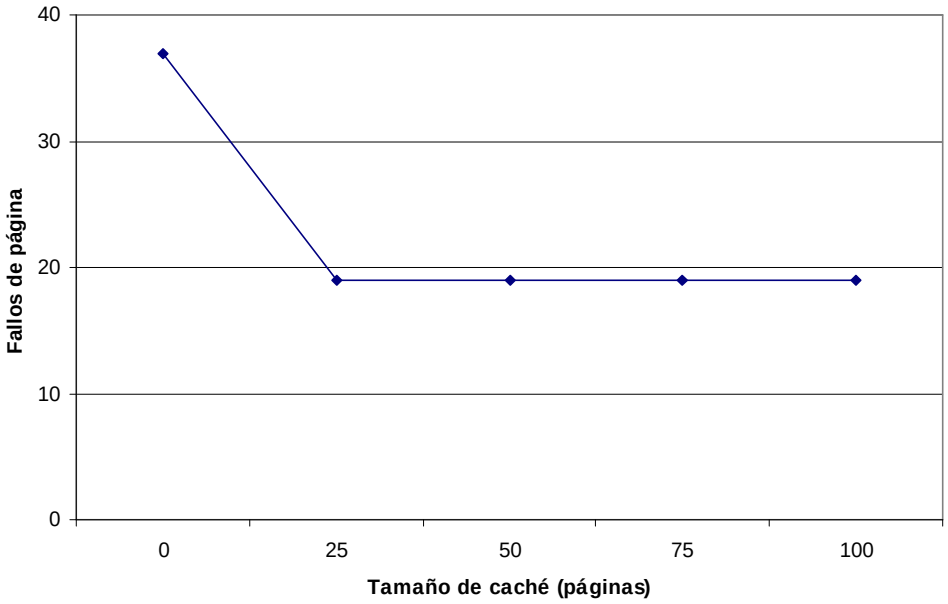
Gráfica 22. Fallos de página frente a cardinalidad para la consulta de kNN TP.

Como es de esperar, tanto el tiempo de respuesta como el número de fallos crecen casi linealmente con el tamaño del conjunto de datos.

Experimento 2. Variación del tamaño de caché. Se han fijado las siguientes variables: cardinalidad del conjunto de datos 10.000, intervalo [0, 5], k = 5. El tamaño de caché toma los siguientes valores: 0, 25, 50, 75, 100.



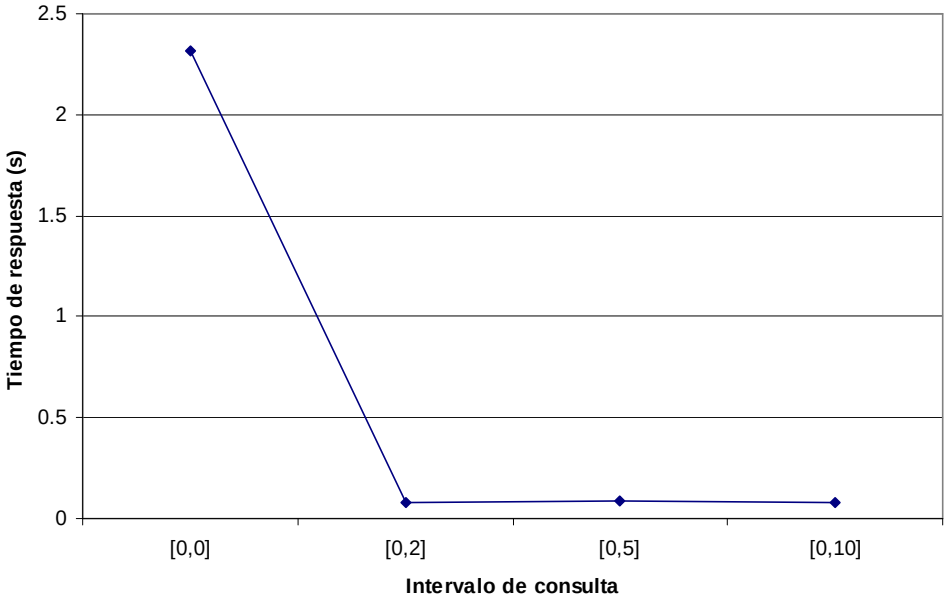
Gráfica 23. Tiempo de respuesta frente a tamaño de caché para la consulta de kNN TP.



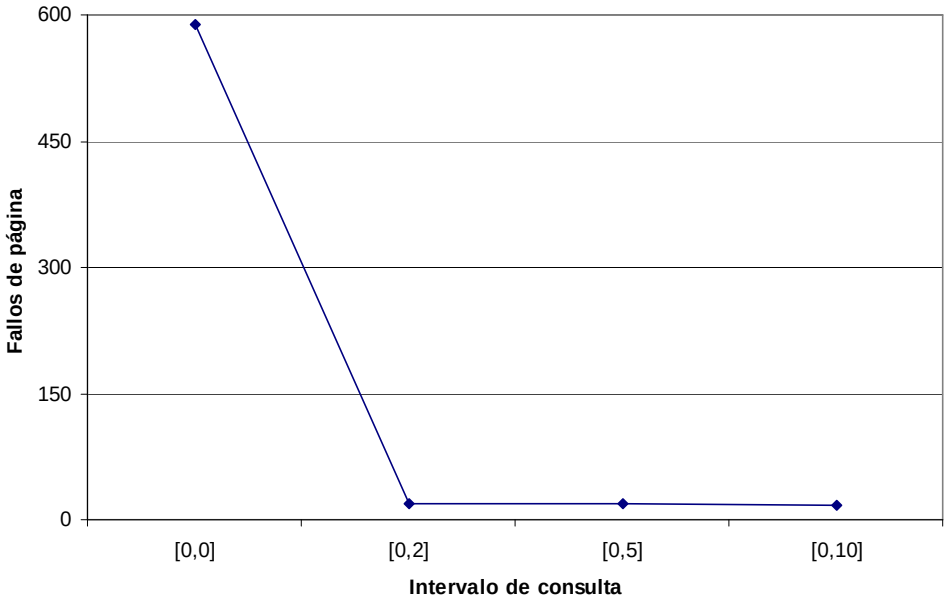
Gráfica 24. Fallos de página frente a tamaño de caché para la consulta de kNN TP.

La variación del tamaño de caché no influye en el tiempo de respuesta. El número de fallos de página si varía aunque se estabiliza muy pronto. También vemos aquí que el número de accesos a disco necesarios para responder a la consulta es pequeño

Experimento 3. Variación del intervalo de consulta. Se han fijado las siguientes variables: tamaño de caché 25, cardinalidad del conjunto de datos 10.000, k = 5. El intervalo de consulta toma los siguientes valores: [0, 0], [0, 2], [0, 5], [0, 10].



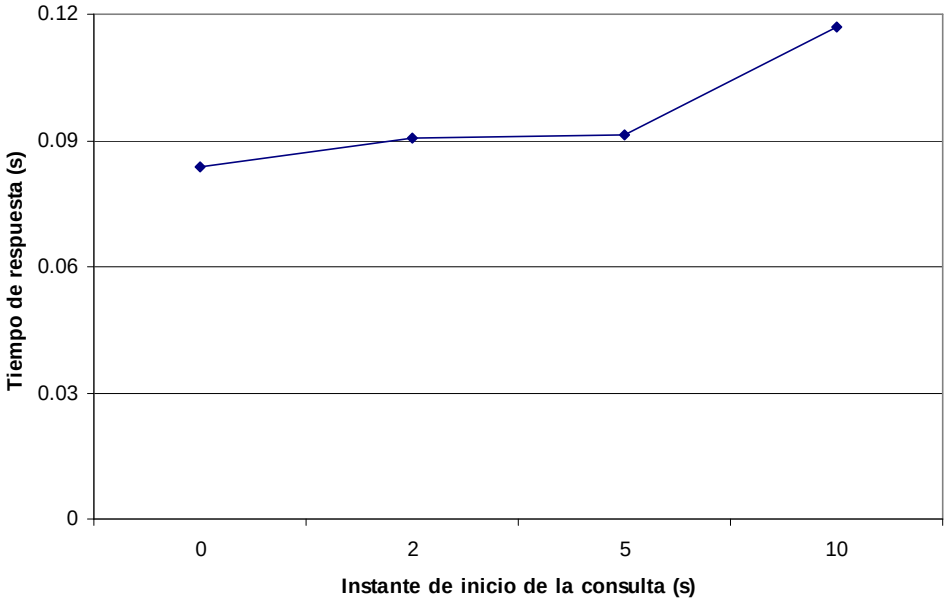
Gráfica 25. Tiempo de respuesta frente a tamaño de intervalo para la consulta de kNN TP.



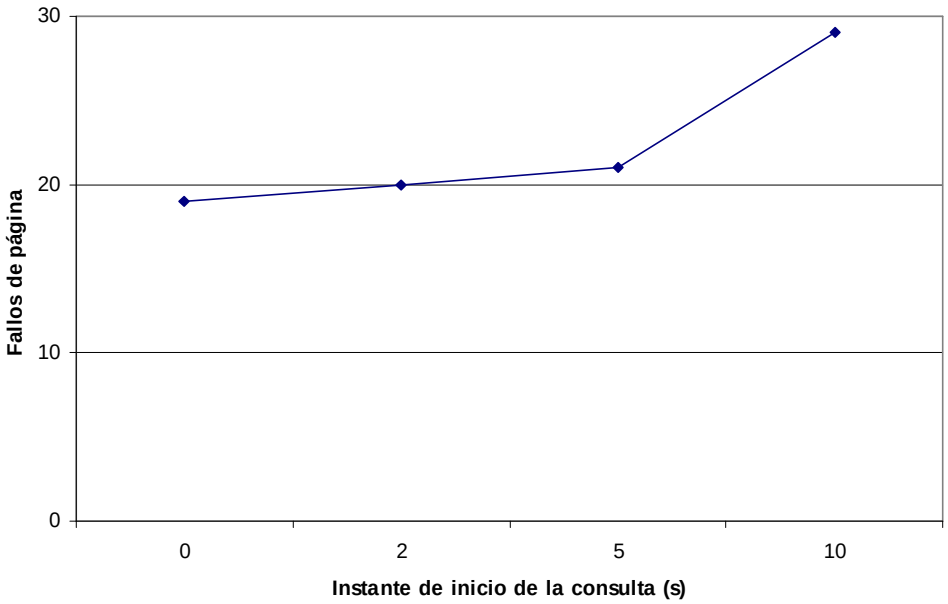
Gráfica 26. Fallos de página frente a tamaño de intervalo para la consulta de kNN TP.

En el intervalo [0, 0] (y en general en intervalos "puntuales") los valores de tiempo de respuesta y fallos de página se disparan debido a que toda la componente R de la consulta se procesa. Para intervalos mayores este efecto no se produce y los valores son reducidos y constantes, ya que sólo se busca la componente TP, que será la misma independientemente de la longitud del intervalo. Por tanto, no es buena idea usar este algoritmo para realizar consultas instantáneas.

Experimento 4. Variación del instante de consulta. Se han fijado las siguientes variables: tamaño de caché 25, cardinalidad del conjunto de datos 10.000, k = 5. El intervalo de consulta toma los siguientes valores: [0, 5], [2, 7], [5, 10], [10, 15].



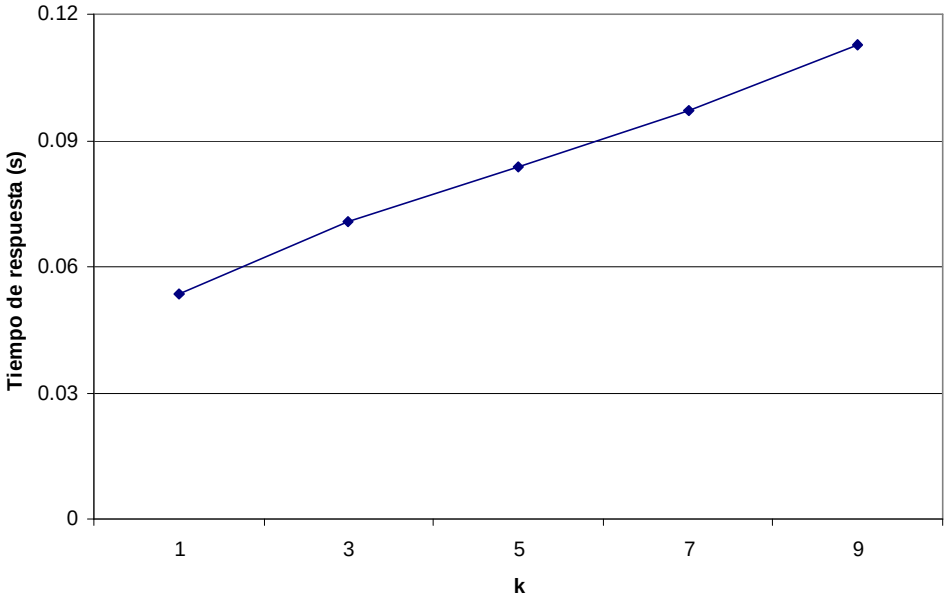
Gráfica 27. Tiempo de respuesta frente a inicio de intervalo para la consulta de kNN TP.



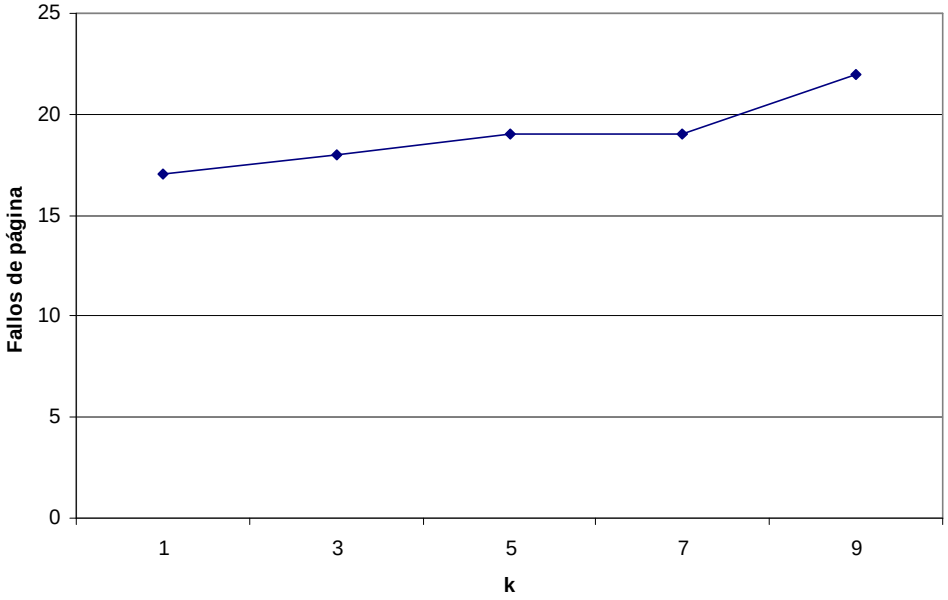
Gráfica 28. Fallos de página frente a inicio de intervalo para la consulta de kNN TP.

Retrasar el instante de inicio de la consulta hace que los MBRs de los objetos se expandan, de manera que se producen más solapes y es necesario acceder a más nodos para su comprobación. Por tanto aumentan tanto el número de fallos de página, así como el tiempo de respuesta empleado en procesar los objetos solapados.

Experimento 5. Variación de k. Se han fijado las siguientes variables: tamaño de caché 25, intervalo [0, 5], cardinalidad del conjunto de datos 10.000. El número de vecinos k toma los siguientes valores: 1, 3, 5, 7, 9.



Gráfica 29. Tiempo de respuesta frente a k para la consulta de kNN TP.



Gráfica 30. Fallos de página frente a k para la consulta de kNN TP.

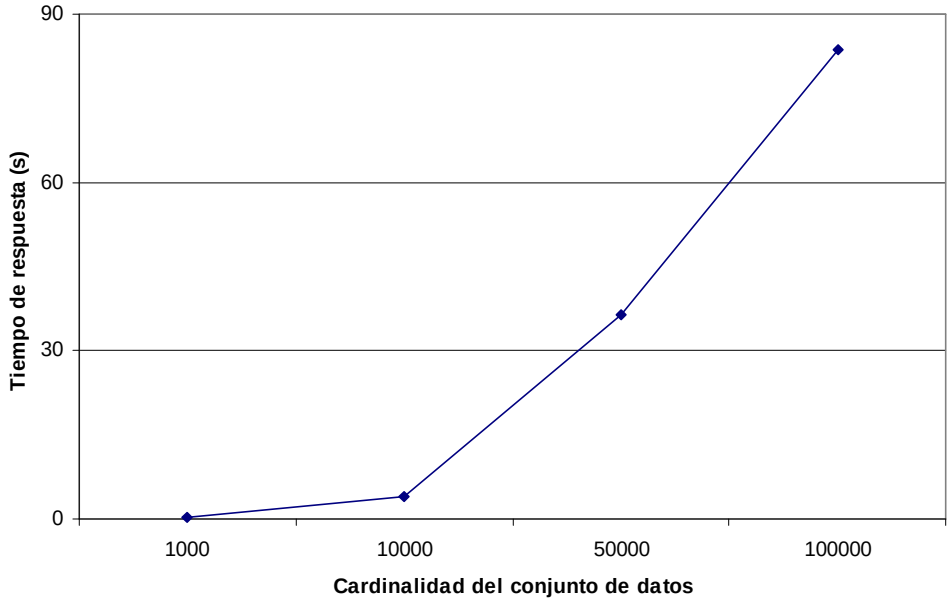
El tiempo de respuesta crece linealmente con k por el peso del cálculo de la componente convencional R del resultado, cuyo tamaño es proporcional a k. Por otra parte el número de fallos de página crece de manera casi lineal también, aunque como hemos indicado antes los fallos de página no son directamente proporcionales a los accesos a disco.

4.3.2 Consulta de los k vecinos más próximos continua

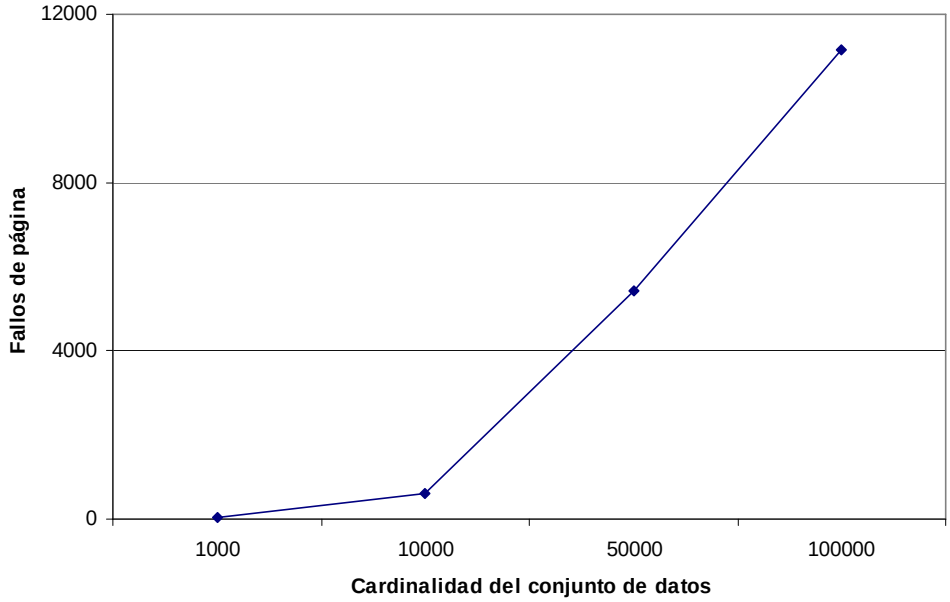
Estos son los resultados de los experimentos realizados con el algoritmo implementado para responder a la consulta de los k vecinos más próximos continua.

Para la serie de experimentos realizados con este algoritmo se han tenido en cuenta las siguientes variables: cardinalidad del conjunto de datos, tamaño de caché, tamaño de intervalo de consulta, instante de consulta y k.

Experimento 1. Variación de la cardinalidad. Se han fijado las siguientes variables: tamaño de caché 25, intervalo [0, 5], k = 5. La cardinalidad del conjunto de datos toma los siguientes valores: 1.000, 10.000, 50.000, 100.000.



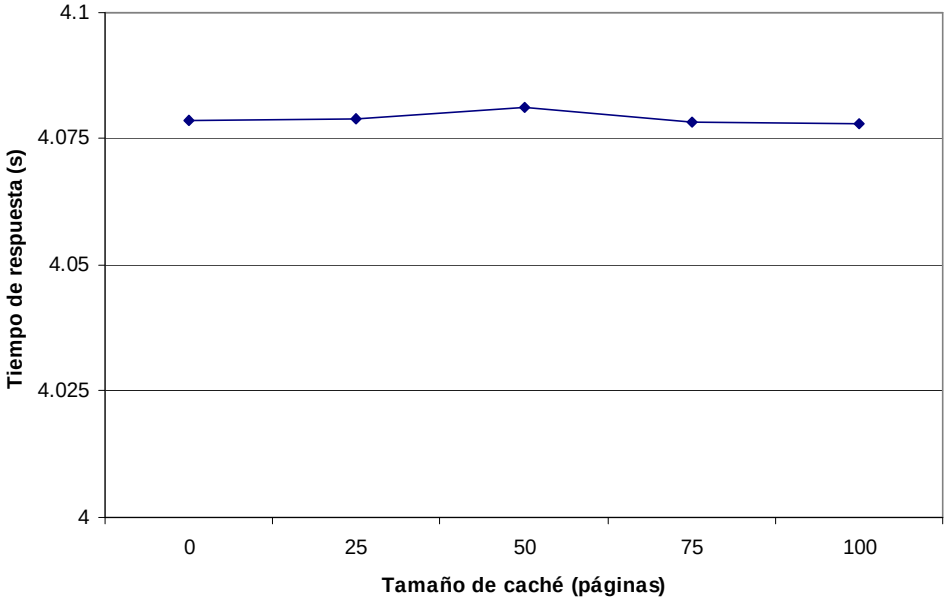
Gráfica 31. Tiempo de respuesta frente a cardinalidad para la consulta kNN Continua.



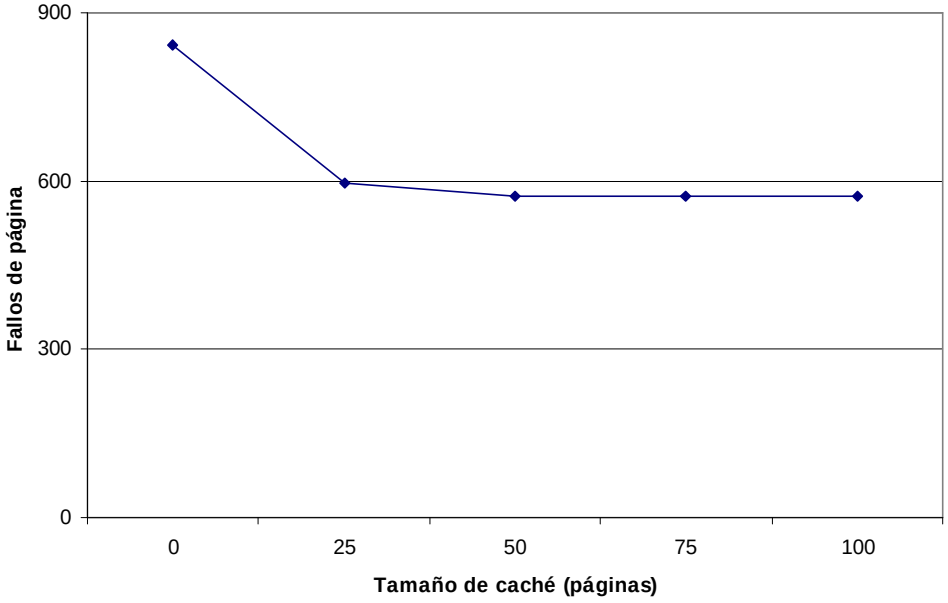
Gráfica 32. Fallos de página frente a cardinalidad para la consulta de kNN Continua.

Como es de esperar el tiempo de respuesta y los fallos de página crecen con la cardinalidad del conjunto de datos, ya que cuanto mayor es esta más objetos habrá que procesar.

Experimento 2. Variación del tamaño de caché. Se han fijado las siguientes variables: cardinalidad del conjunto de datos 10.000, intervalo [0, 5], k = 5. El tamaño de caché toma los siguientes valores: 0, 25, 50, 75, 100.



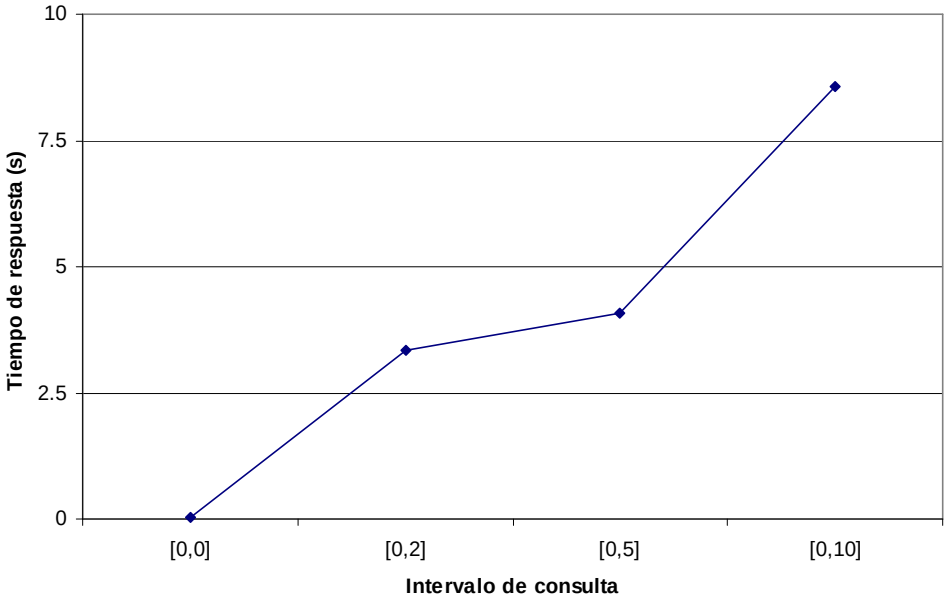
Gráfica 33. Tiempo de respuesta frente a tamaño de caché para la consulta de kNN Continua.



