UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

CLASIFICACIÓN Y COMPARACIÓN DE MÉTODOS DE DETECCIÓN DE COLISIONES DE FASE AMPLIA PARA IMÁGENES SINTÉTICAS EN 2D+TIEMPO

TÉSIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS
MENCIÓN COMPUTACIÓN

HÉCTOR MARCELO MORAGA AROS

PROFESOR GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
BENJAMÍN BUSTOS CÁRDENAS
PATRICIO INOSTROZA FAJARDIN
FERNANDO RANNOU FUENTES

SANTIAGO DE CHILE
2019

Resumen

La detección de colisiones ha sido un tópico de extenso estudio en computación gráfica y geometría computacional. Sus campos de aplicación incluyen la robótica, biología computacional, juegos, y simulaciones en cirugía, física y biología.

Un entorno real complejo para la detección de colisiones corresponde a imágenes biomédicas y astronómicas: imágenes en series de tiempo, de gran tamaño, a veces con gran cantidad de objetos, que se mueven y se deforman, y que pueden aparecer o desaparecer de la escena, o experimentar autocolisiones, aspectos que pueden hacer la detección de colisiones un trabajo demasiado lento de calcular sin las estrategias apropiadas.

Este trabajo realiza un contraste entre dos enfoques de detección de colisiones de fase amplia en 2D+tiempo: usando los métodos estáticos sweep and prune directo, que funcionan cuadro a cuadro, y el método sweep and prune incremental, que mantiene las estructuras a lo largo de la simulación, contra el enfoque cinético del algoritmo sweep and prune cinético. Ambos enfoques fueron probados en simulaciones de objetos con movimiento rectilíneo en direcciones aleatorias. La comparación se hizo siguiendo los lineamientos de experimentación en algoritmos establecidos por Moret [26] respecto a qué parámetros es pertinente medir, las diez sugerencias de Johnson [22] para realizar un experimento con algoritmos y las sugerencias de Sanders [31] para mostrar los resultados en forma de gráficos.

Los resultados muestran que el uso del algoritmo sweep and prune cinético es un orden de magnitud más rápido que los algoritmos sweep and prune estáticos en simulaciones que involucran una distribución aleatoria de objetos en movimiento, sin importar la densidad de objetos en la simulación. También se estudia el efecto de colisiones múltiples dentro de las simulaciones, un suceso que no fue investigado por Coming & Staad [11]. Esto permite hacer sugerencias sobre las instancias en las que un algoritmo es más apropiado que otro.

Dedicado a mis padres y a todos
aquellos de los que he aprendido mucho
en la vida: perseverancia, trabajo y amistad

Agradecimientos

Quiero agradecer especialmente a la profesora Nancy Hitschfeld-Kahler por permitirme realizar un tema de investigación de mi autoría. También me gustaría agradecer a Steffen Härtel por permitirme trabajar en el SCIAN Lab de la facultad de medicina de la Universidad de Chile, lo que me inspiró para el tema de esta tesis. Finalmente, me gustaría agradecer a los profesores que han sido fuente de sabiduría, inspiración y amistad: Jérémy Barbay, Claudio Gutierrez, Sergio Ochoa, Éric Tanter y José Urzúa.

Tabla de Contenido

Índice de figuras

Índice de algoritmos

Índice de tablas

Capítulo 1

Introducción

Por más de treinta años, la detección de colisiones ha sido un tópico importante en geometría computacional. Sus aplicaciones incluyen la robótica, juegos, biología computacional y simulaciones en cirugía y física. Su fin último es acelerar la detección de colisiones al evitar procesar objetos que no están involucrados directamente en una posible colisión. El problema se vuelve más desafiante aún si se desea determinar dónde estos objetos colisionan, si se desplazan a lo largo del tiempo, si por contacto estos objetos se deforman, o incluso si el propio objeto experimenta auto-colisiones [24, 27, 33, 35].

Experimentos en biología y observaciones en astronomía corresponden a entornos reales para la detección de colisiones ya que en ellos se encuentran involucrados varios de los problemas descritos anteriormente. Si se considera además que algunos de estos experimentos en la actualidad involucran pilas(stacks) de imágenes de alta resolución con gran cantidad de objetos (en la escala de TB [32]) y con muchas secuencias temporales [20, 23, 37], el análisis se hace significativamente más costoso sin las herramientas apropiadas.

Dentro de la detección de colisiones existen distintas clasificaciones, dependiendo de si se trata de análisis estáticos o dinámicos, de detección de colisiones de fase amplia o detección de colisiones de fase angosta, rígidos o deformables, entre otros [7, 21, 24, 25].

En esta tesis se propone la comparación de dos métodos de detección de colisiones de fase amplia, el Sweep and Prune estático, basado en el método de Cohen [9] contra el método Sweep and Prune cinético de Coming & Staadt [11]. Para ello se implementó cuatro simulaciones, tres de las cuales son utilizadas para validar el correcto funcionamiento de los algoritmos involucrados: Sweep and Prune estático, en sus versiones directo e incremental, y Sweep and Prune cinético. La última simulación es la encargada de registrar el desempeño de los anteriores algoritmos, usando una cantidad variable de objetos desplazándose en trayectorias lineales. Las comparaciones se llevarán a cabo en cuanto a

cantidad de objetos involucrados, cantidad de operaciones, densidad, entre otras medidas de interés.

La comparación es pertinente debido a que en entornos biológicos de microscopía pueden existir muchos *stacks* de imágenes de alta resolución, a veces con múltiples objetos los que pueden significar un gasto de recursos si entre conjuntos de imágenes consecutivas las escenas permanecen prácticamente con la misma distribución física, lo que es llamado una alta coherencia espacial y temporal.

Mientras que para el uso de las *Jerarquía de Volúmenes Envolventes (bvh)*[1] es necesario que los objetos sean rígidos y estáticos [12], para las estructuras de datos cinéticas se requiere de antemano un "plan de vuelo", que define las trayectorias de las primitivas, y el establecimiento de un certificado cuya falla es la causante de las actualizaciones de la estructura [19]. Si bien en [11] establecen la comparación entre el método de detección de colisiones de fase amplia Sweep and Prune en su versión estática y su versión dinámica, deja de lado la detección de eventos múltiples[2], que de suceder, pueden ser catastróficos para el correcto funcionamiento del algoritmo, como lo establece *Aban et al.* [2]. En esta tesis se incluye un método para detectar de antemano la existencia de estos eventos múltiples en las simulaciones cinéticas. El considerar escenarios con muchos objetos en movimiento hace pertinente estudiar la real ganancia de utilizar métodos cinéticos por sobre métodos estáticos en series de tiempo. En particular, se desea probar si se puede sugerir a priori qué tipo de método seleccionar, basándose en alguna medida inicial. Una vez resuelta la etapa de detección de colisiones de fase amplia en la detección de colisiones, si se desea establecer si existe verdaderamente colisión entre primitivas de un par de objetos, se utilizan los métodos de fase angosta, como por ejemplo, la *bvh* que fue implementada para su uso por el laboratorio de análisis de imágenes científicas (SCIAN-Lab) de la Facultad de Medicina de la Universidad de Chile en la sección 3.7. Respecto a las investigaciones experimentales de algoritmos, esta tesis ha aplicado algunas de las recomendaciones y principios establecidos por los papers de Moret [26], Johnson [22] y Sanders [31].

Moret establece lineamientos para la validación empírica y montaje experimental al trabajar con algoritmos. Si bien su enfoque principalmente fue establecido para afrontar los cada vez más complejos algoritmos teóricos, en especial para afrontar los problemas tratables, es un buen punto de referencia para poder comparar los dos tipos de algoritmos. Sanders da sugerencias sobre la forma en cómo presentar resultados experimentales en algoritmos permitiendo la reproducibilidad del experimento. Johnson aportó con diez principios interrelacionados entre sí que deberían tomarse en cuenta a la hora de crear

[1] forma de asociar una serie de objetos distribuidos espacialmente como un árbol binario.

[2] Situación que se da en una simulación cuando se tienen tres o más elementos con el mismo valor de una coordenada (de una posición) al mismo tiempo. Por ejemplo, cuando tres aristas poseen su coordenada x con el mismo valor en un instante de tiempo determinado.

publicaciones experimentales en el área de algoritmos. Estos principios son:

1. **Desarrollar experimentos relevantes**: en el caso de esta investigación el interés principal tiene que ver con la comprobación experimental del tiempo de ejecución de ambos tipos de algoritmos Sweep And Prune (estático versus cinético), tomando en cuenta un aspecto que investigaciones anteriores [10] expresamente intentan evitar, como es el caso de colisiones múltiples. Por otro lado, se desea comprobar si el algoritmo Sweep and Prune cinético es una alternativa práctica de utilizar contra el algoritmo Sweep and Prune incremental.

2. **Enlazar tu paper con la literatura**: en esta investigación la comparación se hará directamente con los algoritmos de [9] y [10]. Las simulaciones tendrán como parámetros las cantidadades de: objetos en escena y porcentaje inicial de ocupación. En este sentido es diferente al ambiente de pruebas de [10, 11] ya que en este se ejecuta los algoritmos en el período de cada cuadro, precalculando cuales son los eventos que han de ejecutarse en el intervalo correspondiente. En esta investigación sólo se hace correr las simulaciones sin detención ni precálculo de los eventos por cuadro (con la única excepción del cálculo de los eventos iniciales), sino que en la medida que sucede un evento se calculan los siguientes.

3. **Usar instancias de bancos de prueba que puedan apoyar conclusiones generales**: en este sentido se han desarrollado ambientes de prueba para corroborar el correcto funcionamiento de los diferentes algoritmos y una simulación con distribución espacial aleatoria de objetos de diferentes formas para la comparación del tiempo de ejecución de cada algoritmo. Cada banco de pruebas ha sido almacenado para distintas cantidades de objetos en la escena, diferentes porcentajes de ocupación y con un valor constante de 1000 cuadros por segundo. Valor que solamente influye en los algoritmos Sweep and Prune estáticos.

4. **Usar diseños experimentales eficientes y efectivos**. Se ha intentado desarrollar un algoritmo lo más eficiente posible basándose en las estructuras de datos y clases provenientes de la biblioteca estándar de Java. Respecto a todas las construcciones necesarias para la ejecución de los diferentes algoritmos se ha desarrollado de la manera más simple posible. Para mayor profundidad de las diferentes implementaciones, revisar el capítulo 3 y los anexos. Como se indicó en el ítem anterior, cada instancia de simulación generada de manera aleatoria es utilizada en los tres métodos a comparar, de acuerdo a lo recomendado en [22] como técnica de reducción de la varianza.

5. **Usar implementaciones razonablemente efectivas**: A lo largo del diseño y desarrollo de esta investigación este punto fue fundamental en poder desarrollar implementaciones que cumplieran con los límites teóricos. En el capítulo 5 se comparan dos versiones del algoritmo Sweep and Prune estático incremental: el primero implementa listas de colisiones parciales por coordenada, las que utilizan revisión exhaustiva par por par para obtener la lista de colisiones efectiva entre objetos; el segundo, en vez de usar listas utiliza la estructura *TreeSet* de la biblioteca Java que me permite elementos no repetidos en orden, lo que implica una aceleración en la obtención de la lista de pares de objetos cuyas cajas colisionan de manera efectiva.

6. **Asegurar reproductibilidad**: Para ello se graba en un archivo de texto las posiciones iniciales de todos los puntos de los polígonos, sus velocidades por cada eje, el tiempo de la simulación y el área inicial de la simulación. Dentro de los parámetros iniciales manejados por el usuario, se encuentran el porcentaje de ocupación del área total inicial, la cantidad de objetos en la simulación y la cantidad de cuadros por segundo.

7. **Asegurar comparabilidad**: se informa en los anexos de esta memoria el tipo de máquina usada, su sistema operativo, la cantidad de memoria RAM y la configuración de la máquina virtual de Java usada. Para cada conjunto de mediciones se calibró el profiler con la aplicación propia que posee Netbeans.

8. **Reportar toda la historia**: se presenta en las conclusiones el comportamiento del mejor y el peor caso para cada conjunto de parámetros. En los anexos se incluye una tabla con el detalle de los experimentos.

9. **Sacar conclusiones bien justificadas y buscar explicaciones**: como se describió anteriormente, el presente trabajo tiene por finalidad comparar el desempeño en tiempo de ejecución de tres algoritmos de fase amplia, con la finalidad de: corroborar la conclusión obtenida por [10] en un ambiente de pruebas, además de evaluar la factibilidad de que el algoritmo Sweep and Prune cinético sea una alternativa de uso factible a su versión estática incremental.

10. **Presentar los datos de manera informativa**: para ello se implementan las recomendaciones de [31] en tablas y gráficos.

1.1. Objetivo General

Realizar una comparación entre tres métodos de detección de colisiones de fase amplia: dos Sweep and Prune estáticos y un Sweep and Prune dinámico, con la finalidad de sugerir,

dada una distribución inicial de objetos en una imagen, qué método es el más conveniente en cuanto a tiempo de ejecución, mediante la medición inicial de un parámetro que mide la densidad de objetos en una imagen.

1.2. Objetivos Específicos

1. Establecer la necesidad de implementación de algoritmos de detección de colisiones en el procesamiento de imágenes para SCIAN-Lab.
2. Detallar las implementaciones realizadas para los algoritmos de Sweep and Prune estático y dinámico.
3. Calcular los tiempos de ejecución de los métodos Sweep and Prune estático y dinámico en diversos escenarios.
4. Observar si los tiempos de ejecución se encuentran acordes con el uso del método predicho por la densidad de objetos.
5. Corroborar el valor de la constante (muy alta) del algoritmo Sweep and Prune estático [35].

Se espera validar los resultados obtenidos por los experimentos de Coming y Staadt [11] para distintos valores de cantidad de objetos en escena y para distintas densidades. Además se evaluará el efecto de los eventos múltiples en la simulación, los que no están involucrados en [11].

1.3. Aporte

Se hizo una comparación de los algoritmos Sweep and Prune estático y Sweep and Prune cinético de manera similar a lo realizado por [11], en simulaciones en 2 dimensiones y con una mayor variación de los parámetros de área inicial ocupada y cantidad de objetos involucrados en la simulación, tomando en cuenta de manera explícita, las colisiones múltiples que no aparecen nombradas en [11] pero cuya detección cumple un rol fundamental debido a que la resolución incorrecta de este tipo de eventos puede provocar la destrucción irreparable del orden en las listas cinéticas, lo que es crucial para el correcto funcionamiento del método Sweep and Prune cinético [2].

La densidad de objetos por área inicial muestran una relación directa con el tiempo de ejecución del algoritmo, sin embargo no es suficiente para estimar qué algoritmo vale la

pena utilizar, ya que entran en juego otros factores. En el caso del algoritmo Sweep and Prune estático, juega un papel importante la frecuencia de muestreo, porque establece la cantidad de cuadros que se han de procesar, y por el lado del algoritmo Sweep and Prune cinético juega un rol importantísimo la densidad de objetos de la simulación, ya que puede generar muchas colisiones, y por la naturaleza de la construcción de este algoritmo, se encontrarán **todas**, lo que puede incidir en su desempeño desfavorablemente.

El algoritmo Sweep and Prune directo se implementó en C++ en forma de biblioteca dinámica además de la construcción del AABB Tree como Jerarquía de Volúmenes Envolventes para el caso de la detección de colisiones de detección de colisiones de fase angosta, y está disponible para su uso en el laboratorio SCIAN-Lab de la Facultad de Medicina de la Universidad de Chile.

1.4. Contenido de la tesis

El capítulo 2 presenta una visión general de los métodos de detección de colisiones y sus clasificaciones, los distintos métodos y sus limitaciones. Después de ello, se hace un recorrido por los métodos de detección de colisiones Sweep and Prune en sus versiones estático y cinético. Este último, haciendo énfasis en las estructuras de datos cinéticas, su filosofía, que atributos deben cumplir y su forma de creación, ya que es la pieza fundamental para el funcionamiento del algoritmo Sweep and Prune cinético. El capítulo 3 describe la metodología y las medidas de comparación utilizadas y por qué son importantes. En el capítulo 4 se muestra los modelos utilizados para validar cada uno de los algoritmos y el uso de aquellos necesarios para la medición de desempeño, el aporte del trabajo, mostrando los resultados. El capítulo 5 contiene la discusión de los resultados, las conclusiones del trabajo, y una lista de posibles temas para otras investigaciones cuyo punto de partida es la presente investigación.

Capítulo 2

Estado del arte

2.1. Antecedentes

La detección de colisiones es un tópico que lleva muchos años en computación, por lo que existe extensas y variadas revisiones generales del tema, así como también clasificaciones de los distintos métodos y enfoques [12, 17, 21, 24, 25, 35]. Los enfoques para la detección de colisiones y proximidad se pueden clasificar en diversos grupos. Por ejemplo, si se trata de objetos rígidos [21] o deformables, si la búsqueda se hace para descarte rápido de objetos (también llamada de detección de colisiones de fase amplia) o para refinar búsquedas de colisiones (llamada de detección de colisiones de fase angosta) [24]. Otros autores consideran clasificaciones como la naturaleza de los modelos de entrada (lineales versus curvados, convexos versus cóncavos, o de superficies versus volumétricos), la existencia de movimiento (estáticos versus dinámicos), el tipo de consulta de colisión (discreta o continua), y el tipo de arquitectura que utiliza la consulta (CPUs versus GPUs) [25].

De todas las clasificaciones anteriores, en esta tesis interesa abordar los métodos según si son estáticos o dinámicos (donde se verá una comparación de desempeño de ambos tipos de algoritmos en el capítulo 5) y según si son de detección de colisiones de fase amplia o de detección de colisiones de fase angosta (referido a un requerimiento de la facultad de medicina de la U. de Chile, para obtener el proceso de detección de colisiones completo, sin un análisis de por medio). La forma que tienen los métodos de detección de colisiones de fase amplia, -cuya finalidad es reducir al mínimo los posibles pares de objetos en colisión, los que serán posteriormente corroborados por un método de detección de colisiones fase angosta-, de reducir al mínimo la cantidad de comparaciones que deben hacerse entre objetos es usar algún tipo de estructura de datos que aproveche la coherencia espacial. Una vez obtenida la lista de los posibles pares de objetos que colisionan, se utiliza algún método de detección de colisiones de fase angosta, cuya finalidad es dados dos objetos que

posiblemente colisionan, encontrar si efectivamente lo hacen y entre qué elementos que componen a ambos objetos se produce esta colisión, para revisar con detalle qué primitivas de cada par de objetos efectivamente se intersectan. Los métodos estáticos y cinéticos indican si toman en cuenta el movimiento del objeto o no. Los métodos estáticos hacen el análisis en un cuadro de tiempo determinado, sin importar su historia, o a lo más se comparan con el cuadro anterior. En el caso de los métodos cinéticos, se pasa de un modelo basado en tiempo a uno basado en eventos, los que corresponden a algún cambio en la propiedad que se desea medir. En el caso de las colisiones es si algún objeto intersecta con otro o deja de hacerlo.

2.2. Modelos computacionales en microscopía

La experimentación en biología y medicina en la actualidad requiere mejores maneras de cuantificar sus teorías que la simple observación de fenómenos. Cada vez es más necesario poder medir variables físicas sobre organelos o células en los experimentos. Variables como la adhesión celular, estimaciones de fuerzas, velocidades, orientaciones, cálculo de superficies compartidas, y desplazamiento de grupos celulares a lo largo del tiempo, entre otras, fenómenos que ocurren naturalmente en 3D. La gran cantidad de objetos de interés y el tamaño de las imágenes adquiridas hacen indispensable el uso de la computación para la creación de modelos para mediciones y análisis de estas variables.

Si se toma como ejemplo la generación de un modelo computacional basado en estos grupos de imágenes (*stacks*), se obtendrá una serie de objetos volumétricos (células y/o organelos) en 3D, que se desplazan y cambian de forma a lo largo del tiempo. Si no existen buenas técnicas de generación y/o seguimiento de estos modelos, puede pasar que las fronteras de algunos objetos traspasen las de otros produciéndose traslapes que no existen en la realidad [8], la que puede inducir a mediciones incorrectas. Es ahí donde la detección de colisiones cumple un rol fundamental para actuar en consecuencia, en este caso, corregir las superficies de contacto entre los objetos, y si se requiere de una simulación físicamente realista, también debe considerarse que en cada contacto puede ser modificada la dirección y la velocidad del desplazamiento de los objetos involucrados en el contacto. Cómo las membranas de las células no son rígidas, incluirá una deformación, modificando el contacto entre los objetos. Este proceso es continuo a lo largo del tiempo. Si se considera además que algunos de estos experimentos en la actualidad involucran *stacks* de imágenes de alta resolución con gran cantidad de objetos y con muchas secuencias temporales [20, 23, 37], el análisis se hace significativamente costoso sin las herramientas apropiadas.

2.3. Detección de colisiones estática

Dentro de los enfoques estáticos para objetos rígidos se puede encontrar los métodos de subdivisión espacial y los de *bvhs*. Los primeros consisten en una representación jerárquica del espacio y los segundos corresponden a una representación jerárquica del objeto (ambos usualmente como una representación de árbol binario). Como ejemplos de subdivisión espacial existen:

- Octree: es una estructura de datos espacial que divide el espacio en ocho octantes, lográndose una división por dos de cada dimensión (del mismo tamaño).
- Kd-tree: es una estructura de datos de partición espacial que organiza puntos en un espacio k-dimensional. Corresponde a un caso especial de los árboles de partición binaria del espacio.
- BSP Tree: es una estructura de datos de partición espacial que subdivide al espacio de manera recursiva en dos, hasta algún criterio de parada para formar las hojas. La partición binaria del espacio de un BSPTree puede ser mediante un hiperplano en cualquier dirección, a diferencia de los árboles octree o Kd-tree, cuya división está dada por un hiperplano alineado con los ejes del sistema Cartesiano.
- Grillas uniformes.

Como ejemplos de volúmenes envolventes (*Bounding Volumes*) existen:

- Oriented Bounding Boxes (OBBs): llamado también como la caja de volúmen mínimo que envuelve a un objeto. En general, los ejes de esa hipercaja pueden tener cualquier orientación.
- Axes Aligned Bounding Boxes (AABBs): cajas orientadas con los ejes coordenados cartesianos. Los AABB son subconjuntos de los OBB.
- sphere-trees: Una forma particular de la *bvh*, donde cada nodo representa a una hiper-esfera.
- k-Discrete Oriented Polytopes (k-DOPs): generaliza a un volúmen envolvente. Un K-DOP es la intersección Booleana a lo largo de las extensiones de las k-direcciones. Por lo tanto, un K-DOP es la intersección Booleana de los k planos límites y es un politopo convexo que contiene al objeto (en 2-D un polígono; en 3-D a un poliedro). Un rectángulo en 2-D es un caso especial de 2-DOP, y una caja 3-D es un caso especial de un 3-DOP. En general, los ejes de un DOP no tienen que ser ortogonales,

y puede haber más ejes que dimensiones del espacio. Por ejemplo, una caja 3-D que está biselada en todos sus bordes y esquinas puede ser construída como un 13-DOP.

De acuerdo a [24], con el enfoque de subdivisión espacial es inevitable la división de polígonos o poliedros, lo que causa un incremento en la profundidad del árbol y pérdida de desempeño. Además, como no se ajustan bien para cubrir las primitivas de los objetos, cuando éstos están muy cerca es difícil determinar el estado del contacto. Por ello el enfoque de Volumen Envolvente (bv) provee jerarquías más ajustadas y pequeñas que el particionamiento espacial. Además, las *OBB* son más aplicables para detección de colisiones a formas generales (es decir, que no importa si son convexas o no convexas) que los algoritmos basados en símplices o en características. Chang [7] concluye que un enfoque de *bvh* es mejor en comparación a uno de partición del espacio debido principalmente al costo en memoria que tienen las segundas estructuras, porque puede haber muchas celdas que con punteros referencian a un mismo objeto (un ejemplo de ello son los *octree*).

Dentro de todos los tipos de Volúmenes Envolventes (bvs), Gottschalk [18] establece que los OBBs y elipsoides se ajustan mejor a una primitiva que esferas y *AABBs*, pero el costo de cálculo de colisiones es mayor. Chang [7] indica que la construcción de OBBs aproximadas no es costosa usando métodos heurísticos, pero si lo es para OBBs óptimos, por lo que sigue siendo materia de investigación.

2.3.1. Método Sweep and Prune estático

En un escenario con gran cantidad de objetos, si se ejecuta una revisión exhaustiva de todos los objetos contra el resto para detectar colisiones, es una operación de orden $O(n^2)$, con n el número de objetos de la escena. El uso de *bvs* para objetos no es suficiente para simplificar la búsqueda de posibles colisiones, por lo que debe aplicarse en conjunto con otros algoritmos, como por ejemplo, *Sweep and Prune*, también llamado **Sort and Sweep** [12].

Un objeto en 2D está compuesto de: una lista de vértices sin un orden preestablecido, representados como una lista de puntos con coordenadas (x, y) que corresponden a la posición inicial del objeto en la simulación; y un par de valores que representan la magnitud y dirección de la velocidad de desplazamiento rectilíneo del objeto a lo largo del tiempo.