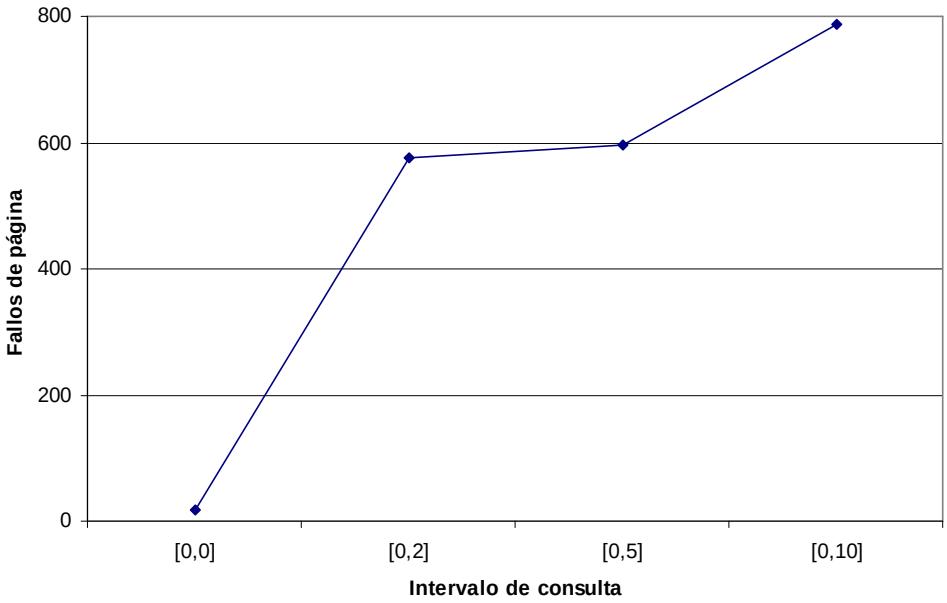
Gráfica 34. Fallos de página frente a tamaño de caché para la consulta de kNN Continua.

La variación del tamaño de caché afecta muy poco al tiempo de respuesta. Como en la consulta TP, el número de fallos de página se estabiliza relativamente pronto.

Experimento 3. Variación del intervalo de consulta. Se han fijado las siguientes variables: tamaño de caché 25, cardinalidad del conjunto de datos 10.000, k = 5. El intervalo de consulta toma los siguientes valores: [0, 0], [0, 2], [0, 5], [0, 10].



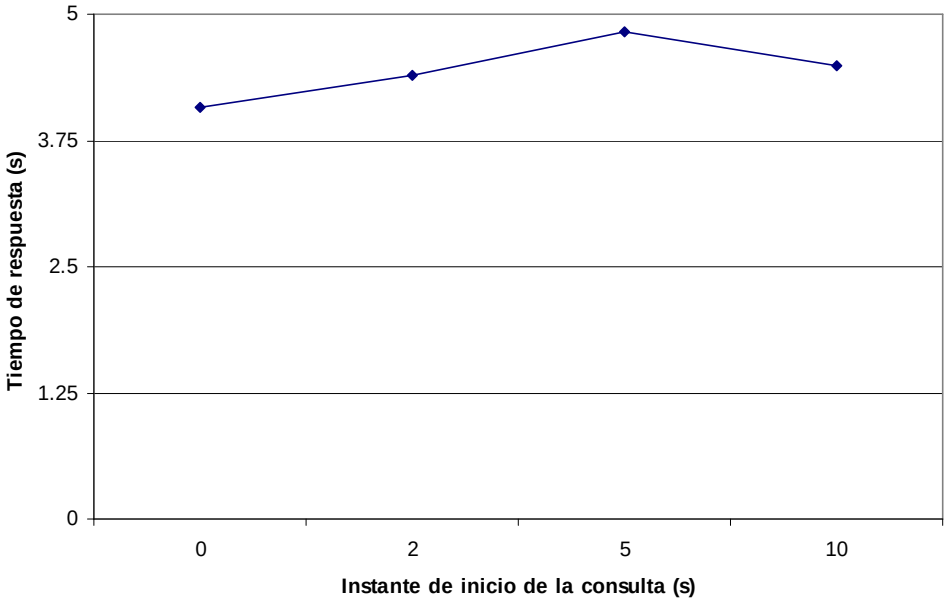
Gráfica 35. Tiempo de respuesta frente a tamaño de intervalo para la consulta de kNN Continua.



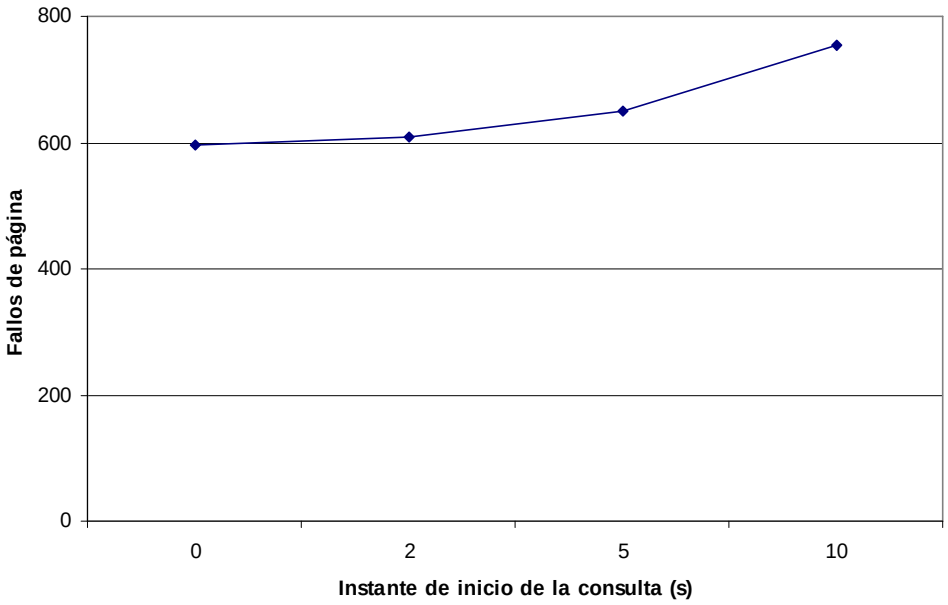
Gráfica 36. Fallos de página frente a tamaño de intervalo para la consulta de kNN Continua.

Cuanto mayor es el intervalo de consulta más cambios se producen y por tanto mayor es el resultado a devolver. Por este motivo aumentan tanto el tiempo de respuesta como los fallos de página. Recordemos que es un algoritmo repetitivo y por tanto el coste de obtener los cambios sucesivos durante cada intervalo es mayor que el los algoritmos incrementales como el de consulta de ventana continua.

Experimento 4. Variación del instante de consulta. Se han fijado las siguientes variables: tamaño de caché 25, cardinalidad del conjunto de datos 10.000, k = 5. El intervalo de consulta toma los siguientes valores: [0, 5], [2, 7], [5, 10], [10, 15].



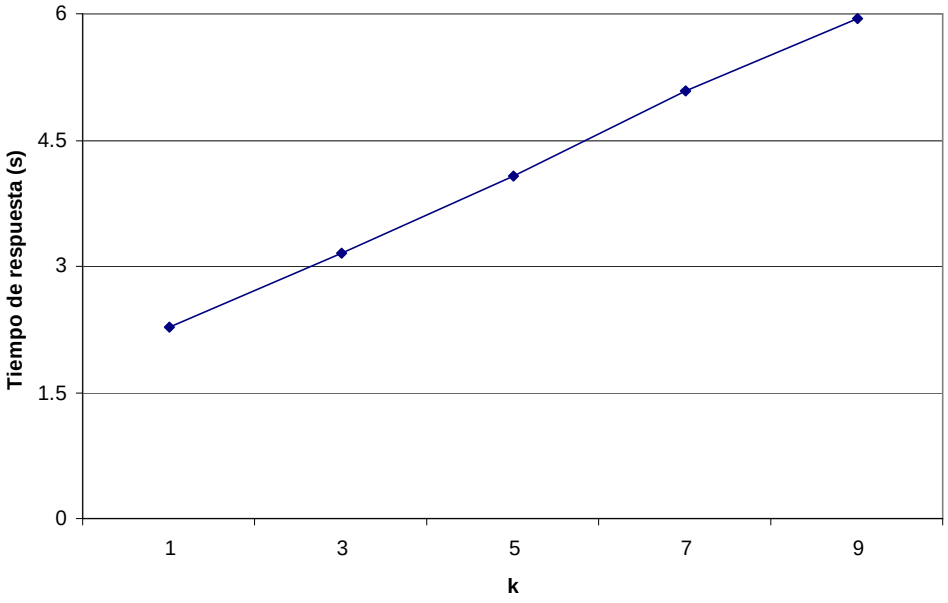
Gráfica 37. Tiempo de respuesta frente a inicio de intervalo para la consulta de kNN Continua.



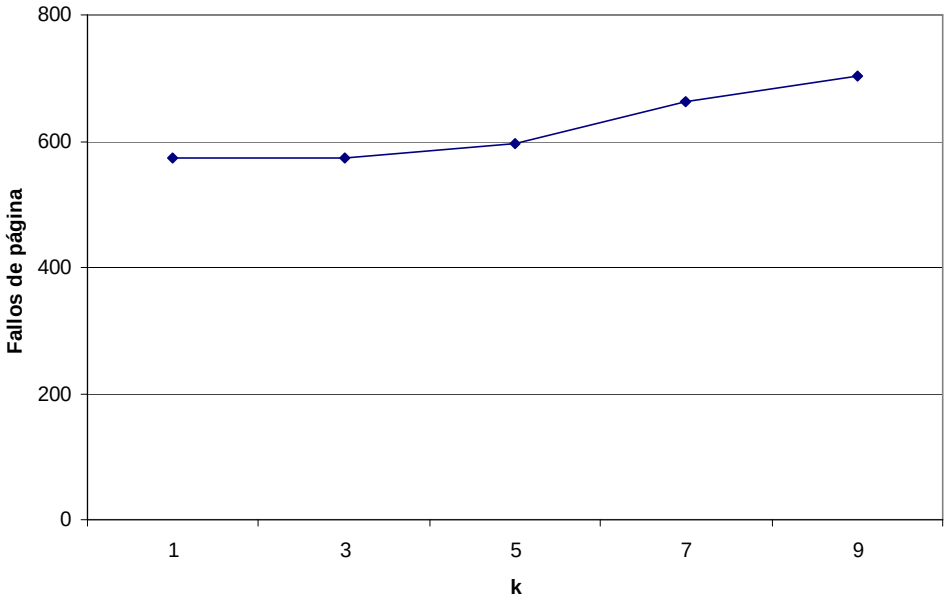
Gráfica 38. Fallos de página frente a inicio de intervalo para la consulta de kNN Continua.

El retraso del inicio de la consulta aumenta los solapes y por tanto el tiempo de respuesta y el número de fallos de página también crece.

Experimento 5. Variación de k. Se han fijado las siguientes variables: tamaño de caché 25, intervalo [0, 5], cardinalidad del conjunto de datos 10.000. El número de vecinos k toma los siguientes valores: 1, 3, 5, 7, 9.



Gráfica 39. Tiempo de respuesta frente a k para la consulta de kNN Continua.



Gráfica 40. Fallos de página frente a k para la consulta de kNN Continua.

El tiempo de respuesta crece de manera casi lineal con k como cabía esperar, debido al cálculo de la componente R. El número de fallos de página también crece con k pero no de forma tan directa, posiblemente debido a las posiciones de los objetos en los nodos.

4.4 Consultas de join en entornos dinámicos

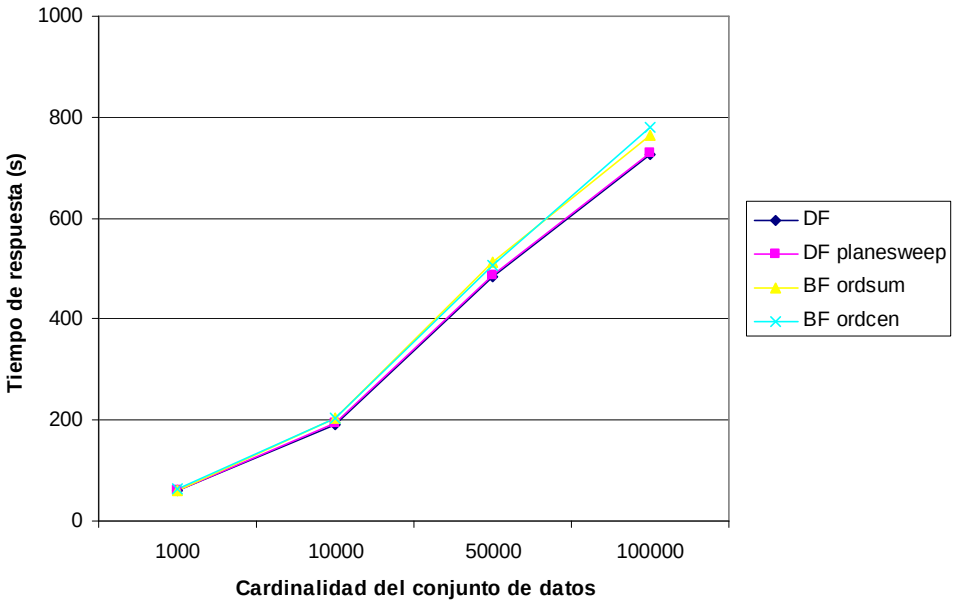
4.4.1 Consulta de join de intervalo

Estos son los resultados de los experimentos realizados con los algoritmos implementados para responder a la consulta join: Join DF simple (DF), Join DF plane sweep (DF planesweep), Join BF ordsum (BF ordsum) y Join BF ordcen (BF ordcen).

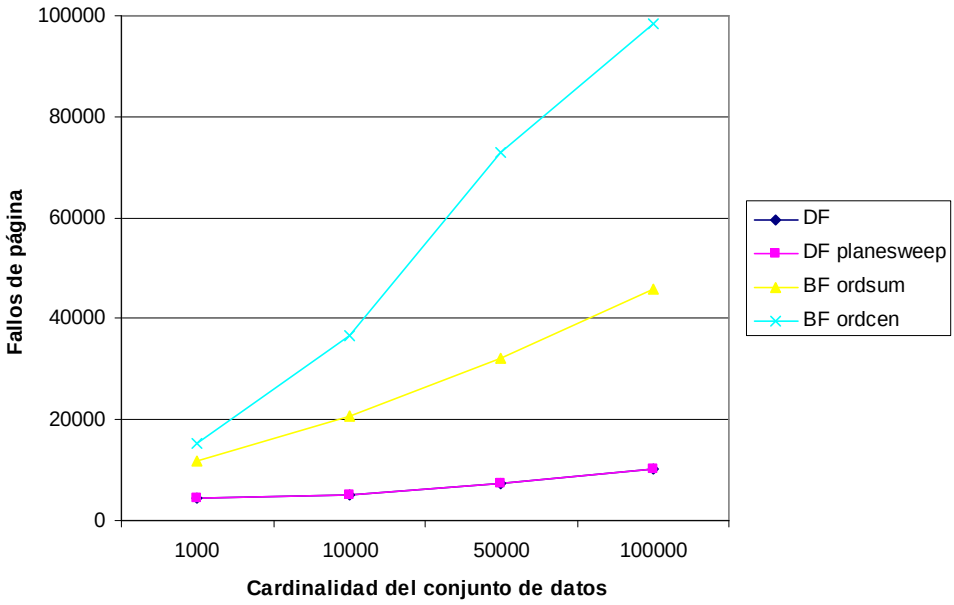
Para la serie de experimentos realizados con esto algoritmos se han tenido en cuenta las siguientes variables: cardinalidad del conjunto de datos, tamaño de caché, tamaño de intervalo de consulta, instante de consulta, y velocidad de la imagen. La cardinalidad del conjunto de datos reales ST es de 131461 objetos. Además, para todos los experimentos, salvo para el 3 (variación del tamaño del intervalo de consulta), los intervalos serán puntuales, es decir, se corresponderán con un instante en el tiempo.

Experimento 1. Variación de la cardinalidad. Se han fijado las siguientes variables: tamaño de caché 250, intervalo [5, 5], velocidad de la imagen (1, 1). La cardinalidad del conjunto de datos toma los siguientes valores: 1.000, 10.000, 50.000, 100.000.

Datos sintéticos



Gráfica 41. Tiempo de respuesta frente a cardinalidad para la consulta join en datos sintéticos.



Gráfica 42. Fallos de página frente a cardinalidad para la consulta de join en datos sintéticos.

Datos reales

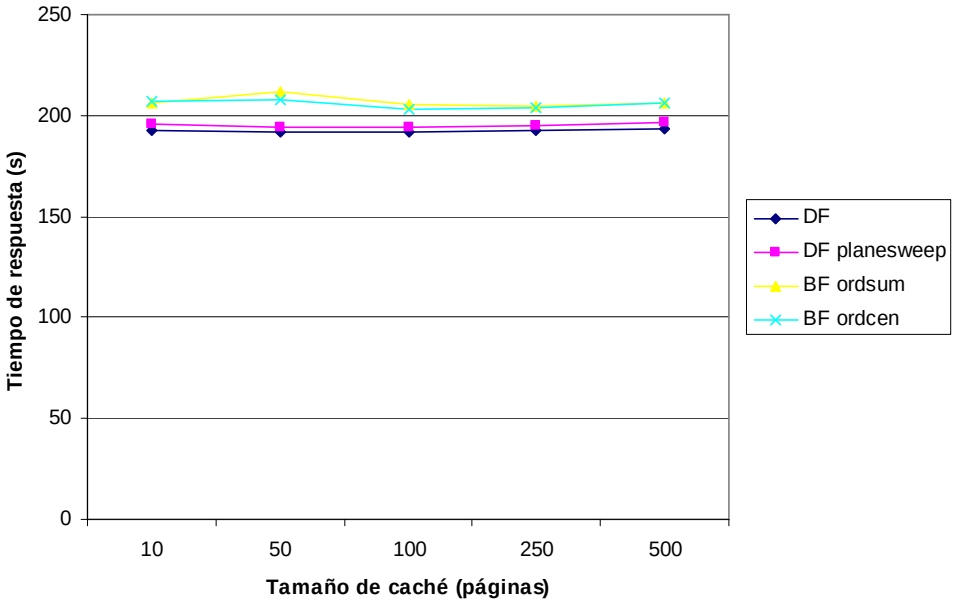
Al disponer de un único conjunto de datos reales no se ha realizado este experimento variando la cardinalidad del conjunto de datos.

Todos los algoritmos producen tiempos de ejecución prácticamente iguales, tardando un poco más los *BF*, que tienen que gestionar el IJI. Los fallos de página, aun con la caché del sistema, tienen cierto peso en el tiempo de ejecución. *DF planesweep* se acerca a *DF* conforme aumenta la cardinalidad.

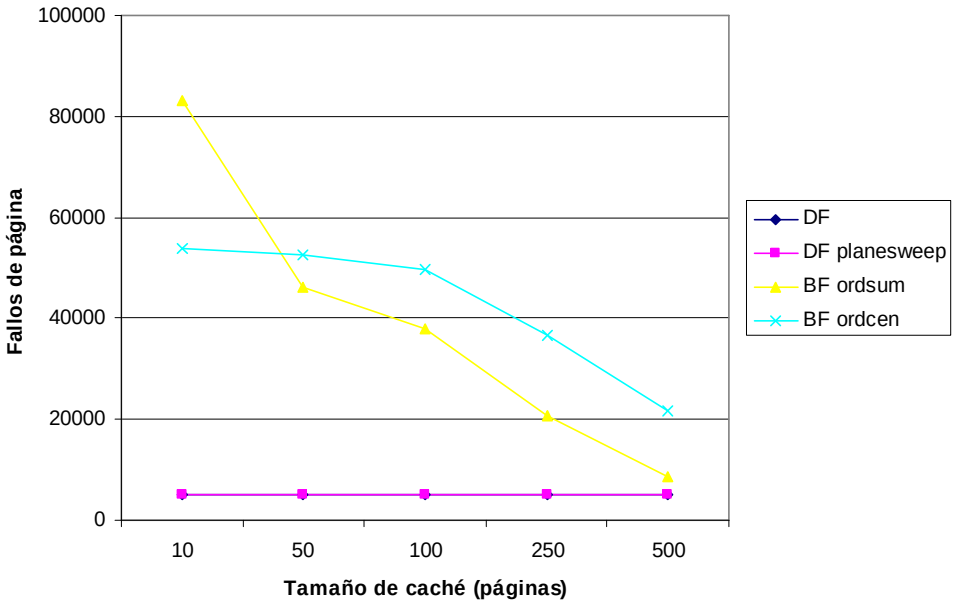
En cuanto a número de fallos de página ambos *DF* tienen el mismo comportamiento. Crecen muy lentamente con la cardinalidad. *BF ordsum* sin embargo digiere peor el aumento de la cardinalidad, multiplicando casi por 5 los fallos de *DF* para 100000 objetos, mientras que *BF ordcen* casi duplica a *BF ordsum*. Esto puede ser debido a dos factores. La recursividad para los *DF* y la ordenación en los *BF*. Los algoritmos recursivos tienen los nodos cargados en memoria en cada retroceso, por lo que ahorran una carga, además, siempre tienen una ordenación por uno de los árboles ya que leen las entradas secuencialmente en los nodos. Los algoritmos *BF* realizan una ordenación global y esto puede hacer que los saltos de un nodo a otro perjudiquen el orden de lectura. Por ejemplo, podemos ver que con el cambio de ordsum a ordcen doblamos el número de fallos, por tanto una ordenación mala a nivel global tiene grandes efectos en el número de fallos.

Experimento 2. Variación del tamaño de caché. Se han fijado las siguientes variables: cardinalidad del conjunto de datos 10.000, tamaño de caché 250, intervalo [5, 5], velocidad de la imagen (1, 1). El tamaño de caché toma los siguientes valores: 10, 50, 100, 250, 500.

Datos sintéticos



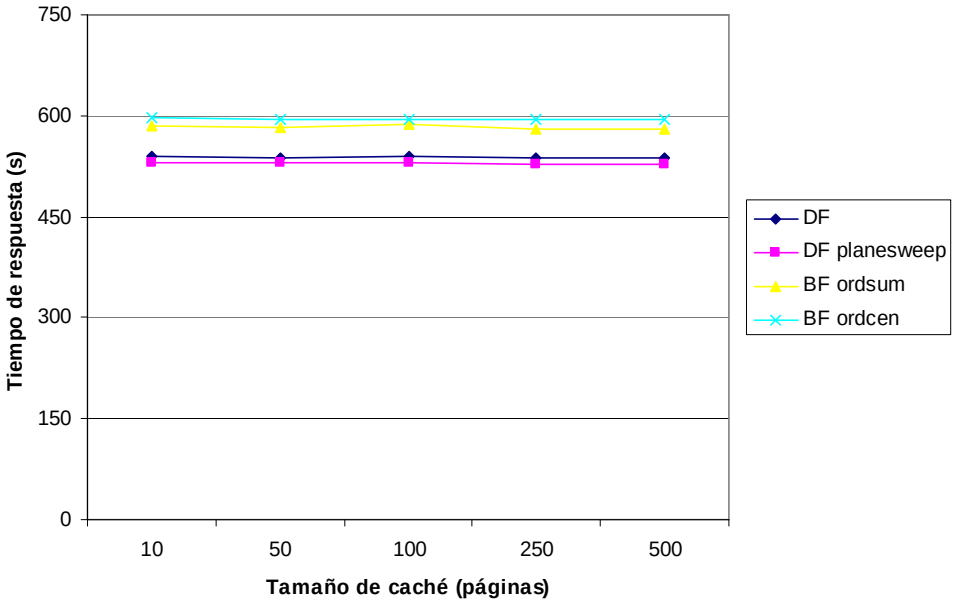
Gráfica 43. Tiempo de respuesta frente a tamaño de caché para la consulta de join en datos sintéticos.



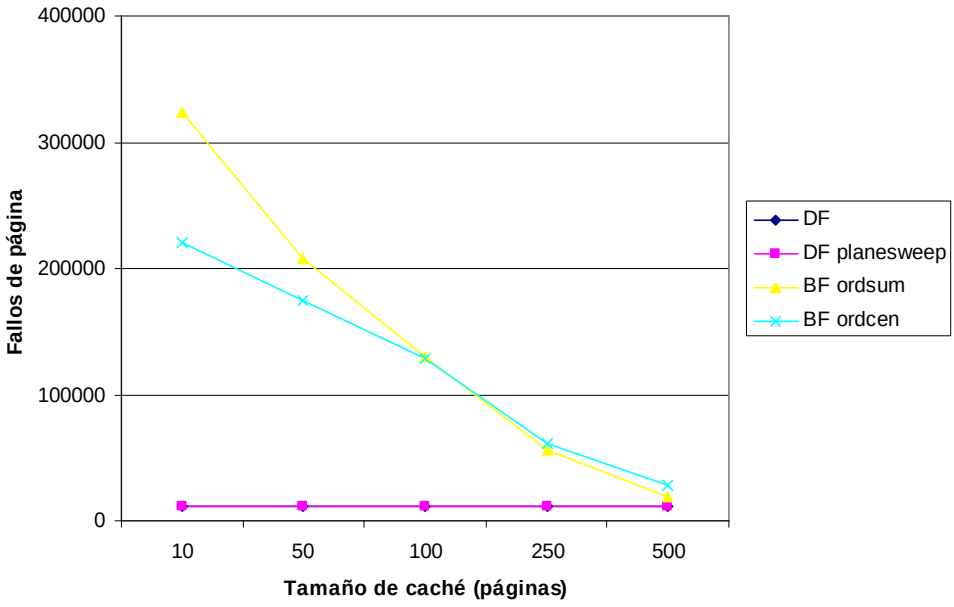
Gráfica 44. Fallos de página frente a tamaño de caché para la consulta de join en datos sintéticos.

Datos reales

Recordemos el conjunto de datos reales utilizado para los experimentos con los algoritmos de join espacio-temporal contiene 131461 MBRs y se la han añadido velocidades (sólo traslación) distribuidas uniformemente en [-v, v] siendo v el 0.5% del tamaño del universo de datos por unidad de tiempo.



Gráfica 45. Tiempo de respuesta frente a tamaño de caché para la consulta de join en datos reales.



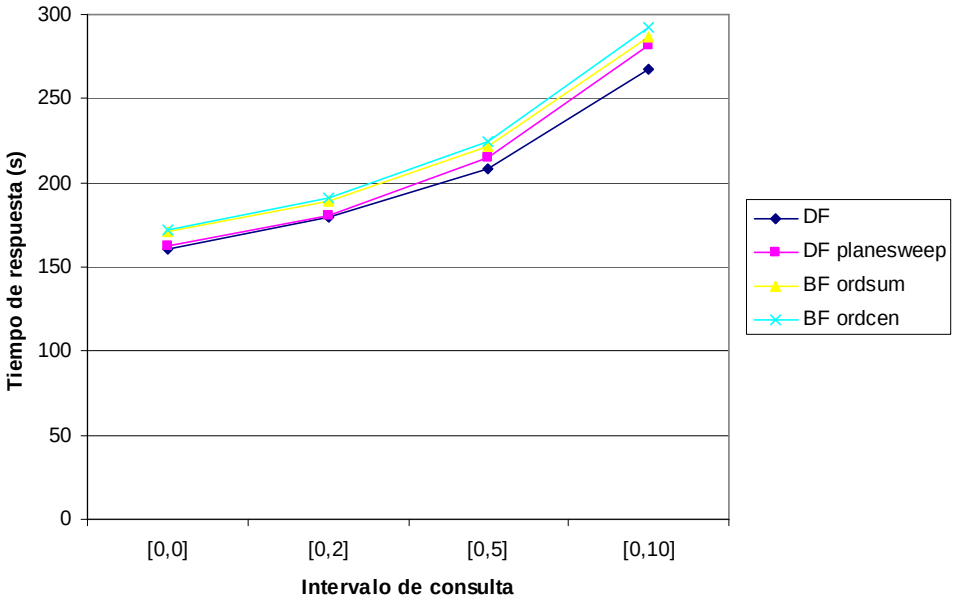
Gráfica 46. Fallos de página frente a tamaño de caché para la consulta de join en datos reales.

Tanto para datos reales como para sintéticos el tiempo de ejecución prácticamente no se ve afectado por los fallos de página. Los cuatro algoritmos tienen un comportamiento parecido, siendo algo mejores los *DF* que los *BF* de nuevo. El reducido número de

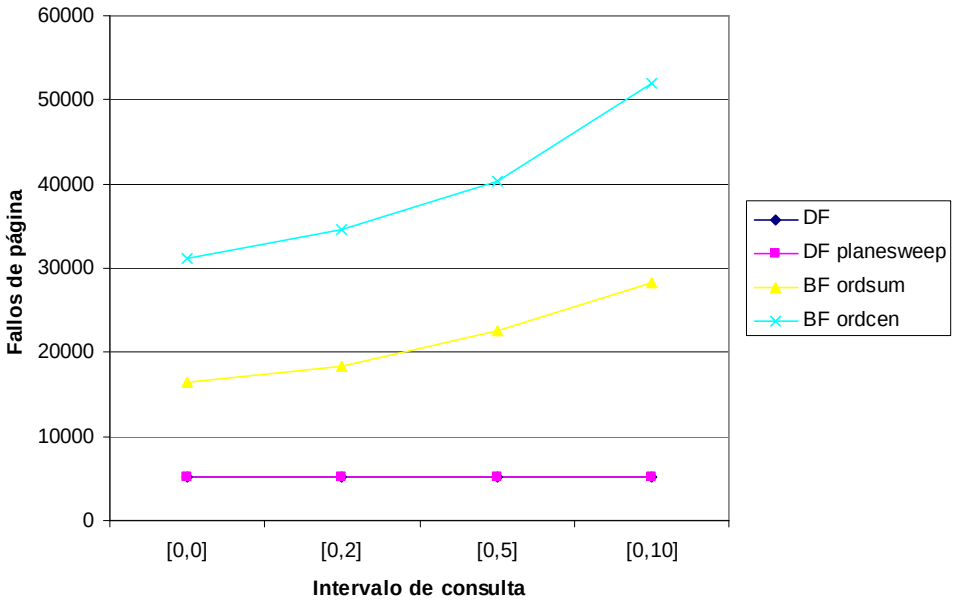
fallos de página de los algoritmos *DF* hace que no se vean afectados por el tamaño de la caché. Los *BF* son diferentes. *BF ordcen* es muy sensible a los cambios en el tamaño de la caché. Para cachés pequeñas es el peor, con más de 80000 fallos para caché de 10 páginas, pero cae conforma aumentamos al tamaño hasta casi igualar a los *DF* para cachés de 500 páginas. *BF ordcen* comienza mejor que *BF ordsum* (60000 fallos) y se muestra estable hasta cachés de 100 páginas. A partir de ahí comienza a caer pero con peores resultados que *BF ordsum*.

Experimento 3. Variación del intervalo de consulta. Se han fijado las siguientes variables: cardinalidad del conjunto de datos 10.000, tamaño de caché 250, velocidad de la imagen (1, 1). El intervalo de consulta toma los siguientes valores: [0, 0], [0, 2], [0, 5], [0, 10].

Datos sintéticos

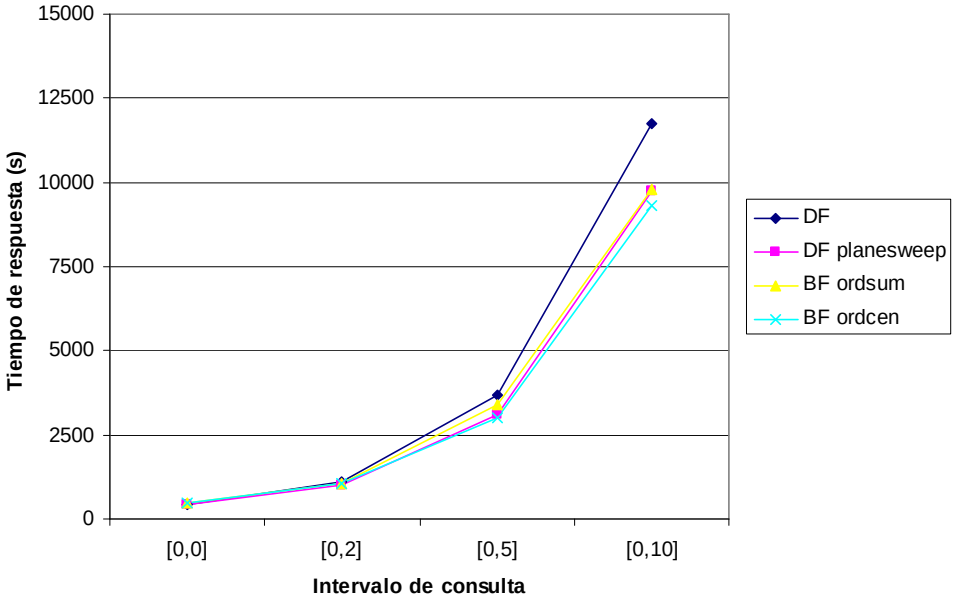


Gráfica 47. Tiempo de respuesta frente a tamaño de intervalo para la consulta de join en datos sintéticos.

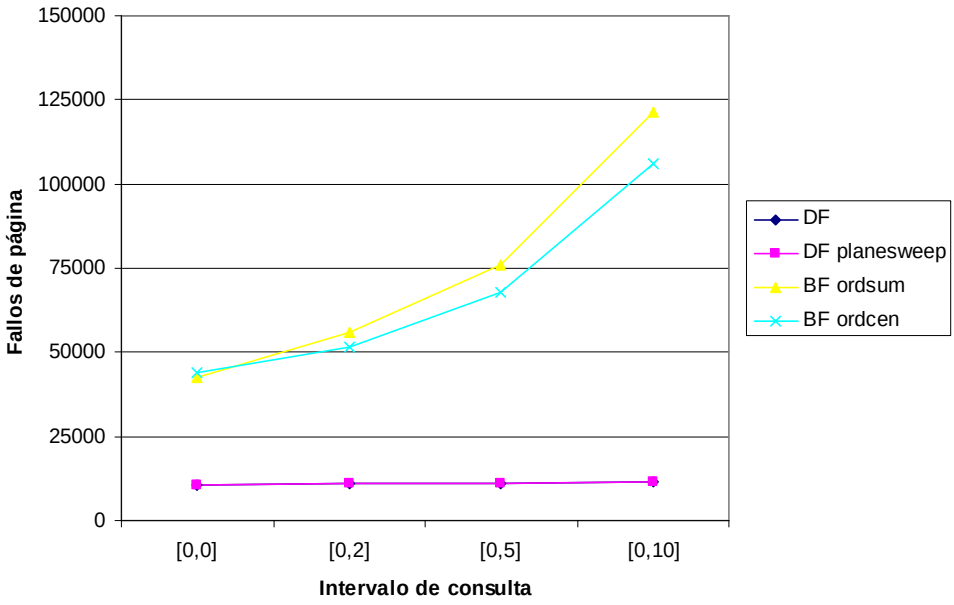


Gráfica 48. Fallos de página frente a tamaño de intervalo para la consulta de join en datos sintéticos.

Datos reales



Gráfica 49. Tiempo de respuesta frente a tamaño de intervalo para la consulta de join en datos reales.



Gráfica 50. Fallos de página frente a tamaño de intervalo para la consulta de join en datos reales.

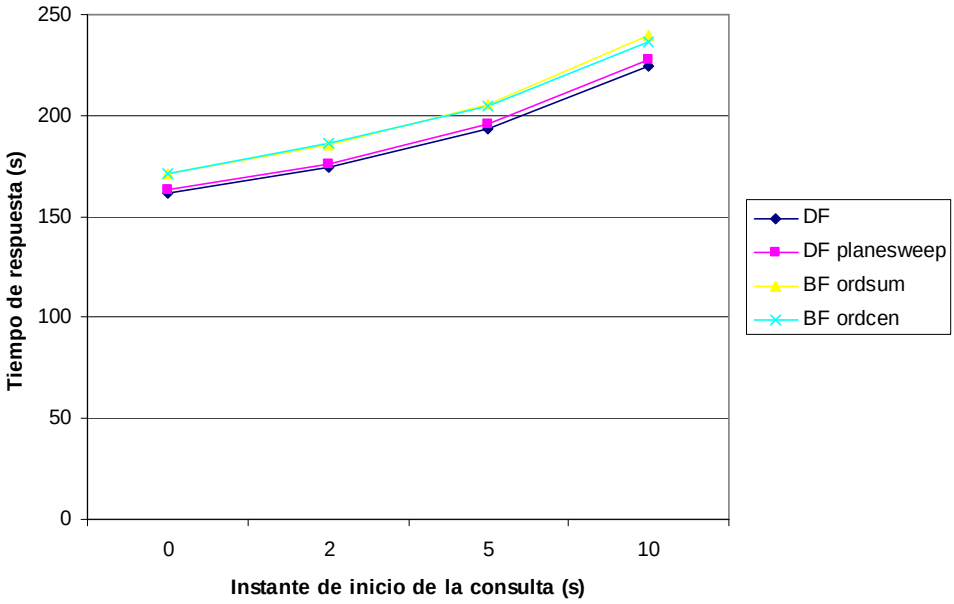
Al considerar intervalos mayores, los MBRs de los nodos aumentan su tamaño y por tanto se producen más solapes a nivel intermedio, aumentando los tiempos de respuesta y número de fallos de página con datos sintéticos. De nuevo todos los algoritmos tienen resultados similares, siendo muy parecidos entre si los *BF* y algo mejor que el resto *DF simple*. Se mantiene la tendencia de *BF ordcen* que casi dobla el nº de fallos de *BF ordsum* y ambos bastante por encima de *DF*.

Con datos reales (recordemos que este conjunto tiene 131461 objetos) se invierte la tendencia anterior para el tiempo de respuesta y *BF ordcen* se muestra como el mejor, muy parejo con *BF ordsum* y *DF planesweep*. *DF* empeora respecto de los demás conforme aumenta el tamaño del intervalo. El mal resultado de *DF* en tiempo de respuesta para datos reales y, en general, el cambio de tendencia respecto del experimento con datos sintéticos, puede deberse a la diferente naturaleza de los datos reales y sintéticos que provoque un tamaño de IJI menor y por tanto un menor tiempo de respuesta en los dos algoritmos *BF*. La tendencia de *DF planesweep* a mejorar a *DF* en tiempo de respuesta para cardinalidades mayores ya se refleja en el experimento 1 de esta sección.

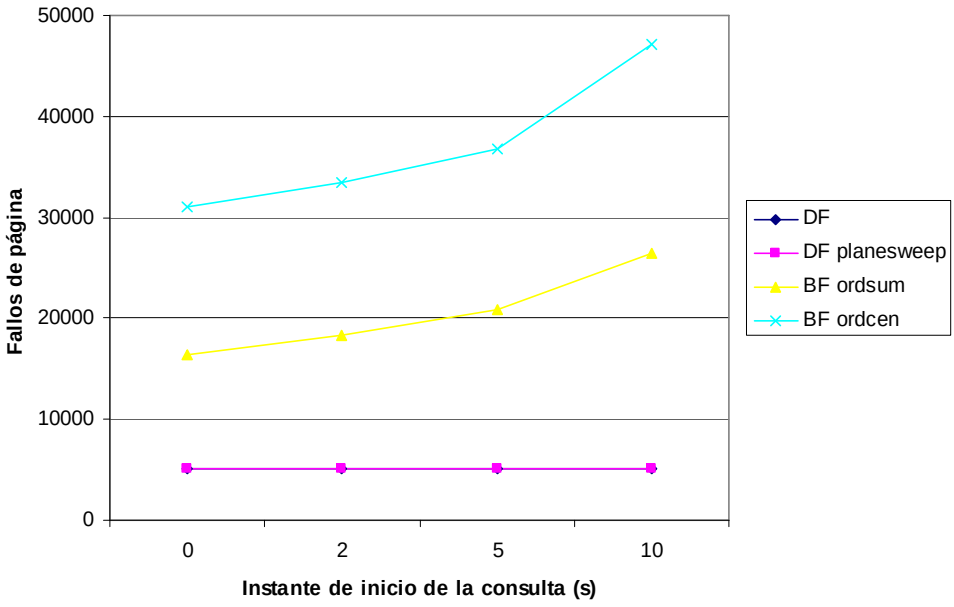
En cuanto a número de fallos de página se confirma la tendencia del experimento anterior y vemos como *BF ordcen* se acerca a los resultados de *BF ordsum* superándolo para todos los intervalos excepto en el [0, 0]. Los dos algoritmos *DF* siguen con pocos fallos y crecen muy ligeramente con el tamaño del intervalo. *DF planesweep* es ligeramente peor que *DF* porque el método usado para obtener los solapes entre nodos da falsos positivos.

Experimento 4. Variación del instante de consulta. Se han fijado las siguientes variables: tamaño de caché 250, cardinalidad del conjunto de datos 10.000, velocidad de la imagen (1, 1). El intervalo de consulta toma los siguientes valores: [0, 0], [2, 2], [5, 5], [10, 10].

Datos sintéticos

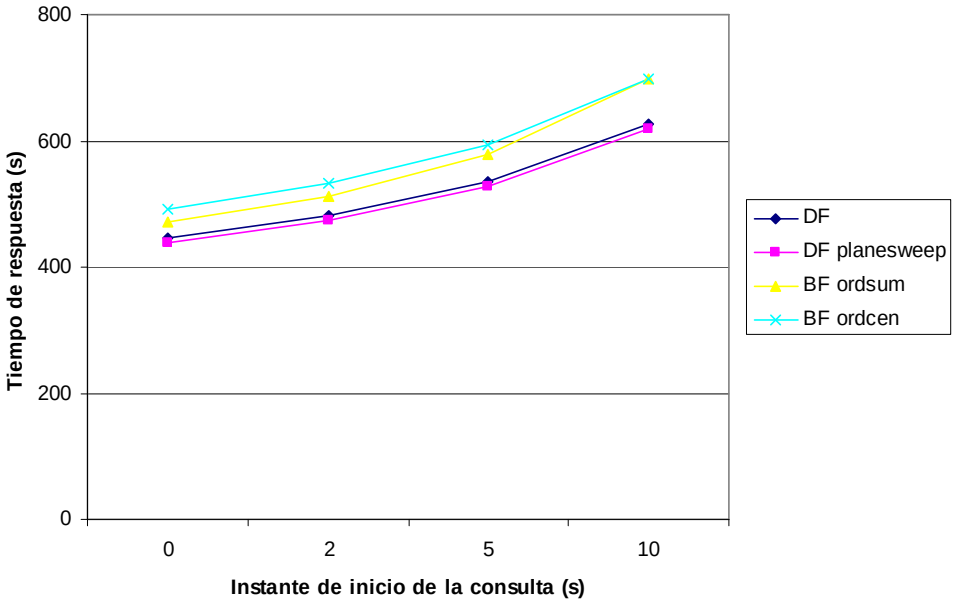


Gráfica 51. Tiempo de respuesta frente a inicio de intervalo para la consulta de join en datos sintéticos.

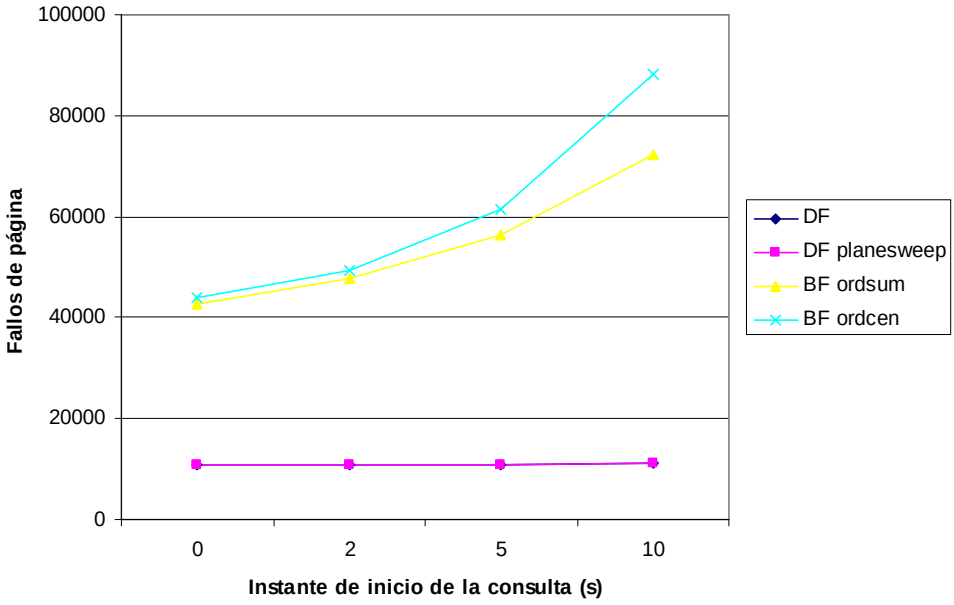


Gráfica 52. Fallos de página frente a inicio de intervalo para la consulta de join en datos sintéticos.

Datos reales



Gráfica 53. Tiempo de respuesta frente a inicio de intervalo para la consulta de join en datos reales.

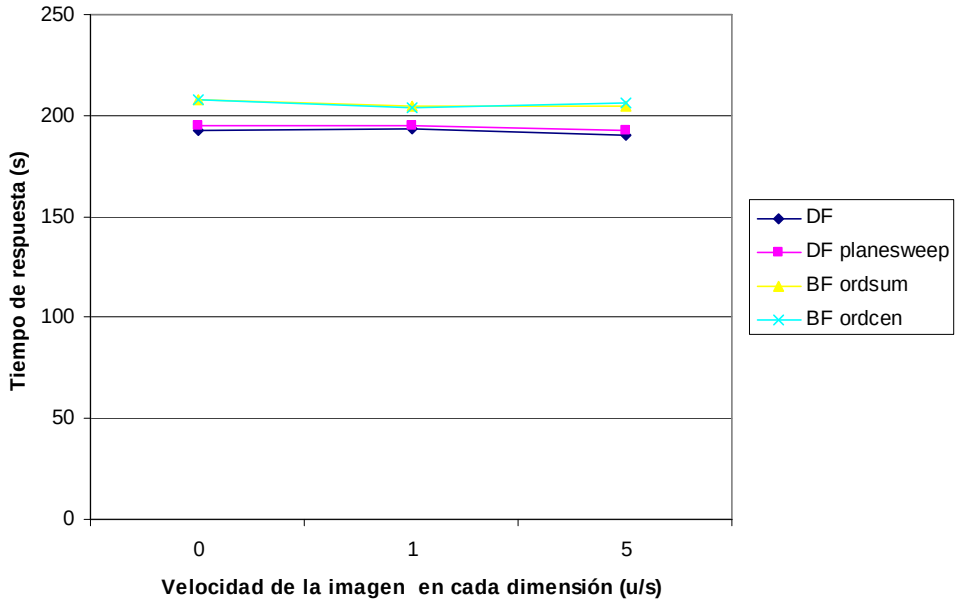


Gráfica 54. Fallos de página frente a inicio de intervalo para la consulta de join en datos reales.

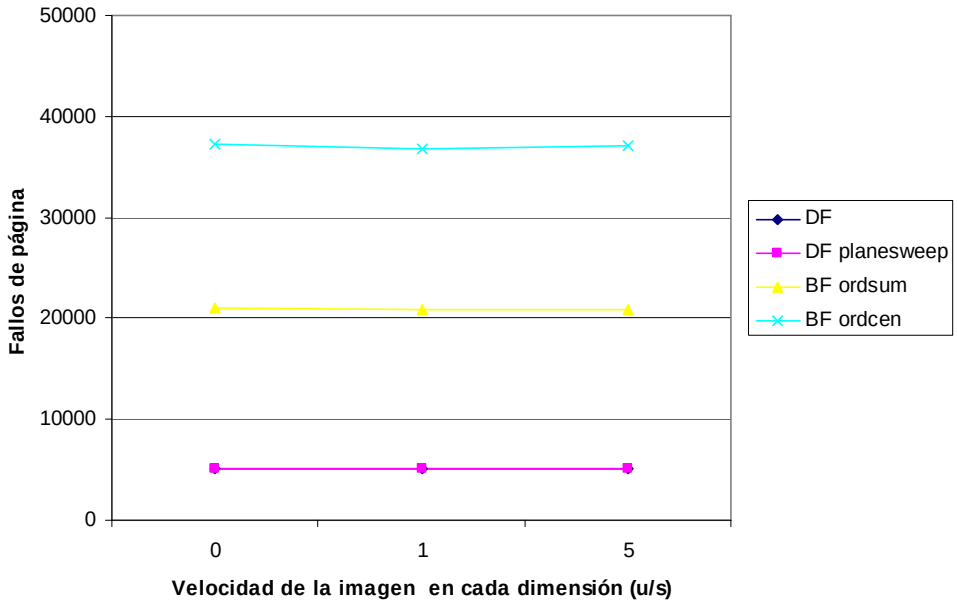
Los resultados son prácticamente iguales a los del experimento anterior, ya que aunque los intervalos tengan tamaño 0 (son instantes) el crecimiento de los MBRs si es el mismo, así que las comprobaciones de solape y accesos a nodos son muy similares.

Experimento 5. Variación de la velocidad de la imagen. Se han fijado las siguientes variables: tamaño de caché 250, cardinalidad del conjunto de datos 10.000, intervalo [5, 5]. La velocidad de la imagen en cada dimensión toma los siguientes valores: (0, 0), (1, 1), (5, 5).

Datos sintéticos

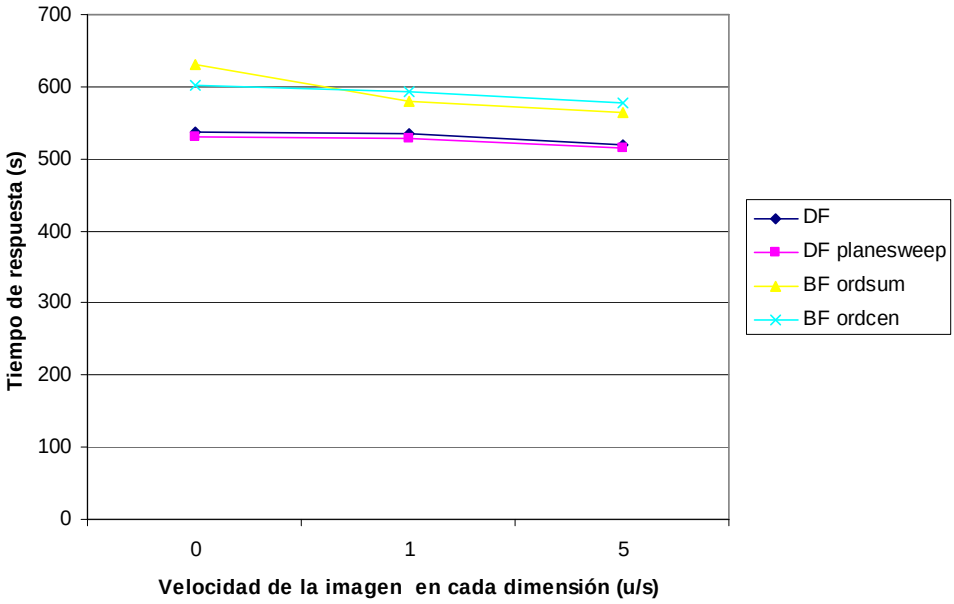


Gráfica 55. Tiempo de respuesta frente a velocidad de la imagen para la consulta de join en datos sintéticos.

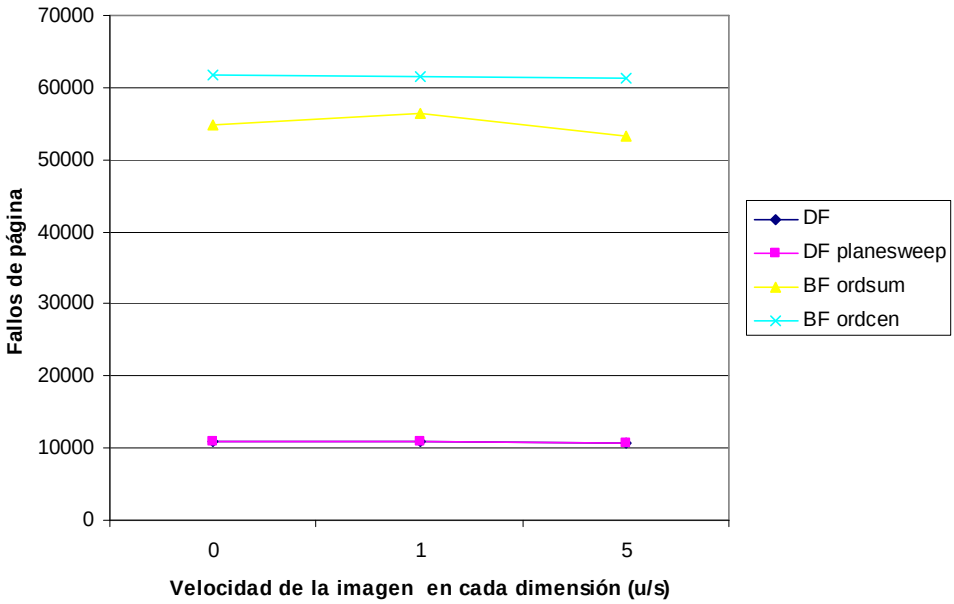


Gráfica 56. Fallos de página frente a velocidad de la imagen para la consulta de join en datos sintéticos.