Sweep and Prune es un algoritmo de detección de colisiones de fase amplia cuya finalidad es reducir el número de posibles objetos en colisión, de manera de procesarlos posteriormente con algún método de fase angosta. Este algoritmo es de complejidad $O(n + o)$ [34], con n=total de *AABBs* y o=los pares de *AABBs* que se traslapan. Este método sirve para 2D y 3D. El resultado es una lista de pares de objetos que posiblemente colisionan. Consiste en calcular los extremos del AABB de cada objeto, es decir, su valor mínimo y máximo por coordenada. Estos valores se encuentran ordenados en una lista

por coordenada, para detectar si existen posibles colisiones entre volúmenes envolventes, mecanismo que se realiza para cada dimensión.

Existen dos tipos de Sweep and Prune estático: el directo, que entrega la lista completa de volúmenes envolventes que colisionan en un frame, y el incremental, que muestra la lista de pares que han sido agregados y/o borrados a partir del resultado de un frame anterior. Sweep and Prune estático establece los siguientes pasos:

i Calcular el volúmen envolvente de cada objeto (en el caso de esta tesis se usará AABB 2D, siendo posible extenderse a 3D). Este paso es mostrado en el algoritmo 1, donde se obtiene como resultado el par de valores $\{\text{mín}, \text{máx}\}$ en el eje coordenado seleccionado.

Algoritmo 1: Obtener los valores máximo y mínimo en una coordenada de un polígono

Data: $poly_i$: lista de puntos por objeto i
Data: dim: dimensión, x, y
Result: par: par de valores (mínimo y máximo)
Function *obtenerMinimoMaximo*$(polygon, dim)$

```
/* min y max obtienen el valor mínimo y máximo en la
   dimensión dim de la lista de todos los puntos del
   polígono                                          */
```

$p \leftarrow \text{mín}\,(poly_i, dim)$;
$q \leftarrow \text{máx}\,(poly_i, dim)$;
return $par \leftarrow$ *new Pair* (p, q);

ii Extraer de cada volúmen envolvente el mínimo y el máximo, por cada eje, y son agregados a una lista de proyecciones, donde existe una lista por cada eje. Se procesan estas listas de proyecciones, obteniendo una lista de pares de colisiones por eje. Este paso es mostrado en el algoritmo 2.

Algoritmo 2: Llenado de listas de colisiones por dimensión.

Data: $listObjects \leftarrow poly_1, poly_2, \ldots poly_n$: lista de objetos
Result: $listaPares_i$: lista de pares de objetos que posiblemente colisionan, por cada eje coordenado i
Procedure *obtenerListaColisionesPorDimension*$(listObjects, dim)$

```
/* Inicializar listas de intervalos por eje                    */
listaPares ← new ArrayList<Pair>();
listaIntervalos ← new ArrayList<Elements>();
/* Llenado de listas de Elementos. Los objetos de la clase
   Element son simples contenedores con el id del objeto, el
   valor de la proyección en la dimensión dim y un valor
   booleano que indica si es el extremo mayor del intervalo.
   */
foreach i ∈ listObjects do
    parValores_i ← obtenerMinimoMaximo(poly_i, dim);
    /* Algoritmo 1                                             */
    b ← parValores_{i,dim}.first;
    e ← parValores_{i,dim}.second;
    listaIntervalos.insertionSort(new Element(i, b, false));
    listaIntervalos.insertionSort(new Element(i, e, true));
listaIndices ← {};
foreach elemento ∈ listaIntervalos_dim do
    if elemento is extremo inferior then
        listaIndices.add(elemento.getId());
    else
        indice ← listaIndices.remove(elemento.getId());
        /* generarPares retorna todas las listas de pares con
           valor (indice, otro), donde otro es cada elemento
           restante en la lista de índices. Si el par o su par
           invertido existe en la lista de pares no se agrega.
           */
        listaPares.add(generarPares(indice, listaIndices_dim));
return listaPares
```

iii De las listas, por cada eje coordenado se extrae la que posee menos elementos, y es comparada con el resto. Si un par existe en todas las listas es agregado en una lista única de salida, (ver algoritmo 3).

Algoritmo 3: Sweep and Prune estático.

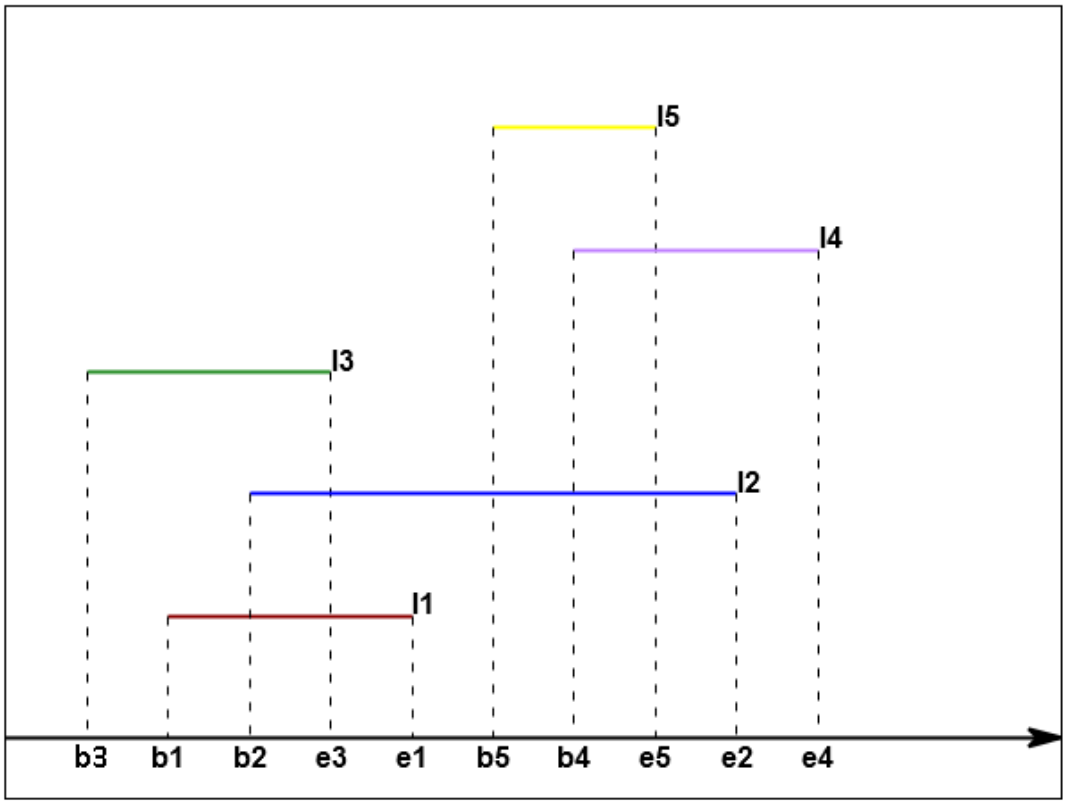
Data: $listObjects \leftarrow poly_1, poly_2, \ldots poly_n$: lista de objetos
Result: $listaPares$: lista de pares de objetos que posiblemente colisionan
Function *SweepAndPruneEstatico*($listObjects$)

```
/* Inicializar listas de intervalos por cada eje y almacenar
   en una lista de listas de colisiones                    */
Inicializar listaDeListasDeColisiones;
foreach dim ∈ {x, y} do
    /* Algoritmo 2                                         */
    listaColisionesDimension =
      obtenerListaColisionesPorDimension(listObjects, dim);
    listaDelistasDeColisiones.add(listaColisionesDimension);
/* obtener lista con menos elementos y extraerla de la lista
   de listas                                               */
listaMasPequena ←
  listaDelistasDeColisiones.extraerListaMenosElementos();
foreach (par ∈ listaMasPequena) do
    if par exists in listaDelistasDeColisiones then
        listaPares.add(par);
return listaPares
```

En la figura 2.1 se ejemplifica una instancia del método Sweep and Prune directo acotado a segmentos en una dimensión, en un instante de tiempo cualquiera de una simulación. Esta figura posee los intervalos que están definidos por I_j con $j = \{1, 2, 3, 4, 5\}$. Cada uno de estos intervalos posee b_j y e_j que corresponden a las proyecciones de inicio y final de estos intervalos respectivamente.

El ejemplo, al ser una instancia unidimensional, sólo necesita ejecutar el algoritmo 2 para obtener la lista de posibles colisiones, que deben ser corroboradas mediante la ejecución de un algoritmo de detección de colisiones de fase angosta.

Figura 2.1: Ejemplo de una instancia del algoritmo Sweep and Prune directo. Cada intervalo I_j tiene su proyección en el eje X, llamados b_j y e_j.



Observando la figura 2.1 y de acuerdo al algoritmo 2 tenemos:

1. Los extremos de cada intervalo se ingresan a la lista de intervalos mediante insertionSort.

   - $listaIntervalos = \emptyset$ al iniciar el ingreso.
   - $listaIntervalos = \{b_1, e_1\}$ al ingresar el intervalo I_1.
   - $listaIntervalos = \{b_1, b_2, e_1, e_2\}$ al ingresar I_2.
   - $listaIntervalos = \{b_3, b_1, b_2, e_3, e_1, e_2\}$ al ingresar I_3.
   - $listaIntervalos = \{b_3, b_1, b_2, e_3, e_1, b_4, e_2, e_4\}$ al ingresar I_4.
   - $listaIntervalos = \{b_3, b_1, b_2, e_3, e_1, b_5, b_4, e_5, e_2, e_4\}$ al ingresar I_5.

2. Se inicializa la lista de índices que me dirá los pares de objetos cuyos intervalos están en colisión. $listaIndices = \emptyset$.

3. Por cada elemento de la lista de intervalos se revisa si es un extremo inferior o superior. Si es un tremo inferior del intervalo (b_j), se agrega el índice a la lista de índices. Si el elemento es un extremo superior de un intervalo (e_j), se elimina el índice j de la lista de índices y se genera una serie de pares con la combinación de $(j, \{\text{listaIndices que quedan}\})$, los que son agregados a la lista de pares que colisionan.

   - b_3 es elemento de inicio de un intervalo, por lo que es agregado a $listaIndices = \{3\}$.
   - b_1 es agregado a $listaIndices = \{1, 3\}$.
   - b_2 es agregado a $listaIndices = \{1, 2, 3\}$.
   - e_3. Se elimina 3 de $listaIndices$, quedando $listaIndices = \{1, 2\}$. Se genera las combinaciones de pares entre $(3, \{1, 2\})$, lo que me da la lista de pares $listaPares = \{(1, 3), (2, 3)\}$.
   - e_1. Se elimina 1 de $listaIndices$, quedando $listaIndices = \{2\}$. Se genera las combinaciones de pares entre $(1, \{2\})$, lo que me da la lista de pares $listaPares = \{(1, 2), (1, 3), (2, 3)\}$.
   - b_5 es agregado a $listaIndices = \{2, 5\}$.
   - b_4 es agregado a $listaIndices = \{2, 4, 5\}$.
   - e_5. Se elimina 5 de $listaIndices$, quedando $listaIndices = \{2, 4\}$. Se genera las combinaciones de pares entre $(5, \{2, 4\})$, lo que me da la lista de pares $listaPares = \{(1, 2), (1, 3), (2, 3), (2, 5), (4, 5)\}$.
   - e_2. Se elimina 2 de $listaIndices$, quedando $listaIndices = \{4\}$. Se genera las combinaciones de pares entre $(2, \{4\})$, lo que me da la lista de pares $listaPares = \{(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (4, 5)\}$.
   - e_4. Se elimina 4 de $listaIndices$, quedando $listaIndices = \emptyset$.

4. Se retorna la lista de pares obtenida, es decir: $listaPares = \{(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (4, 5)\}$.

Para casos de dos o más dimensiones el proceso se realiza de manera análoga, obteniéndose una lista de pares por cada dimensión. Esta lista de pares es consolidada mediante el algoritmo 3, obteniendo la lista final de objetos cuyas AABB colisionan. Para que un par de objetos tengan una colisión efectiva, el par debe aparecer colisionando en todos los ejes.

2.3.2. Análisis asintótico del algoritmo Sweep and Prune estático

La complejidad del algoritmo Sweep and Prune directo está dada por [6]:

$$O\left(n+k\right)\log n, \tag{2.1}$$

donde:

$$\begin{aligned} n &= \text{cantidad de objetos} \\ k &= \text{cantidad de pares que se traslapan} \end{aligned}$$

La ecuación 2.1 se demuestra dado que para ordenar, en una imagen o cuadro, n cajas de acuerdo a su mínima coordenada requiere $O\left(n\log n\right)$, realizar el barrido de las coordenadas de inicio y fin de los objetos requiere $O\left(n\right)$ y la creación de los k pares de salida requieren en total $O\left(k\right)$.

En el caso de usar este algoritmo en una serie (o stack) de imágenes, es óptimo para el caso inicial, es decir, el primer cuadro. Si se hace uso de la coherencia espacial de las imágenes (pensando que en cada cuadro la situación no cambia demasiado), se puede demostrar que el algoritmo se ejecuta en tiempo $O\left(n+k\right)$ [4].

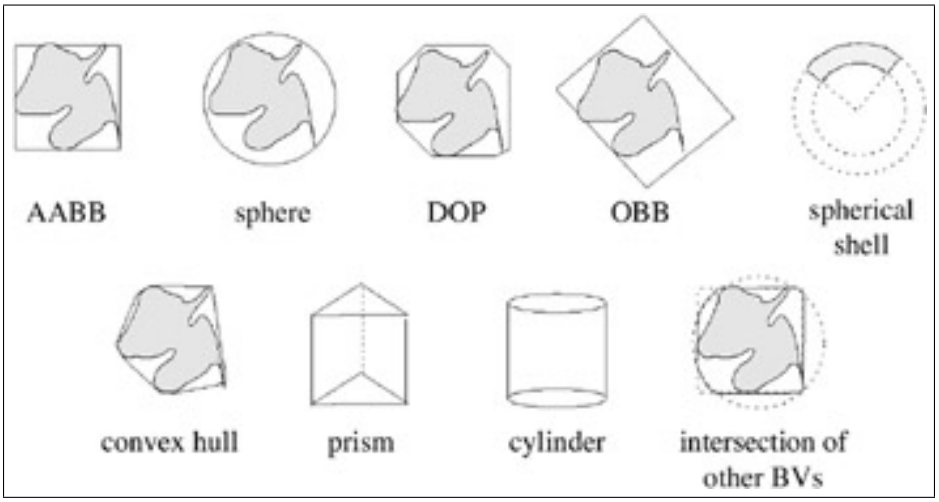
En el caso de esta tesis, se prueba la implementación del algoritmo Sweep and Prune directo, es decir, todos los frames se calculan como si fuera el primero, lo que nos da que el tiempo de ejecución se ejecuta en $O\left(m\left(n+k\right)\right)$ con m los cuadros por segundo de la simulación; y el algoritmo Sweep and Prune incremental, en el que las construcciones se hacen en el primer frame y luego se va actualizando la información en la medida que se va alterando el orden de los AABB de los objetos.

2.3.3. Volúmenes envolventes de un objeto y Jerarquía de Volúmenes Envolventes

Una forma de simplificar los cálculos de detección de colisiones, tanto de detección de colisiones de fase amplia como de detección de colisiones de fase angosta es utilizar los volúmenes envolventes, que son una geometría más simple para representar a un objeto. Dentro de los *bvs* más usados se tienen, en orden de creciente de complejidad y cantidad de parámetros que los definen: esferas, *AABBs*, *OBB*, *k-DOPs* y cerraduras convexas (*convex-hulls*).

Para el caso de la detección de colisiones de fase amplia, se reemplaza el objeto original por su volumen envolvente, mientras que para la detección de colisiones de fase angosta, lo que se reemplaza son las primitivas por estos volúmenes, los cuales se organizan de manera jerárquica, creando las *bvhs*. Estas jerarquías se organizan en una estructura

Figura 2.2: Ejemplos de volúmenes envolventes. Imagen extraída de [33]



de árbol binario.

Para el modelo de esferas existen variadas formas de crear esta jerarquía. Por ejemplo, mediante el uso de *octrees*, donde en cada subdivisión del espacio se puede colocar una esfera. Un método particularmente interesante es el de Welzl [36] debido a que es un método aleatorio, incremental y con tiempo esperado lineal que no requiere ninguna estructura adicional y sólo necesita los vértices de los objetos para la construcción de la mínima esfera (en el caso 2D, la circunferencia de tamaño mínimo). Para el caso de las *AABB* se considera las proyecciones del punto mínimo y el punto máximo en cada eje coordenado para la construcción de la caja.

Algunas aplicaciones que ya utilizan de alguna manera las *AABBs* en la detección de colisiones son [13, 14, 28, 30].

2.4. Detección de colisiones cinéticas

Cuando se emplea detección de colisiones estática para analizar un escenario no estático, como una simulación o un experimento que consta de stacks de imágenes a lo largo del tiempo, el análisis es sensible al parámetro **tiempo de muestreo**. Un valor muy pequeño del tiempo de muestreo puede resultar en un sobremuestreo de la aplicación, con lo que se desperdician recursos de tiempo y memoria en revisiones innecesarias; por el contrario, si se usa un tiempo de muestreo demasiado grande es posible que se pierdan eventos importantes debido al submuestreo. Por otro lado, en los métodos dinámicos de detección de colisiones, las jerarquías estáticas de volúmenes envolventes no sacan parti-

do de la coherencia espacial de los objetos, por lo que cualquier jerarquía es invalidada al tiempo siguiente y debe ser reconstruída por completo [38].

Con las anteriores desventajas en mente, Basch [5] ideó un nuevo enfoque que no está basado en el muestreo de la aplicación, sino en los **eventos** que ocurren en ella, enfoque llamado estructuras de datos cinéticas.

2.4.1. Estructuras de Datos Cinéticas

Las estructuras de datos cinéticas fueron introducidas por Basch *et al.* [5] como una forma elegante y ventajosa de evitar a la recreación de jerarquías (espaciales o de objetos) en cada paso temporal, al introducir un enfoque basado en eventos. La *Estructura de datos Cinética (KDS)* establece que los objetos no están estáticos sino que en *movimiento de tiempo continuo*, es decir, no se detienen y los objetos no saltan de una parte del espacio a otra. Las estructuras de datos cinéticas son una clase de técnicas algorítmicas que sacan partido de la *coherencia espacial* y *temporal* a las estructuras geométricas, en contrapartida a las alternativas estáticas. Una *KDS* está compuesta de dos partes:

i) Un *atributo*, (llamado también función de configuración del sistema), que corresponde a una propiedad geométrica que se quiere observar o medir, y que cambia una cantidad discreta de veces, dentro de las cuales están por ejemplo: mantener una lista ordenada, un heap, la cerradura convexa [3], el par más cercano y los vecinos más cercanos [29], el *minimum spanning tree*, el orden de los polígonos dentro del algoritmo *Sweep and Prune* [10], entre otros.

ii) Una serie de certificados, que verifican la validez del atributo en el momento actual, los cuales son probados en cada evento.

Las *KDS* asumen que se conoce de antemano la trayectoria temporal de las primitivas (representada por lo general como un polinomio en función del tiempo) y cuyo movimiento es continuo, de manera que es posible calcular el instante de falla para cada certificado.

Cada vez que falla un certificado -proceso llamado *evento*- la *KDS* debe ser actualizada. Así, la *KDS* permanece válida hasta el próximo evento. El objetivo cuando se diseña una *KDS* es asegurar que no hay demasiados eventos y que el tiempo de actualización es pequeño. En cada evento se revisan los *certificados* del atributo en cuestión. Cuando éstos fallan se produce una recreación de la estructura cinética y de la cola de eventos, eliminando los futuros eventos asociados al objeto que quedaban en la cola y siendo reemplazados por unos nuevos (producto de la falla del certificado). Los enfoques de *KDS* no consideran como una limitante a la rigidez o deformación del objeto, sin embargo, la principal dificultad se encuentra en generar un certificado para el atributo que se debe mantener en

la estructura cinética. Guibas [6, 19] estableció que para evaluar las *KDS* se debe cumplir cuatro criterios:

C1 **Sensibilidad**: se refiere a la cantidad de tiempo en el peor caso para actualizar la prueba de un conjunto de certificados después de una falla de alguno de ellos. Esto puede requerir el descubrimiento del nuevo valor del atributo, así como la actualización del conjunto de certificados, removiendo los que ya no son válidos en el tiempo actual y agregando aquellos que son parte de la nueva prueba. Se dice que una *KDS* es *sensible* cuando este tiempo de actualización de la prueba es pequeño.

C2 **Eficiencia**: es el número de eventos procesados en el peor caso. La meta es tener un número total de eventos procesados por la estructura en el peor caso que sea asintóticamente del mismo orden, o ligeramente superior, al número de eventos externos (aquellos eventos que cambian la prueba) en el peor caso.

C3 **Localidad**: es el máximo número de certificados en que un objeto cualquiera puede aparecer. Si este número es pequeño se dice que la *KDS* es *local*.

C4 **Compactibilidad**: el tamaño de una *KDS* es el máximo número de certificados que pueden presentarse en una prueba, también refleja el tamaño de la cola de eventos que necesita ser mantenida. Se dice que una *KDS* es *compacta* si el máximo número de certificados presente alguna vez en una prueba es cercano a $O(n)$.

En la siguiente sección se analizará como ejemplo de una estructura de datos cinética a la lista ordenada cinética, la que además tiene importancia porque varios algoritmos cinéticos se basan en ella, incluído el algoritmo Sweep and Prune cinético.

2.4.2. La lista ordenada cinética

La lista ordenada cinética consiste en una lista de objetos que deben estar ordenados en todo momento, por ejemplo, puntos en un espacio n-dimensional que se desplazan a lo largo del tiempo. Supondremos por simplicidad que estos puntos son valores unidimensionales que se desplazan mediante una función polinomial de primer orden, es decir, movimiento rectilíneo. Dada esta lista ordenada, se calculan los eventos iniciales, los que son obtenidos resolviendo el tiempo de cruce entre vecinos. Estos eventos iniciales son guardados en una cola de eventos, los cuales serán procesados en orden temporal, de menor a mayor. La cola de eventos puede ser implementada con una cola de prioridad, un two-pass pairing heap [11] o un árbol AVL [38] cuya clave es el tiempo del evento. Luego, se procesa cada evento, hasta que la lista esté vacía. Los pasos anteriormente descritos se pueden observar en el algoritmo 4. Se puede demostrar que la lista ordenada es [3]:

- **sensible**: una falla en un certificado, causa un swap (el que toma tiempo $O(1)$) y $O(1)$ cambios en los certificados, los que toman tiempo $O(\log n)$ en replanificarse.
- **local**: cada elemento está involucrado en a lo más dos certificados.
- **compacta**: existen exactamente $n-1$ certificados para una lista de n elementos.
- **eficiente**: esta estructura no causa eventos internos indeseados, ya que cada cambio en el orden de los elementos causa exactamente la falla de un certificado.

Algoritmo 4: mantención de lista cinética

Data: $p_0(t), p_1(t), \ldots p_{n-1}(t)$: lista de puntos cinéticos, donde $p_i(t) = a_i * t + b_i$
Data: $t \in [t_{min}, t_{max}]$
Procedure *mantenerKineticSortedList(*$\{p_0(t), p_1(t), \ldots p_{n-1}(t)\}$*, t)*

```
/* ingreso puntos cinéticos                                          */
ksl ← {p_0(t), p_1(t), ... p_{n-1}(t)};
/* ordeno la lista en t = t_min                                      */
ksl.sort(t_min);
/* calculo lista de eventos inicial                                  */
colaEventos ← calcularListaEventosInicial (ksl);
/* proceso la lista de eventos                                       */
while colaEventos ≠ ∅ do
    /* extraigo el evento con menor tiempo de la cola                */
    evento ← colaEventos.pop ();
    if evento is simple then
        eliminar futuros eventos que involucran al par de elementos de la lista;
        realizar el swap de elementos involucrados en la ksl;
        calcular nuevos eventos con los vecinos de los elementos;
    else
        /* buscar lista de índices involucrados en la colisión
           múltiple                                                  */
        listaIndicesParaEliminar ←
          obtenerListaColisionMultiple (t_evento);
        /* eliminar de la cola de eventos todos los eventos
           que involucren los índices encontrados                    */
        colaEventos.eliminarElementos (listaIndicesParaEliminar);
        /* invalidar la ksl, ordenarla nuevamente para t + Δt        */
        ksl.sort (t + Δt);
        /* buscar nuevos eventos con los vecinos de los
           extremos del rango, i y j son posiciones dentro de
           la ksl                                                    */
        listaEventosNuevos ← obtenerEventosExtremos (i, j);
        /* agregar eventos nuevos a la cola de eventos               */
        colaEventos.addAll (listaEventosNuevos);
```

2.4.3. El caso de eventos múltiples

Un punto importante que evita Basch [6] es el caso de eventos múltiples dentro de las simulaciones, tema que aborda Abam *et al* [2], donde el incorrecto orden del procesamiento de los eventos puede dejar al algoritmo en un ciclo infinito e invalidar irremediablemente el atributo. Un modo de resolver la situación es:

- Determinar los índices de la lista ordenada cinética involucrados en el evento múltiple.
- Eliminar de la cola de eventos los eventos que corresponden a este evento múltiple. Proceso que puede tomar $O(n)$.
- Resolver el trozo de la lista cinética invalidado por este evento múltiple, para un tiempo $t = t_{evento} + \epsilon$.