Datos reales.



Gráfica 57. Tiempo de respuesta frente a velocidad de la imagen para la consulta de join en datos reales.



Gráfica 58. Fallos de página frente a velocidad de la imagen para la consulta de join en datos reales.

La variación de la velocidad de la imagen (recordemos solo traslación) no hace que los MBRs crezcan en tamaño, puesto que al ser las velocidades de los extremos del MBR iguales en cada dimensión esto sólo lo desplaza. Además, el hecho de ser una consulta puntual hace que el aumento de velocidad simplemente "desplace" las posiciones en que se darán los resultados. Las gráficas muestran que los tiempos de ejecución y fallos de página permanecen estables ante el cambio de velocidad y que los tiempos de ejecución y fallos se comportan como en los experimentos anteriores,

teniendo los algoritmos *BF* peores resultados en todos los experimentos. Además para datos reales los fallos de página de *BF ordcen* se acercan a los de *BF ordsum* como en los experimentos anteriores.

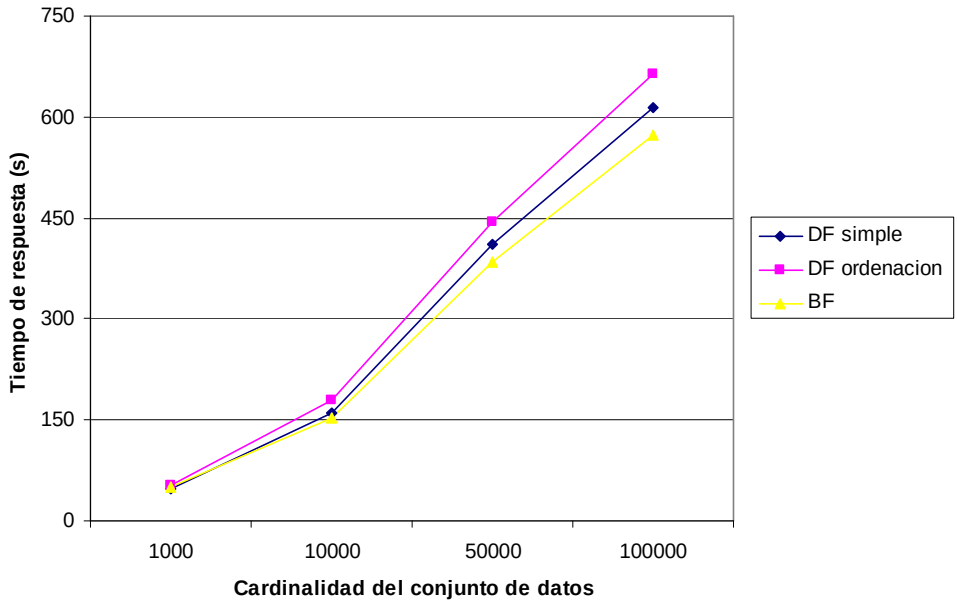
4.4.2 Consulta de join TP

Estos son los resultados de los experimentos realizados con los algoritmos implementados para responder a la consulta join TP: Join TP DF simple (DF simple), Join TP DF ordenación (DF ordenación) y Join TP BF (BF).

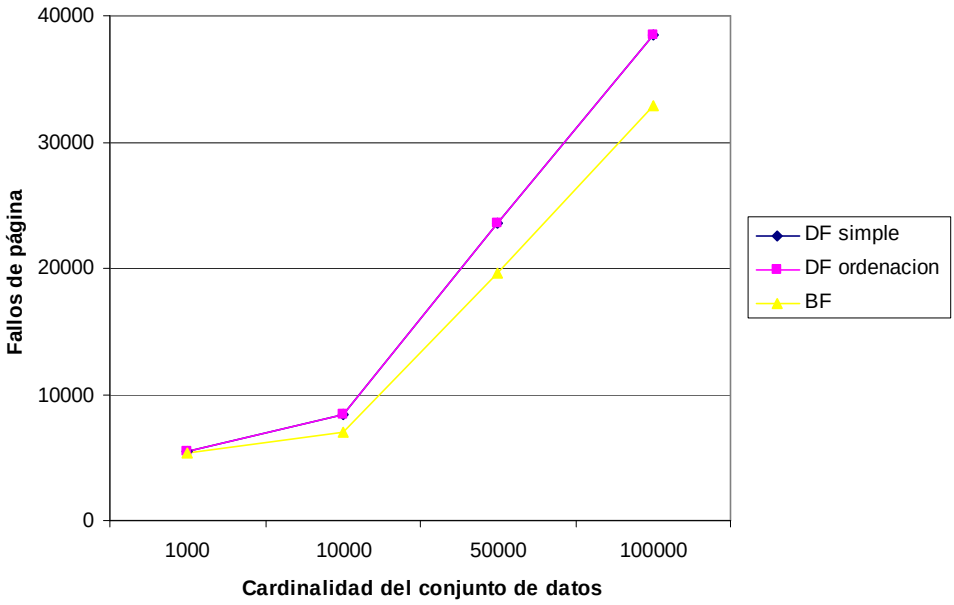
Para la serie de experimentos realizados con esto algoritmos se han tenido en cuenta las siguientes variables: cardinalidad del conjunto de datos, tamaño de caché, tamaño de intervalo de consulta, instante de consulta, y velocidad de la imagen. La cardinalidad del conjunto de datos reales ST es de 131461 objetos.

Experimento 1. Variación de la cardinalidad. Se han fijado las siguientes variables: tamaño de caché 25, intervalo [0, 5], velocidad de la imagen (1, 1). La cardinalidad del conjunto de datos toma los siguientes valores: 1.000, 10.000, 50.000, 100.000.

Datos sintéticos



Gráfica 59. Tiempo de respuesta frente a cardinalidad para la consulta join TP en datos sintéticos.



Gráfica 60. Fallos de página frente a cardinalidad para la consulta de join TP en datos sintéticos.

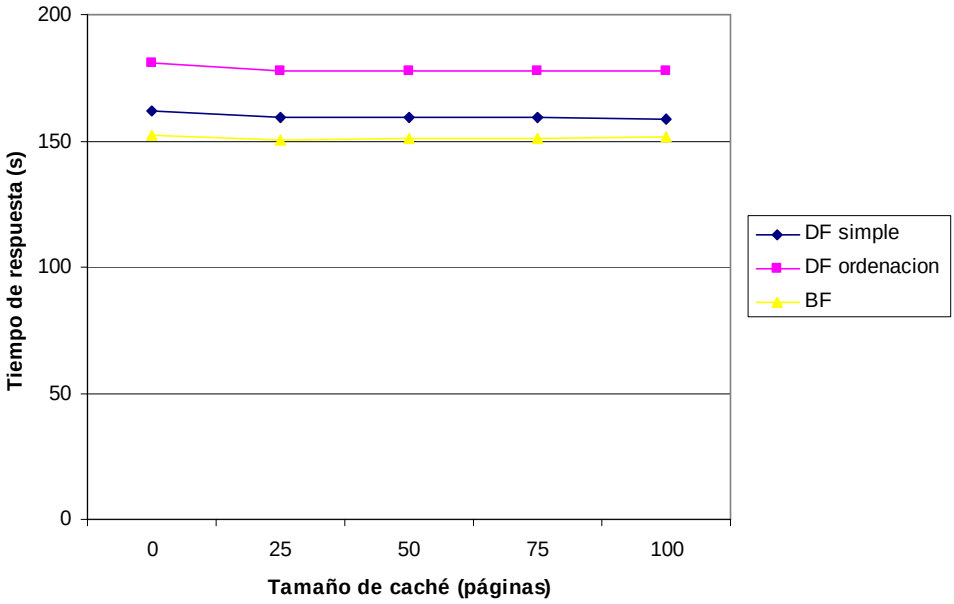
Datos reales

Al disponer de un único conjunto de datos reales no se ha realizado este experimento.

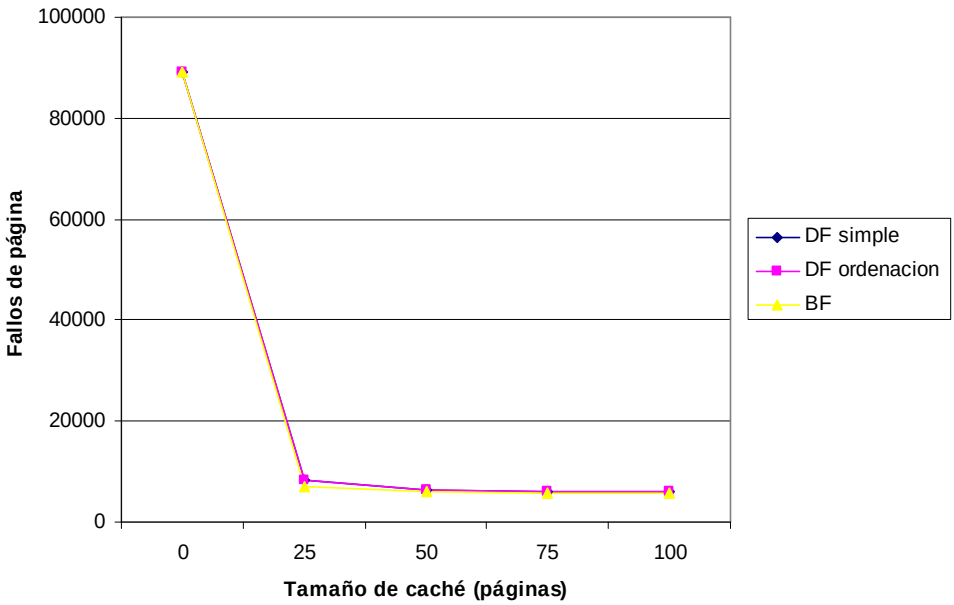
Vemos como *BF*, que en teoría es óptimo respecto al número de accesos porque sólo accede a los nodos necesarios para obtener el resultado, cumple las expectativas y mejora a las dos versiones *DF* en tiempo de ejecución. Además, *DF ordenación* empeora a *DF simple* ya que con el mismo número de accesos requiere tiempo extra para ordenar las entradas, cosa que no se ve compensada por el menor número de comparaciones que debería realizar una vez ordenadas las parejas de entradas por su tiempo de influencia. También, el hecho de que no sólo busque el par con menor tiempo de influencia (componente TP), si no que también busque los pares que se solapan (componente convencional) influya en la cercanía de todos los algoritmos, pues esta búsqueda puede tener mayor peso en el resultado final. Respecto al número de fallos de página *BF* también se comporta mejor que los dos *DF* (que tiene el mismo número de fallos) por las mismas razones que antes. En ambos casos conforme aumenta la cardinalidad del conjunto aumentan las diferencias.

Experimento 2. Variación del tamaño de caché. Se han fijado las siguientes variables: cardinalidad del conjunto de datos 10.000, intervalo [0, 5], velocidad de la imagen (1, 1). El tamaño de caché toma los siguientes valores: 0, 25, 50, 75, 100.

Datos sintéticos

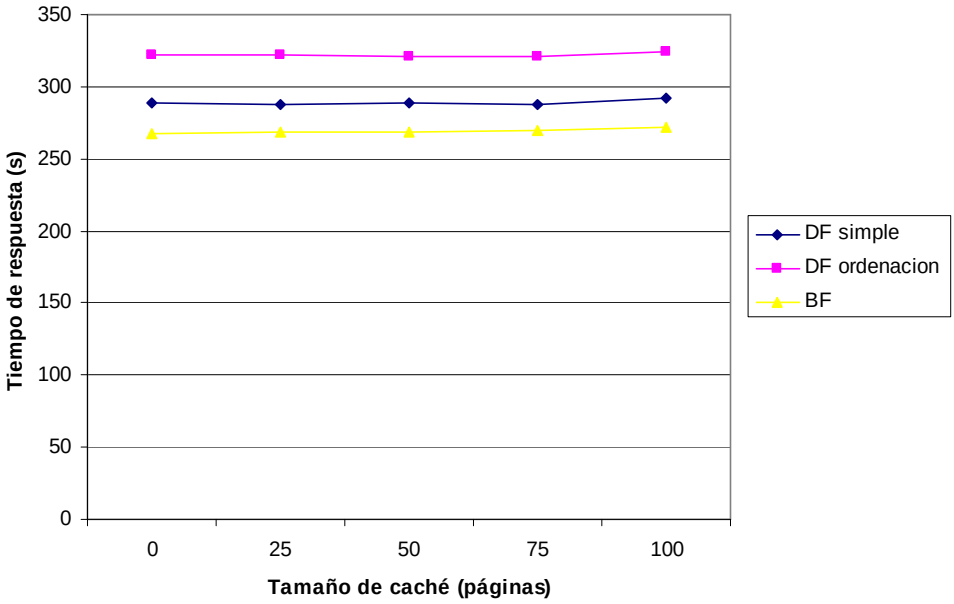


Gráfica 61. Tiempo de respuesta frente a tamaño de caché para la consulta de join TP en datos sintéticos.

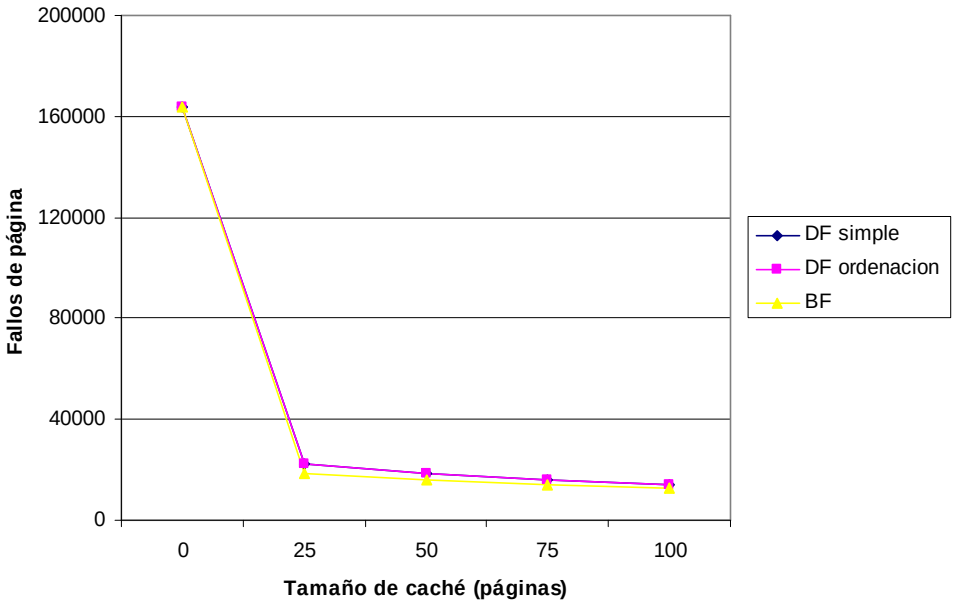


Gráfica 62. Fallos de página frente a tamaño de caché para la consulta de join TP en datos sintéticos.

Datos reales



Gráfica 63. Tiempo de respuesta frente a tamaño de caché para la consulta de join TP en datos reales.



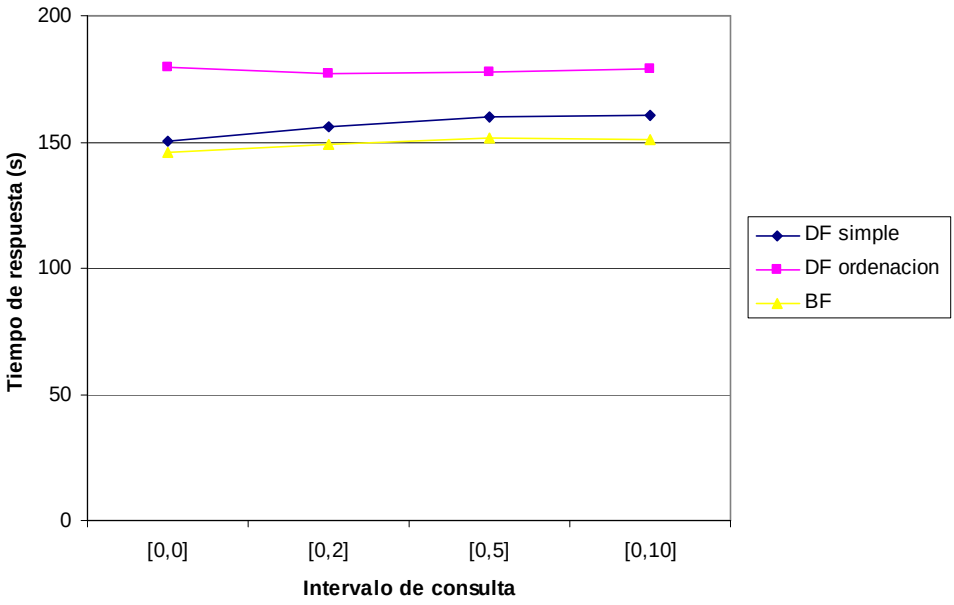
Gráfica 64. Fallos de página frente a tamaño de caché para la consulta de join TP en datos reales.

En este experimento podemos apreciar claramente el efecto de la caché del sistema operativo en el tiempo de respuesta. Mientras que los fallos de página tienen su máximo (tantos como accesos al árbol necesarios para responder a la consulta, unos 90000) para 0 páginas de caché y decaen rápidamente (10000) para mantenerse casi constantes. Los tiempos de ejecución permanecen estables independientemente del número de fallos y por tanto del tamaño de caché. Al igual que antes, *BF* mejora a ambos *DF* (probablemente por no ser recursivo y tener un montículo que aporta poca sobrecarga

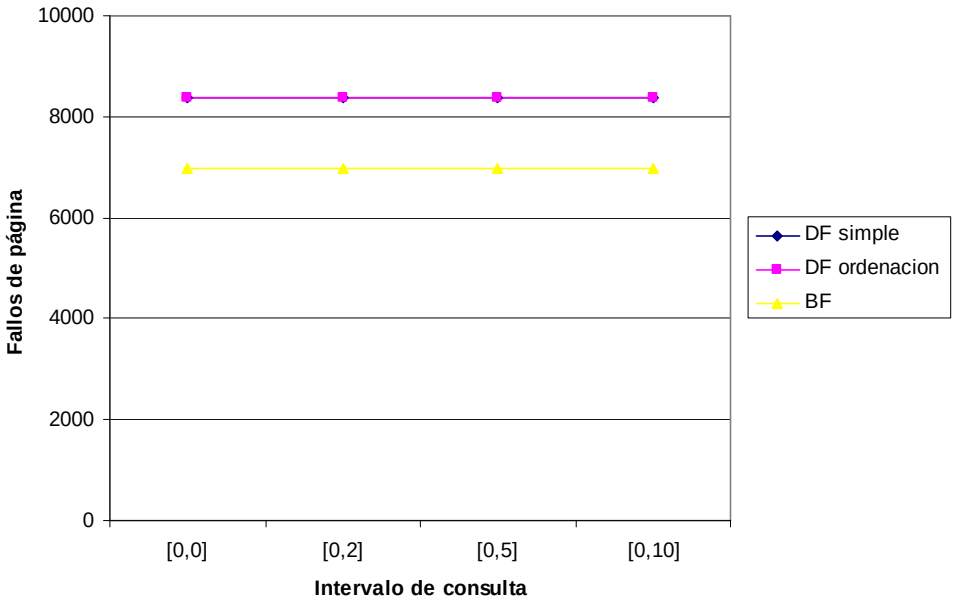
para su gestión) y *DF ordenación* vuelve a no compensar el tiempo empleado en la ordenación de los pares respecto al resultado de *DF simple* con el ahorro en exploración de ramas. El hecho de que para caché 0 *DF ordenación* y *BF* necesiten los mismos accesos a disco (cuando en teoría *BF* es óptimo y *DF ordenación* no) se puede deber a que como se dijo antes, no solo se busca el par de objetos que hace cambiar el resultado de la consulta, si no que también se busca componente convencional R del resultado.

Experimento 3. Variación del intervalo de consulta. Se han fijado las siguientes variables: tamaño de caché 25, cardinalidad del conjunto de datos 10.000, velocidad de la imagen (1, 1). El intervalo de consulta toma los siguientes valores: [0, 0], [0, 2], [0, 5], [0, 10].

Datos sintéticos

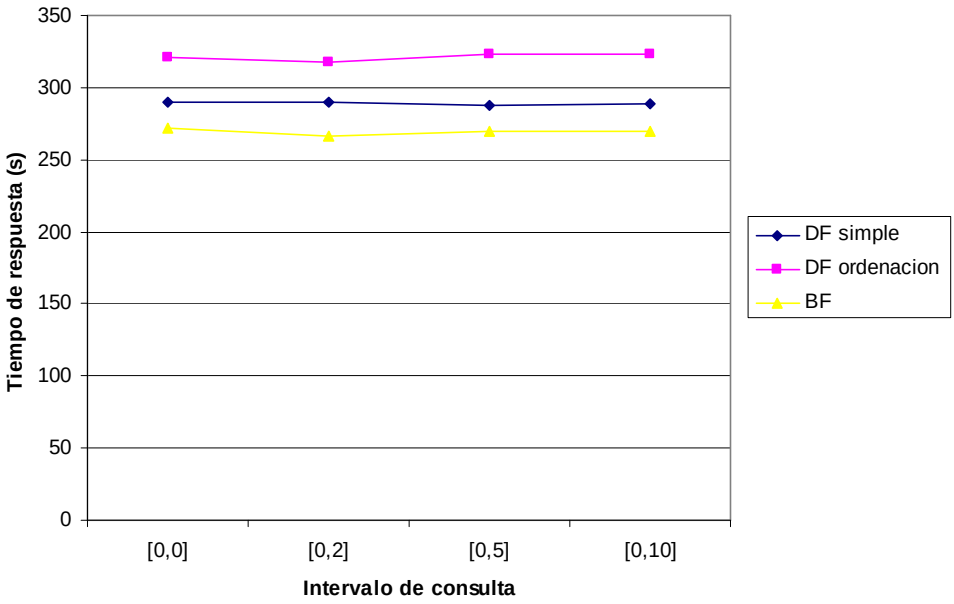


Gráfica 65. Tiempo de respuesta frente a tamaño de intervalo para la consulta de join TP en datos sintéticos.

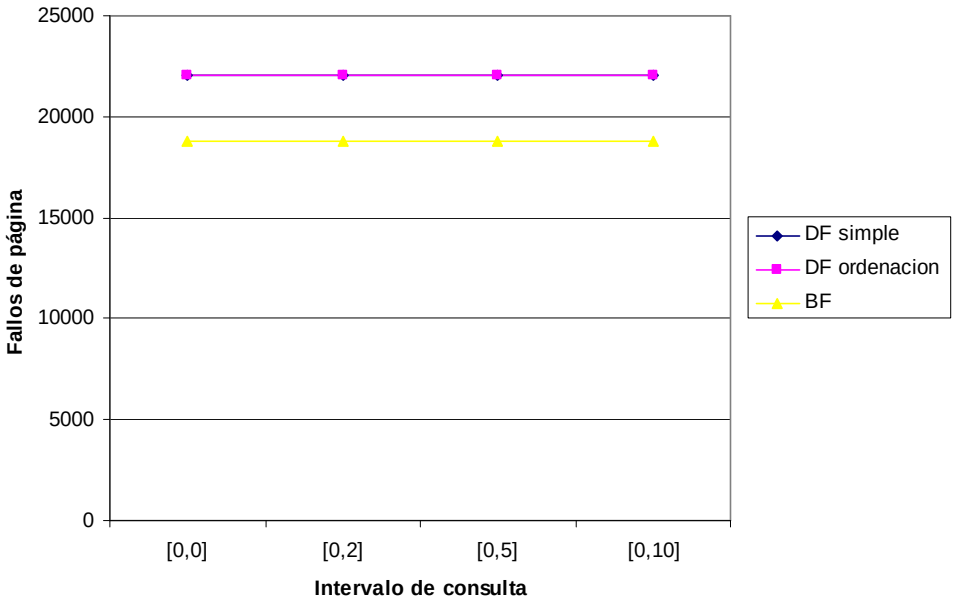


Gráfica 66. Fallos de página frente a tamaño de intervalo para la consulta de join TP en datos sintéticos.

Datos reales



Gráfica 67. Tiempo de respuesta frente a tamaño de intervalo para la consulta de join TP en datos reales.

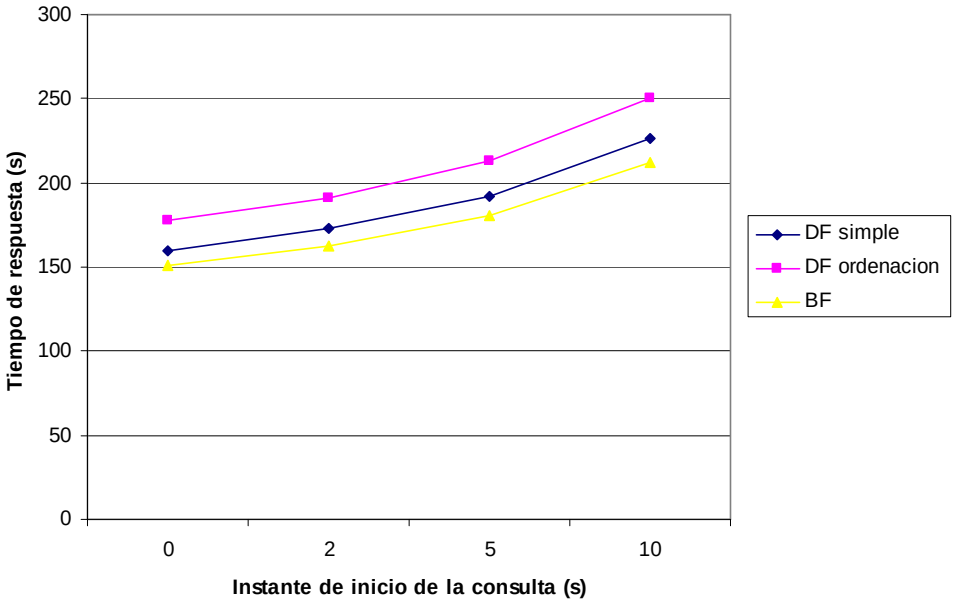


Gráfica 68. Fallos de página frente a tamaño de intervalo para la consulta de join TP en datos reales.