Una vez realizado el procedimiento anterior, se vuelve a obtener los eventos con los nuevos vecinos y se agregan a la lista de eventos de la misma forma que un evento simple.

2.4.4. Ejemplo de una lista ordenada cinética

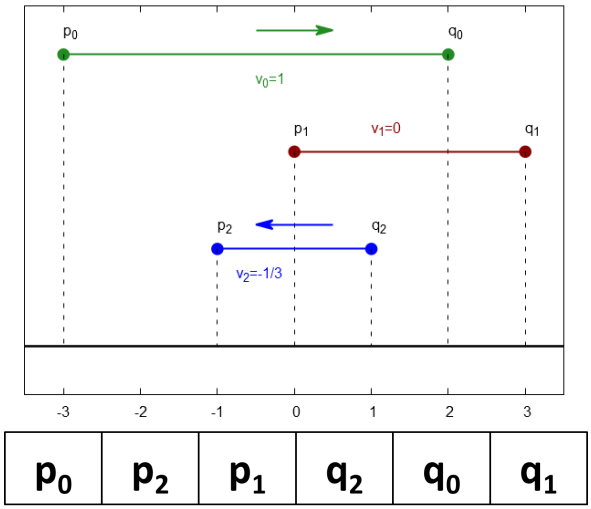
Se asume el ejemplo en 1D para mostrar el mecanismo de la lista ordenada cinética. Sea $EV_{(e_0,e_1)dim}\,(t = t_n)$ un evento que involucra los elementos en las posiciones e_0 y e_1 de la lista cinética en la dimensión dim, en el tiempo $t = t_n$. Dados $p_i = v_i t + p_{0i}$ y $q_i = v_i t + q_{0i}$, puntos móviles unidimensionales, donde p_i y q_i corresponden a extremos de un *AABB*, y sea una lista ksl ordenada en todo momento, la que se cumpla para todo tiempo t: $e_i(t) < e_{i+1}(t), \forall i = 0 \ldots n-1$.

- **atributo**: mantener la lista ordenada ascendente.
- **certificado**: la propia lista ordenada, probando que $e_i(t) < e_{i+1}(t) \forall t$.

Supóngase que los puntos cinéticos son los siguientes: $p_0 = t - 3$, $q_0 = t + 2$, $p_1 = 0$, $q_1 = 3$, $p_2 = -\frac{1}{3}t - 1$ y $q_2 = -\frac{1}{3}t + 1$, estos están acompañados de la lista cinética ordenada inicial, los que se pueden ver en la figura 2.3. El tiempo de simulación está definido con $t = [0 \ldots 10]$.

La lista de eventos inicial se obtiene calculando los tiempos de intersección entre puntos adyacentes, que es la siguiente:

Figura 2.3: Condición inicial de la lista cinética



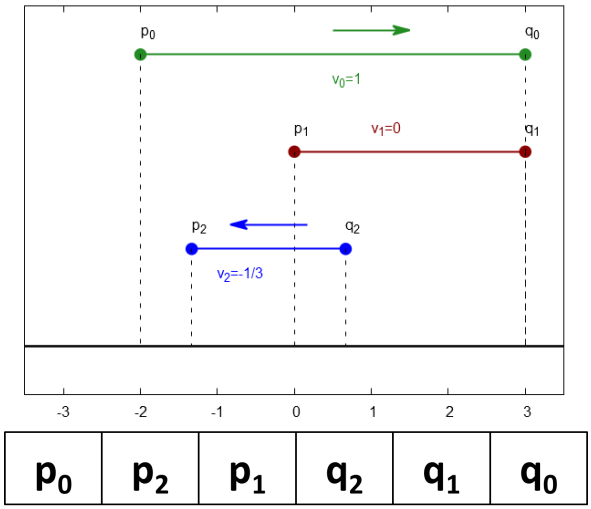
$$
\begin{aligned}
p_0 \cap p_2 &: t - 3 = -\frac{1}{3} - 1 \rightarrow t = \frac{3}{2} \\
p_1 \cap q_2 &: 0 = -\frac{t}{3} + 1 \rightarrow t = 3 \\
q_0 \cap q_1 &: t + 2 = 3 \rightarrow t = 1
\end{aligned}
\tag{2.2}
$$

Por lo tanto, se obtiene los eventos (revisar la figura 2.3):

- $EV_{(4,5)X}\,(t = 1)$, es decir, el evento ocurre entre q_0 y q_1 en $t = 1$.
- $EV_{(0,1)X}\left(t = \frac{3}{2}\right)$, es decir, el evento ocurre entre p_0 y p_2 en $t = \frac{3}{2}$.
- $EV_{(2,3)X}\,(t = 3)$, es decir, el evento ocurre entre p_1 y q_2 en $t = 3$.

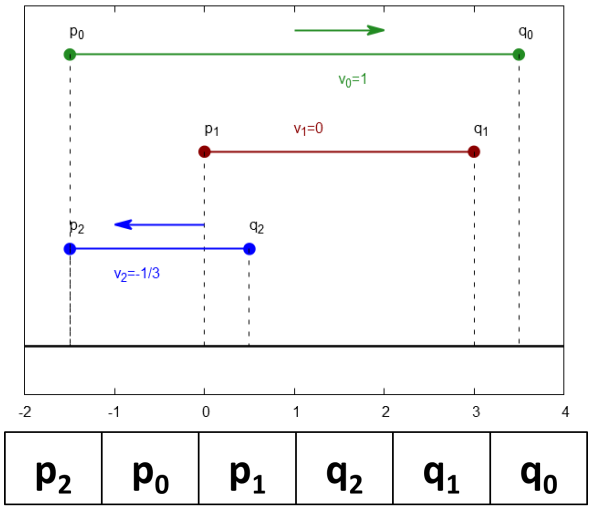
El resto de las ecuaciones no dan valores con tiempos válidos para la simulación, es decir, no se encuentran entre el tiempo mínimo y máximo definidos. En el caso del primer evento, mostrado por la figura 2.4, se procesa el evento $EV_{(4,5)X}\,(t = 1)$, donde $(4, 5)$ son los índices de la lista cinética involucrados.

Figura 2.4: Condición después del primer evento, en t=1



Después de la ejecución del swap (o intercambio) dado por el evento $EV_{(4,5)X}\ (t = 1)$, se tiene la situación de la figura 2.4. En este instante, se deben calcular los nuevos eventos, si existen, con los nuevos vecinos. En este caso, se debe revisar $q_2 \cap q_1$, lo que no da un evento válido.
Para el segundo evento, $EV_{(0,1)X}\ \left(t = \frac{3}{2}\right)$, mostrado por la figura 2.5:

Figura 2.5: Condición después del segundo evento, en t=1.5



Se calcula los nuevos eventos, si existen, con los nuevos vecinos. En este caso, se debe revisar $p_0 \cap p_1$, lo que genera el evento $EV_{(1,2)X}\,(t=3)$. Revisando en $t=3$, los eventos $EV_{(1,2)X}\,(t=3)$ y $EV_{(2,3)X}\,(t=3)$ corresponden a un evento múltiple (figura 2.6), llamado así porque en un tiempo y en una posición determinada común confluyen más de dos puntos de un evento simple. De acuerdo al algoritmo 4:

i) obtener la lista de índices involucrados en el evento múltiple: $listaIndicesParaEliminar \leftarrow \{1,2,3\}$.

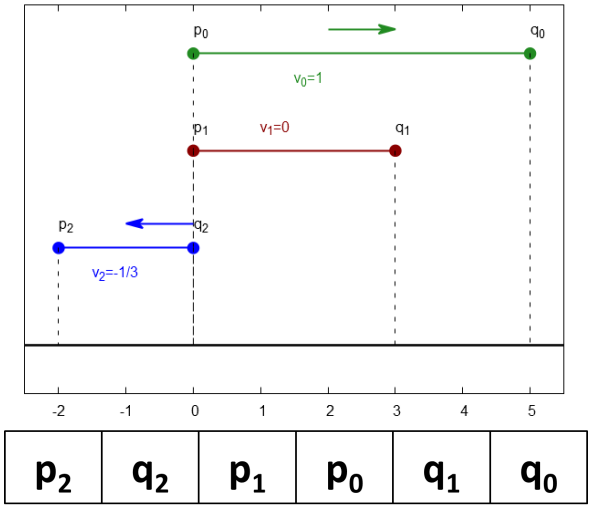
ii) eliminar los eventos que tienen esos índices de la cola de eventos, la que, en este caso, queda vacía.

iii) invalidar la zona de la lista cinética donde está involucrada la colisión múltiple. Y es resuelta para $t = 3+\epsilon$, donde ϵ corresponde a un valor positivo muy pequño.

$$
\begin{aligned}
p_0\,(3+\epsilon) &= 3+\epsilon-3=\epsilon \\
p_1\,(3+\epsilon) &= 0 \\
q_2\,(3+\epsilon) &= -\frac{1}{3}\,(3+\epsilon)+1=-\frac{\epsilon}{3}
\end{aligned}
$$

Lo que define $q_2 < p_1 < p_0$ para $t > 3$.

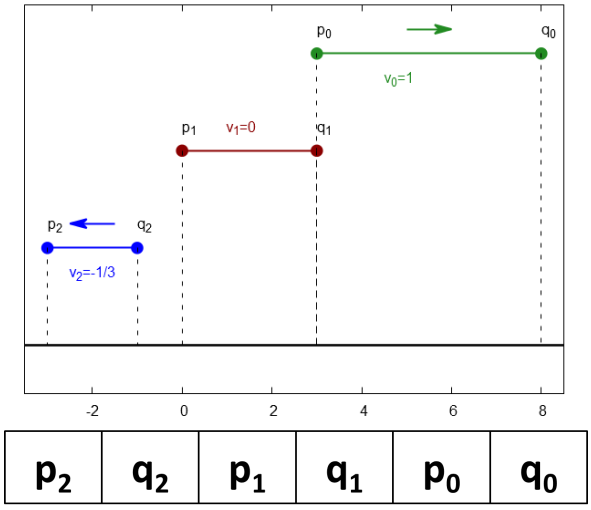
Figura 2.6: Condición de la lista resuelta para el evento múltiple en $t = 3+\epsilon$



iv) buscar nuevos eventos con los vecinos de los extremos del rango, es decir, los índices 1 y 3 de la lista cinética, lo que implica resolver, $p_2 \cap q_2$ y $p_0 \cap q_1$. En el primer caso, no se obtienen eventos ya que ambos elementos tienen la misma velocidad en el mismo sentido; en el segundo caso, se obtiene el evento $EV_{(3,4)}\,(t = 6)$.

El último evento está representado por la figura 2.7, Y como no quedan más eventos que realizar, se detiene la simulación:

Figura 2.7: Condición después del último evento, para $t > 6$



Para el caso de múltiples dimensiones (2D o 3D), el mecanismo es el siguiente:

- Existe una lista cinética por coordenada.
- Existe una lista de colisiones por coordenada.
- La lista de eventos tendrá eventos que ocurren por coordenada. Estos eventos pueden ser ordenados, o bien, en una única cola de prioridad en base al tiempo de falla de la lista cinética correspondiente o, tener una cola de prioridad por cada coordenada. En este segundo caso, se debe revisar de todas las colas de prioridad, cuál evento sucede primero.
- La detección y reparación de la cola de prioridad, dependiendo si un evento es simple o múltiple se realiza de la manera usual para 1D y debe hacerse en la coordenada relativa al evento a ejecutar.

- Una vez ejecutado el evento, se debe consolidar la lista de colisiones que efectivamente suceden. Para ello, el par de índices de objetos que colisionan debe existir en todas las listas de colisiones. Para este filtrado, se elige la lista que contenga menor cantidad de colisiones. Se debe hacer notar que este proceso puede hacer al algoritmo extremadamente ineficiente.

2.4.5. Sweep and Prune Cinético

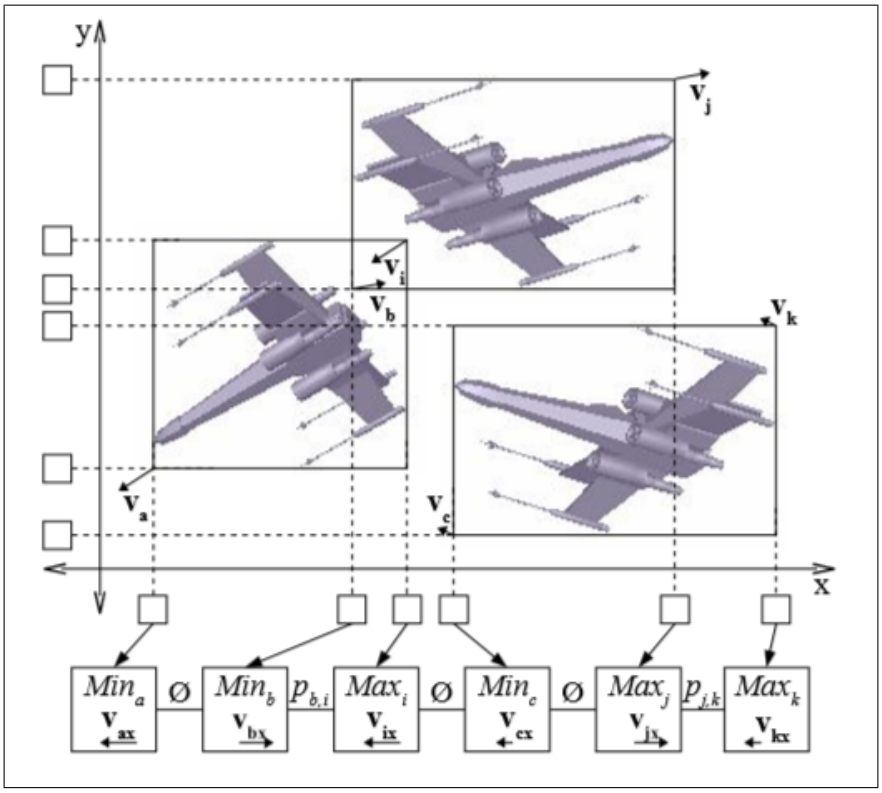
El algoritmo de Sweep and Prune cinético de Coming y Staadt [11] consiste en mantener ordenadas listas cinéticas -una por cada eje coordenado- de valores asociados a los puntos extremos del *AABBs* de cada objeto a lo largo de la simulación, como lo muestra la figura 2.8. Cada punto posee una trayectoria polinomial (de acuerdo a la teoría de KDS [19]), pero en el artículo de Coming y Staadt se establecen trayectorias lineales de objetos completos no deformables con una posición y velocidad iniciales y, la premisa de evitar las *colisiones múltiples*. Para la configuración inicial, se calculan:

- Las colisiones entre los volúmenes envolventes de cada polígono inicial, lo que genera una lista de colisiones iniciales (una por eje coordenado), de la que se puede calcular la lista de colisiones efectiva.

- Todos los tiempos de intersección para cada elemento de la lista con sus elementos adyacentes, proceso que es realizado para cada lista cinética, los que constituyen una lista única de eventos inicial.

El hecho de tener de antemano los tiempos de los eventos elimina la necesidad de generar muestreos en la simulación. La lista de eventos es procesada en orden creciente en el tiempo y cada evento procesado de la lista da lugar a intercambios de posición entre los elementos de las listas cinéticas, como lo muestra la figura 2.9, resultando en la destrucción de dos adyacencias y la creación de otras dos nuevas. El nuevo orden de la lista ordenada cinética induce la desprogramación de hasta dos eventos, asociados a las antiguas adyacencias, y la programación de hasta dos nuevos eventos, asociados a las dos nuevas.

Figura 2.8: Ejemplo de dos listas cinéticas, una en el eje X y la otra en el eje Y, para tres objetos. Se puede observar que la lista tiene los extremos de cada objeto (marcados como Min_n o Max_n), las velocidades y direcciones por coordenada de cada objeto (V_{nx}), y entre medio de estos, existen los eventos de intersección $p_{a,b}$ entre el objeto a y el b, o bien, no existen intersecciones, marcadas por $\emptyset$. Ilustración tomada de [11]

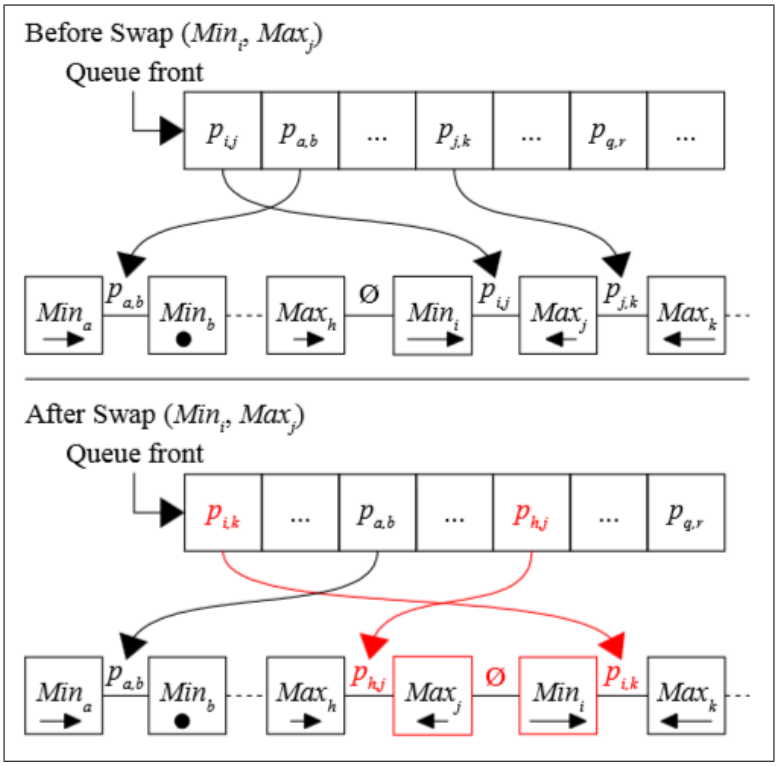


El hecho que existan cambios en las adyacencias de las listas cinéticas no siempre implican un cambio en la lista de colisiones calculadas inicialmente. Sólo en aquellos casos donde un extremo superior de un objeto intercambia posición con el extremo inferior de otro se da un cambio en la lista de colisiones creando un nuevo par de objetos colisionando o eliminando uno ya existente, como puede notarse en la figura 2.9. La necesidad de actualizaciones externas de las estructuras se reduce a las notificaciones cuando el movimiento de un objeto cambia y no cuando cambia de posición. En todo otro sentido, el método Sweep and Prune permanece sin cambio. Los cambios de movimiento se producen cuando hay colisiones entre los objetos o cuando el usuario induce un cambio.

El análisis de esta *KDS* de acuerdo a los aspectos definidos en 2.4.1 establece que es [11]:

- **sensible**: ya que al cambiar el movimiento de un objeto, resulta en el cambio de hasta 2 eventos por eje coordenado, cada uno de los cuales requiere $O(\log n)$ trabajo de programación.
- **local**: el número máximo de eventos que dependen de un objeto en movimiento es de dos eventos por eje, lo que implica $O(1)$.
- **compacta**: requiere sólo $O(n)$ de espacio.
- **eficiente**: genera el mismo número de eventos internos que externos, los swaps.

Figura 2.9: Ejemplo de un evento de intercambio en una lista cinética, sacado de [11]



Capítulo 3

Análisis, diseño e implementación de los algoritmos

En este capítulo se describe las decisiones de diseño e implementación de la aplicación, el plan de pruebas de integración descrito por el proyecto *scenarios*, y el modelo con el que se hará los posteriores análisis de desempeño. Si se desea revisar con más detalle las implementaciones, visitar el anexo B.

3.1. Análisis

Las cualidades deseables de un algoritmo son: eficiencia, robustez, extensibilidad, legibilidad y usabilidad. Debido a lo anterior se decidió la creación de cuatro bibliotecas escritas en Java: una biblioteca que contiene lo necesario para el cálculo de Sweep and Prune estático, llamada *StaticLibrary*, una biblioteca para el cálculo de Sweep and Prune cinético, llamada *KineticLibrary*, un proyecto para generar y mostrar por pantalla un experimento, llamado *ShowModel*, y un proyecto donde se encuentran las pruebas de integración y el modelo desde donde se hará la evaluación de los tres algoritmos, llamado *Scenarios*. Además de la implementación respectiva de cada paquete, se realizó una serie de pruebas unitarias en JUnit 4.x para asegurar el correcto funcionamiento de las clases y sus métodos, así como la documentación en Javadoc de todos los métodos importantes. Como requerimiento especial del Laboratorio de Análisis de Imágenes Científicas de la Facultad de Medicina de la Universidad de Chile (SCIAN-Lab), se implementó una versión del modelo Sweep and Prune estático en C++ de Windows, generando una biblioteca de vínculo dinámico para utilizarse de manera integrada con el software Scian-Soft/IDL.

Esta versión del modelo estático está definida para 2D, acoplándose un método de detección de colisiones de fase angosta (AABB Tree) para detectar las aristas de los objetos que entran en contacto.

3.2. Implementación del ambiente de las simulaciones

Para las simulaciones se creó una interfaz *Simulation* dentro de la biblioteca **Scenarios**, con los métodos:

- **exec**: método abstracto que será el encargado de ejecutar el algoritmo en cuestión en cada una de las realizaciones de la interfaz Simulation.
- **writeStatisticsToFile**: método encargado de grabar las estadísticas en la ruta y el nombre de archivo estipulados como parámetros de entrada.
- **getStatistics**: método encargado de devolver el objeto de las estadísticas respecto a la simulación ejecutada.

Esta interfaz fue implementada por cada uno de los métodos a comparar: Sweep and Prune directo, Sweep and Prune incremental y Sweep and Prune cinético.

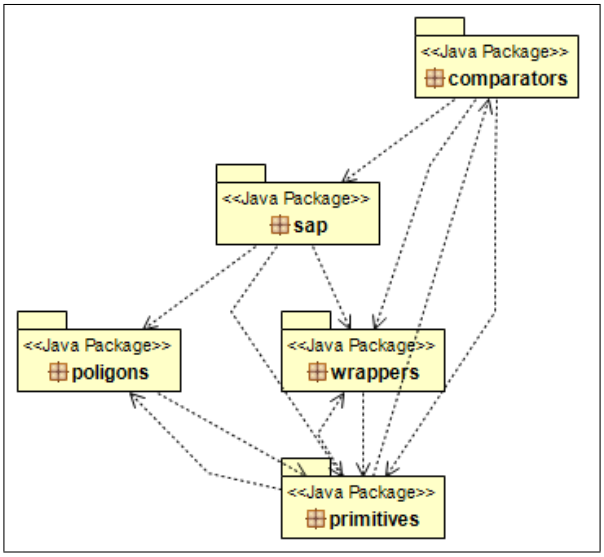
3.3. Diseño de la biblioteca estática

En la biblioteca **StaticLibrary** se desarrolló desde cero el uso de puntos y aristas en 2D. Si bien otros *frameworks* como [13] o [14] poseen estas primitivas, se optó por desarrollar todo completamente, eliminando características innecesarias. La creación de las *AABBs* fueron desarrolladas desde cero, pensando en la posible integración de otros volúmenes envolventes más adelante, como esferas, *OBBs*, cerraduras convexas, entre otras; así también la futura expansión del sistema al análisis 3D. Esta biblioteca también está encargada de la creación de diversos polígonos en 2D parametrizados, que están construídos en base al radio unitario, siendo posible cambiar su escala mediante un parámetro en el constructor y elegir una rotación al azar de la figura. Estos polígonos fueron creados usando el patrón *fábrica abstracta* [15]. Finalmente, también dentro de esta biblioteca se encuentra la implementación del algoritmo sweep and prune estático. La figura 3.1 muestra la relación existente entre los paquetes de la biblioteca.

3.3.1. Objetos y métodos claves

Los paquetes que componen la biblioteca **staticLib** son: comparators, sap, polygons, wrappers y primitives, como lo muestra la figura 3.1. Los más importantes son descritos a continuación:

Figura 3.1: Diagrama de paquetes de la biblioteca staticLib



- **primitives** (figura 3.2): posee todas las clases necesarias para representar elementos en 2D y su futura extensión a 3D. Definiciones de punto, arista y su *AABB*. Se implementaron interfaces para Point, Edge, Primitive y BV, de las cuales se hereda la implementación para Point2D, Edge2D, Primitive2D y el volumen envolvente AABB2D. La interfaz genérica Pair es usada en muchas clases dentro de las primitivas. Adicionalmente se definió una carpeta para contener los test unitarios de todas las clases relevantes y sus métodos. A continuación se da una breve descripción de las clases más importantes:

  - **Point2D**: representada como un par de valores de precisión doble. Incluye métodos para sumar coordenadas entre dos puntos, calcular distancias entre puntos, multiplicar por un escalar, entre otros.
  - **Edge2D**: representada como un arreglo de dos objetos Point2D. Incluye métodos para el cálculo de: área con signo entre la arista y un punto externo, intersecciones entre dos aristas, longitud de la arista, entre otros.

- **AABB2D**: representada por dos objetos Point2D y un objeto Primitive, este último, es un objeto Edge2D. Incluye métodos para intersectar o unirse con otros AABB2D, obtener el largo y el ancho del objeto, comparaciones con otros AABB2D por eje coordenado y por área, entre otros.
- **Element**: clase utilizada en los AABB2D. Es un wrapper que envuelve el valor de una coordenada de un punto y un valor booleano indicando si es o no el extremo de máximo valor.
- **BVTree**: necesaria para generar el árbol de volúmenes envolventes, para uso futuro.
- **BVNode**: contiene un volumen envolvente que es parte del BVTree, hasta ahora el único volumen envolvente implementado es el AABB2D.

- **polygons** (figura 3.3): implementa una fábrica de polígonos regulares e irregulares para su utilización en los diferentes escenarios. Los objetos generados heredan de la clase abstracta *Polygon2D*. Todos los puntos que definen a un polígono se encuentran dentro del círculo unitario. Un atributo importante de estos objetos es el *factor de escala*, cuyo valor por defecto es 1.0, es decir, se encuentran definidos dentro del círculo unitario anteriormente descrito. Los métodos implementados permiten trasladar, cambiar la escala y rotar a los polígonos. Las clases importantes de este *package*:

  - **Polygon2D**: formada por una lista de Point2D, posee métodos para calcular el área, desplazar el polígono, escalarlo, obtener el AABB2D que lo envuelve, entre otros. Este polígono posteriormente se puede escalar de acuerdo al valor del área final y desplazar sumando un objeto Point2D dado como parámetro.
  - **PolygonFactory**: obtiene un polígono 2D usando el patrón *factory method*. Como opciones están triángulo, cuadrado, pentágono, hexágono, estrella de 4 puntas, estrella de 6 puntas y un boomerang.

Figura 3.2: Diagrama de clases de las primitivas estáticas en 2D

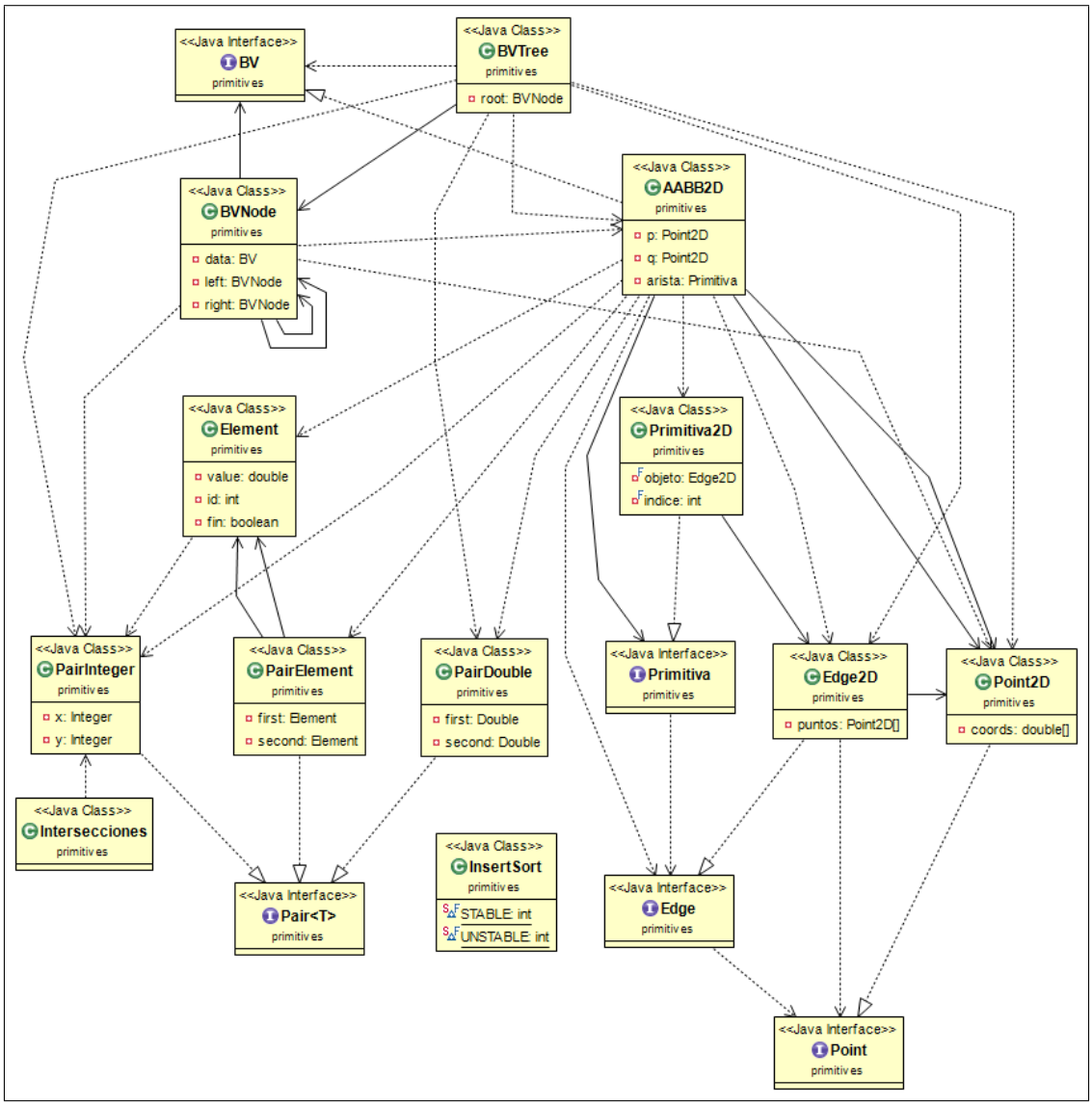
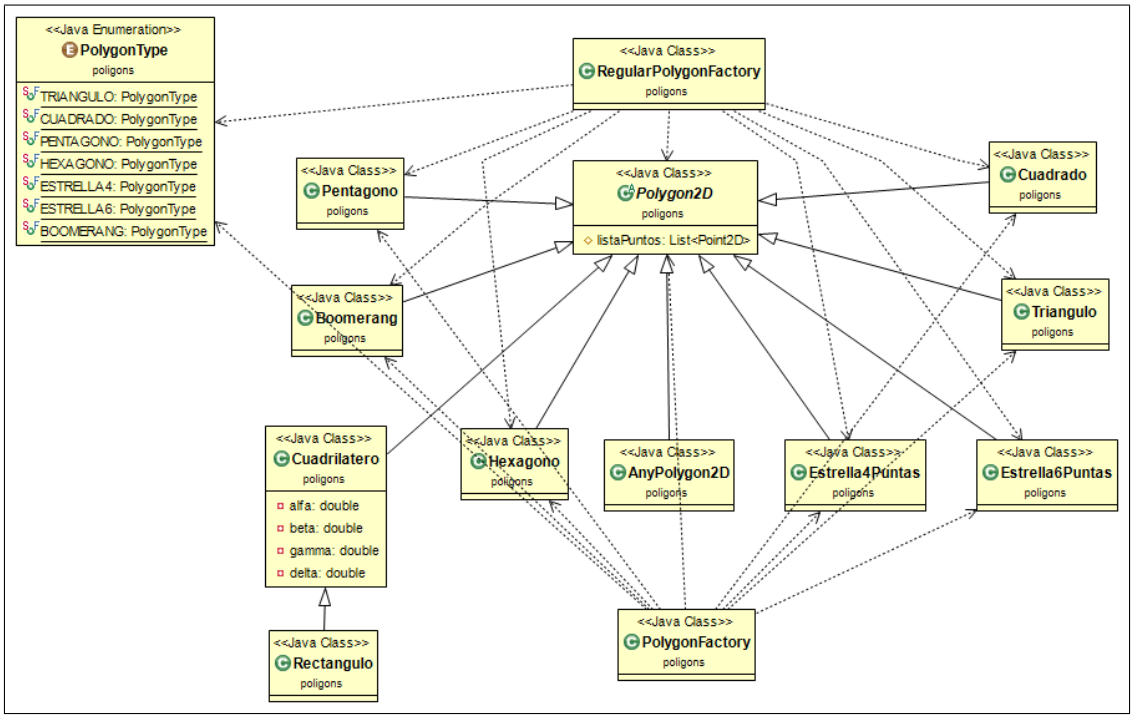


Figura 3.3: Diagrama de clases del paquete polygons

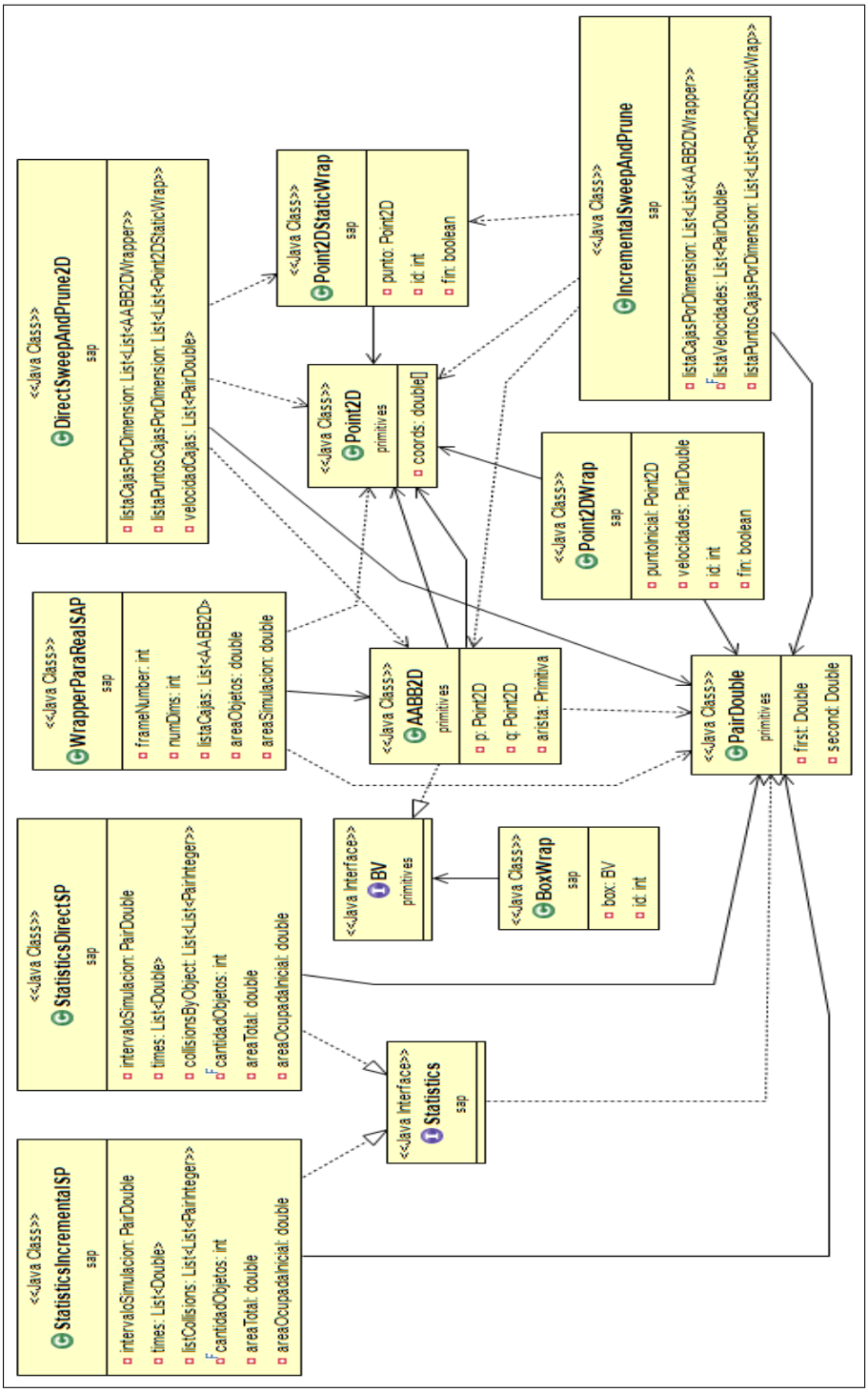


- **sap** (figura 3.4): posee lo necesario para la implementación de las versiones Sweep and Prune estáticas directa e incremental, para así obtener la lista de colisiones como resultado. Los métodos importantes de este *package* son:

  - **DirectSweepAndPrune2D**: implementación del método sweep and prune directo. Posee como atributos una lista de cajas AABB2D que se mueven con el paso del tiempo, existiendo una lista por coordenada. Una lista de Puntos2D obtenidas desde las cajas anteriores, también una lista por coordenada y, la lista de velocidades de cada caja. Los métodos más importantes son:
    - getListaParesEfectivosColisionando: retorna una lista de pares de objetos que efectivamente colisionan.
    - insertionSort: ordena la lista de puntos en el tiempo actual, en base al valor de su coordenada.
    - inicializarEstructuras: método que inicializa la lista de cajas que se actualizará a lo largo de la simulación.
    - updateBoxesPositions: método que actualiza la posición dado un delta de tiempo, desde la posición actual de los objetos.

- **IncrementalSweepAndPrune**: implementación del método sweep and prune incremental. Al igual que en el caso directo, sus atributos son una lista de cajas AABB2D que se mueven con el paso del tiempo, existiendo una lista por coordenada, una lista de Puntos2D obtenidas desde las cajas anteriores, también una lista por coordenada y, la lista de velocidades de cada caja. Los métodos más importantes son:
    - insertionSort: ordena la lista de puntos en el tiempo actual, en base al valor de su coordenada. La versión de insertionSort usada en en base a un centinela, que permite apurar el algoritmo.
    - updateBoxesPositions: método que actualiza la posición dado un delta de tiempo, desde la posición actual de los objetos.
    - validarOrdenListaPuntos: revisa si la lista está ordenada en el tiempo actual. Si está desordenada llama a **insertionSort** para ordenarla.

Figura 3.4: Diagrama de clases del paquete sap (sweep and prune)



3.3.2. Implementación del algoritmo Sweep and Prune directo

Para ejecutar el algoritmo Sweep and Prune directo el punto de entrada es la clase *SimulationStaticDirectSweepAndPrune*, la que es una realización de la interfaz *Simulation*. Esta clase tiene como parámetros de entrada de su constructor:

- la lista de objetos, representada por una lista de listas de puntos2D.
- la lista de dimensiones, que corresponde a un par de valores por dimensión, que muestra los límites físicos de la simulación, que por ahora no son más que valores para las estadísticas.
- la lista de velocidades de cada objeto (representada como una lista de pares de valores double), los tiempos inicial y final de la simulación.
- un valor booleano que permite mostrar mensajes extra por pantalla en la medida que se ejecuta la aplicación.

Dentro de este constructor se inicializa la lista de cajas necesarias para la ejecución del algoritmo, que corresponden a los AABB en 2D de cada objeto, además se calcula los datos de las áreas iniciales de simulación y el área ocupada por los objetos, necesarios para las estadísticas de la densidad de los objetos dentro de la escena.

El método *exec* es el que ejecuta el algoritmo Sweep and Prune directo. Este algoritmo ejecuta el cálculo de las listas de colisiones para cada tiempo como si fuera el único. Los pasos que realiza el algoritmo son los siguientes:

- En el tiempo actual, se actualiza la posición de las cajas.
- Se ejecuta el método *structuresInitialization* que extrae los extremos máximos y mínimos de cada caja y los ordena en forma ascendente.
- Se ejecuta el método *getPairsListCollision* que retorna la lista de colisiones efectivas.
- Se guardan todas las estadísticas en un archivo de texto.

La clase *DirectSweepAndPrune2D* contiene todos los métodos necesarios para ejecutar el algoritmo Sweep and Prune directo en 2D. Internamente está formado por:

- Una lista de listas de puntos2D, que corresponden a la lista ordenada de los puntos extremos de cada caja que envuelve a cada objeto. Existe una lista por eje coordenado.

- Una lista de listas de pares de enteros, que indican la posición de los puntos extremos de cada caja de cada objeto. Existe una lista por cada eje coordenado.
- Una lista de pares de enteros, con la lista de identificadores de los objetos que colisionan efectivamente.
- Dos listas de pares de enteros auxiliares, para almacenar las colisiones entre objetos en cada eje coordenado.

El método *initializeStructures* es el encargado de extraer de la lista de cajas los puntos extremos de cada una. Estos puntos son ordenados en tiempo $O(n \log n)$, usando la biblioteca **sort** de Java. Posteriormente se extrae la lista de posiciones de cada objeto, es decir, la posición dentro de la lista de puntos para cada objeto (una lista por coordenada). De esta lista de posiciones se extrae la lista de colisiones efectivas (que demora $O(n^2)$).

3.3.3. Implementación del algoritmo Sweep and Prune incremental

La diferencia principal entre este método y el anterior es que una vez establecida la lista ordenada de puntos de los extremos de las cajas de cada objeto, se mantiene ésta a lo largo de toda la simulación, evaluando en cada tiempo (fijado por los tiempos inicial y final de la simulación en conjunto con el paso temporal), si existe un cambio en la posición de alguno de los puntos. Cuando se detecta que el orden de la lista no se mantiene, se recalcula la lista de colisiones efectiva de los objetos. Para hacer esto posible, se definió puntos 2D que fueran móviles a lo largo de la simulación, de manera que se actualiza la posición de cada punto en cada intervalo de tiempo. Para ejecutar la simulación del algoritmo Sweep and Prune incremental, el punto de entrada es la clase *SimulationStaticIncrementalSweepAndPrune*, la que es una realización de la interfaz *Simulation*. Esta clase tiene como parámetros de entrada de su constructor:

- Una lista de listas de cajas acompañadas del identificador del objeto al que está asociado cada caja.
- Una lista de velocidades de cada objeto por coordenada.
- Una lista de listas de puntos2D que representan los extremos de cada caja de cada objeto.
- Una lista con las posiciones de inicio y final de cada caja de cada objeto.

3.4. Diseño de la biblioteca cinética

3.4.1. Objetos y métodos claves

Como se observó en 2.4.1, la lista cinética en conjunto con una cola de prioridades de los eventos es la base del método Sweep and Prune cinético. La biblioteca la completa un actuador y un observador, necesarios para desacoplar la interacción entre la lista cinética, la cola de eventos, la lista de colisiones y el objeto que almacena las estadísticas. Las clases más importantes de esta biblioteca son:

- **DimensionKinetic**: representan el plan de vuelo de cada coordenada de un punto2DKinetic. Según [11], los objetos en movimiento son polígonos, donde cada vértice tiene un "plan de vuelo" en función del tiempo que puede cambiar durante la simulación. En nuestro caso, por simplicidad, se estableció trayectorias rectilíneas uniformes que no cambian a lo largo del tiempo, es decir, cada coordenada tiene una trayectoria del tipo $\vec{v} \cdot t + \vec{k}$, con $\vec{v}$ la velocidad del objeto, y $\vec{k}$ la coordenada inicial en la simulación.

- **Point2DKinetic**: clase que representa a un Punto2D cinético, implementado como un arreglo de largo dos de DimensionKinetic. Método importante es el getPosition, que devuelve la posición del punto en un instante de tiempo ingresado como parámetro.

- **Wrapper**: clase que contiene un objeto DimensionKinetic, un identificador numérico del objeto y un atributo booleano que indica si es o no extremo superior.

- **KineticSortedList**: clase que representa la lista cinética en una dimensión. Está formada por una lista de objetos Wrapper.

- **KSLList**: clase que tiene una lista de KineticSortedList, representando al conjunto de listas cinéticas de la simulación, donde existe una lista por dimensión. Este objeto además cumple con extender la clase Observable, necesaria para la implementación del patrón *Observer* que desacople las funcionalidades entre la lista cinética y la cola de eventos.

- **Event**: clase abstracta que engloba a todos los tipos de eventos, tiene como atributos el tiempo en que se produce el evento y la dimensión en la que ocurre. Con esta clase se puede definir eventos de distinto tipo, como el choque con los límites de la simulación o un evento de intercambio de posiciones. Un evento de intercambio de posición (o *swap*), desde ahora representados por $EV_{i,j,dim}(t = t_k)$ corresponde

al intercambio de dos objetos en las posiciones i y j en la lista cinética en una dimensión dim en el tiempo t_k. Este evento de swap está definido por una única subclase de Evento, llamada EventoSwap, que tiene como atributos extra a los de la clase Evento, los índices de la lista cinética que deben intercambiarse en el momento del evento.

- **EventsQueue**: clase que contiene una cola de prioridad de objetos de la clase Evento. La cola está ordenada primero por tiempo, y en el caso de empate, se ordena por la dimensión, y en el caso de existir aún un empate, se ordena por el mínimo índice de la lista cinética que debe ser intercambiado. Esta cola de eventos también es un objeto *Observable*. Como métodos importantes esta clase tiene:

  - **deleteMultipleEvents**: permite eliminar eventos múltiples, indicando el tiempo del evento, la dimensión y el rango de los índices a eliminar ingresados como parámetros.
  - **calculateCurrentEvents**: dada la configuración actual de las listas, permite calcular los eventos que son generados desde el tiempo actual que es ingresado como parámetro.
  - **processEvents**: procesa la lista de eventos hasta que la cola queda vacía, esto incluye los eventos que se generan con los vecinos después de efectuar un intercambio de posición en alguna de las listas KineticSortedList.

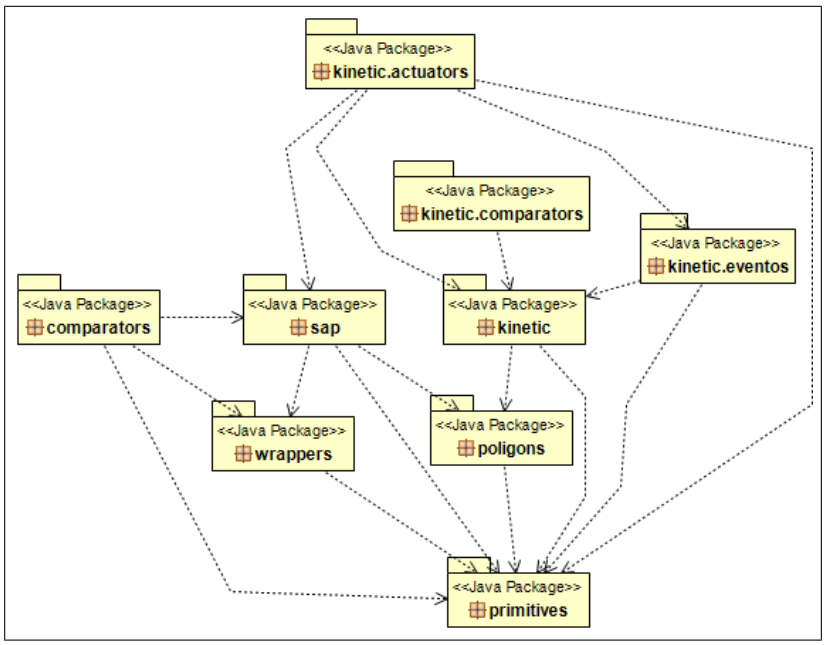
- **EventsObserver**: clase que implementa al *Observer*, y contiene referencias a los objetos observados: ListaKSL (de la clase KSLList), colisiones (de la clase IntersectionsList), colaEventos (de la clase EventsQueue) y estadísticas (de la clase KineticStatistics). Esta clase funciona en conjunto con la clase *Actuator*, la que ejecuta los eventos en los diferentes objetos observados de acuerdo a la señal que el objeto observable le envía al observador. Las señales que atiende el *Actuator* son iniciar las estructuras, resolver un evento simple y resolver un evento múltiple.

3.4.2. Diagramas de clases

Según la teoría, los objetos en movimiento son polígonos, donde cada vértice tiene un "plan de vuelo" en función del tiempo que puede cambiar durante la simulación. Se asume que el plan de vuelo requiere espacio $O(1)$ y que la intersección entre dos planes de vuelo puede ser calculada en tiempo $O(1)$. El plan de vuelo puede cambiar debido a intevención del usuario o debido a fenómenos físicos como una colisión, en cuyo caso se debe efectuar una actualización de todos los eventos que involucran al vértice. Como se

observó en 2.4.5, el atributo a mantener es la lista cinética ordenada de menor a mayor, la que está compuesta por todos los puntos cinéticos extremos de los objetos (el máximo y el mínimo). En la implementación de la aplicación se definió polígonos bidimensionales de diferentes tamaños, no deformables y con un plan de vuelo lineal en función del tiempo por objeto. Si existen colisiones no se cambia la dirección de los objetos. La figura 3.5 muestra la distribución de los paquetes de la biblioteca cinética *kineticLibrary* y su relación con la biblioteca *staticLibrary*.

Figura 3.5: Diagrama de paquetes de la biblioteca kineticLibrary (arriba) y su relación con la biblioteca staticLibrary (abajo)



La implementación de esta biblioteca involucra la creación de listas cinéticas, una cola de eventos, listas de colisiones (una por eje coordenado), una clase que albergue las estadísticas y un mecanismo que asegure un bajo acoplamiento entre las partes. La lista cinética tiene por función mantener la relación de orden de los puntos mínimo y máximo de cada objeto en cada instante de tiempo de la simulación. Está implementada en base a puntos 2D cinéticos, es decir puntos que además tienen la velocidad de desplazamiento (por segundo) de cada coordenada. A esta lista se le ingresa el tiempo en que se desea evaluar el orden de la lista. Si no se mantiene se hace un nuevo ordenamiento, lo que a su vez, puede incidir en la creación de nuevos eventos que deben ser agregados a la cola de eventos.