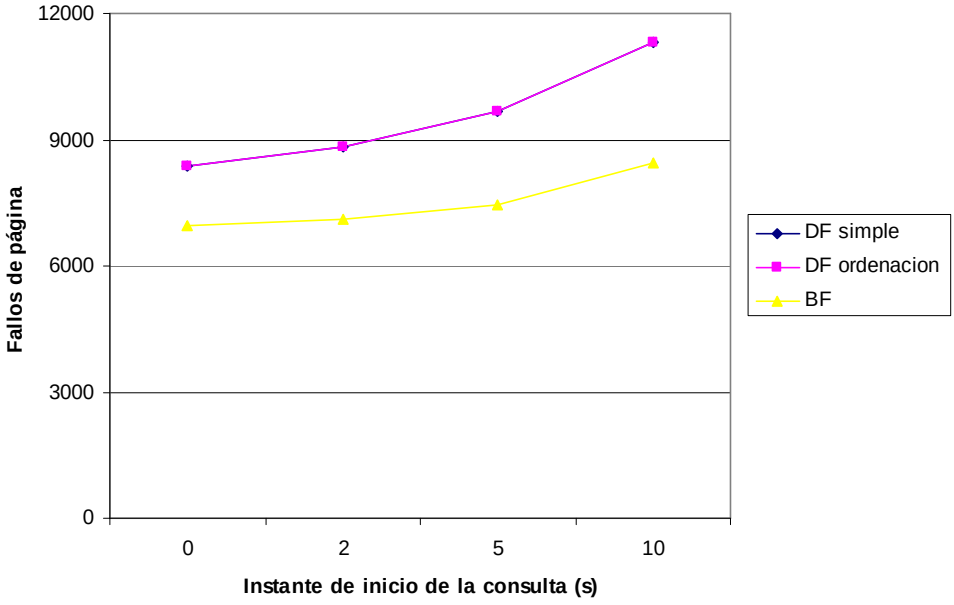
El tiempo de ejecución permanece estable, no se ve afectado, salvo para *DF simple* que para datos sintéticos empeora conforme aumenta el tamaño del intervalo. Esto se puede deber a que procese más pares de los necesarios debido a que no los trata en el orden óptimo. La tendencia general es la misma, *BF* mejor que *DF simple* mejor que *DF ordenación*. El número de fallos de página se mantiene constante para las tres versiones con *BF* siendo el mejor y *DF simple* y *DF ordenación* con idénticos resultados. El rendimiento de los algoritmos no empeora con el tiempo (en general) porque el resultado no depende del tiempo. Los objetos que intersecan y sus tiempos de influencia son los mismos en todos los casos, ya que siempre empezamos en el instante 0.

Experimento 4. Variación del instante de consulta. Se han fijado las siguientes variables: tamaño de caché 25, cardinalidad del conjunto de datos 10.000, velocidad de la imagen (1, 1). El intervalo de consulta toma los siguientes valores: [0, 5], [2, 7], [5, 10], [10, 15].

Datos sintéticos

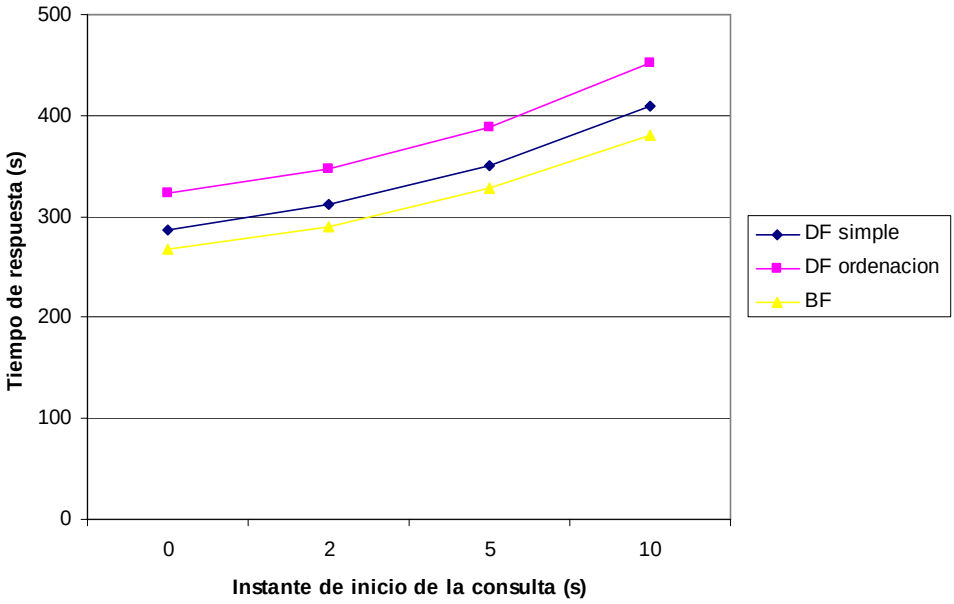


Gráfica 69. Tiempo de respuesta frente a inicio de intervalo para la consulta de join TP en datos sintéticos.

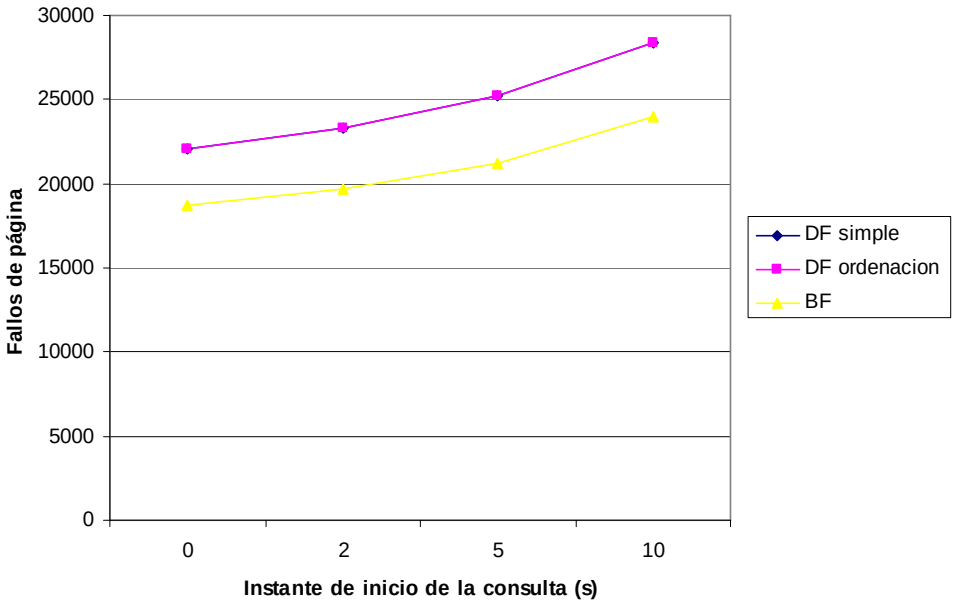


Gráfica 70. Fallos de página frente a inicio de intervalo para la consulta de join TP en datos sintéticos.

Datos reales



Gráfica 71. Tiempo de respuesta frente a inicio de intervalo para la consulta de join TP en datos reales.

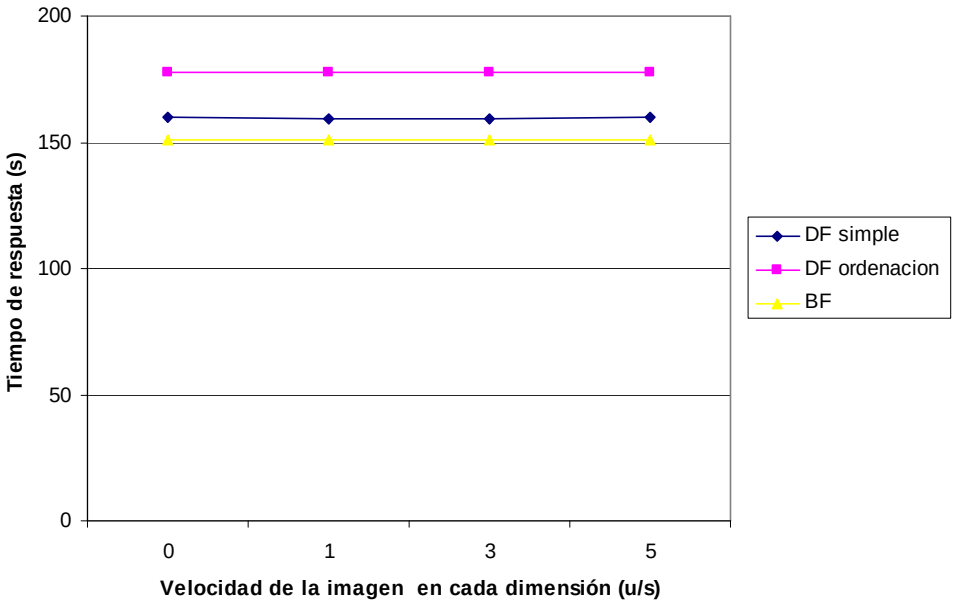


Gráfica 72. Fallos de página frente a inicio de intervalo para la consulta de join TP en datos reales.

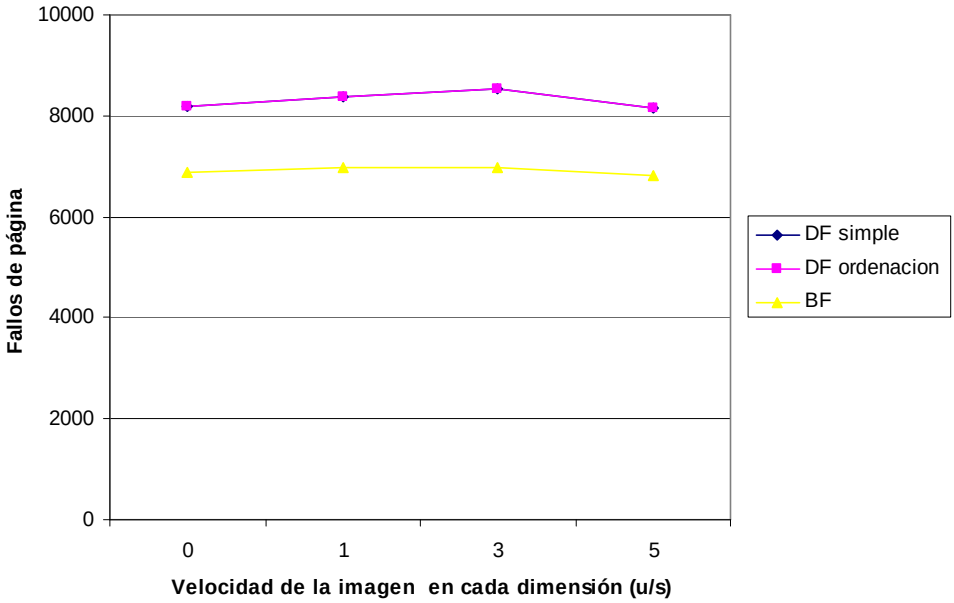
Podemos ver que tanto para datos sintéticos como para datos reales se mantiene el comportamiento de los experimentos anteriores, es decir, *BF mejora* a *DF simple* y éste a *DF ordenación*. Aquí si influye el retraso en el inicio del intervalo. Este provoca que los MBRs sean mayores y como consecuencia aumentan los solapes. Por tanto, habrá que acceder a más nodos para compararlos, aumentando el tiempo de respuesta y el número de fallos de página.

Experimento 5. Variación de la velocidad de la imagen. Se han fijado las siguientes variables: tamaño de caché 25, cardinalidad del conjunto de datos 10.000, intervalo fijo: [0, 5]. La velocidad de la imagen toma los siguientes valores: (0, 0), (1, 1), (3, 3), (5, 5).

Datos sintéticos

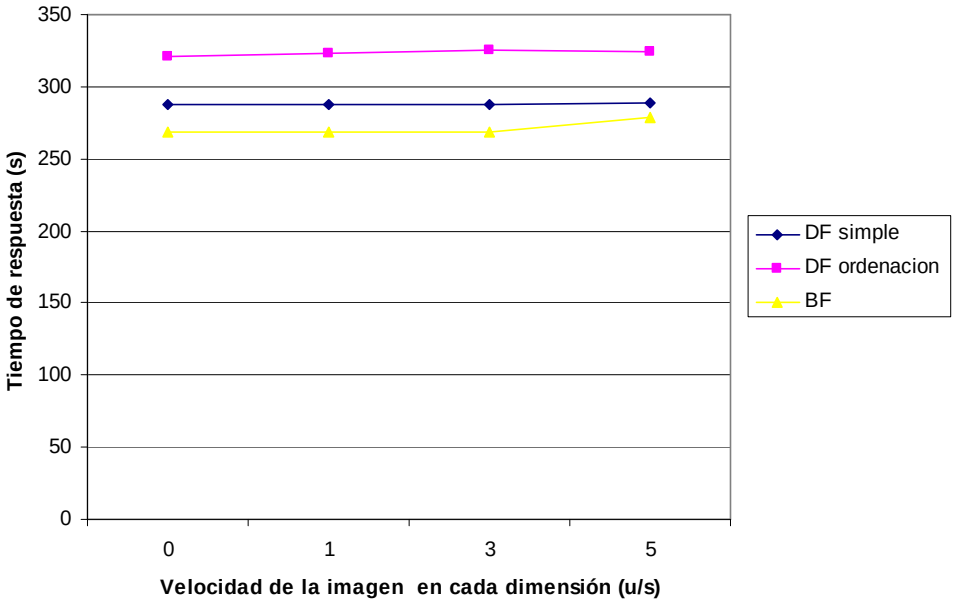


Gráfica 73. Tiempo de respuesta frente a velocidad de la imagen para la consulta de join TP en datos sintéticos.

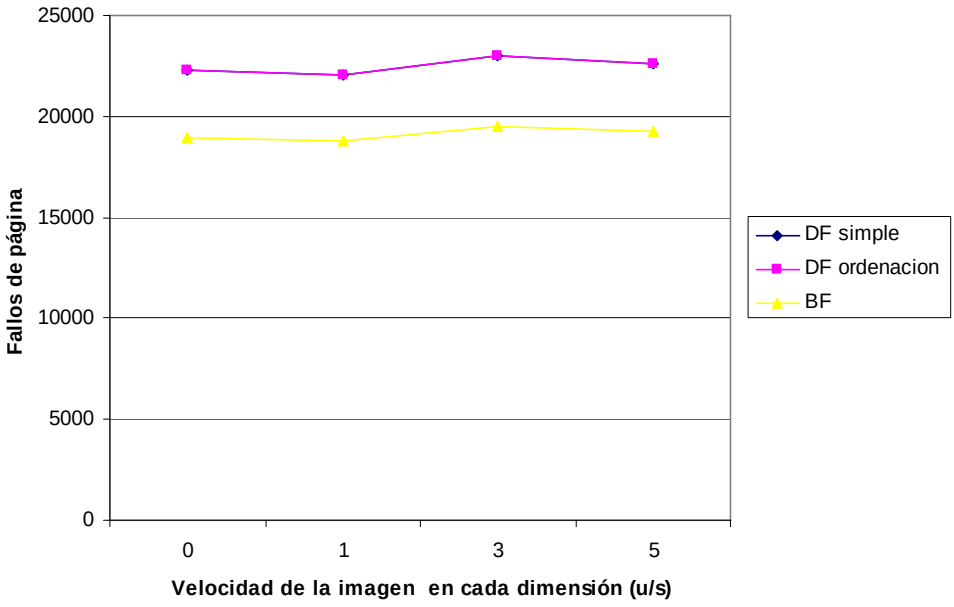


Gráfica 74. Fallos de página frente a velocidad de la imagen para la consulta de join TP en datos sintéticos.

Datos reales



Gráfica 75. Tiempo de respuesta frente a velocidad de la imagen para la consulta de join TP en datos reales.



Gráfica 76. Fallos de página frente a velocidad de la imagen para la consulta de join TP en datos reales.

Este parámetro (velocidad de la imagen) no parece tener efecto ni en el tiempo de respuesta ni en el número de fallos de página. Se mantienen la tendencia *BF* mejor que *DF simple* y éste mejor que *DF ordenación*. En este caso el aumento de la velocidad no hace crecer a los MBRs, lo que hace es reducir los tiempos de influencia, pero no los solapes iniciales (compone convencional de la consulta).

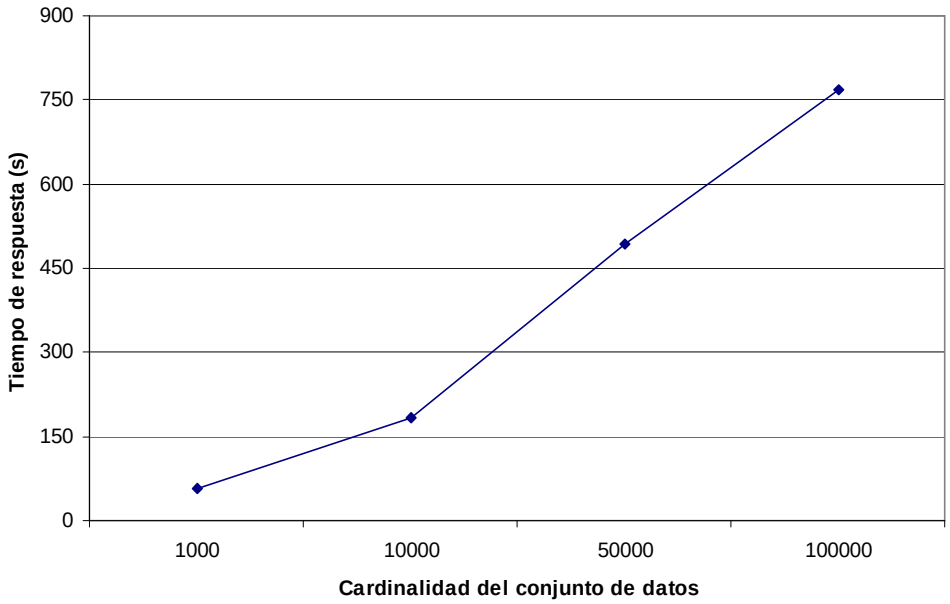
4.4.3 Consulta de join continua

Estos son los resultados de los experimentos realizados con el algoritmo implementado para responder a la consulta de join continua.

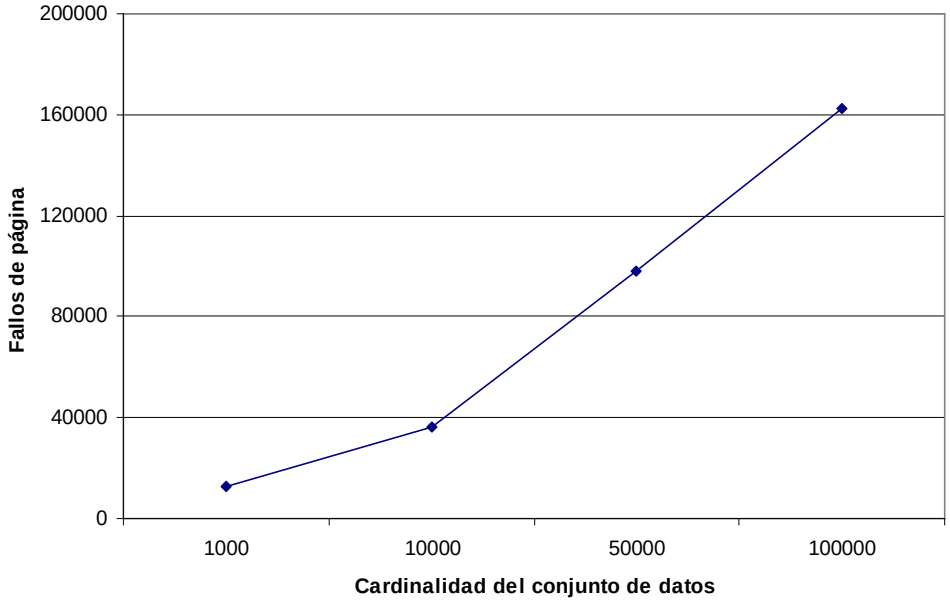
Para la serie de experimentos realizados con este algoritmo se han tenido en cuenta las siguientes variables: cardinalidad del conjunto de datos, tamaño de caché, tamaño de intervalo de consulta, instante de consulta, y velocidad de la imagen. La cardinalidad del conjunto de datos reales ST es de 131461 objetos.

Experimento 1. Variación de la cardinalidad. Se han fijado las siguientes variables: tamaño de caché 25, intervalo [0, 5], velocidad de la imagen (1, 1). La cardinalidad del conjunto de datos toma los siguientes valores: 1.000, 10.000, 50.000, 100.000.

Datos sintéticos



Gráfica 77. Tiempo de respuesta frente a cardinalidad para la consulta join Continua en datos sintéticos.



Gráfica 78. Fallos de página frente a cardinalidad para la consulta de join Continua en datos sintéticos.

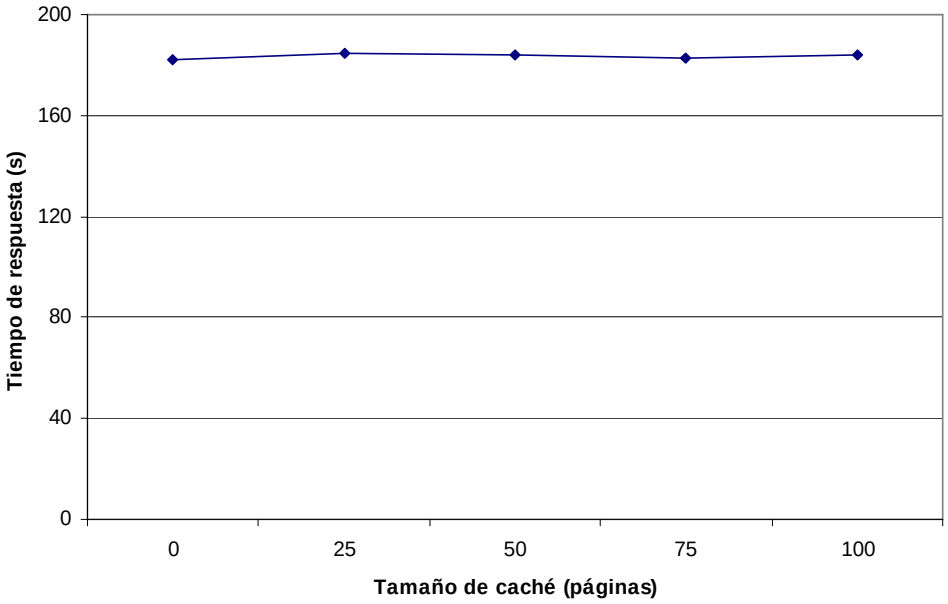
Datos reales

Al disponer de un único conjunto de datos reales no se ha realizado este experimento.

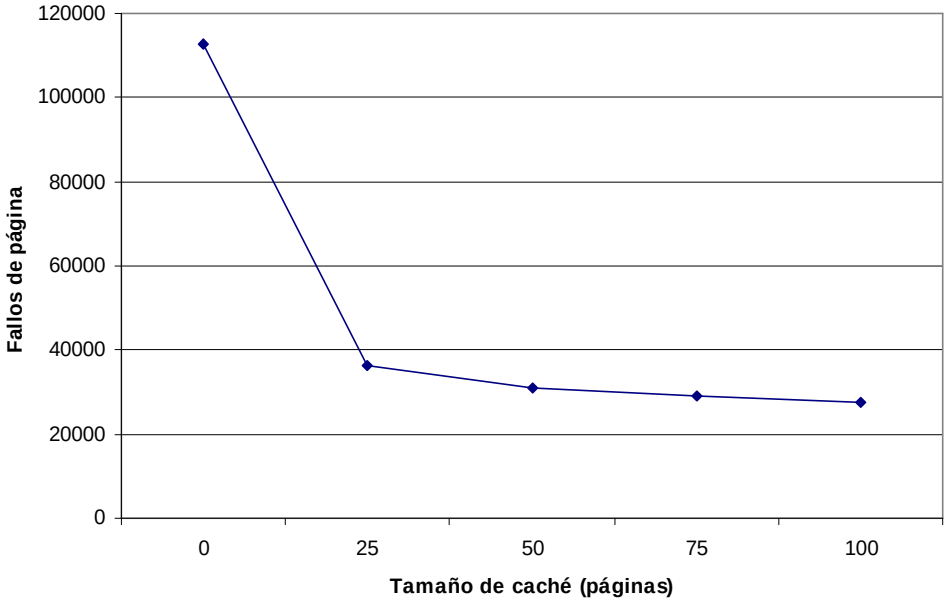
Tanto los fallos de página como el tiempo de respuesta aumentan con la cardinalidad del conjunto de datos como era de esperar, puesto que se deben procesar más objetos. Para la cardinalidad máxima (100000 objetos móviles) el tiempo de respuesta, más de 12 minutos, si bien es elevado, para aplicaciones donde el tiempo de respuesta no sea crítico, no parece excesivo para un uso real del algoritmo, teniendo en cuenta el gran número de objetos procesados.

Experimento 2. Variación del tamaño de caché. Se han fijado las siguientes variables: cardinalidad del conjunto de datos 10.000, intervalo [0, 5], velocidad de la imagen (1, 1). El tamaño de caché toma los siguientes valores: 0, 25, 50, 75, 100.

Datos sintéticos

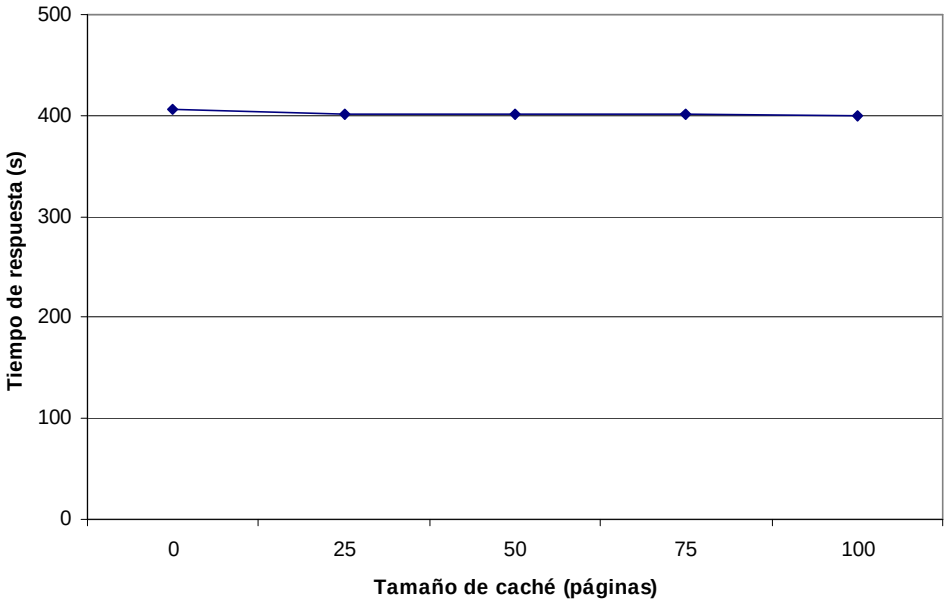


Gráfica 79. Tiempo de respuesta frente a tamaño de caché para la consulta de join Continua en datos sintéticos.