La cola de eventos tiene por función alimentar al método Sweep and Prune Cinético con los eventos de intersección de puntos en la simulación. Esta cola fue implementada con la clase PriorityQueue de Java, definiendo la interfaz Comparable en los eventos para establecer el orden. Notar que la cola de eventos puede ser implementada por una cola de prioridad, una two-pass pairing heap, o un árbol AVL, de acuerdo a la sección 2.4.2.

La lista de colisiones es una colección de pares de enteros, que indican los índices de los objetos involucrados en una colisión. Existe una lista por cada coordenada.

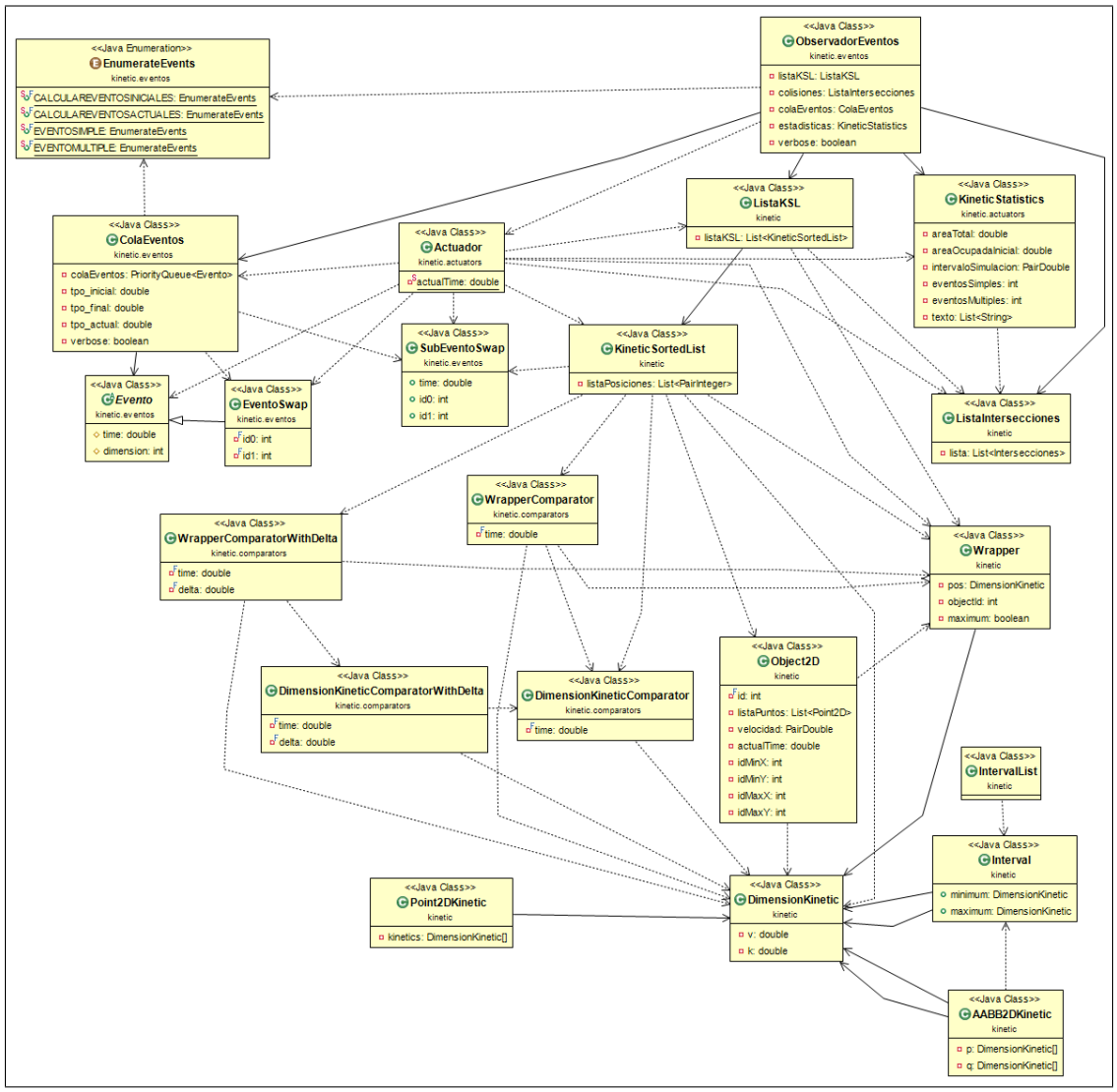
Existe una estrecha relación entre todos los objetos anteriores. Por ejemplo, un evento de la cola puede provocar un cambio en la lista de colisiones y en las listas cinéticas. A su vez, estos cambios en las listas cinéticas pueden generar nuevos eventos que deben ser incorporados en la cola. Como una forma de mantener el desacoplamiento de las clases, se optó por crear una implementación con el patrón *Observer*, en donde cada objeto observado (listas cinéticas, cola de eventos, estadísticas) avisan de algún tipo de evento a un Observador, el que delega la ejecución a un *Actuator* (figura 3.6).

La biblioteca **kinetic** posee:

- La definición de los puntos cinéticos, que corresponden a pares de valores, por cada coordenada, que indican la velocidad del objeto y su posición.
- La definición de la lista cinética ordenada que alberga a los puntos cinéticos extremos de cada objeto (máximo y mínimo por cada dimensión) de acuerdo a cómo están ordenados en un instante determinado de la simulación.
- La cola de prioridad que contiene a todos los eventos de la simulación.
- La lista de colisiones por dimensión, que es constantemente actualizada dependiendo de los eventos.

La biblioteca cinética necesita de la biblioteca estática para su funcionamiento, por lo que debe ser incluída dentro del proyecto, como muestra la figura 3.5.

Figura 3.6: Diagrama de clases con la implementación del patrón observador para las clases ColaEventos, ListaKSL, kineticStatstics y ListaColisiones



3.4.3. Implementación del algoritmo Sweep and Prune Cinético

Para la simulación del algoritmo Sweep and Prune cinético, el punto de entrada es la clase *SimulationKinetic*, la que es una realización de la interfaz *Simulation*. Esta clase tiene como parámetros de entrada de su constructor:

- Una lista de object2D. Cada uno de estos objetos tiene asociados: un identificador numérico del objeto, una lista de puntos2D, las velocidades del objeto en cada eje coordenado y el tiempo actual del objeto.
- Los tiempos inicial y final de la simulación.
- La lista de dimensiones, es decir los límites iniciales de la simulación.
- Una variable booleana para mostrar información extra en la simulación. Esta permite revisar detalladamente el proceso de intercambios y detecciones de nuevos eventos a medida que la simulación se ejecuta.

El constructor de la clase *SimulationKinetic* inicializa las estadísticas con el área total de los objetos, el área de la simulación y el tiempo de duración de la simulación.

El método *exec* inicializa los objetos listaKSL, colisiones, la colaEventos y estadísticas. Estos objetos se comunican entre sí por medio del patrón Observer, de manera de aislar responsabilidades entre las clases. La clase EventsObserver es la encargada de ser el observador para las clases anteriores, las que comunican tipos de eventos mediante las señales: CALCULAREVENTOSINICIALES, EVENTOSIMPLE o EVENTOMULTIPLE, a un objeto de la clase *Actuator* que genera los cambios necesarios en las clases asociadas al observador.

El método *initializeStructures* de la clase SimulationKinetic arma las listas cinéticas (una por coordenada), la lista de colisiones y calcula todos los eventos iniciales dados por la configuración inicial de las listas cinéticas.

Una vez que se han establecido los eventos iniciales, se procede a la ejecución del método *processEvents*, que me procesará los eventos que existan en la cola de eventos hasta cumplirse la simulación. El método *processEvents* funciona de la siguiente manera:

Algoritmo 5: metodo *processEvents* de la clase *EventsQueue*

```
Procedure processEvents
    while !cola.isEmpty() do
        super.setChanged();
        super.notifyObservers(EnumerateEvents.EVENTOSIMPLE);
```

Lo que hace **super.setChanged()** y **super.notifyObservers(...)** es mandarle al objeto de la clase Observer una señal de que se debe ejecutar un procedimiento determinado por el mensaje **EnumerateEvents.EVENTOSIMPLE**.

La clase EventsObserver

Esta clase corresponde al observador que recibe los mensajes enviados por los objetos observados. Los distintos mensajes que se pueden enviar al observador, detallando lo que hace cada uno:

- CALCULAREVENTOSINICIALES: mensaje para calcular los eventos iniciales. Se delega al objeto Actuador la inicialización de las distintas estructuras: la cola de eventos, la lista de KSL, lista de colisiones y el objeto de estadísticas.
- CALCULAREVENTOSACTUALES: dado un valor del tiempo, se calcula los eventos futuros entre este valor del tiempo y el final de la simulación. Este mensaje no es usado en ninguna simulación.
- EVENTOSIMPLE: se avisa que el evento a procesar es un evento simple, es decir involucra solamente a un par de puntos en una dimensión específica.
- EVENTOMULTIPLE: se avisa que el evento a procesar es un evento múltiple, por lo que debe generarse los mecanismos para mantener el orden de la listaKSL.

En el algoritmo 5 el mensaje enviado es EVENTOSIMPLE, lo que gatilla los siguientes pasos:

- Extraer el evento de la cola de eventos.
- Consultar a la lista cinética si el evento extraído es un evento simple o múltiple. Esto se logra consultando si alrededor de los puntos involucrados en el evento en cuestión existen otros puntos que están en la misma posición y en el mismo tiempo de ocurrido el evento.
- Si el evento es simple, se pide al Actuador que procese un evento simple.
- si el evento es múltiple, se pide al Actuador que procese un evento múltiple.

La clase Actuator

Clase encargada de realizar los cambios en las listas cinéticas, la cola de eventos y en las listas de colisiones. Dependiendo del mensaje recibido en el observador, se puede ejecutar uno de los siguientes métodos de la clase:

- **initializeStructures**: las listas ordenadas cinéticas se crean ordenadas en base al tiempo inicial de la cola de eventos (valor que usualmente es 0). Luego se hace un recorrido entre todos los pares vecinos en la lista cinética (como se muestra en la figura 2.3) obteniendo así una lista de eventos y el tiempo de ejecución del evento. Esto implica resolver el tiempo de las ecuaciones lineales que representan la trayectoria de cada uno de los puntos que se encuentran en las posiciones i e $i+1$ de cada KSL. Es decir, se debe resolver:

$$v_i * t + k_i = v_{i+1} * t + k_{i+1} \tag{3.1}$$

  o lo que es lo mismo,

$$t_{i,i+1} = -\frac{k_{i+1} - k_i}{v_{i+1} - v_i} \tag{3.2}$$

  Hay que aclarar que los eventos corresponden a un intercambio de posiciones entre dos puntos adyacentes, sin perjuicio que en otra investigación se agregue otro tipo de eventos como el choque con límites de la simulación.

- **solveSimpleEvent**: con parámetros de entrada la cola de eventos, la lista ordenada cinética, la lista de colisiones, la lista de estadísticas y el parámetro booleano que permite mostrar información adicional. Este método extrae el evento de la cola y con él se verifica en la lista cinética si es un evento simple o múltiple. Si el evento es simple:

  1. Se extrae la información del par de elementos, de la clase *Wrapper*, involucrados en el evento. Datos importantes de los *Wrapper* son los id asociados a los polígonos involucrados y si corresponden al inicio o al fin del polígono.
  2. Se revisa la lista de colisiones en donde se produce un cambio solamente si el intercambio de posiciones de la lista cinética se da entre un inicio y un final de dos objetos. En el caso del intercambio entre dos inicios o dos finales no hay cambio en la lista de colisiones.

3. Se procede a revisar si el par de objetos involucrados se encuentra en la lista de colisiones, en cuyo caso se elimina de la lista y, en caso contrario, se agrega a la lista de colisiones.
4. Se realiza el intercambio de posiciones entre los dos elementos de la KSL.
5. Se actualiza el tiempo actual de la cola de eventos (indicando que se ha procesado el evento).
6. Se calculan los eventos de las nuevas vecindades debidas al intercambio de posiciones en la lista cinética, como se establece en la sección 2.4.2, los que son agregados a la cola de eventos.

Si el evento es múltiple, se deriva su ejecución al método *solveMultipleEvent*.

- **solveMultipleEvent**: con parámetros de entrada la cola de eventos, la lista ordenada cinética, la lista de colisiones, la lista de estadísticas y el parámetro booleano que permite mostrar información adicional. El procedimiento para procesar un evento múltiple:

  1. Se extrae el evento de la cola de eventos. Esta es implementada como una cola de prioridad, clase que viene por defecto en la biblioteca estándar de Java.
  2. Se extrae el rango de índices invalidados de la lista cinética.
  3. Se agrega a las estadísticas los índices de las listas cinéticas que están involucradas en el evento múltiple.
  4. Se elimina los elementos de la cola de eventos que están dentro del rango. Al ser una cola de prioridad ordenada primero por tiempo y luego por la dimensión, empieza a extraese eventos que corresponden al tiempo del evento y dimensión específicada.
  5. Se reordena todos los elementos involucrados en la colisión múltiple de la lista cinética, con un pequeño ϵ de tiempo ingresado como parámetro para desempatar.
  6. Se valida que la nueva lista queda bien ordenada (paso que no es obligatorio y que es de $O(n)$).
  7. Se calcula los nuevos eventos a ingresar en la cola, generados por el intercambio de posiciones en la lista cinética de los puntos involucrados en el evento múltiple, los que son agregados a la cola de eventos.

3.5. La biblioteca scenarios: pruebas de integración y modelo

Esta biblioteca fue creada con la intención de hacer pruebas de integración de las bibliotecas anteriores, además de crear el modelo sobre el cuál se hará las mediciones. Posee clases para leer y escribir los distintos archivos de texto necesarios para los experimentos, tanto los archivos iniciales, como los resultados, lo que hará reproductibles los experimentos realizados; un generador de objetos al azar, el que utiliza la fábrica de objetos que se encuentra en el staticLib en conjunto con otros parámetros como las velocidades por cada eje, si se quiere el objeto rotado o no y la escala del objeto y, las clases que generan la simulación. En este caso, se diseñó un sistema con una variación del patrón de diseño observador [15], el cual tiene como sujetos a la lista cinética ordenada, la cola de eventos y la lista de colisiones, como una forma de separar la interacción entre ellas. La figura 3.7 muestra la distribución general de las clases de la biblioteca **scenarios**, que se encuentra dividida en dos *packages*:

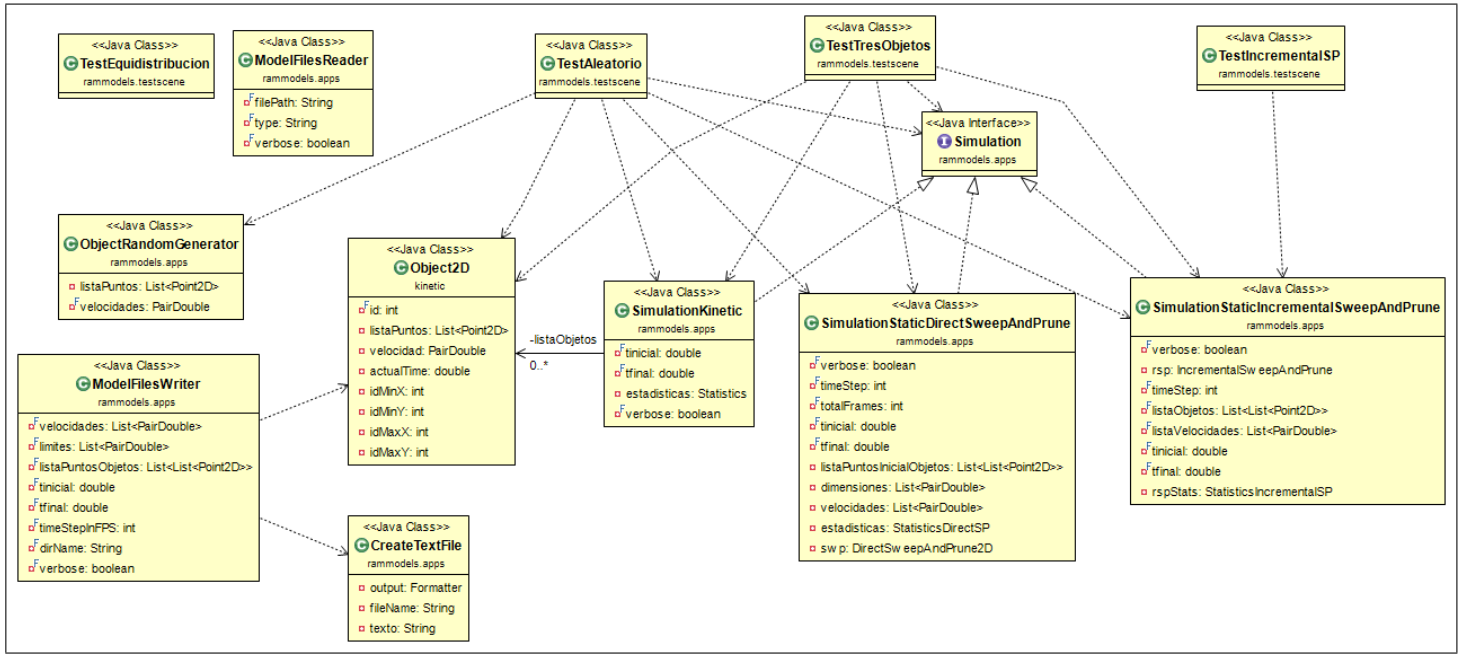
1. **apps**: que posee varias clases de apoyo, principalmente para crear y leer archivos de texto, traspasar datos a los modelos para iniciar las simulaciones y guardar estadísticas como resultado de las simulaciones. Las clases más importantes son:

    - **CreateTextFile**: clase encargada de crear archivos de texto. Necesaria para la creación de los archivos iniciales de los modelos a ejecutar. Esto se hace para tener un respaldo de los modelos iniciales, de manera que si se desea volver a ejecutar los mismos, se parta desde esos archivos.
    - **ModelFilesReader** y **ModelFilesWriter**: clases que se encargan de leer y escribir los archivos de los modelos para su posterior ejecución.
    - **Simulation**: interfaz que posee tres métodos importantes: **exec**, que ejecuta la simulación en cuestión, **getStatistics**, que retorna un objeto de la interfaz **Statistics**, con las estadísticas de la ejecución y, **writeStatisticsToFile**, que graba las estadísticas a un archivo de texto.
    - **SimulationKinetic**, **SimulationStaticDirectSweepAndPrune** y **SimulationStaticIncrementalSweepAndPrune**: implementaciones de la interfaz **Simulation** para los tres algoritmos a simular.

2. **testscene**: posee los cuatro modelos a ejecutar, los cuales son:

   - **TestIncrementalSP**: simulación en la que se genera rombos de distinto tamaño centrados en el origen del sistema de coordenadas y se deplazan a lo largo del eje X.
   - **TestEquidistribucion**: simulación en la que se genera una cantidad de triángulos ingresada como parámetro, equidistribuídos con una velocidad constante y una dirección aleatoria.
   - **TestTresObjetos**: Clase que simula el movimiento de tres objetos y ejecuta los tres algoritmos en ellos. Se trata más extensamente en 4.2.3.
   - **TestAleatorio**: Clase sobre la que se hacen las mediciones en esta tesis. Como parámetros se incluyen: la cantidad de objetos en la simulación, el porcentaje de ocupación del área total inicial en la simulación y los cuadros por segundo de la simulacón.

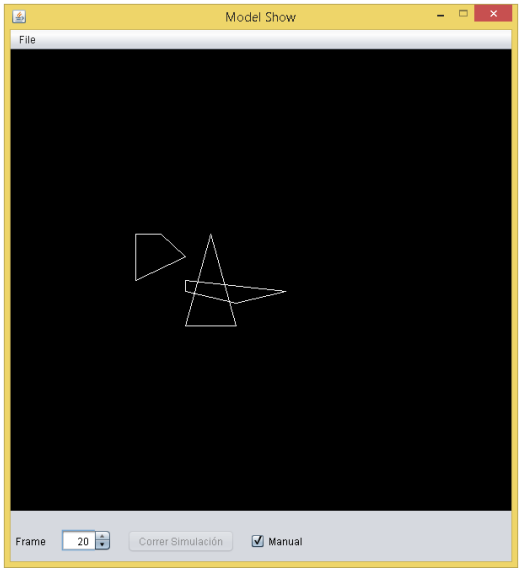
Figura 3.7: Biblioteca de escenarios



3.6. La biblioteca showModel

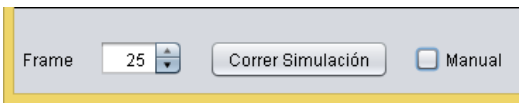
La biblioteca showModel surgió como necesidad de tener visibilidad de los modelos que estaban siendo generados, con la única finalidad de mostrar la simulación dados el paso temporal entre cada cuadro de la simulación y la duración de esta como parámetros iniciales. Esta biblioteca tiene lo necesario para cargar un archivo de texto inicial de la simulación y mostrar el cuadro a cuadro, o simulación continua. Está hecha con la biblioteca Swing de Java utilizando la clase Graphics2D. La figura 3.8 muestra la ejecución de la biblioteca **showModel** con el modelo de tres objetos.

Figura 3.8: ShowModelFrame con la simulación del escenario de tres objetos (4.2.3) en el momento del evento múltiple en $t = 2$.



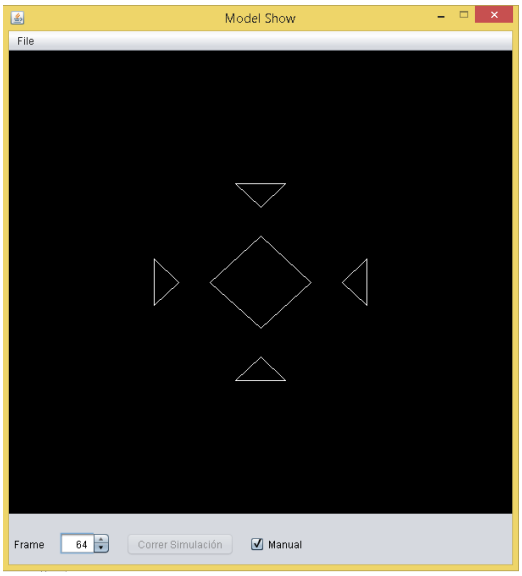
La aplicación tiene dos formas de ejecución: automática y manual, la que se activa mediante un checkbox. Posee un spinner con el número del frame, el cuál puede ser cambiado a voluntad en el modo manual. También contiene un botón que permite correr la simulación automáticamente cuando no está activado el modo manual, como lo muestra la figura 3.9.

Figura 3.9: Panel de control de la aplicación showModel



La biblioteca también posee una demo de la creación de un modelo(figura 3.10), la que puede ser cargada después en la aplicación para su visualización.

Figura 3.10: Demo de ejemplo cargada en la aplicación



3.7. Biblioteca de vínculos dinámicos con AABBTree

En el laboratorio de análisis de imágenes científicas de la facultad de medicina de la Universidad de Chile (SCIAN-Lab) se implementó una solución del algoritmo Sweep and Prune estático pero basándose solamente en el primer paso, que consiste en la creación de las listas ordenadas de cajas (una lista por eje coordenado) para su posterior barrido y creación de la lista de pares de cajas que posiblemente colisionan. Esto, debido a que entre una imagen y la siguiente, dentro de la lista de experimentos no hay seguridad de que exista la coherencia espacial y temporal necesaria para la ejecución del algoritmo completo. Una vez obtenidos los posibles pares de objetos que colisionan se procede a verificar qué aristas son las que están en contacto, utilizando un método de detección de colisiones de fase angosta, como lo es el AABB Tree, consistente en una *bvh* compuesta de cajas alineadas por los ejes (*AABBs*).

Ambos métodos debieron ser implementados en C++ para Windows como una biblioteca de vínculos dinámicos (dll) debido a que se utiliza como función externa al programa Scian/IDL.

Capítulo 4

Validación de los algoritmos

4.1. Análisis asintótico de las implementaciones de los algoritmos

Para hacer una justa comparación entre la teoría y las implementaciones, se hará una revisión de los principales métodos de los diferentes algoritmos.

4.1.1. Sweep and Prune Directo

Para ello hay que analizar el comportamiento de la clase **SimulationStaticDirectSweepAndPrune** y su método **exec**, ya que este método fue el medido en todos los algoritmos implementados. El método exec podría ser resumido como sigue:

Algoritmo 6: metodo *exec* de Sweep and Prune Directo

```
Data: swp: objeto de la clase DirectSweepAndPrune2D
Procedure exec()
    foreach frame do
        if frame es el primero then
            swp.inicializarEstructuras(listaPuntosIniciales)
            swp.setVelocidadCajas(velocidades)
            swp.getListaParesEfectivosColisionando()
        else
            swp.updateBoxesPositions(1/timestep)
            swp.insertionSort()
            swp.getListaParesEfectivosColisionando()
```

Si f es el número de frames, n la cantidad de polígonos, tenemos:

- **initializeStructures**: método que se ejecuta una vez. Crea una lista de AABB2D para cada dimensión, de cada polígono involucrado en la simulación. Son dos dimensiones, por lo que el método tiene una $O(n/f)$ amortizado, donde f son la cantidad de cuadros de la simulación.
- **setVelocidadCajas**: método O(1), debido a que se ingresa la lista de pares de velocidades completo, de una sola vez.
- **getPairsListCollision**: método que se ejecuta en todos los frames, y está dividido en dos partes: la primera lee la lista, por coordenada, de puntos extremos (máximo y mínimo) de las cajas que rodean a cada polígono, la cuál está ordenada de menor a mayor. A esta lista se le ejecuta el barrido indicado en 2.3.1. Para apurar las búsquedas y borrados se estableció listas definidas por la clase HashSet de Java, la que tiene una complejidad de $O(1)$[1] para los métodos add, remove y contains que son usados en esta parte. Este barrido genera una lista de pares colisionando en la dimensión dada. La segunda parte consiste en buscar en ambas listas los pares de índices comunes, que son copiados a una lista de salida. Esta operación tiene una complejidad de $O(k)$, con k la cantidad máxima de pares generados en alguna dimensión. De acuerdo a los tiempos vistos en las simulaciones esta es la operación más costosa en los algoritmos sweep and prune estáticos.

En resumen, la cantidad k de pares de colisiones resultantes tiene una complejidad de $O(n^2)$, y si multiplicamos por la cantidad de frames a ejecutar este método nos da $O(f \cdot n^2)$.

4.1.2. Sweep and Prune Incremental

El mecanismo de este algoritmo es muy similar al caso anterior, sólo que la lista ordenada de puntos por cada dimensión debe ser revisada en cada cuadro temporal para validar que se mantiene el orden, que en caso de ser negativo, se aplica un nuevo ordenamiento de la lista y se recalculan las listas de colisiones como se indicó en 4.1.1.

El peor caso se da cuando la lista de puntos extremos de los objetos no mantiene el orden de menor a mayor. En ese caso se debe ejecutar el ordenamiento de los puntos, que está implementado usando el método insertionSort de acuerdo a lo propuesto por [9], cuya complejidad es $O(n^2)$. Posteriormente se requiere el cálculo de la lista de pares de objetos colisionando, lo que también tiene complejidad $O(n^2)$. Como esto puede suceder, en el

[1] `https://gist.github.com/psayre23/c30a821239f4818b0709`

peor caso, en cada frame de la simulación, tenemos que la complejidad del algoritmo está dada por $O(f \cdot n^2)$.

4.1.3. Sweep and Prune cinético

Según [11], el método que ellos proponen tiene una complejidad de $O((s + u + c) \log n + c \log c + b)$, con n, el número de objetos, s el número de swaps, c el número de colisiones, u el número de actualizaciones de movimiento que ocurren en un intervalo de tiempo y b el costo de todos los tests de intersecciones. Para ello hay que analizar el comportamiento de la clase **SimulationKinetic** y su método **exec**. El método exec podría ser resumido como sigue:

Algoritmo 7: método *exec* de Kinetic Sweep and Prune

```
Data: colaEventos: cola de Eventos, implementada como una cola de prioridad
Data: colisiones: lista de Intersecciones
Data: listaKSL: lista de Kinetic Sorted Lists
Procedure exec()
    crearListaKSL
    crearListaIntersecciones
    crear colaEventos
    crear observadorEventos
    agregar el observador a todos los objetos observados
    inicializar estructuras (colaEventos, listaKSL, colisiones)
    colaEventos.procesarEventos()
```

Si n es el número de polígonos:

- Crear **ListaKSL**, **listaIntersecciones**, **colaEventos** y **observadorEventos**: todos los métodos son $O(1)$.
- agregar observador a todos los objetos observados: son cuatro objetos en total (listaKSL, colaEventos, listaIntersecciones y objeto de estadísticas), tiene complejidad $O(1)$.
- inicializar estructuras (listaKSL, colaEventos, listaIntersecciones): este evento sucede una sola vez, al inicio de la simulación. Para asegurar el correcto funcionamiento de las listas, el acceso de todos los métodos que manipulan las listas tienen la palabra reservada **synchronized**. La inicialización de la listaKSL consiste en el llenado de dos listas de largo $2n$ con el equivalente cinético de los Wrappers utilizados en 4.1.1, es decir, contienen un objeto DimensionKinetic (de acuerdo a lo descrito en 3.4), el

id numérico que lo relaciona con el polígono de origen y si el punto en cuestión es un extremo inicial o final de la caja que contiene al polígono. Estas listas de objetos DimensionKinetic ordenadas en base al tiempo inicial dado por la simulación, que para obtener los puntos de la caja de un polígono tiene complejidad $O(1)$ (ya que a lo más son veinte puntos en un objeto), y como son n polígonos a buscar, esta inicialización tiene complejidad $O(n \log n)$. La inicialización de la cola de eventos necesita procesar la listaKSL recién creada para llenar la cola de prioridad con los eventos iniciales. Esta operación para cada par de puntos es de $O(1)$, por lo que para cada KSL es de $O(n)$. Para cada tiempo resuelto que sea mayor que el tiempo inicial t_0 de la simulación, se crea el evento de tipo $EV_{i,i+1,dim}\ (t = t_k)$. La creación de este evento tiene complejidad $O(1)$, e ingresarlo a la cola de prioridad tiene complejidad $O(\log n)$.
La inicialización de la lista de colisiones implica recorrer las listas KSL, y en esta parte inicial, el mecanismo es idéntico al de los algoritmos estáticos, por lo que la creación de la lista cinética inicial tiene complejidad $O(n^2)$.

- **colaEventos.procesarEventos**: Este método es ejecutado hasta que la cola de eventos haya quedado vacía, o hasta que se alcanza el final del tiempo de ejecución. Consiste en sacar uno por uno los eventos de la cola de prioridad y procesarlos como eventos simples, haciendo la respectiva evaluación de que esto es así. En el caso que no sea un evento simple, se ejecuta el proceso de reparación de la lista cinética afectada por el evento múltiple y se eliminan de la cola los eventos simples que conforman este evento múltiple, después de lo cual se sigue con el procesamiento normal de los eventos, de acuerdo a como se describe en el algoritmo 5.

4.2. Escenarios

4.2.1. Cómo crear los modelos de prueba

La biblioteca *Scenarios* es la que tiene almacenados los modelos de prueba. Se encuentra dividido en dos paquetes: **apps** y **testscene**. El paquete **apps** posee todas las clases necesarias para la construcción y ejecución de un modelo. Una breve descripción de sus clases y utilidad:

- **CreateTextFile**: permite crear un archivo de texto de acceso secuencial[2].

[2]Basado en un ejemplo del libro Java SE 8 for programmers, Paul & Harvey Deitel. (Cap. 15.4 Sequencial-Access Text Files)

- **ObjectRandomGenerator**: clase que genera un objeto de manera aleatoria. Necesita como parámetros de entrada las dimensiones de la simulación, el porcentaje del área total de la simulación que cubrirá el objeto, el intervalo de tiempo que dura la simulación. Esto genera la lista de Puntos2D inicial y la velocidad, por cada eje coordenado, del objeto.

- **Simulation**: interfaz que crea el contrato para los métodos exec (al que se mide el desempeño en todos los algoritmos), writeStatisticsToFile y getStatistics para todas las simulaciones.

- **SimulationKinetic**: implementación de la interfaz Simulation para el algoritmo Sweep and Prune Cinético.

- **SimulationStaticDirectSweepAndPrune**: implementación de la interfaz Simulation para el algoritmo Sweep and Prune estático directo, es decir, para la versión cuadro a cuadro.

- **SimulationStaticIncrementalSweepAndPrune**: implementación de la interfaz Simulation para el algoritmo Sweep and Prune estático incremental.

El paquete **testscene** posee los escenarios de simulación. Los escenarios de esta tesis corresponden a cuatro situaciones:

- **TestSPDirecto**: creado para probar el correcto funcionamiento del algoritmo sweep and prune directo y el método de detección de colisiones de detección de colisiones de fase angosta basado en la Jerarquía de Volúmenes Envolventes.

- **TestTresObjetos**: prueba el correcto funcionamiento de los tres algoritmos involucrados en esta tesis en el movimiento de tres objetos simples, incluyendo la detección de un evento múltiple por el algoritmo Sweep and Prune cinético.

- **TestEquidistribucion**: genera una grilla con objetos (triángulos) del mismo tamaño con velocidades de desplazamiento lentas en dirección al azar, que impiden que dentro del tiempo de la simulación (que corresponde a un segundo) exista alguna colisión.

- **TestAleatorio**: despliega objetos al azar con posiciones y velocidades al azar, pero que cumplen con dos parámetros dados al inicio de la simulación: cantidad de objetos y el área inicial cubierta por estos objetos como un porcentaje del área de la simulación. Este último modelo es el utilizado en la medición de desempeño de los algoritmos.

En general, una simulación necesita como puntos de partida la lista de puntos2D, las velocidades de cada objeto, el intervalo de tiempo de la simulación y las dimensiones de la simulación. Estos datos quedarán guardados en un único archivo de texto como respaldo de las simulaciones realizadas, las que podrían ser ejecutadas posteriormente. Estos archivos tienen el siguiente formato, el que está explicado usando patrones regulares:

```
1    marco:\\(\\((-?\\d+?.\\d+),(-?\\d+?.\\d+)\\),
           \\((-?\\d+?.\\d+),(-?\\d+?.\\d+)\\)\\)
2    timeInterval:(\\d+.?\\d+)-(\\d+.?\\d+)
3    framesPerSecond:(\\d+)
     //dados n objetos en la simulaci\'{o}n con i=1...n
2i+2 (-?\\d+?.\\d+)\\s+(-?\\d+?.\\d+) //velocidades del objeto
2i+3 (\\d+)\\s+(-?\\d+?.\\d+)\\s+(-?\\d+?.\\d+)... //puntos del objeto
...
```

La primera línea corresponde a la definición del marco de la simulación y corresponde a los límites por cada dimensión. Por ejemplo: $marco : ((-10.0, 10.0), (-10.0, 10.0))$. La segunda línea corresponde al intervalo de tiempo en que se debe realizar la simulación, incluidos los extremos. Por ejemplo: $timeInterval : 0.0-10.0$. La tercera línea corresponde a los cuadros por segundo de la simulación, valor necesario para las simulaciones de los algoritmos sweep & prune estáticos. Por ejemplo: $framesPerSecond : 100$. Luego de la tercera línea se generan pares de líneas para describir cada objeto de la simulación, por lo que si existen n objetos en la simulación existirá $2n$ líneas. La línea $2i + 1$ con $i = 1 \ldots n$, corresponde a las velocidades por coordenada del objeto i. Por ejemplo: $-3.5\ 4.5$. La línea $2i + 2$ con $i = 1 \ldots n$, corresponde a las coordenadas de los puntos. Si hay k puntos del polígono, el formato será: $k\ x_1\ y_1\ x_2\ y_2\ \ldots x_k\ y_k$. El primer valor de la línea corresponde a la cantidad de puntos del polígono, y los siguientes $2k$ números corresponden a los valores de las coordenadas de los puntos. Por ejemplo: $3\ 0.0\ 0.0\ 2.0\ 0.0\ 1.0\ 3.0$.

Una vez creado el archivo, se ejecutan las simulaciones de los tres algoritmos, mediante la ejecución de los objetos de las clases **SimulationStaticDirectSweepAndPrune**, **SimulationStaticIncrementalSweepAndPrune** y **SimulationKinetic**. En cada una de las clases anteriores el método **exec** es el que permite la ejecución del algoritmo.

El hecho que las distintas simulaciones revisan distintos aspectos de los tres algoritmos a comparar no hizo posible crear una clase única para la ejecución automática y repetitiva de los experimentos. Además, la recopilación de tiempos de ejecución en cada experimento genera un archivo XML en vivo, que debió ser almacenado a mano antes de realizar la siguiente ejecución. Si uno quisiera repetir los experimentos realizados, se debe utilizar las clases **ModelFileReader** y **ModelFileWriter** las cuales no son usadas en estas simulaciones y su código se encuentra en etapa de desarrollo.

4.2.2. TestSPDirecto: prueba del método sweep and prune estático directo

Este escenario de prueba para el método estático revisa colisiones entre cuatro grupos de objetos:

- Un cuadrado y un hexágono. El proceso retorna que el objeto 0 (cuadrado) choca con el objeto 1 (hexágono) sin determinar en detalle qué vértice(s) del cuadrado colisiona(n) con qué vértice del hexágono. Una imágen del escenario está dado en la figura 4.1.
- Un triángulo y un cuadrado. El proceso retorna que el objeto 0 (triángulo) choca con el objeto 1 (cuadrado). Se incluye la revisión de si existe un choque real entre vértices de ambos objetos, mediante el uso de un algoritmo de detección de colisiones de fase angosta. El resultado es que no existe una verdadera colisión entre los objetos. Se muestra la imágen del escenario en la figura 4.2.
- Dos estrellas regulares, una de 4 puntas y una de 6 puntas. Se incluye la revisión detallada de si existe un choque real entre vértices de ambos objetos, cosa que sucede efectivamente.
- Una punta de flecha y un rectángulo. Se incluye la revisión detallada si existe un choque real entre vértices de ambos objetos. Se muestra una imagen del escenario en la figura 4.3.

Figura 4.1: Imagen artificialmente creada en gnuplot mostrando la prueba de colisión entre un cuadrado y un hexágono

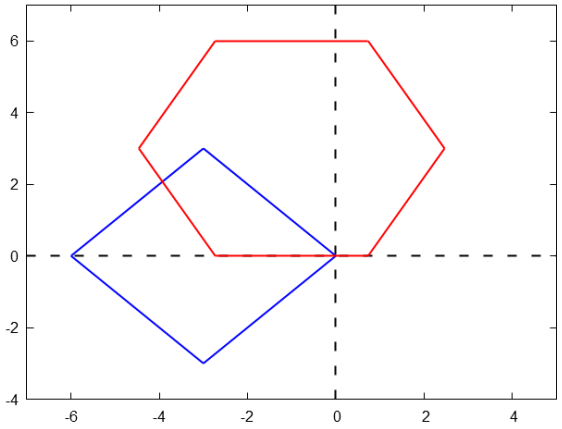


Figura 4.2: Imagen artificialmente creada en gnuplot mostrando la prueba de colisión entre un triángulo y un cuadrado

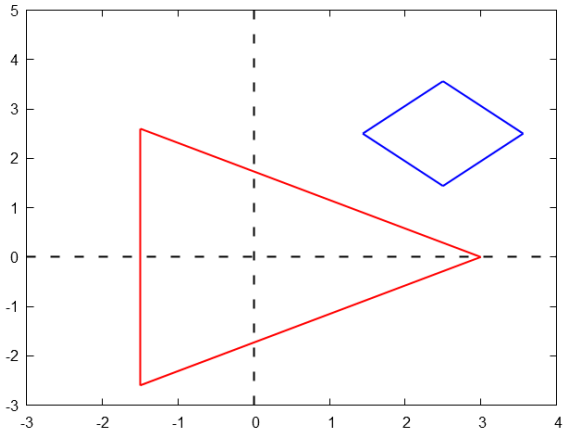
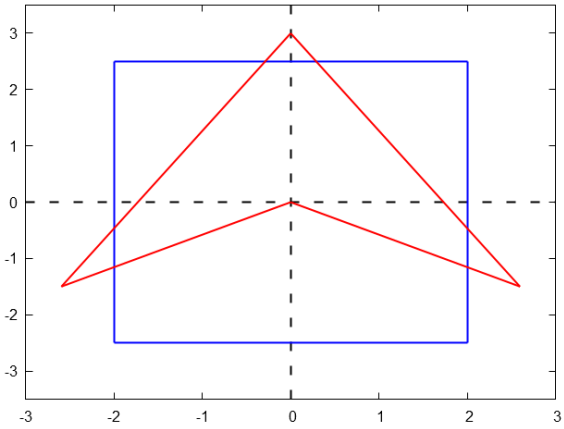


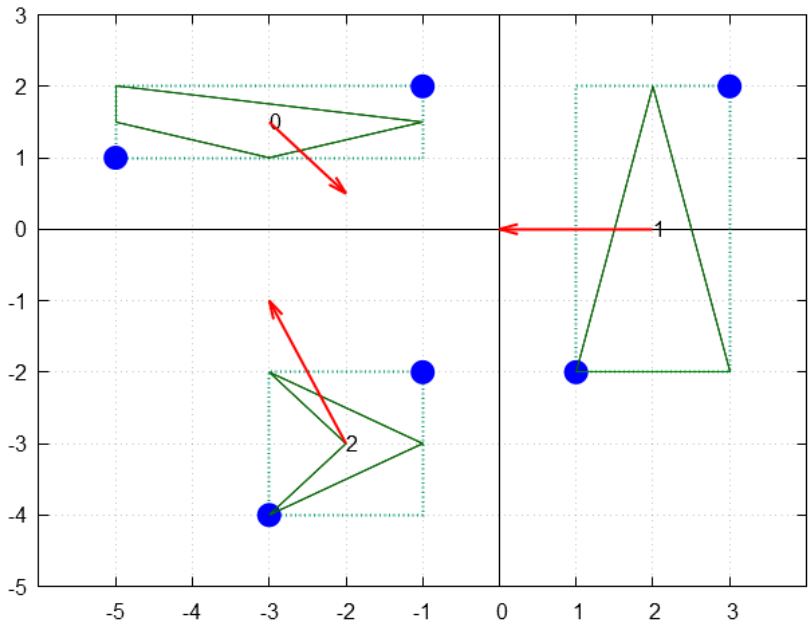
Figura 4.3: Imagen artificialmente creada en gnuplot mostrando la prueba de colisión entre una punta de flecha y un rectángulo



4.2.3. TestTresObjetos: tres objetos con trayectoria bien determinada

Este escenario se realizó para probar el correcto funcionamiento de la biblioteca **kineticLib**. Contiene tres polígonos, uno de ellos no convexo, en movimiento rectilíneo (ver figura 4.4), que se desplazan a lo largo de un área cuadrada de 20x20 con un tiempo de simulación de 10 segundos. Por motivos de rendimiento no se permitió que los objetos rebotaran en la simulación, por lo que desde el inicio poseen una trayectoria en línea recta y no cambian su dirección de desplazamiento. El ejercicio es importante porque muestra 17 eventos a lo largo de la simulación, uno de los cuales es un evento múltiple (ver figura 4.6c).

Figura 4.4: Imagen artificialmente creada en gnuplot mostrando las posiciones iniciales de los objetos y sus cajas envolventes (AABB2D). Las flechas de color rojo indican el vector desplazamiento



Este escenario integra además la generación de un archivo para el análisis estático y otro para el análisis dinámico. Se graban en un archivo las posiciones iniciales de cada punto para cada objeto además de su vector velocidad, logrando así que la situación para las tres simulaciones sea la misma.

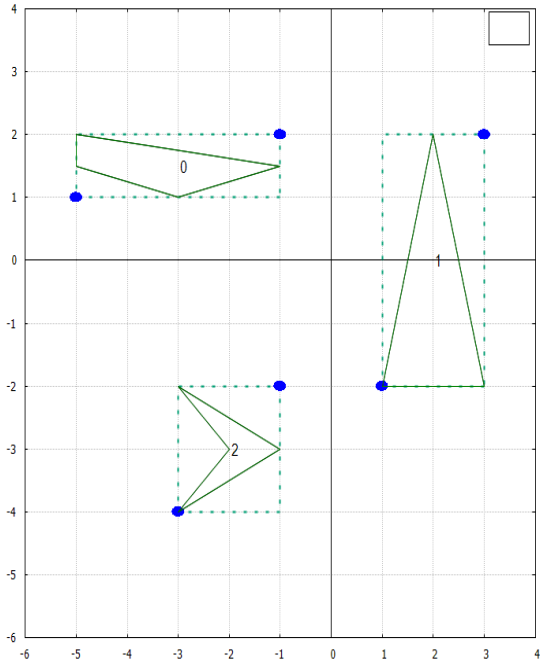
Después de generar los archivos se ejecutan las simulaciones, de las que se guardan los resultados de las colisiones en un archivo de salida, debido a que se pide la reproductibilidad de las simulaciones. En el caso cinético se genera un consolidado de los eventos.

En el método estático se calcula la posición de cada objeto en base al tiempo actual y su velocidad. Sea p_i, v_i la posición y velocidad respectivamente, de un objeto en la coordenada i y, sea t_k y t_{k-1} el intervalo de tiempo actual y anterior respectivamente:

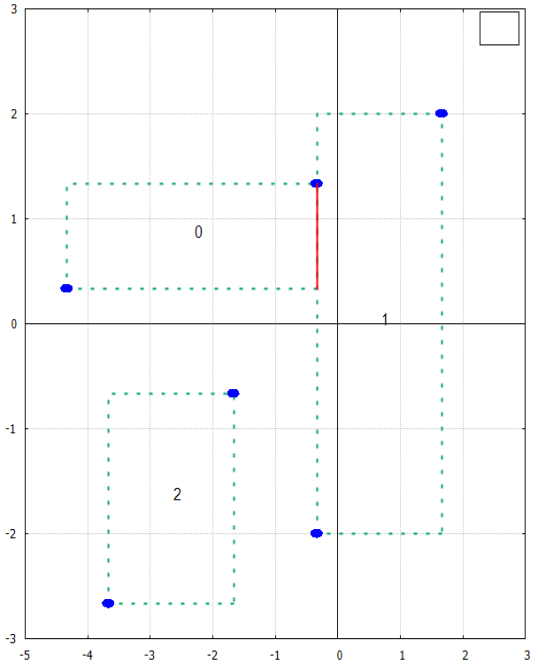
$$p_i\left(t_k\right) = v_i\left(t_k - t_{k-1}\right) + p_i\left(t_{k-1}\right) \tag{4.1}$$

La diferencia $\Delta t = t_k - t_{k_1}$, nuestro intervalo de tiempo, llamado *timestep* dentro de la aplicación y corresponde a un valor constante. En cada instante de tiempo, dado por $k \cdot \Delta t$, se graba en un archivo las posiciones de cada punto de cada objeto. Las figuras 4.5 a 4.8 muestran los tiempos en donde se producen los eventos.

Figura 4.5: Secuencia de imágenes artificialmente creadas en gnuplot, mostrando los eventos del escenario 2

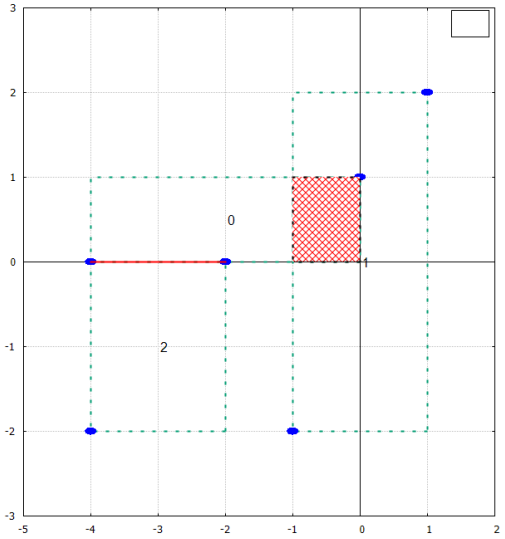


(a) Configuración inicial $t = 0$.

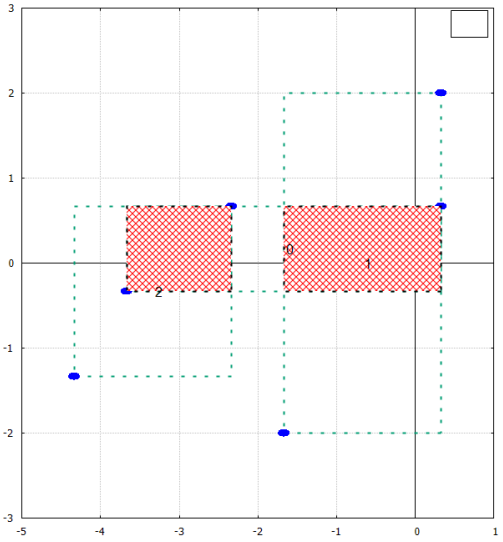


(b) Posiciones de las cajas en t=$\frac{2}{3}$, momento en que sucede la primera colisión.

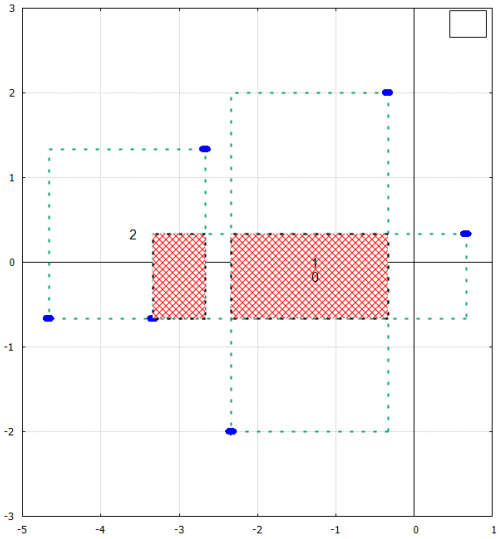
Figura 4.6: Secuencia de imágenes artificialmente creadas en gnuplot, mostrando los eventos del escenario 2, para un tiempo entre $t = 1$ y $t = 2$. En f), en $t = 2$, ocurre un evento múltiple.