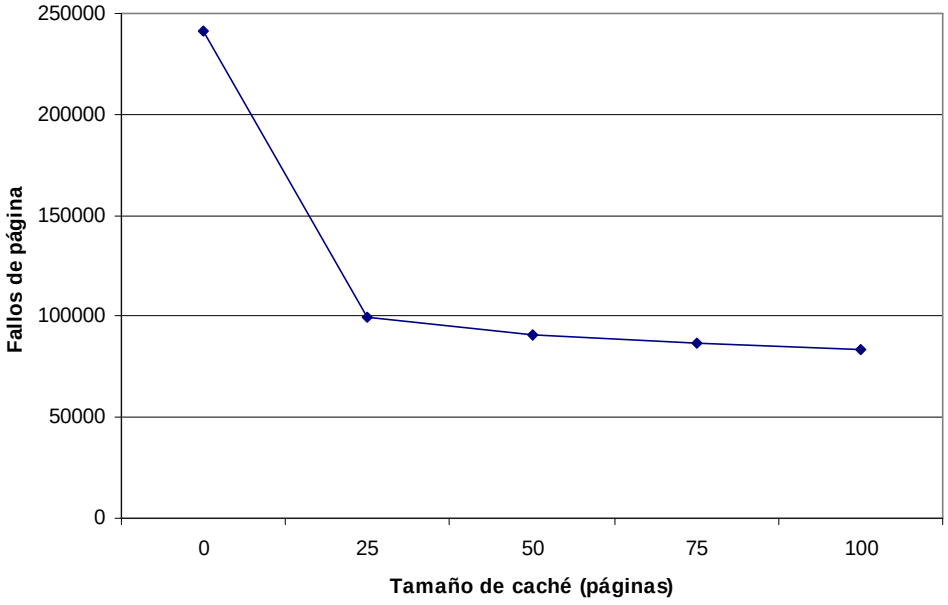


Gráfica 80. Fallos de página frente a tamaño de caché para la consulta de join Continua en datos sintéticos.

Datos reales



Gráfica 81. Tiempo de respuesta frente a tamaño de caché para la consulta de join Continua en datos reales.



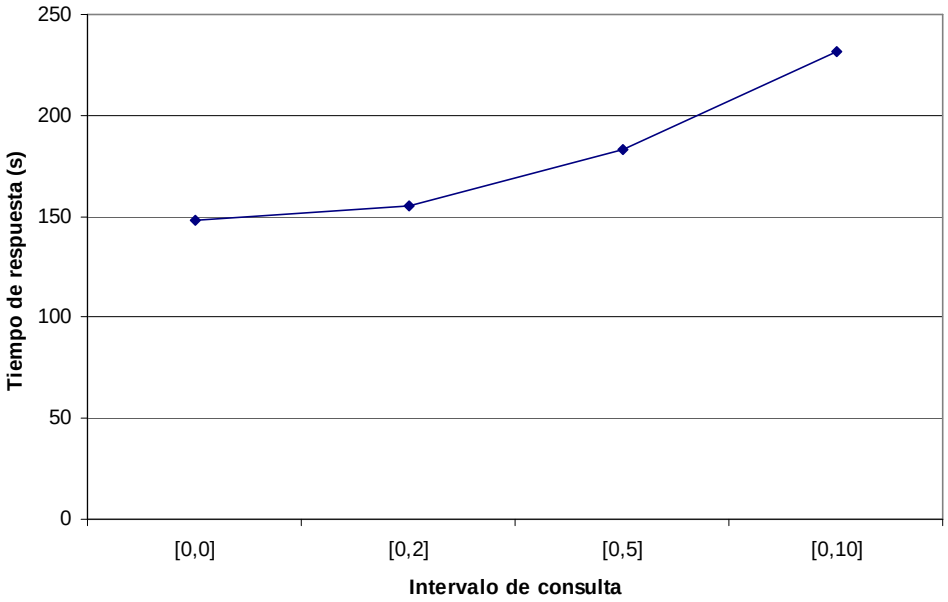
Gráfica 82. Fallos de página frente a tamaño de caché para la consulta de join Continua en datos reales.

Para caché 0, como en el join TP (al fin y al cabo este algoritmo es muy parecido, sólo sigue obteniendo incrementalmente los resultados del montículo hasta alcanzar la condición de terminación) tiene un número de fallos de página que cae rápidamente y entonces mantiene un descenso suave. Esto parece indicar que el algoritmo tiene unas necesidades de caché definidas y a partir de entonces aumentar su tamaño no nos lleva a evitar más fallos de página. Podemos ver que con el máximo de caché utilizado en el

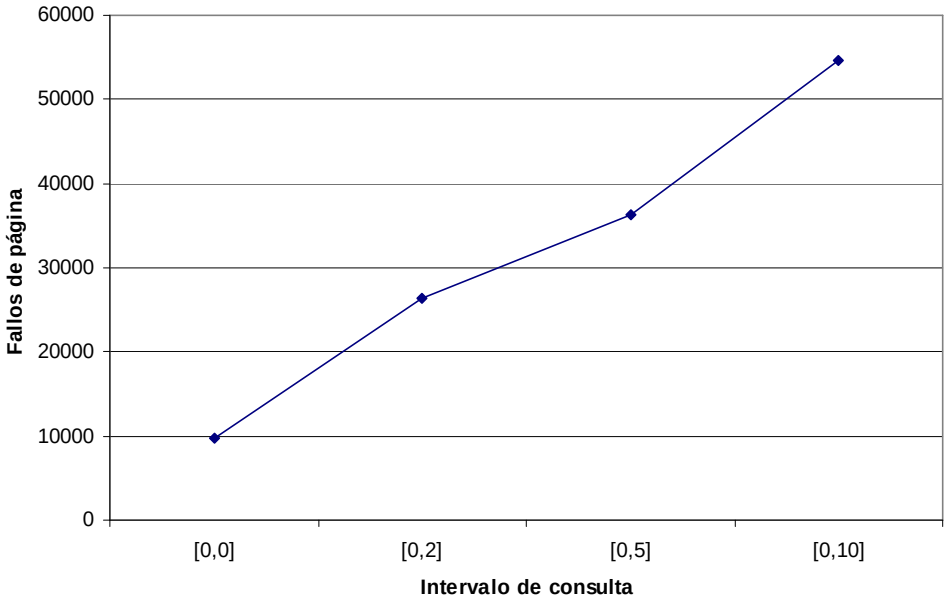
experimento aún no se ha alcanzado ese máximo. El tiempo de respuesta permanece estable posiblemente debido a la caché del sistema operativo.

Experimento 3. Variación del intervalo de consulta. Se han fijado las siguientes variables: tamaño de caché 25, velocidad de la imagen (1, 1), cardinalidad del conjunto de datos 10.000. El intervalo de consulta toma los siguientes valores: [0, 0], [0, 2], [0, 5], [0, 10].

Datos sintéticos

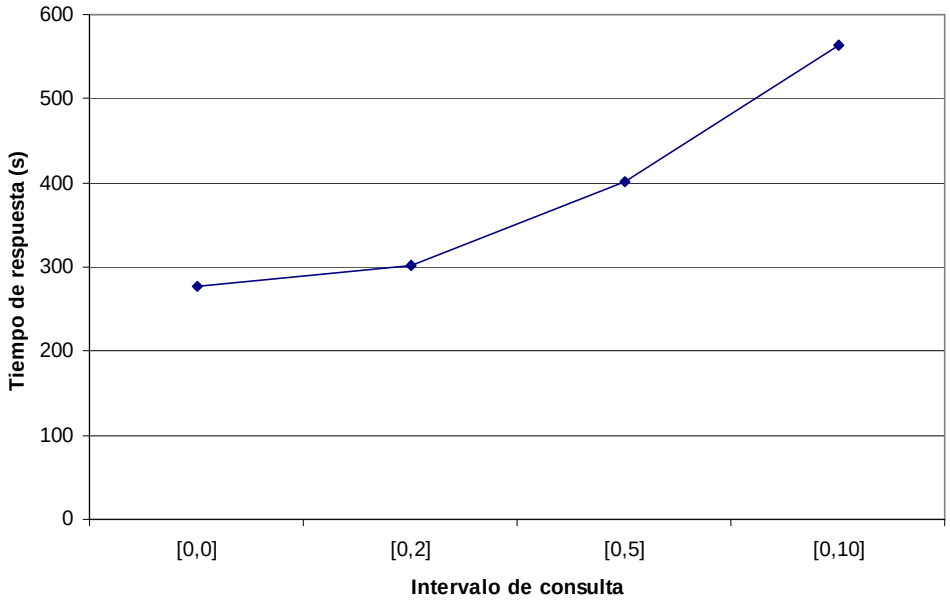


Gráfica 83. Tiempo de respuesta frente a tamaño de intervalo para la consulta de join Continua en datos sintéticos.

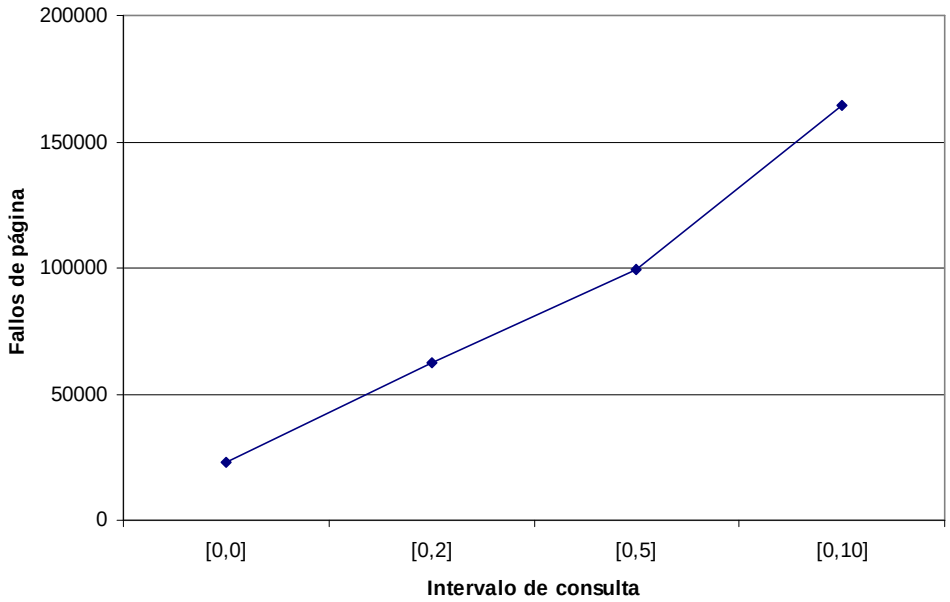


Gráfica 84. Fallos de página frente a tamaño de intervalo para la consulta de join Continua en datos sintéticos.

Datos reales



Gráfica 85. Tiempo de respuesta frente a tamaño de intervalo para la consulta de join Continua en datos reales.

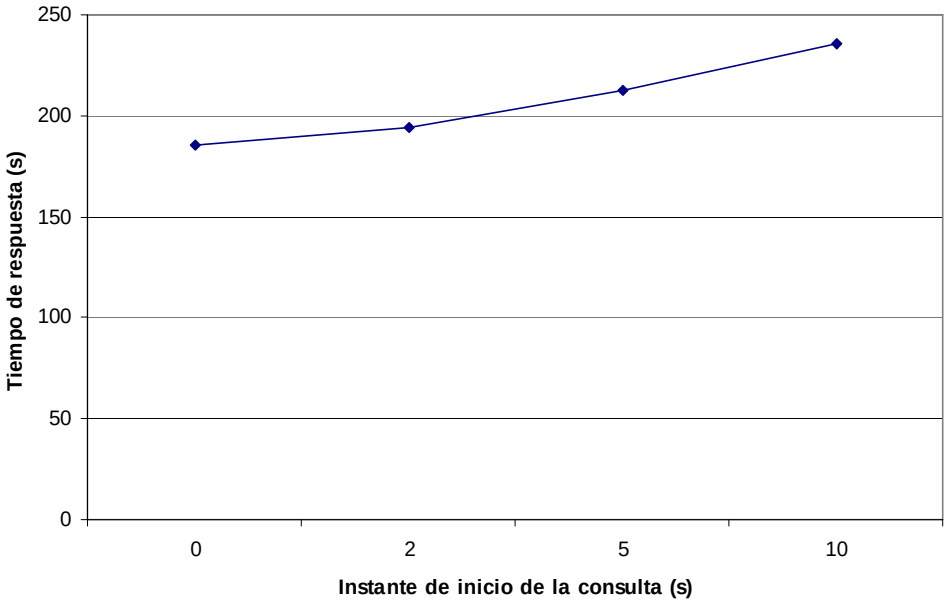


Gráfica 86. Fallos de página frente a tamaño de intervalo para la consulta de join Continua en datos reales.

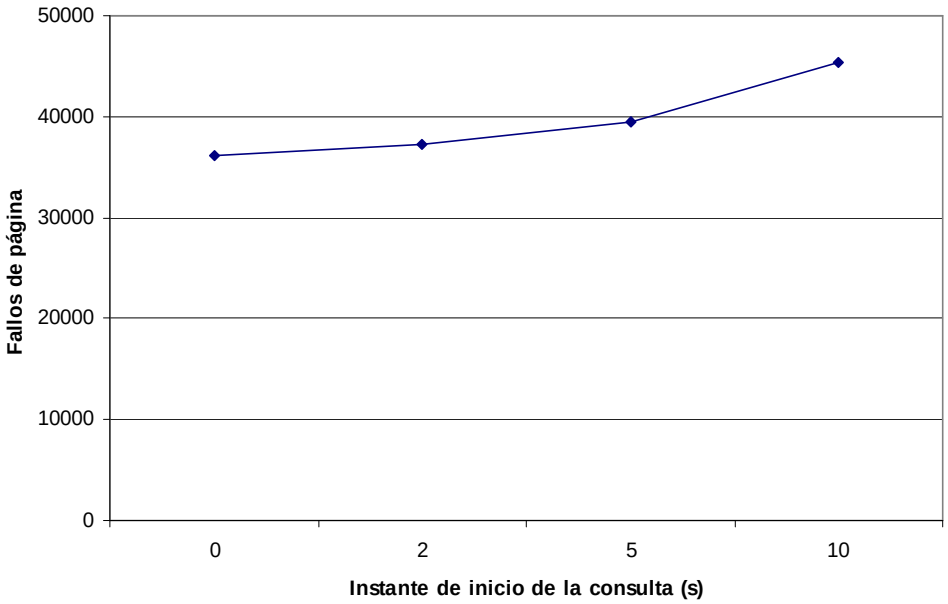
Para este caso, el aumento del tamaño del intervalo hace que aumenten los solapes y además que el número de cambios a devolver en el resultado aumente también. Este algoritmo devuelve los resultados de forma incremental, con un sobrecoste mínimo. Por tanto el tiempo de respuesta aumenta de manera suave. Esto afecta de forma muy parecida al número de fallos de página, que también crece pero más despacio.

Experimento 4. Variación del instante de consulta. Se han fijado las siguientes variables: tamaño de caché 25, cardinalidad del conjunto de datos 10.000, velocidad de la imagen (1, 1). El intervalo de consulta toma los siguientes valores: [0, 0], [2, 7], [5, 10], [10, 15].

Datos sintéticos.

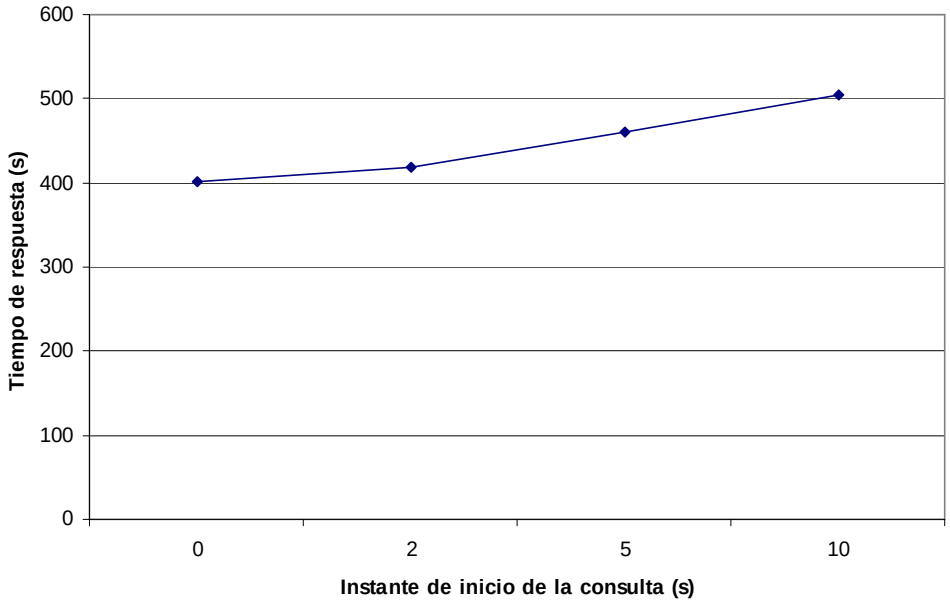


Gráfica 87. Tiempo de respuesta frente a inicio de intervalo para la consulta de join Continua en datos sintéticos.

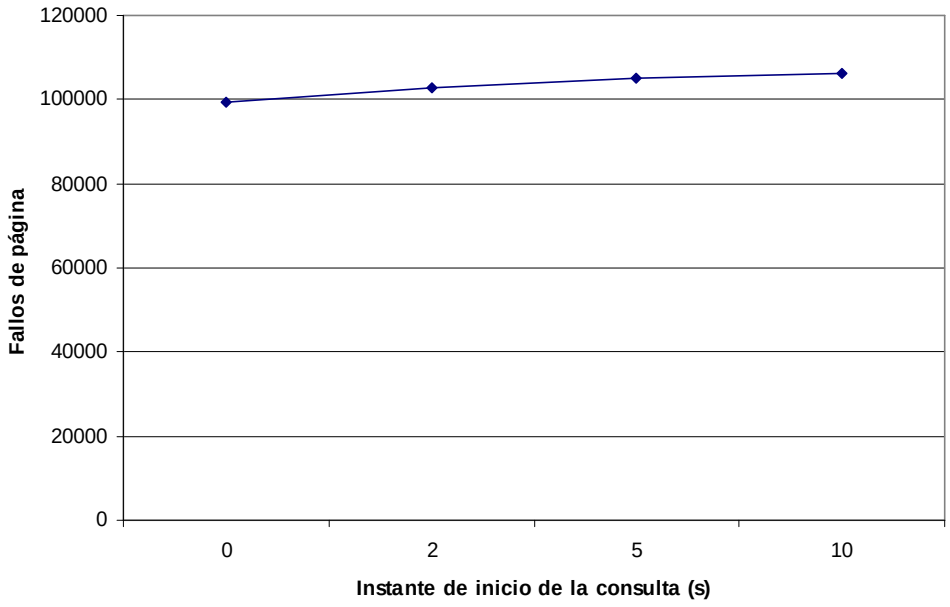


Gráfica 88. Fallos de página frente a inicio de intervalo para la consulta de join Continua en datos sintéticos.

Datos reales



Gráfica 89. Tiempo de respuesta frente a inicio de intervalo para la consulta de join Continua en datos reales.

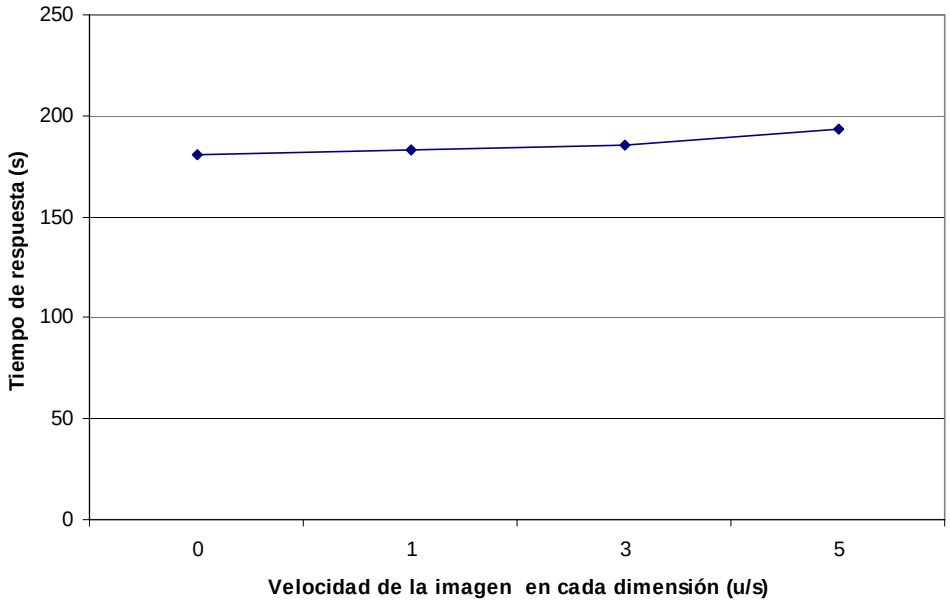


Gráfica 90. Fallos de página frente a inicio de intervalo para la consulta de join Continua en datos reales.

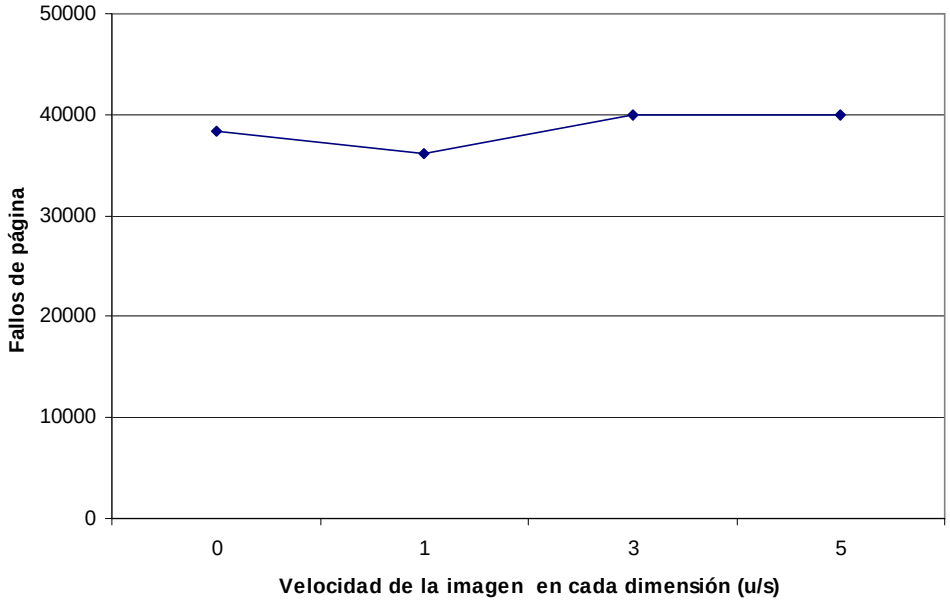
En este caso, a diferencia del experimento anterior, los intervalos son del mismo tamaño, sólo que comienzan en instantes distintos, por eso el número de fallos crece de forma similar al tiempo de respuesta, porque se debe al crecimiento de los MBRs debido al tiempo y no a la búsqueda incremental de objetos con tiempos de influencia posteriores.

Experimento 5. Variación de la velocidad de la imagen. Se han fijado las siguientes variables: tamaño de caché 25, cardinalidad del conjunto de datos 10.000, intervalo [0, 5]. La velocidad de la imagen toma los siguientes valores: (0, 0), (1, 1), (3, 3), (5, 5).

Datos sintéticos

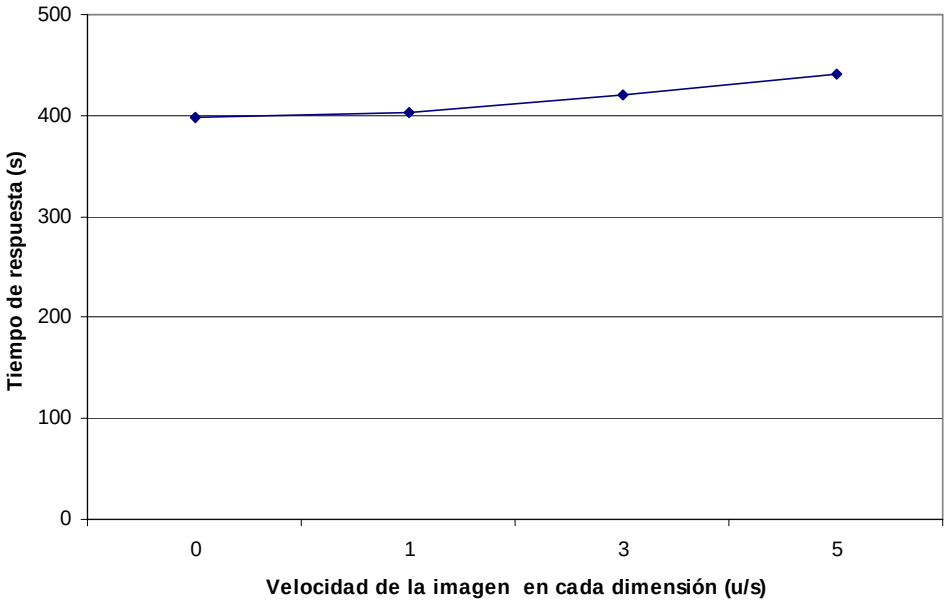


Gráfica 91. Tiempo de respuesta frente a velocidad de la imagen para la consulta de join Continua en datos sintéticos.