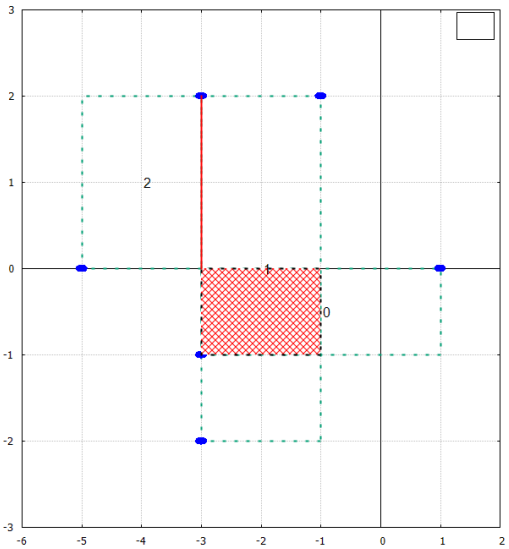
(c) $t = 1$



(d) $t = \frac{4}{3}$

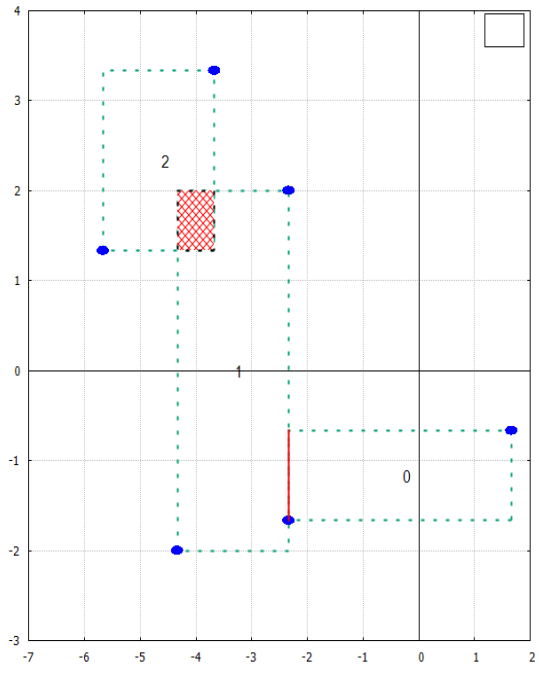


(e) $t = \frac{5}{3}$

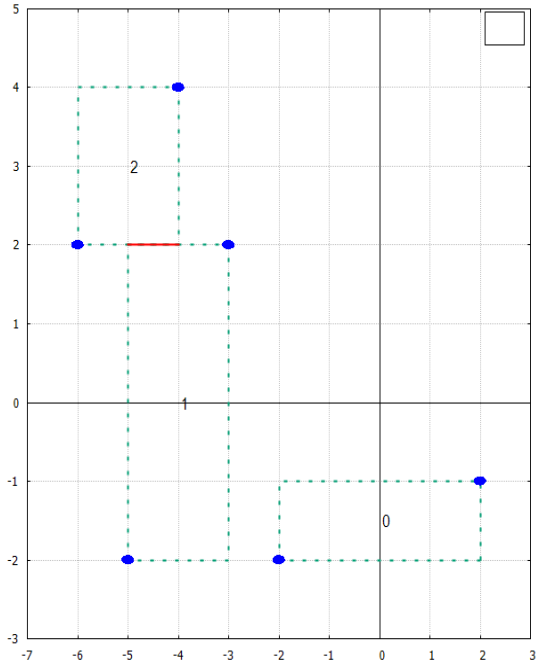


(f) $t = 2$. Evento múltiple entre los objetos 0,1 y 2 en $x = -3$

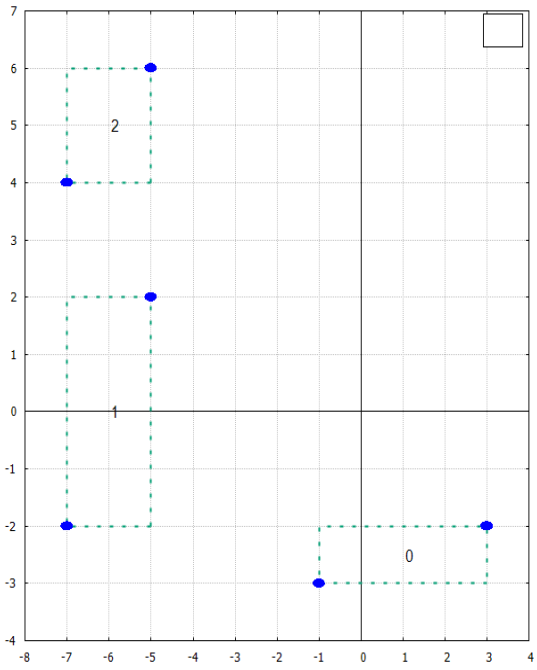
Figura 4.7: Secuencia de imágenes mostrando los eventos del escenario 2, entre $t = \frac{8}{3}$ y $t = \frac{11}{2}$



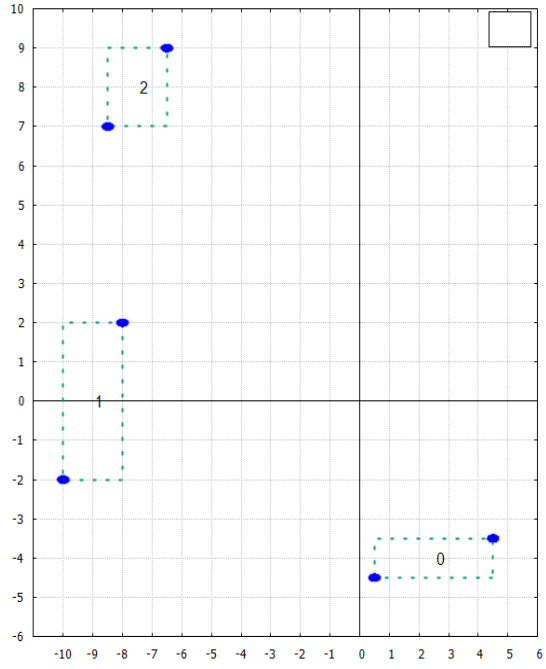
(g) $t = \frac{8}{3}$



(h) $t = 3$

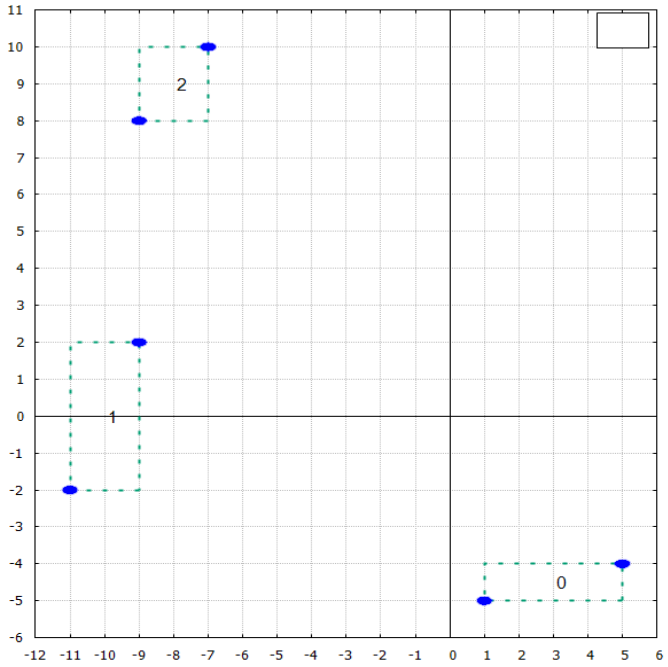


(i) $t = 4$



(j) $t = \frac{11}{2}$

Figura 4.8: $t = 6$, momento donde ocurre el último evento entre el objeto 1 y el 2



4.2.4. TestAleatorio: múltiples objetos generados en posiciones y con trayectorias rectilíneas aleatorias

Para esta simulación se implementó un programa que crea los polígonos mediante un generador aleatorio de objetos (utilizando el patrón *factory method*), que incluye las velocidades de desplazamiento en las dos dimensiones. Se debe incluir en la simulación como parámetro el porcentaje del área inicial que cubrirán las figuras, la cantidad de objetos en la simulación y el intervalo de tiempo.

Se optó por este método debido a la posibilidad de reproducir todos los resultados generados. Dentro de las opciones de polígonos generados del factory method existen: triángulo, cuadrado, pentágono, hexágono, estrellas de 5 o 6 puntas, flecha tipo boomerang y la posibilidad de generar polígonos2D personalizados. Cada uno de los polígonos anteriores está definido dentro del círculo unitario y posee como parámetro un factor de escala, si son regulares o no (algunos tienen esa opción), y si se puede rotar una cantidad de radianes iniciales al azar. Además, existe la posibilidad de desplazar estos polígonos. Los datos iniciales de la simulación necesarios son:

- Las dimensiones por cada coordenada, es decir, el marco por donde se moverán los objetos.
- La duración en de la simulación, en segundos.
- La cantidad de objetos.
- El paso temporal entre cuadro y cuadro (*timestep*).
- Por cada objeto se indica la velocidad en cada uno de los ejes, la cantidad de puntos y las coordenadas iniciales de cada uno de ellos.

Se ejecuta la simulación y se genera un archivo por cada paso temporal. Se procesa cada archivo con el método Sweep and Prune estático y se guardan las estadísticas en un archivo para su posterior análisis. En el caso cinético se lee el archivo inicial y se generan los objetos cinéticos iniciales necesarios para la simulación del sweep and prune cinético y finalmente se graba un consolidado con la estadística.

4.3. Mediciones de desempeño

El desempeño de ambos métodos (estático y cinético) se comparó en base al tiempo de ejecución. Para tales efectos se utilizó el profiler de Netbeans para Java (similar profiler existe para Eclipse). En este caso particular se utilizó el profiler que viene con Netbeans 8.1. La comparación en base a la memoria consumida no se pudo realizar debido a que requiere la comprensión y manejo del stack de memoria de Java, calculando en cada paso importante de la simulación la diferencia de memoria utilizada en el stack. Por otro lado, el uso de memoria por instancias de clases en Java no existe de manera explícita, por lo que en muchos casos, sólo se conoce la referencia al objeto en cuestión y no el tamaño que utiliza. Como ejemplo se muestra a continuación el tiempo de ejecución de ambos algoritmos sweep and prune en el escenario 2 de la sección 4.2.3.

4.3.1. Tiempos de ejecución

El profiler de Netbeans es utilizado profesionalmente como herramienta para descubrir problemas de desempeño de aplicaciones hechas en Java. Posee varios modos de operación:

- **Telemetry**: el cual permite revisar el desempeño de la aplicación on-line (figura 4.9), incluso de aplicaciones de uso remoto.

- **Methods**: para medir el tiempo de ejecución de una aplicación y conteo de invocaciones, incluídos árboles de llamados. Esta es la opción elegida para medir el desempeño de los algoritmos debido a que permite seleccionar en qué métodos y en qué clases medir el tiempo transcurrido y/o el timestamp del momento en que la ejecución alcanza un punto de medición. El único factor en contra es que solo mide con un máximo de 2 dígitos decimales valores menores a 10 [ms], perdiendo precisión para valores mayores a 100[ms]

- **Objects**: me permite medir el tamaño y cantidad de los objetos involucrados en la simulación. Esta opción tiene en contra que con objetos complejos, como colecciones, sólo establece el tamaño de la referencia a esa variable y no el tamaño real de la colección, razón por la que no se utilizó esta herramienta.

Figura 4.9: Profiler de Netbeans en su opción telemetría

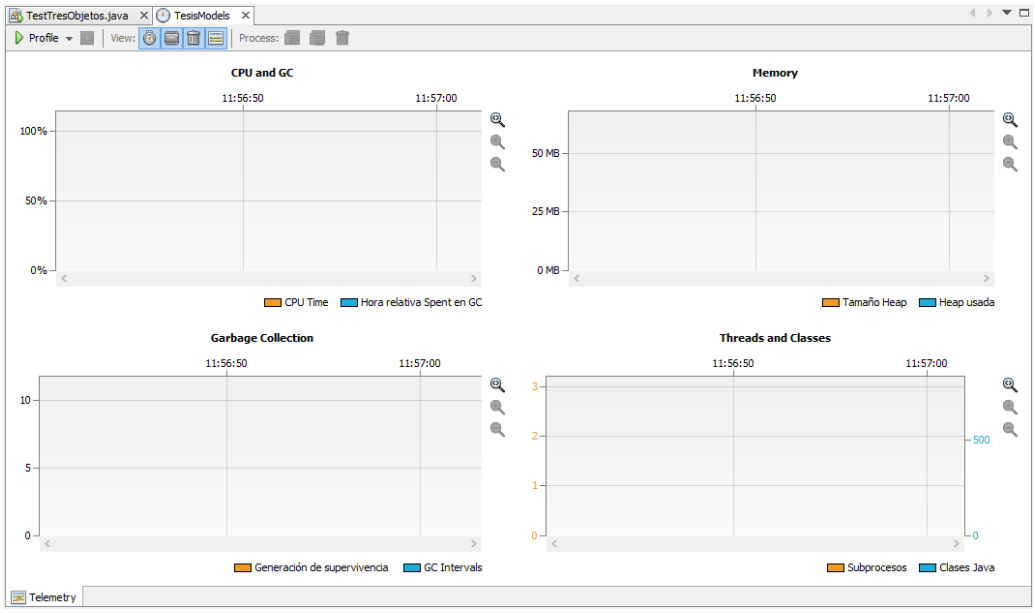
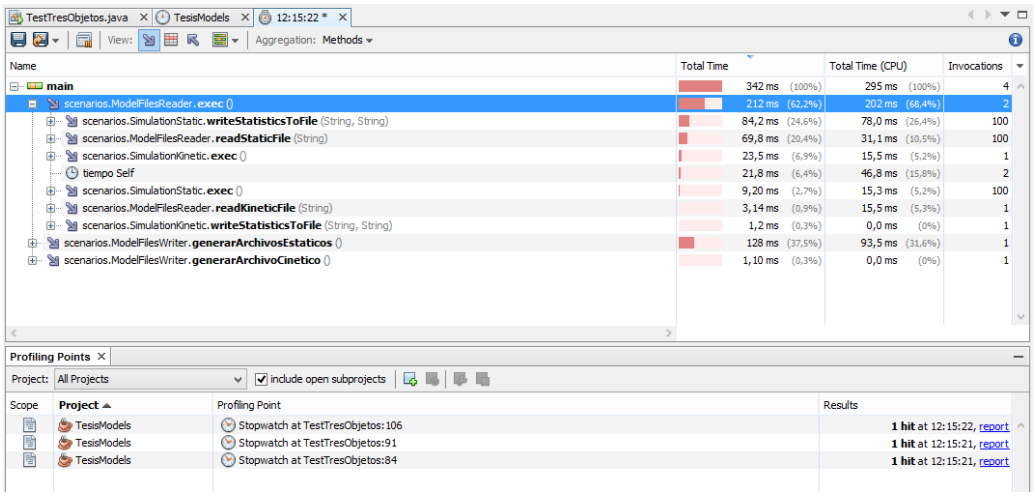


Figura 4.10: Profiler de Netbeans en su opción Methods



Para los experimentos aleatorios se registró el tiempo de ejecución de los algoritmos, incluídos los tiempos de inicialización de las distintas estructuras y el tiempo de creación de los archivos con los resultados.

4.4. Resultados

Se realizó las simulaciones de 4.2.4 con una cantidad variable de objetos dentro de la escena: 1, 2, 5, 10, 20, 50, 100, 200, 500 y 1000 objetos iniciales; y una ocupación (o densidad) variable, representada como un porcentaje del área total de la escena, con: 1 %, 2 %, 5 % y 10 %. Todos los experimentos de esta simulación fueron realizados con 1000 cuadros por segundo (FPS), con un tiempo total de simulación de 1 segundo. Las velocidades de los objetos fueron direcciones aleatorias con magnitudes entre 0.3 y 3.5 veces el valor mínimo entre el ancho y el alto del AABB que rodea al polígono. El tamaño de cada polígono (en área de ocupación) está definido por el porcentaje de ocupación de la escena multiplicado por el área de la escena y dividido por la cantidad de objetos.

4.4.1. Sweep and Prune directo

En las figuras 4.11 a 4.18 se tiene las curvas de tiempo de ejecución del mejor y peor desempeño del algoritmo Sweep and Prune directo para distintos porcentajes de ocupación y cantidad de objetos. Cada figura incluye la mejor curva ajustada de acuerdo al método

de mínimos cuadrados no lineales de Levenberg-Marquad que posee MatlabR2014a®[3].
El algoritmo Sweep and Prune directo tiene un tiempo de ejecución de $O(n^2)$ debido a que la etapa de ordenamiento de los AABB utiliza el método insertionSort en cada cuadro. La otra etapa crítica dentro del algoritmo es la obtención de los pares que efectivamente colisionan, la que fue calculada mediante el uso de los métodos *addAll* y *retainAll* que operan sobre colecciones en Java.
Dentro de los análisis realizados se encuentra la comparación de la curva de ajuste que calza mejor con los datos experimentales para los procesos de ordenamiento y cálculo de colisiones, como una manera de justificar el orden de los desempeños de los algoritmos, los que pueden observarse en el anexo B.5. Para ello, se calcularon las estadísticas de suma de los errores al cuadrado (SSE), R^2, Grados de libertad de R^2 ajustado(dfe), R^2 ajustado y Error cuadrático medio (RMSE). Además se compara los valores de los parámetros obtenidos por el método de mínimos cuadrados de acuerdo a [1]. De acuerdo a [1] la forma de poder comparar las distintas curvas generadas es en base a las estadísticas de SSE y R^2_{adj} además de la observación de los intervalos de confianza de los parámetros de cada curva. Se detallará para este primer caso, el comportamiento de la curva promedio para el método de ordenamiento con el 1 % de densidad de objetos, y los demás se tratarán de manera análoga.
Se puede observar en la tabla B.2 que para el ajuste cuadrático, los valores del intervalo de confianza del coeficiente a **no cruzan** por 0, lo que sugiere que para el proceso de cálculo de colisiones, el ajuste es de tipo **cuadrático**. De acuerdo a la tablaB.9, se puede observar que los valores de SSE se van reduciendo en la medida que se va subiendo el orden de la ecuación de ajuste, sin embargo, el valor de R^2_{adj} sufre poca variación desde el ajuste lineal hacia arriba. Para elegir la curva que mejor ajusta con los datos se debe revisar los coeficientes, y así evitar curvas de sobreajuste. En la tabla B.10 Se puede observar que para el ajuste subcuadrático, los valores del intervalo de confianza del coeficiente a cruzan por 0, lo que sugiere que esta curva es un sobreajuste del modelo. Esto significa que la curva de mejor ajuste para el promedio de los datos es la lineal.

[3] `http://www.mathstat.dal.ca/cluster/manuals/matlab/toolbox/curvefit/ch_fitt5.html`

Figura 4.11: Resultados de S&P directo con ocupación del 1 % para curva del máximo en los experimentos.

El ajuste cuadrático de la curva superior obtenida corresponde a $0.0074x^2 + 75.8604x\log(x) - 128.4384x + 813.6912\log(x) + 145.9597$, con un $SSE = 9.8868 * 10^6$, $R^2 = 0.9999$, $dfe = 5$, $R^2_{adj} = 0.9999$ y $RMSE = 1.4062 * 10^3$, mientras que el ajuste superlineal corresponde a $82.7x\log(x) - 170x + 1024.2\log(x) + 72.5$ con un $SSE = 1.0762 * 10^7$, $R^2 = 0.9999$, $dfe = 6$, $R^2_{adj} = 0.9999$ y $RMSE = 1.3393 * 10^3$.

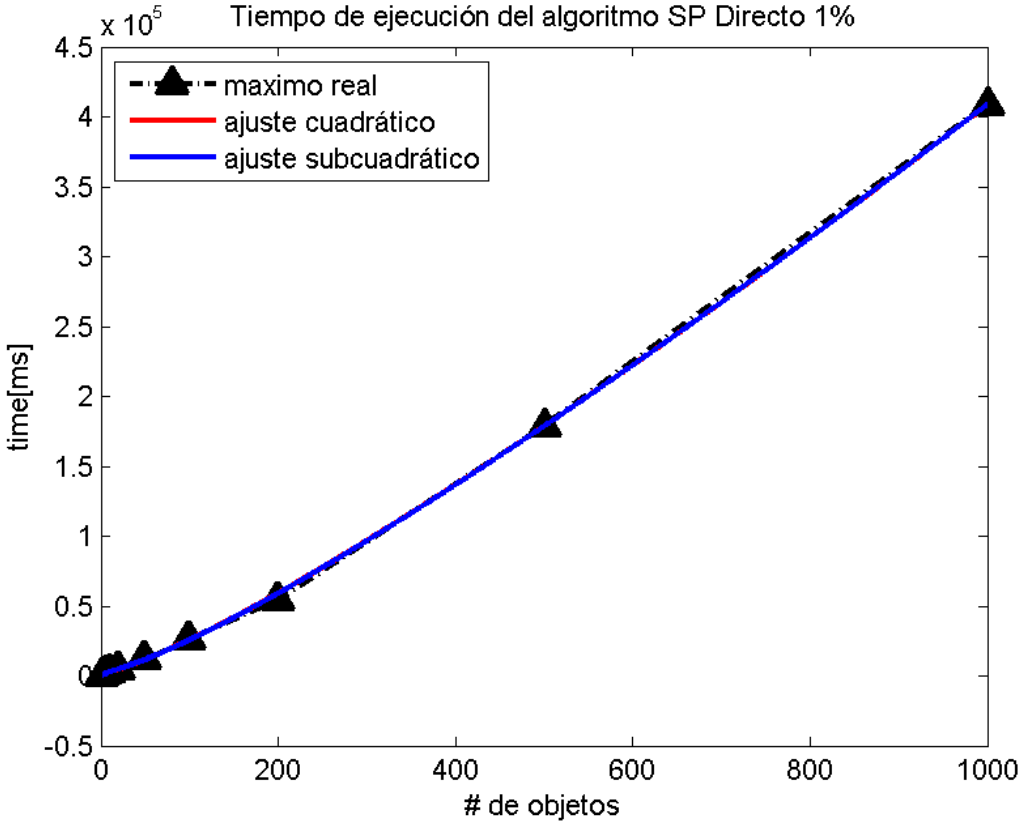


Figura 4.12: Resultados de S&P directo con ocupación del 1 % para curva del mínimo en los experimentos.

El ajuste cuadrático de la curva inferior obtenida corresponde a $0.0283x^2 + 64.2615x\log(x) - 97.7356x + 669.5367\log(x) - 14.8140$, con un $SSE = 5.1819 * 10^6$, $R^2 = 1.0$, $dfe = 5$, $R^2_{adj} = 0.9999$ y $RMSE = 1.018 * 10^3$, mientras que el ajuste superlineal corresponde a $89x\log(x) - 244.2x + 1239.6\log(x) - 151.2$ con un $SSE = 8.0973 * 10^6$, $R^2 = 0.9999$, $dfe = 6$, $R^2_{adj} = 0.9999$ y $RMSE = 1.1617 * 10^3$.