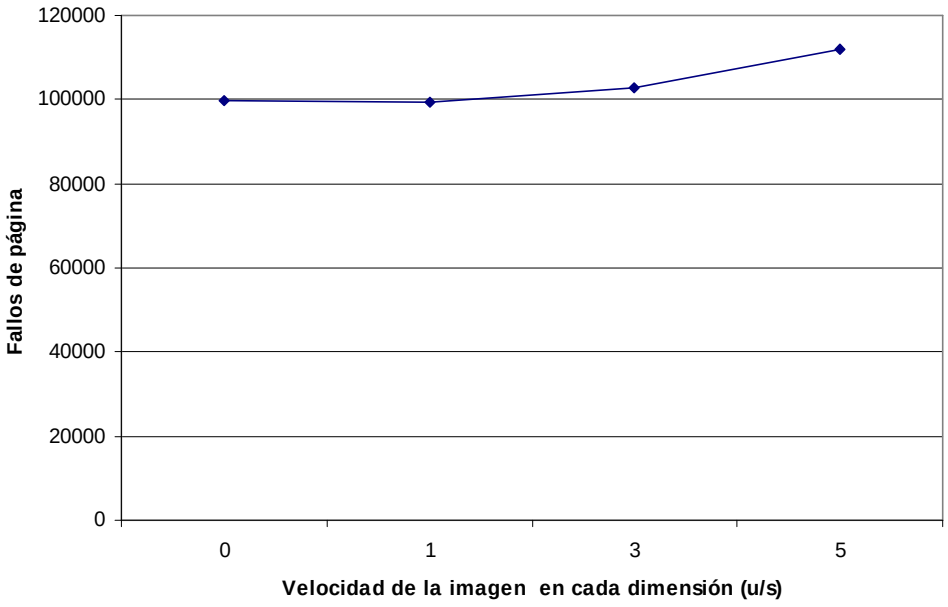


Gráfica 92. Fallos de página frente a velocidad de la imagen para la consulta de join Continua en datos sintéticos.

Datos reales



Gráfica 93. Tiempo de respuesta frente a velocidad de la imagen para la consulta de join Continua en datos reales.



Gráfica 94. Fallos de página frente a velocidad de la imagen para la consulta de join Continua en datos reales.

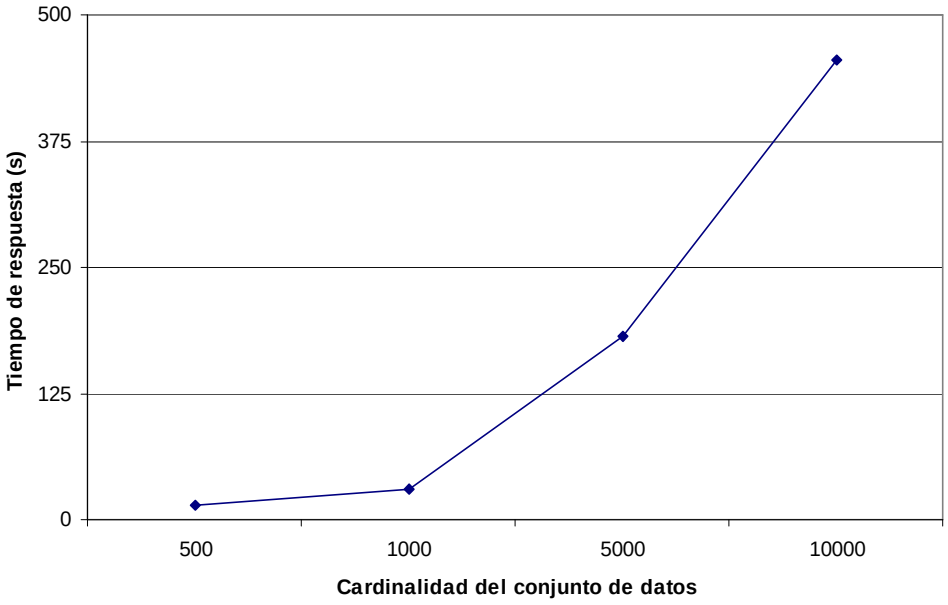
El tiempo de respuesta y el número de fallos de página aumentan con la velocidad debido posiblemente a que ésta influye acortando los tiempos de influencia. Por tanto, a mayor velocidad de la imagen más objetos que devolver en el resultado final y mayores tiempos de respuesta y fallos de página.

4.5 Consulta de semijoin espacial de los k vecinos más próximos continua

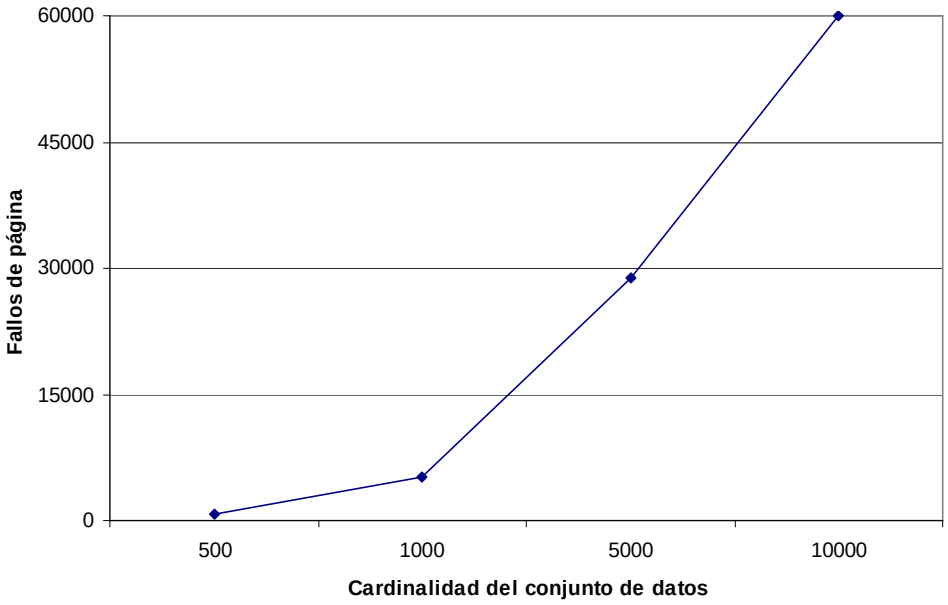
Estos son los resultados de los experimentos realizados con el algoritmo implementado para responder a la consulta semijoin espacial de los k vecinos más próximos. Su comportamiento parecido al de la consulta de los k vecinos más proximos continua, pues no es más que una ejecución repetitiva de este

Para la serie de experimentos realizados con esto algoritmos se han tenido en cuenta las siguientes variables: cardinalidad del conjunto de datos, tamaño de caché, tamaño de intervalo de consulta, instante de consulta, k y velocidad de la imagen.

Experimento 1. Variación de la cardinalidad. Se han fijado las variables siguiente: tamaño de caché 25, intervalo [0, 5], k 5, velocidad de la imagen (1, 1). La cardinalidad del conjunto de datos toma los siguientes valores: 500, 1.000, 5.000, 10.000.



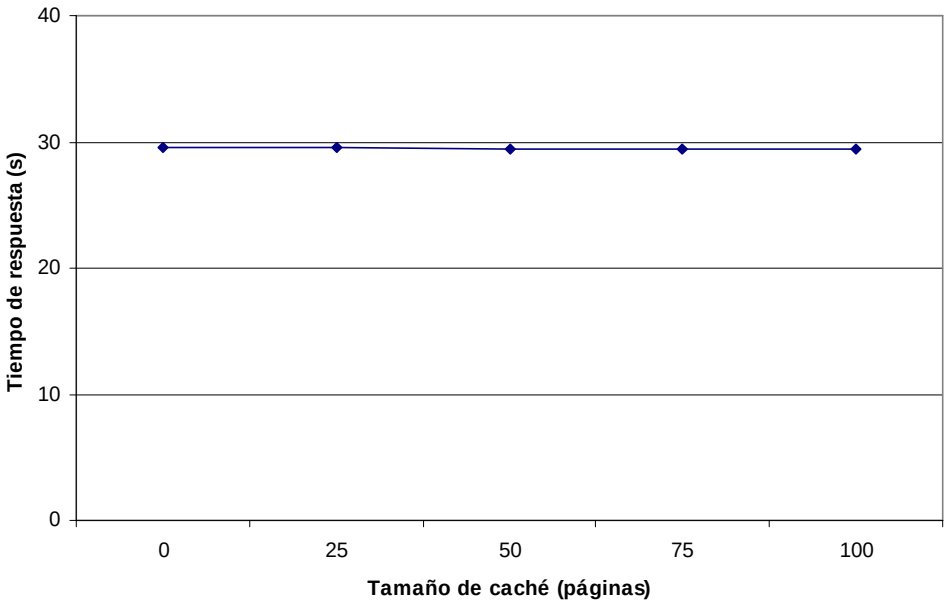
Gráfica 95. Tiempo de respuesta frente a cardinalidad para la consulta de semijoin kNN continua.



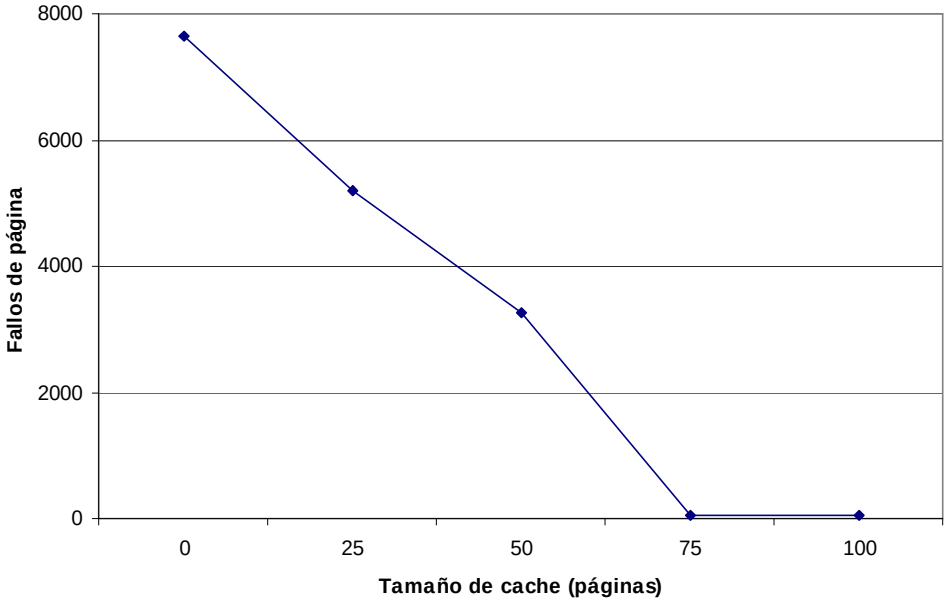
Gráfica 96. Fallos de página frente a cardinalidad para la consulta de semijoin kNN continua.

Como es de esperar, cuanto mayor es el número de objetos a procesar mayor es el tiempo de respuesta, así como el número de fallos de página. Notar la similitud de la forma de la curva con la del experimento para la consulta de los k vecinos más próximos (es una ejecución repetitiva de ese algoritmo) en la sección 4.3.2.

Experimento 2. Variación del tamaño de caché. Se han fijado las variables siguientes: cardinalidad del conjunto de datos 1.000, intervalo [0, 5], k 5, velocidad de la imagen (1, 1). El tamaño de caché toma los siguientes valores: 0, 25, 50, 75, 100.



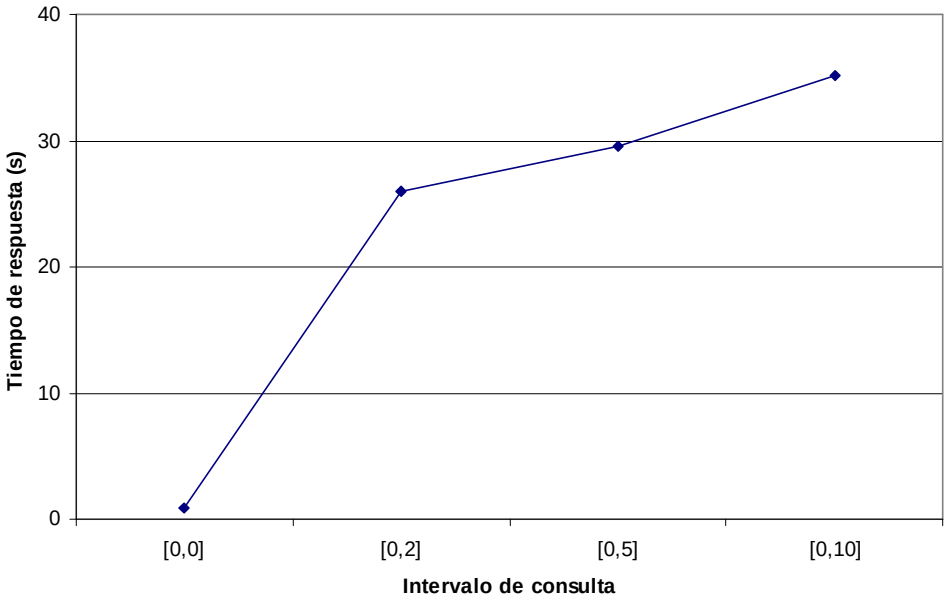
Gráfica 97. Tiempo de respuesta frente a tamaño de caché para la consulta de semijoin kNN continua.



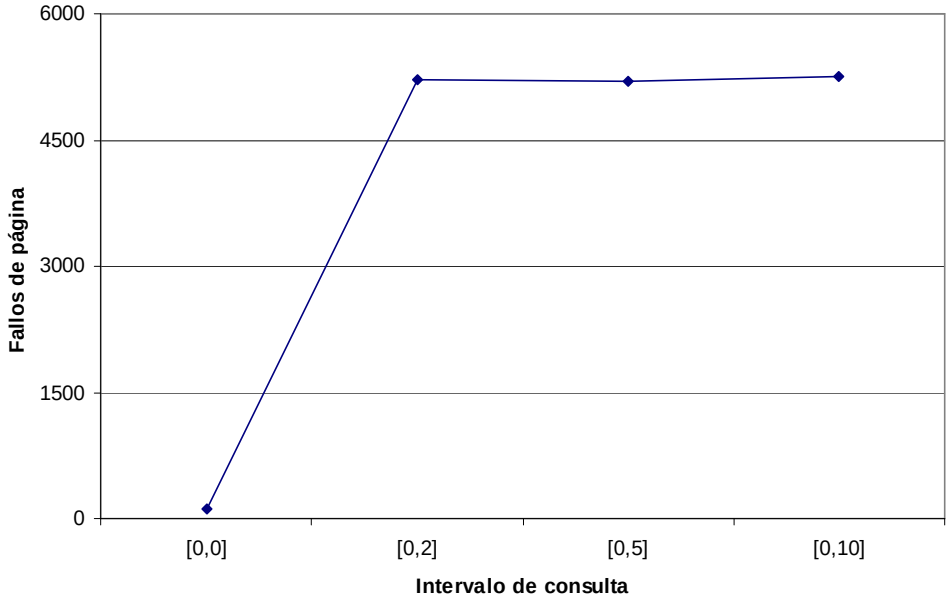
Gráfica 98. Fallos de página frente a tamaño de caché para la consulta de semijoin kNN continua.

En este experimento vemos que, al igual que en la sección 4.3.2, el número de fallos de página se estabiliza, aunque ahora lo hace para un tamaño de caché mayor, pues se están procesando más datos y se producen más fallos. De nuevo el tiempo de respuesta parece independiente del número de fallos.

Experimento 3. Variación del intervalo de consulta. Se han fijado las siguientes variables: tamaño de caché 25, cardinalidad del conjunto de datos 1.000, k 5, velocidad de la imagen (1, 1). El intervalo de consulta toma los valores: [0, 0], [0, 2], [0, 5], [0, 10].



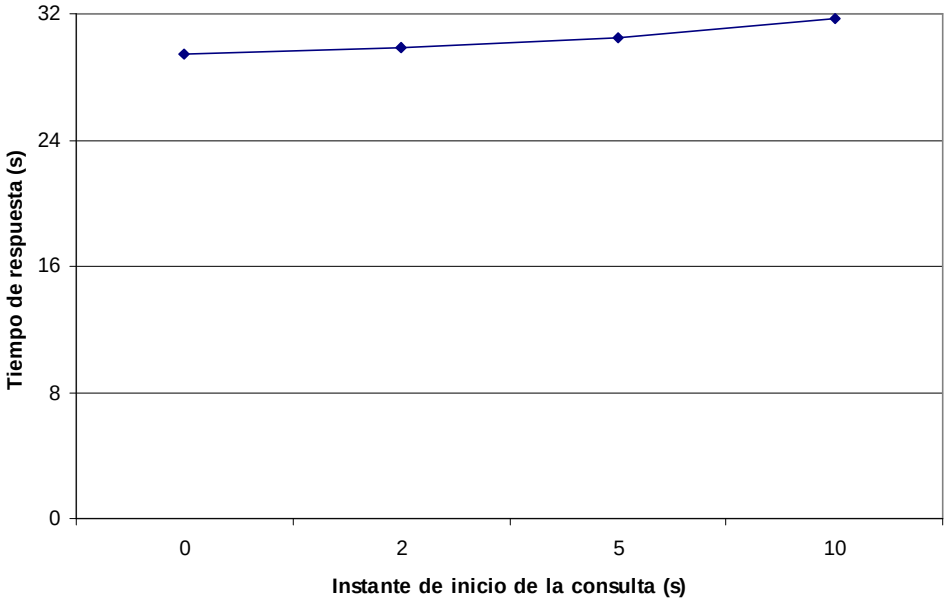
Gráfica 99. Tiempo de respuesta frente a tamaño de intervalo para la consulta de semijoin kNN continua.



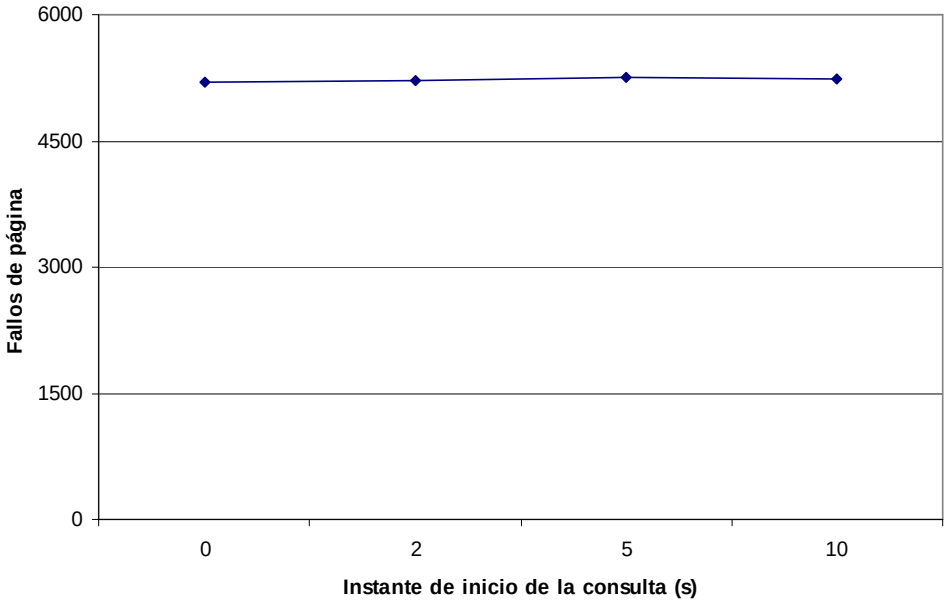
Gráfica 100. Fallos de página frente a tamaño de intervalo para la consulta de semijoin kNN continua.

Conforme crece el tamaño de los intervalos aumenta el número de cambios a devolver en el resultado y con ellos el tiempo de respuesta y el número de fallos de página. Al contrario que el la consulta de los k vecinos más próximos (sección 4.3.2), aquí tanto el número de fallos como el tiempo de respuesta crece de manera mucho más ligera para el intervalo de mayor tamaño. Esto se puede deber a que al estar considerando conjuntos más pequeños sobre los que realizar la consulta de los k vecinos más próximos continua (1000 frente a 10000 en la sección 4.3.2) el número de cambios en el resultado para este tipo de consulta sea más pequeño conforme avanza el tiempo.

Experimento 4. Variación del instante de consulta. Se han fijado las siguientes variables: tamaño de caché 25, cardinalidad del conjunto de datos 1.000, k 5, velocidad de la imagen (1, 1). El intervalo de consulta toma las siguientes variables: [0, 5], [2, 7], [5, 10], [10, 15].



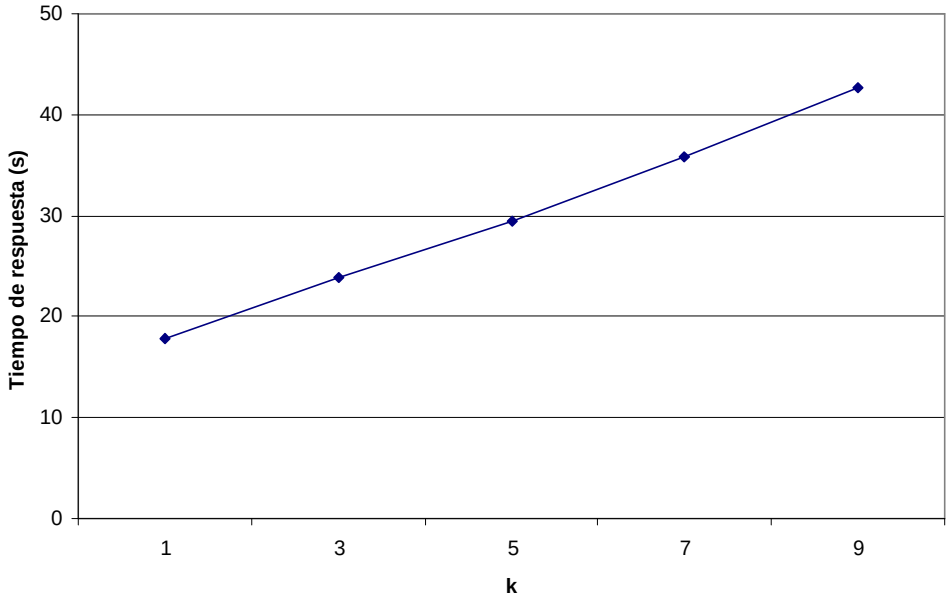
Gráfica 101. Tiempo de respuesta frente a inicio de intervalo para la consulta de semijoin kNN continua.



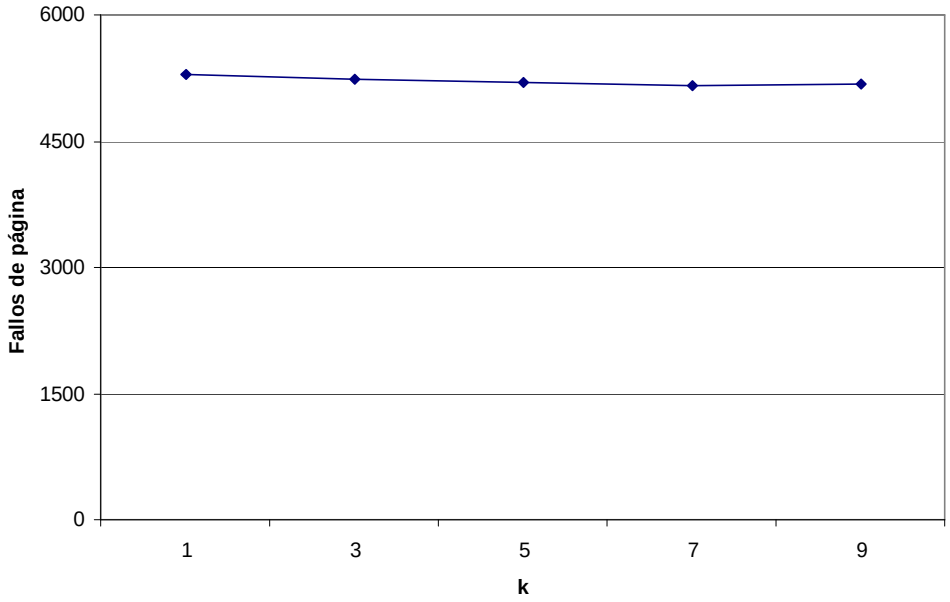
Gráfica 102. Fallos de página frente a inicio de intervalo para la consulta de semijoin kNN continua.

El retraso del inicio del intervalo de consulta hace que aumenten el tiempo de respuesta y los fallos de página, aunque de forma muy ligera. La causa de esto se puede ser la misma que en el experimento anterior.

Experimento 5. Variación de k. Se han fijado las siguientes variables: tamaño de caché 25, intervalo [0, 5], velocidad de la imagen (1, 1), cardinalidad del conjunto de datos 1.000. El número de vecinos k toma los siguientes valores: 1, 3, 5, 7, 9.



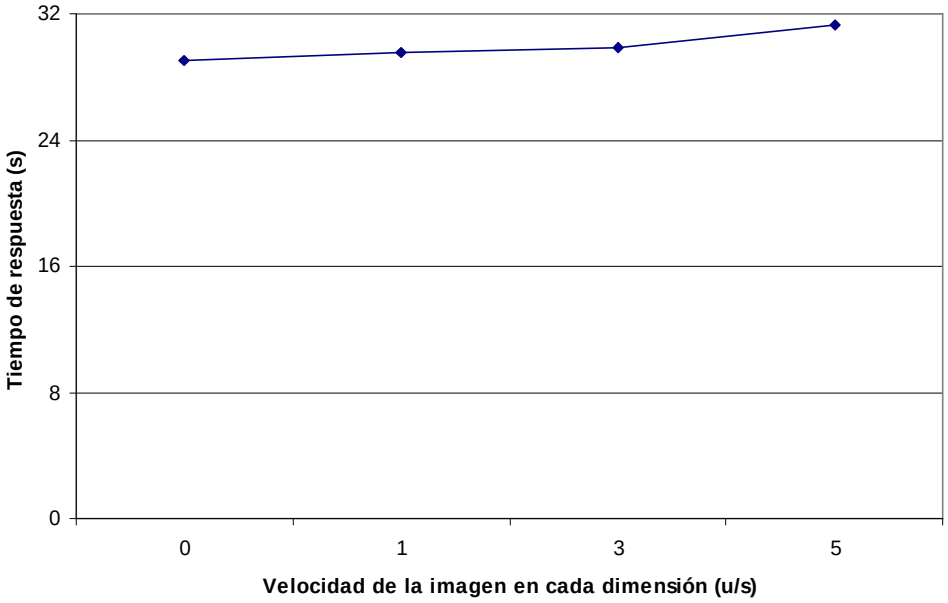
Gráfica 103. Tiempo de respuesta frente a k para la consulta de semijoin kNN continua.



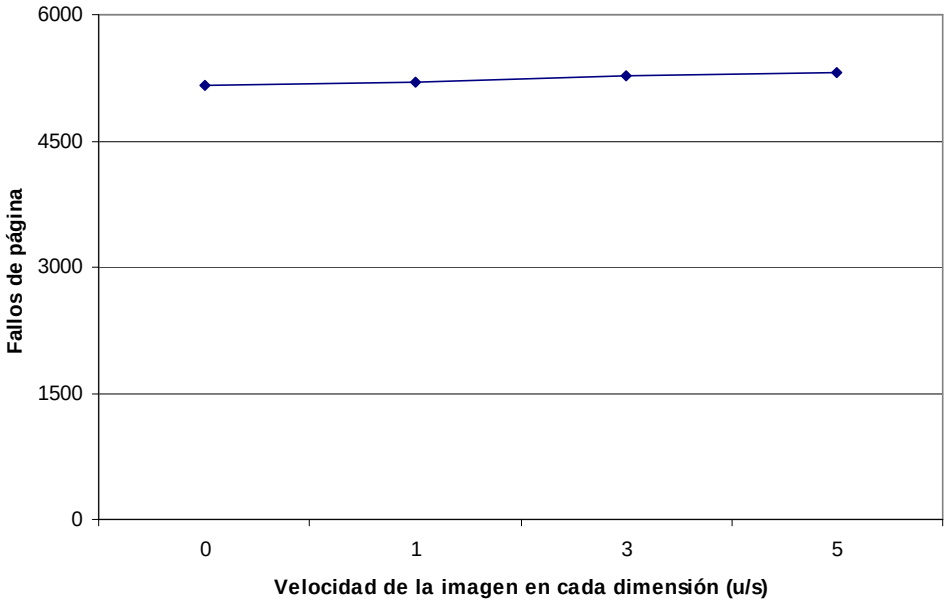
Gráfica 104. Fallos de página frente a k para la consulta de semijoin kNN continua.

El tiempo de respuesta crece de forma casi lineal con el valor de k, al igual que en la seción 4.3.2.

Experimento 6. Variación de la velocidad de la imagen. Se han fijado las siguientes variables: tamaño de caché 25, intervalo [0, 5], cardinalidad del conjunto de datos 1.000, k 5. La velocidad de la imagen toma los siguientes valores: (0, 0), (1, 1), (3, 3), (5, 5).



Gráfica 105. Tiempo de respuesta frente a velocidad de la imagen para la consulta de semijoin kNN continua.



Gráfica 106. Fallos de página frente a k para la consulta de semijoin kNN continua.

El aumento de la velocidad hace que se produzcan más solapes, por tanto se deben procesar más objetos. Por este motivo crecen tanto el tiempo de respuesta como el número de fallos de página.

4.6 Conclusiones de los experimentos

A continuación analizaremos los resultados de los diferentes algoritmos durante los experimentos, con el objetivo de extraer conclusiones acerca de sus comportamientos.

Para la *consulta de ventana parametrizada en el tiempo* se han implementado tres algoritmos: *DF simple*, *DF ordenación* (ambos recursivos) y *BF* (no recursivo). Los mejores resultados son obtenidos por *BF*, que tiene un menor número de fallos de página y un menor tiempo de respuesta en todas las circunstancias. En cuanto a las dos versiones *DF*, el menor número de fallos de página de *DF* ordenación no compensa el mayor tiempo empleado en el procesamiento de los datos. El comportamiento de los tres algoritmos es lineal con el tamaño de la ventana de consulta y todos tienen un muy pequeño número de accesos a disco, por lo que este no parece ser un factor determinante. Las variaciones en el intervalo de consulta, tanto su momento de inicio como su tamaño, tienen poca influencia en el tiempo de respuesta y en el número de fallos de página. El tiempo de respuesta es del orden de milésimas de segundo, por lo que podría ser aplicado en muchas situaciones.

Para la *consulta de ventana continua* se ha implementado un único algoritmo no recursivo. Su comportamiento es muy parecido al algoritmo que responde a la consulta TP, ya que es similar a éste y devuelve los resultados adicionales de manera incremental. Empeora de forma lineal con el tamaño de la ventana de consulta e incurre en pocos fallos de página, pero se ve más afectado por el aumento de tamaño del intervalo de consulta. Su tiempo de respuesta es del orden de centésimas de segundo. Podría ser utilizado en lugar del algoritmo TP, ya que tiene un tiempo de respuesta y número de fallos de página parecidos y su resultado da más información que el de éste con un coste muy parecido.

Para la *consulta de los k vecinos más próximos parametrizada en el tiempo* se ha implementado un único algoritmo no recursivo. Tanto el tiempo de respuesta como el número de fallos de página crece de forma lineal con el tamaño del conjunto de datos sobre el que se realiza la consulta. También tiene un comportamiento lineal con el valor de k, el número de vecinos buscados. El número de fallos de página no se ve muy afectado por el tamaño de la caché salvo para cuando este es 0, por lo que en sistemas con poca memoria su comportamiento podría empeorar. Se ve muy poco afectado por el aumento de tamaño del intervalo de consulta, mientras que el retraso de su inicio tiene más influencia, aunque no de manera exagerada. Para intervalos puntuales su comportamiento no es muy bueno, por lo que no es recomendable para realizar consultas puntuales. Su tiempo de respuesta es del orden de medio segundo en el peor de los casos (salvo para intervalos puntuales como se ha dicho antes), por lo que podría ser aplicado a un amplio rango de situaciones.

Para la *consulta de los k vecinos más próximos continua* se ha implementado un único algoritmo no recursivo y repetitivo. Su tiempo de respuesta y su número de fallos de página crece linealmente con el tamaño del conjunto de datos, pero más rápido que en problemas anteriores. El número de fallos de página se ve poco afectado por el tamaño de la caché. Por la naturaleza de la consulta, el aumento del tamaño del intervalo de consulta hace que aumenten rápidamente el tiempo de respuesta y el número de fallos de página, aunque tienden a estabilizarse. El retraso del inicio del intervalo de consulta tiene menor influencia. Con un tiempo de respuesta en torno al minuto, para problemas que traten con conjuntos de datos grandes (mayores de 10000) quizás su uso no sea recomendable si el tiempo de respuesta es un factor crítico. De nuevo es un algoritmo que aporta información más completa que su homónimo TP, y si el tiempo de respuesta no es un factor determinarse podría usarse en lugar de éste.

Para la *consulta de join espacial de intervalo* se han implementado cuatro algoritmos: *DF*, *DF planesweep* (realizan recorridos en profundidad del árbol), *BF ordsum* y *BF ordcen* (realizan recorridos en anchura del árbol). En general, los algoritmos *DF* incurren en un número de fallos de página muy inferior a *BF*, cosa que no se ve reflejada en el tiempo de respuesta, que es similar, aunque los *DF* siguen siendo mejores. El aumento del tamaño del conjunto de datos afecta de manera importante al número de fallos de página de los algoritmos *BF*, en especial a *BF ordcen*, que dobla a *BF ordsum*, mientras que los algoritmos *DF* crecen con una pendiente muy suave. El tamaño de caché también afecta mucho a los fallos de página de los algoritmos *BF*, sobre todo para *BF ordsum*, que para caché de 500 páginas tienen un número de fallos muy cercano a los *DF*. Las variaciones en el tamaño y el instante de inicio de consulta afecta a las dos métricas de rendimiento salvo en el caso de los fallos de página de los algoritmos *DF*, que permanecen casi constantes bajo toda circunstancia. En sistemas donde el acceso a disco sea un factor limitante es recomendable usar los algoritmos *DF*, puesto que el número de fallos de página es drásticamente menor que el de los *BF* y se mantiene estable en las distintas situaciones. Para intervalos grandes los *BF* se han comportado mejor en experimentos con datos reales. Podemos pensar que en estos casos son más idóneos, pero debemos recordar que su número de fallos es mucho mayor que el de las versiones *DF* y además su consumo de memoria también es mayor, debido a la creación del IJI. Por tanto, sólo para consultas sobre intervalos grandes y en sistemas con una gran caché y memoria suficiente los algoritmos *BF* ofrecen un mejor desempeño.

Para la *consulta de join espacial parametrizada en el tiempo* se han implementado tres algoritmos: *DF simple*, *DF ordenación* (ambos recursivos), y *BF* (no recursivo). El tiempo de respuesta y el número de fallos de página de *BF* es siempre mejor que el de los dos algoritmos *BF*, por lo que éste es una mejor opción. El comportamiento de los tres es parecido, llamando la atención que para experimentos con caché de tamaño 0 el número de fallos de página es muy elevado, aunque cae rápidamente a valores pequeños y se mantiene estable en cuanto se empieza a aumentar el tamaño de la caché. En general todos se ven más afectados por el retraso del inicio del intervalo de consulta que por cualquier otro factor. En cualquier caso el algoritmo BF demuestra comportarse mejor, por lo que sería la elección a realizar para su aplicación en cualquier circunstancia por encima de los demás.

Para la *consulta de join espacial continua* se ha implementado un único algoritmo no recursivo. Su comportamiento es prácticamente igual al del algoritmo de join TP BF, con la principal característica de una rápida caída y estabilización del número de fallos de página al aumentar el tamaño de la caché, salvo que el tiempo de respuesta y el número de fallos de página se ven más afectados por el aumento de tamaño del intervalo de consulta. Para consultas en las que se retrasa el inicio del intervalo de consulta, el tiempo de respuesta, en general, es algo menor que el de los algoritmos para join de intervalo y para join TP, por lo que podría ser más interesante usar este algoritmo para responder a todas ellas, puesto que da una información más completa y por tanto más útil. Para intervalos de consulta mayores duplica o triplica los resultados de los algoritmos para join de intervalo y para join TP, por lo que dependiendo de las necesidades de la aplicación también podría usarse en lugar de ellos.

En cuanto al uso de memoria, los algoritmos recursivos hacen uso de ella mediante las llamadas recursivas, aunque esta cantidad está relacionada con la profundidad del índice, mientras que los algoritmos *BF* (*Best-First*, no *Breadth-First*) hacen uso de un

montículo cuyo tamaño está relacionado con el número de objetos en el índice. Por tanto, a la hora de elegir uno u otro es posible que si la memoria disponible es un factor limitante, pueda ser mejor usar las versiones *DF*.

Para la *consulta de semijoin de los k vecinos más próximos continua* se ha implementado un único algoritmo. Su comportamiento es similar al del algoritmo de la consulta de los k vecinos más próximos continua, pues no es más que una ejecución repetitiva de este. Sin embargo los valores de tiempo de respuesta y fallos de página son elevados, alcanzando el primero unos 500 segundos en el peor de los casos, por lo que quizás no sea de utilidad práctica es situaciones donde el tiempo de respuesta pequeño sea un factor a tener en cuenta.

En general, salvo para la consulta de los k vecinos más próximos, parece interesante usar el algoritmo de consulta continua en lugar del algoritmo TP, puesto que con tiempos de respuesta bastante parecidos dan una información más completa.

Capítulo 5

CONCLUSIONES Y TRABAJOS FUTUROS

A continuación se exponen las conclusiones obtenidas del trabajo realizado y algunos posibles trabajos futuros para continuar en la línea de este proyecto.

5.1 Conclusiones

A lo largo de este proyecto se han estudiado métodos de acceso espacio-temporales como el TPR*-tree y se han estudiado estrategias para responder a consultas espacio-temporales aprovechando su estructura. Se han estudiado e implementado algoritmos para resolver diversas consultas espacio-temporales en TPR*-trees tales como consultas parametrizadas en el tiempo y consultas continuas. Para algunas consultas se han implementado más de un algoritmo, de manera que podemos comparar su comportamiento en diversas circunstancias. Para otras solamente se ha implementado una versión, pero aún así podemos extraer algunas conclusiones interesantes.

Para la *consulta de ventana parametrizada en el tiempo* se han implementado tres algoritmos: *DF simple*, *DF ordenación* (ambos recursivos) y *BF* (no recursivo). Los mejores resultados son obtenidos por *BF*, que tiene un menor número de fallos de página y un menor tiempo de respuesta en todas las circunstancias. En cuanto a las dos versiones *DF*, el menor número de fallos de página de *DF* ordenación no compensa el mayor tiempo empleado en el procesamiento de los datos. El comportamiento de los tres algoritmos es lineal con el tamaño de la ventana de consulta y todos tienen un muy pequeño número de fallos de página, por lo que este no parece ser un factor determinante. Las variaciones en el intervalo de consulta, tanto su momento de inicio como su tamaño, tienen poca influencia en el tiempo de respuesta y en el número de fallos de página. El tiempo de respuesta es del orden de milésimas de segundo, por lo que podría ser aplicado en una amplia variedad de prolemas. El pequeño número de fallos de página lo hace utilizable tanto para sistemas con caché como para sistemas sin caché.

Para la *consulta de ventana continua* se ha implementado un único algoritmo no recursivo. Su comportamiento es muy parecido al algoritmo *BF* que responde a la consulta de ventana TP, ya que es similar a éste y devuelve los resultados adicionales de manera incremental. Empeora de forma lineal con el tamaño de la ventana de consulta e incurre en pocos fallos de página, pero se ve más afectado por el aumento de tamaño del intervalo de consulta. Su tiempo de respuesta es del orden de centésimas de segundo.

Podría ser utilizado en lugar del algoritmo TP, ya que tiene un tiempo de respuesta parecido y su resultado da más información que el de éste.

Para la *consulta de los k vecinos más próximos parametrizada en el tiempo* se ha implementado un único algoritmo no recursivo. Tanto el tiempo de respuesta como el número de fallos de página crece de forma lineal con el tamaño del conjunto de datos sobre el que se realiza la consulta. También tiene un comportamiento lineal con el valor de k, el número de vecinos buscados. El número de fallos de página no se ve muy afectado por el tamaño de la caché salvo para cuando éste es 0, por lo que en sistemas con poca memoria su comportamiento podría empeorar. Se ve muy poco afectado por el aumento de tamaño del intervalo de consulta, mientras que el retraso de su inicio tiene más influencia, aunque no de manera exagerada. Para intervalos puntuales su comportamiento no es muy bueno, por lo que no es recomendable para realizar consultas puntuales. Su tiempo de respuesta es del orden de medio segundo en el peor de los casos (salvo para intervalos puntuales como se ha dicho antes), por lo que podría tener muchas aplicaciones útiles.

Para la *consulta de los k vecinos más próximos continua* se ha implementado un único algoritmo no recursivo. Su tiempo de respuesta y su número de fallos de página crece linealmente con el tamaño del conjunto de datos, pero más rápido que en problemas anteriores. El número de fallos de página se ve poco afectado por el tamaño de la caché. Por la naturaleza de la consulta, el aumento del tamaño del intervalo de consulta hace que aumenten rápidamente el tiempo de respuesta y el número de fallos de página, aunque tienden a estabilizarse. El retraso del inicio del intervalo de consulta tiene menor influencia. Con un tiempo de respuesta en torno al minuto, para problemas que traten con conjuntos de datos grandes quizás su uso no sea recomendable si el tiempo de respuesta es un factor crítico.

Para la *consulta de join espacial de intervalo* se han implementado cuatro algoritmos: *DF*, *DF planesweep* (realizan recorridos en profundidad del árbol), *BF ordsum* y *BF ordcen* (realizan recorridos en anchura del árbol). En general, los algoritmos DF incurren en un número de fallos de página muy inferior a *BF*, cosa que no se ve reflejada en el tiempo de respuesta, que es similar, aunque los *DF* siguen siendo mejores. El aumento del tamaño del conjunto de datos afecta de manera importante al número de fallos de página de los algoritmos *BF*, en especial a *BF ordcen*, que dobla a *BF ordsum*, mientras que los algoritmos *DF* crecen con una pendiente muy suave. El tamaño de caché también afecta mucho a los fallos de página de los algoritmos *BF*, sobre todo para *BF ordsum*, que para caché de 500 páginas tienen un número de fallos muy cercano a los *DF*. Las variaciones en el tamaño y el instante de inicio de consulta afecta a las dos métricas de rendimiento salvo en el caso de los fallos de página de los algoritmos *DF*, que permanecen casi constantes bajo toda circunstancia. En sistemas donde el acceso a disco sea un factor limitante es recomendable usar los algoritmos *DF*, puesto que el número de fallos de página es drásticamente menor que el de los *BF* y apenas varía en las distintas situaciones.

Para la *consulta de join espacial parametrizada en el tiempo* se han implementado tres algoritmos: *DF simple*, *DF ordenación* (ambos recursivos), y *BF* (no recursivo). El tiempo de respuesta y el número de fallos de página de *BF* es siempre mejor que el de los dos algoritmos *BF*, por lo que éste es una mejor opción. El comportamiento de los tres es parecido, llamando la atención que para experimentos con caché de tamaño 0 el número de fallos de página es muy elevado, aunque cae rápidamente a valores pequeños

y se mantiene estable en cuanto se empieza a aumentar el tamaño de la caché. En general todos se ven más afectados por el retraso del inicio del intervalo de consulta que por cualquier otro factor. Su tiempo de respuesta, en general, es algo menor que el de los algoritmos para join de intervalo, por lo que, aunque responden a consultas ligeramente diferentes, podría ser más interesante usar este algoritmo para responder a ambas consultas.

Para la *consulta de join espacial continua* se ha implementado un único algoritmo no recursivo. Su comportamiento es prácticamente igual al del algoritmo de join TP BF, con la principal característica de una rápida caída y estabilización del número de fallos de página al aumentar el tamaño de la caché, salvo que el tiempo de respuesta y el número de fallos de página se ven más afectados por el aumento de tamaño del intervalo de consulta.

Para la *consulta de semijoin de los k vecinos más próximos continua* se ha implementado un único algoritmo. Su comportamiento es similar al del algoritmo de la consulta de los k vecinos más próximos continua, pues no es más que una ejecución repetitiva de este. Sin embargo los valores de tiempo de respuesta y fallos de página son elevados, alcanzando el primero unos 500 segundos en el peor de los casos, por lo que quizás no sea de utilidad práctica es situaciones donde el tiempo de respuesta pequeño sea un factor a tener en cuenta.

En general, salvo para la consulta de los k vecinos más próximos, parece interesante usar el algoritmo de consulta continua en lugar del algoritmo TP, puesto que con tiempos de respuesta bastante parecidos dan una información más completa.

5.2 Trabajos Futuros

Estos serían posibles trabajos a desarrollar continuando en la línea de este proyecto:

- Implementar la versión single-pass del algoritmo de los k vecinos más próximos continuo descrita en la sección 3.5.2 y comparar su comportamiento con el de la versión repetitiva.

- Desarrollar e implementar una versión del algoritmo para la consulta de semijoin de los k vecinos más próximos continua que aproveche la estructura subyacente del TPR*-tree para realizar la consulta de manera más eficiente y comparar su comportamiento con la versión implementada en este proyecto.

- Desarrollar e implementar versiones para entornos dinámicos de otras consultas como la consulta del vecino más próximo inversa [BJKS02] o la consulta de join de similitud.

BIBLIOGRAFÍA

[AAE00] P. K. Agarwal, L. Arge y J. Erickson: "*Indexing moving points"*. Proc. of the ACM Symp. on Principles of Database Systems (PODS), páginas 175-186, 2000.

[BJKS02] R. Benetis, C. Jensen, G. Karciauskas y S Saltenis: "*Nearest neighbor and reverse nearest neighbor queries for moving objects*". Proceedings of International Database Engineering and Applications Symposium (July), páginas 44-53, 2002.

[BKS93] T. Brinkhoff, H.-P. Kriegel y B. Seeger: "*Efficient processing of spatial joins using R-trees"*. ACM SIGMOD, páginas 237-246, 1993.

[BKSS90] N. Beckmann, H.P. Kriegel, R. Schneider y B. Seeger: "*The R*-tree: an Effcient and Robust Access Method for Points and Rectangles*". ACM SIGMOD Conference, páginas 322-331, 1990.

[Cam07] C. Campos: "*Procesamiento de Consultas entre Regiones y Objetos Móviles*". PFC Universidad de Almería, 2007.

[CJL08] S. Chen, C.S. Jensen y D. Lin. "*A Benchmark for Evaluating Moving Object Indexes*". PVLDB '08, August 23-28, 2008, Auckland, New Zealand. 2008.

[Cor02] A. Corral: "*Algorithms for Processing of Spatial Queries using R-trees. The Closest Pairs Query and its Application on Spatial Databases"*. Tesis Doctoral, Universidad de Almería, España, 2002.

[CTVM08] A. Corral, M. Torres, M. Vassilakopoulos, Y. Manolopoulos. "*Predictive Join Processing between Regions and Moving Objects"*. ADBIS Conference, LNCS, Vol. 5207, páginas. 46-61. 2008.

[GuS05] R.H. Guting y M. Schneider: "*Moving Objects Databases*". Morgan Kaufmann, 2005.

[Gut84] A. Guttman: "*R-trees: a Dynamic Index Structure for Spatial Searching*". *ACM* SIGMOD Conference 1984, paginas 47-57, 1984.

[HJR97] Y.W. Huang, N. Jing y E.A. Rundensteiner: "*Spatial joins using R-trees: Breadth-first traversal with global optimizations"*. Proceedings 23rd VLDB Conference, páginas 396–405. 1997.

[HS99] G. Hjaltason y H. Samet: "*Distance browsing in spatial databases"*. ACM Trans. Datab.Syst. 24, 2, páginas 265-318, 1999.

[JaS07] E.H. Jacox y H.Samet: "*Spatial Join Techniques*". TODS 32(1) 7, páginas 1-44, 2007.

[KGT99] G. Kollios, D. Gunopulos y V. J. Tsotras: "*On indexing mobile objects"*. Proc. of the ACM Symp. on Principles of Database Systems (PODS), páginas 261-272, 1999.

[Lau05] A. Lau: "*Processing frequent updates with the TPR*-Tree using Bottom-Up Updates*". http://softbase.uwaterloo.ca/~atklau/bu_tprs_tree.pdf, 2005.

[MGA03] M. F. Mokbel , T. M. Ghanem y W. G. Aref: "*Spatio-Tempotal Access Methods*". 2003.

[MNP06] Y. Manolopoulos, A. Nanopoulos, A.N. Papadopoulos, Y. Theodoridis: "*R-Trees: Theory and Applications*". Springer, 2006.

[MPV05] Y. Manolopoulos, A. N. Papadopoulos, M. G. Vassilakopoulos: "*Spatial Databases: Technologies, techniques and trends*". Ideal Group, 2005.

[NAM10] L.-V. Nguyen-Dinh, W. G. Aref y M. F. Mokbel: "*Spatio-Tempotal Access Methods: Part 2(2003-2010)*", 2010.

[NS98] M. A. Nascimento y J. R. Silva: "*Towards historical r-trees*". Proc. of the ACM Symp. on Applied Computing (SAC), páginas 235-240, 1998.

[PAH02] C. M. Procopiuc, P. K. Agarwal y S. Har-Peled: "*Star-tree: An ef_cient self-adjusting index for moving objects*". Proc. of the Workshop on Alg. Eng. and Experimentation (ALENEX), páginas 178-193, 2002.

[PJT00] D. Pfoser, C. S. Jensen y Y. Theodoridis: "*Novel approaches to the indexing of moving object trajectories*". Proc. of the 26th Intl. Conf. on Very Large Data Bases (VLDB), 2000.

[PM97] A. Papadopoulos y Y. Manolopoulos: "*Performance of nearest neighbor queries in R-trees*". Proceedings of International Conference on Database Theory (ICDT) (Jan.), páginas 394-408, 1997.

[PXKAH02] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref y S. E. Hambrusch: "*Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects"*. IEEE Transactions on Computers 51, páginas 1124-1140, 2002.

[RKV95] N. Rousopoulos, S. Kelly y F. Vincent: "*Nearest neighbor queries*". Proceedings of the ACM SIGMOD Conference (May), ACM, New York, páginas 71-79, 1995.

[SJ01] S. Saltenis y C. S. Jensen: "*Indexing of moving objects for location-based services"*. Technical Report TR-63, Time Center, 2001.

[SJLL00] S. Saltenis, C.S. Jensen, S.T. Leutenegger y M.A. Lopez: "*Indexing the Positions of Continuously Moving Objects*". ACM SIGMOD Conference, páginas 331-342, 2000.

[Sam90] H. Samet: "*Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*". Addison-Wesley, 1990.

[TP01] Y. Tao y D. Papadias: "*The mv3r-tree : A spatio-temporal access method for timestamp and interval queries"*. Proc. of the 27th Intl. Conf. on Very Large Data Bases (VLDB), 2001.

[TP02] Y. Tao y D. Papadias: "*Time-Parametrized Queries in Spatio-Temporal Databases*". ACM SIGMOD'2002 June 4-6, 2002.

[TP03] Y. Tao y D. Papadias: "*Spatial Queries in Dynamic Enviroments*". ACM Transactions on Database Systems, Nº 2,Vol. 28, páginas 101-139, 2003.

[TPS02] Y. Tao, D. Papadias y Q. Shen: "*Continuous Nearest Neighbor Search*". Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002.

[TPS03] Y. Tao, D. Papadias y J. Sun: "*The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries*". VLDB Conference, páginas 790-801, 2003.

[TSP03] Y. Tao, J. Sun, y D. Papadias: *"Selectivity Estimation for Predictive Spatio-Temporal Queries"*. ICDE, 2003.

[TUW98] J. Tayeb, O. Ulusoy y O. Wolfson: "*A quadtree-based dynamic attribute indexing method"*. The Computer Journal 41**,** páginas 185-200, 1998.

[TVM98] T. Tzouramanis, M. Vassilakopoulos y Y. MAnolopoulos: "*Overlapping Linear Quadtrees: a Spatio-temporal Access Method*". ACM GIS '98, Washington, D.C., USA, 1998.

[TVS96] Y. Theodoridis, M. Vazirgiannis y T. Sellis: "*Spatio-temporal indexing for large multimedia applications*". Proc. of the IEEE Conf. on Multimedia Computing and Systems (ICMSC), 1996.

[XHL90] X. Xu, J. Han y W. Lu: "*Rt-tree: An improved r-tree index structure for spatiotemporal databases*". Proc. of the Intl. Symp. on Spatial Data Handling (SDH), páginas 1040-1049, 1990.

[ZLRB08] R. Zhang, D. Lin, K. Ramamohanarao, E. Bertino: "*Continuous Intersection Joins Over Moving Objects*". ICDE Conference, páginas 863-872, 2008.