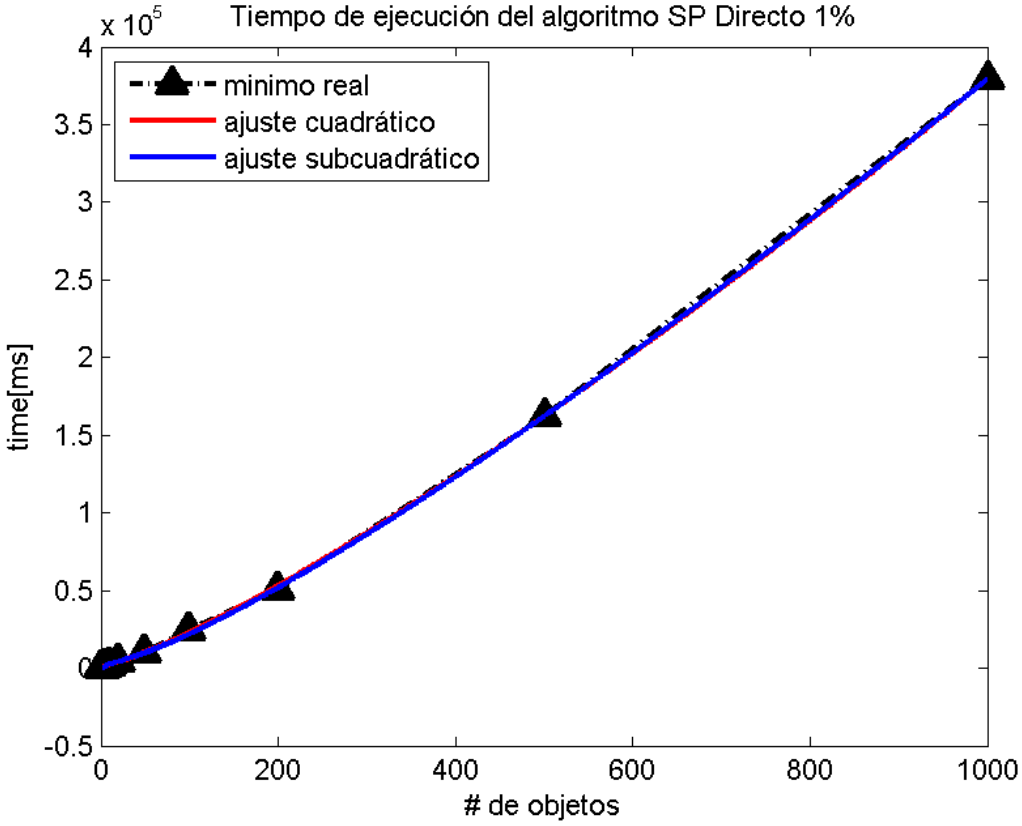
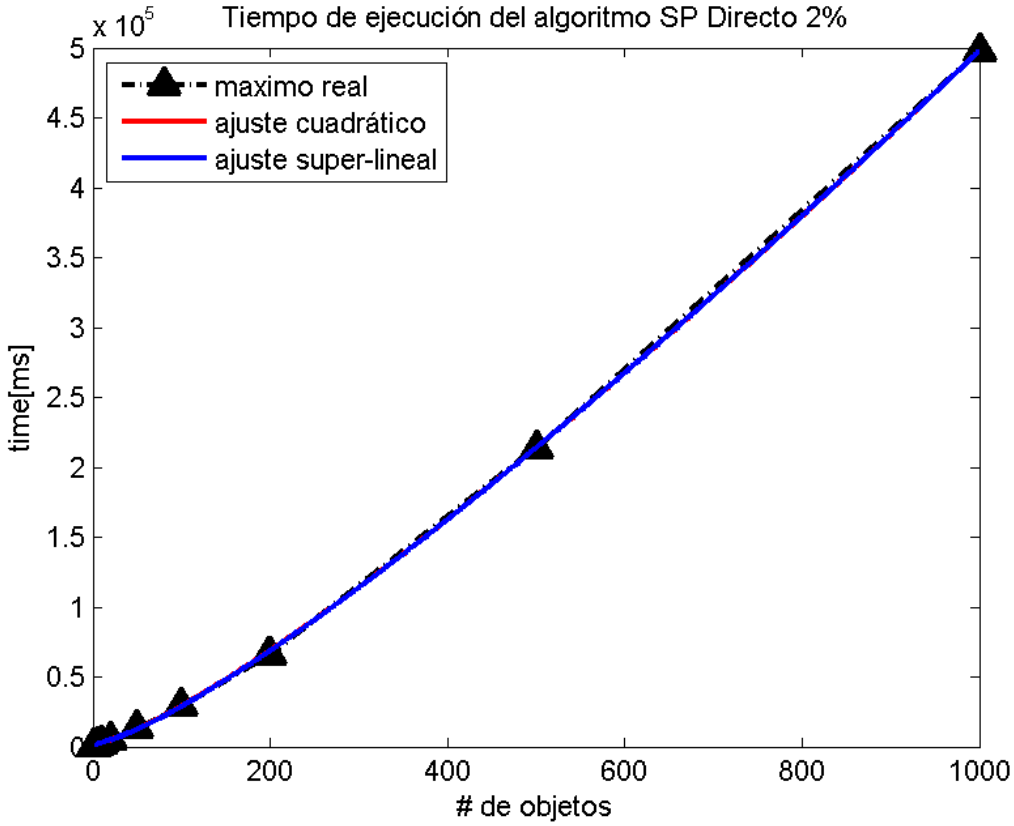


Figura 4.13: Resultados de S&P directo con ocupación del 2 % para curva del máximo en los experimentos.

El ajuste cuadrático de la curva superior obtenida corresponde a $-0x^2+116.6x\log(x)-327.7x+1154.2\log(x)+86.9$, con un $SSE = 4.725*10^6$, $R^2 = 1.0$, $dfe = 5$, $R^2_{adj} = 1.0$ y $RMSE = 972.1157$, mientras que el ajuste superlineal corresponde a $95.2999x\log(x)-214.7624x+722.6294\log(x)+387.3442$ con un $SSE = 9.0829*10^6$, $R^2 = 1.0$, $dfe = 6$, $R^2_{adj} = 0.9999$ y $RMSE = 1230.4$.



Para las colisiones con un 2 % de ocupación, según la tabla B.3, no es necesario ver el ajuste de los coeficientes, ya que con un ajuste subcuadrático se tiene un calce perfecto con los datos ($R^2_{adj} = 1$). Para el algoritmo de ordenamiento se puede observar en la tabla B.12 que, tanto para el ajuste cuadrático, los valores del intervalo de confianza del coeficiente a cruzan por 0, lo que sugiere que esta curva es un sobreajuste del modelo. En este caso, la curva de mejor ajuste es subcuadrática.

Figura 4.14: Resultados de S&P directo con ocupación del 2 % para curva del mínimo en los experimentos.

El ajuste cuadrático de la curva inferior obtenida corresponde a $0.0283x^2 + 64.2615x\log(x) - 97.7356x + 669.5367\log(x) - 14.8140$, con un $SSE = 5.1819 * 10^6$, $R^2 = 1.0$, $dfe = 5$, $R^2_{adj} = 0.9999$ y $RMSE = 1.018 * 10^3$, mientras que el ajuste superlineal corresponde a $89x\log(x) - 244.2x + 1239.6\log(x) - 151.2$ con un $SSE = 8.0973 * 10^6$, $R^2 = 0.9999$, $dfe = 6$, $R^2_{adj} = 0.9999$ y $RMSE = 1.1617 * 10^3$.

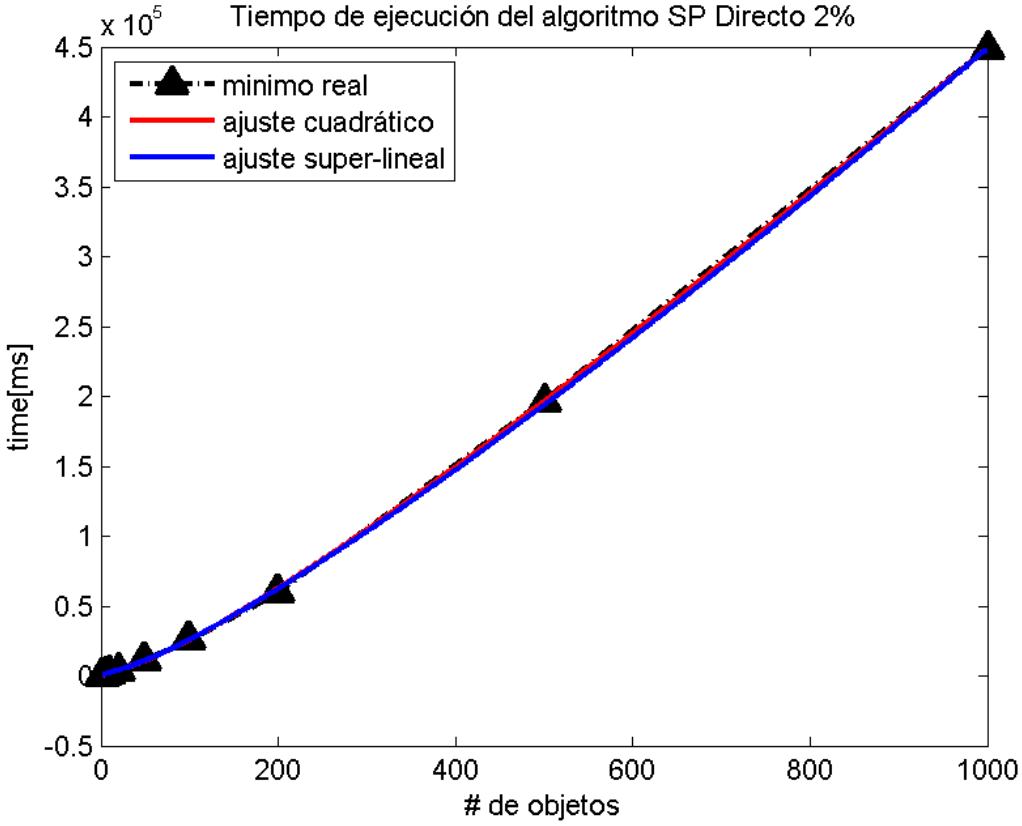
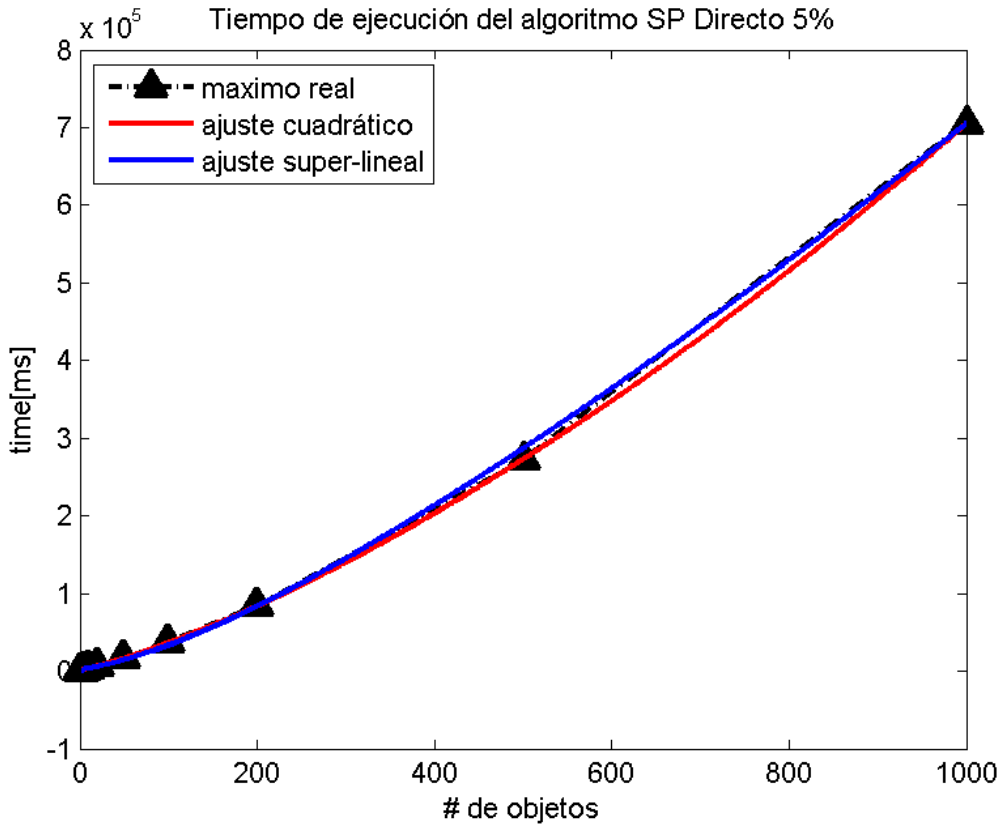


Figura 4.15: Resultados de S&P directo con ocupación del 5 % para curva del máximo en los experimentos.

El ajuste cuadrático de la curva superior obtenida corresponde a $0.2257x^2 + 73.6558x\log(x) - 34.4712x + 682.7995\log(x) + 268.9477$, con un $SSE = 2.9425 * 10^5$, $R^2 = 1.0$, $dfe = 5$, $R^2_{adj} = 1.0$ y $RMSE = 242.5905$, mientras que el ajuste superlineal corresponde a $215.1x\log(x) - 802.6x + 3166.6\log(x) - 124.9$ con un $SSE = 6.5756 * 10^7$, $R^2 = 0.9999$, $dfe = 6$, $R^2_{adj} = 0.9999$ y $RMSE = 3.3105 * 10^3$.



Para el algoritmo de colisiones, en la tabla B.6, dado que entre la curva subcuadrática y la cuadrática, al agregar coeficientes no se obtiene ningún intervalo de confianza que cruce por 0, la curva de mejor ajuste es la cuadrática. En el algoritmo de ordenamiento, de la tabla B.13 se observa que al aumentar el grado de la ecuación de ajuste mayor a lineal, el R^2_{adj} se mantiene inalterado, pero su valor de $RMSE$ **aumenta** al agregar más coeficientes, por lo que la curva de mejor ajuste es lineal.

Figura 4.16: Resultados de S&P directo con ocupación del 5 % para curva del mínimo en los experimentos.

El ajuste cuadrático de la curva inferior obtenida corresponde a $x^2 + 103.4x\log(x) - 235.3x + 1071.9\log(x) - 106.3$, con un $SSE = 1.4043 * 10^6$, $R^2 = 1.0$, $dfe = 5$, $R^2_{adj} = 1.0$ y $RMSE = 529.959$, mientras que el ajuste superlineal corresponde a $190.9x\log(x) - 717.3x + 2849.8\log(x) - 495.4$ con un $SSE = 7.2732 * 10^7$, $R^2 = 0.9998$, $dfe = 6$, $R^2_{adj} = 0.9997$ y $RMSE = 3.4817 * 10^3$.

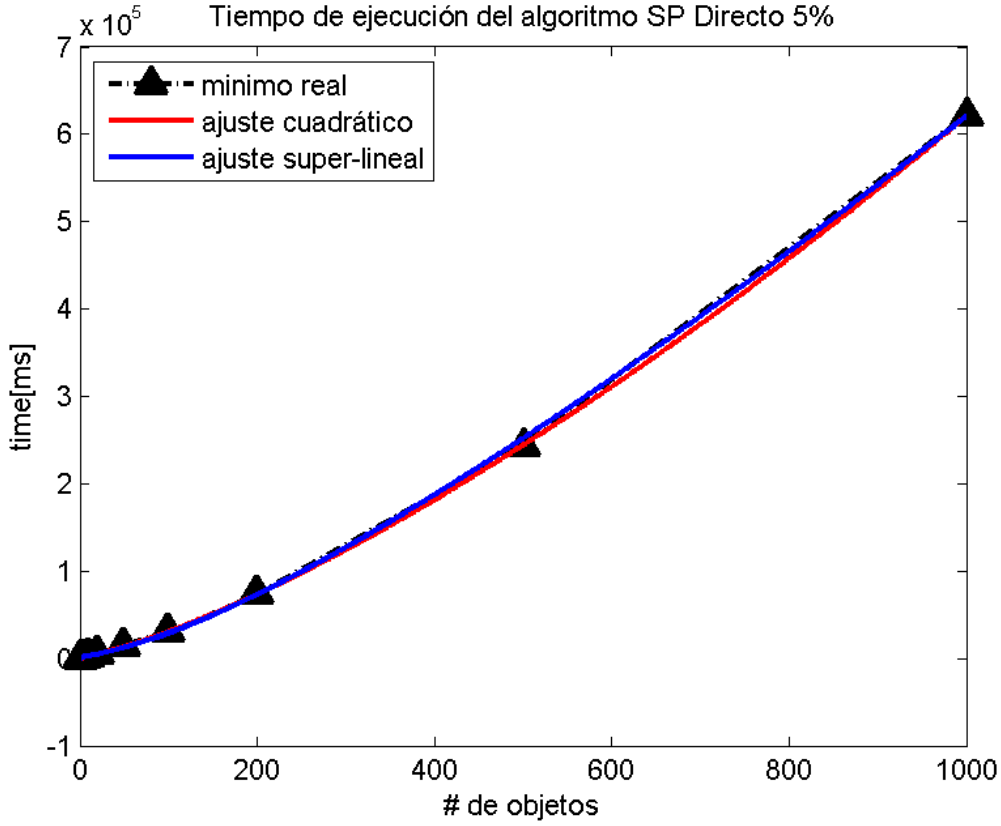
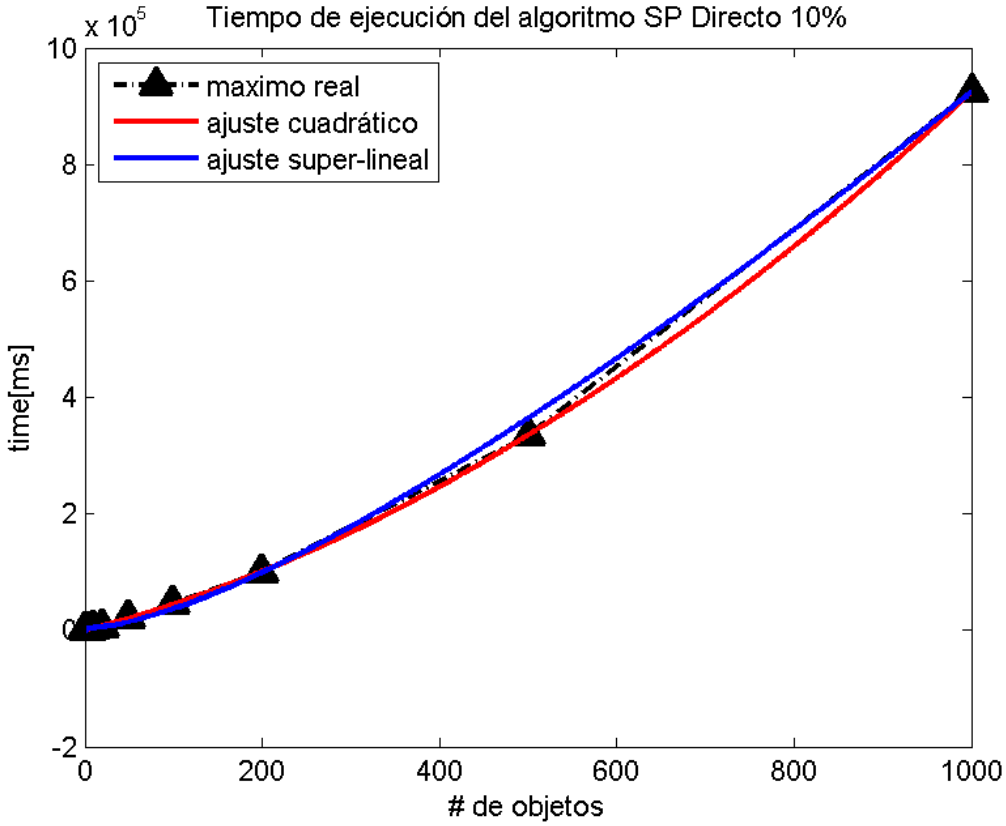


Figura 4.17: Resultados de S&P directo con ocupación del 10 % para curva del máximo en los experimentos.

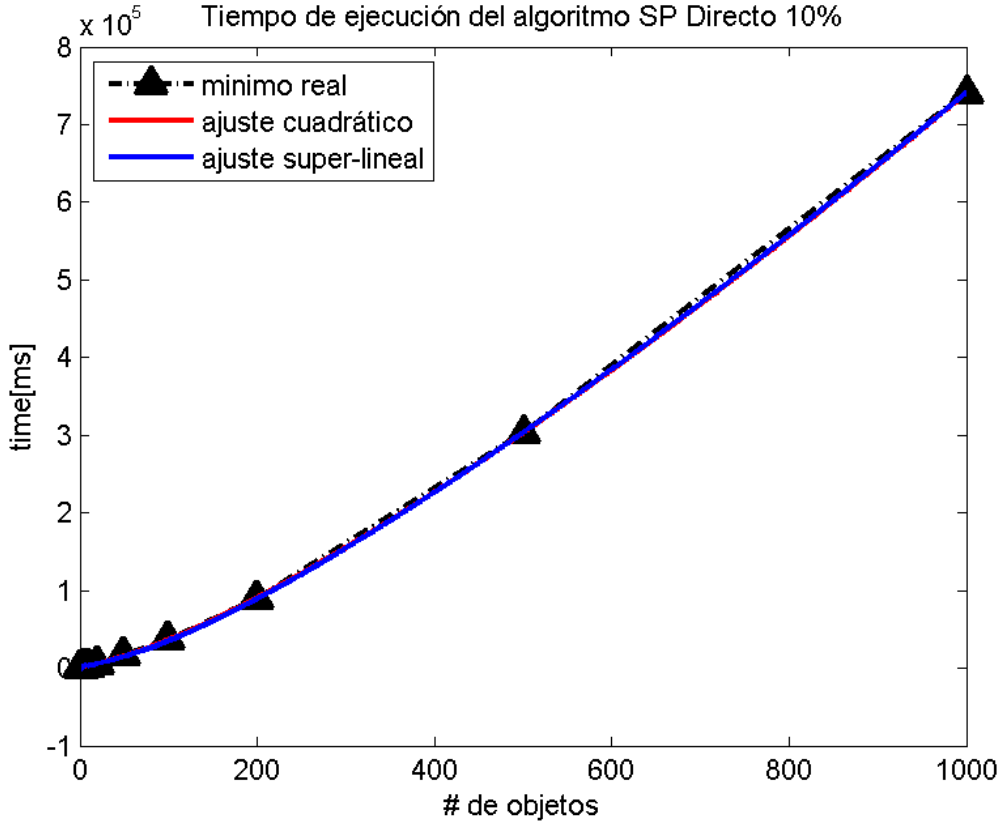
El ajuste cuadrático de la curva superior obtenida corresponde a $0.4766x^2 + 23.5727x\log(x) + 285.2343x - 23.3204\log(x) - 7.7521$, con un $SSE = 5.1849 * 10^6$, $R^2 = 1.0$, $dfe = 5$, $R^2_{adj} = 1.0$ y $RMSE = 1.0183{*}10^3$, mientras que el ajuste superlineal corresponde a $320.3x\log(x) - 1318.8x + 4630.4\log(x) - 347$ con un $SSE = 2.8411{*}10^8$, $R^2 = 0.9996$, $dfe = 6$, $R^2_{adj} = 0.9994$ y $RMSE = 6.8813 * 10^3$.



Para el algoritmo de cálculo de colisiones, la tabla B.7 muestra que se reduce el valor de RMSE y aumenta el valor de R^2_{adj} en la medida que aumenta el orden de la ecuación. Si se observa los coeficientes su intervalo de confianza, en la tabla B.8 se tiene que ninguno de los coeficientes mayores al lineal tiene un cruce por 0, por lo que la curva de mejor ajuste es la cuadrática. Para el ordenamiento, el análisis de los coeficientes de la tabla B.16 muestra que las curva subcuadrática es la que posee el mejor ajuste.

Figura 4.18: Resultados de S&P directo con ocupación del 10 % para curva del mínimo en los experimentos.

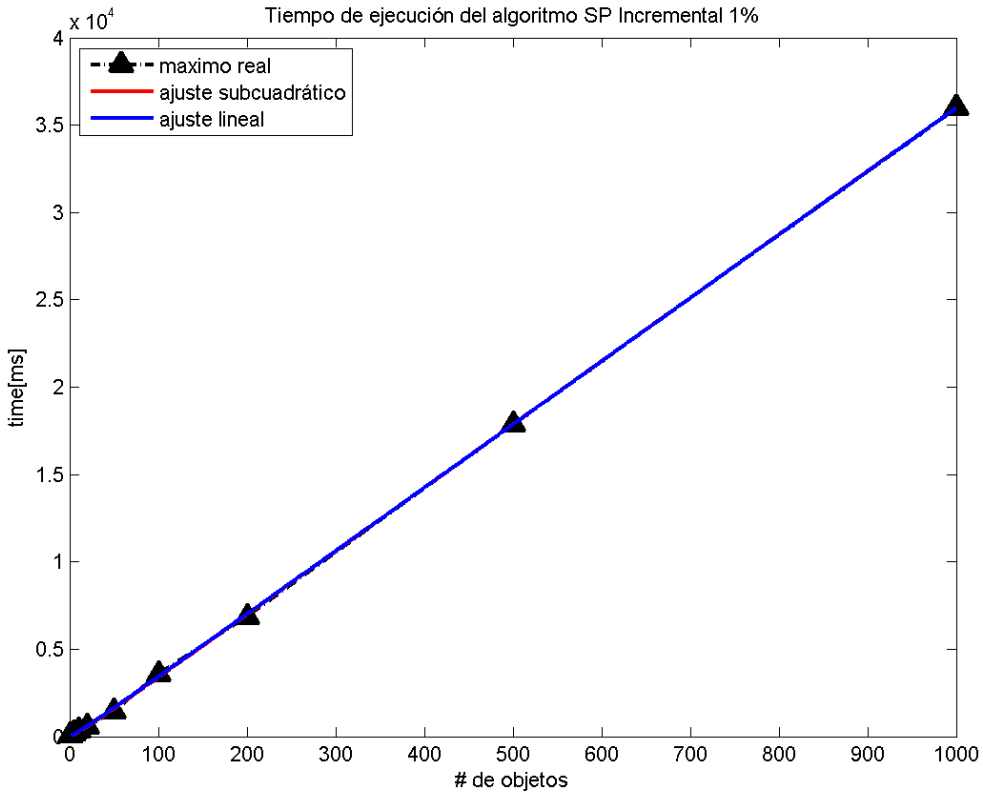
El ajuste cuadrático de la curva inferior obtenida corresponde a $0x^2 + 186.8x\log(x) - 600.7x + 2225.4\log(x) - 172.2$, con un $SSE = 6.6367 * 10^6$, $R^2 = 1.0$, $dfe = 5$, $R^2_{adj} = 1.0$ y $RMSE = 1.1521 * 10^3$, mientras que el ajuste superlineal corresponde a $216.1x\log(x) - 771.4x + 2828\log(x) - 289.1$ con un $SSE = 8.0973 * 10^6$, $R^2 = 0.9999$, $dfe = 6$, $R^2_{adj} = 0.9999$ y $RMSE = 1.3169 * 10^3$.



4.4.2. Sweep and Prune incremental

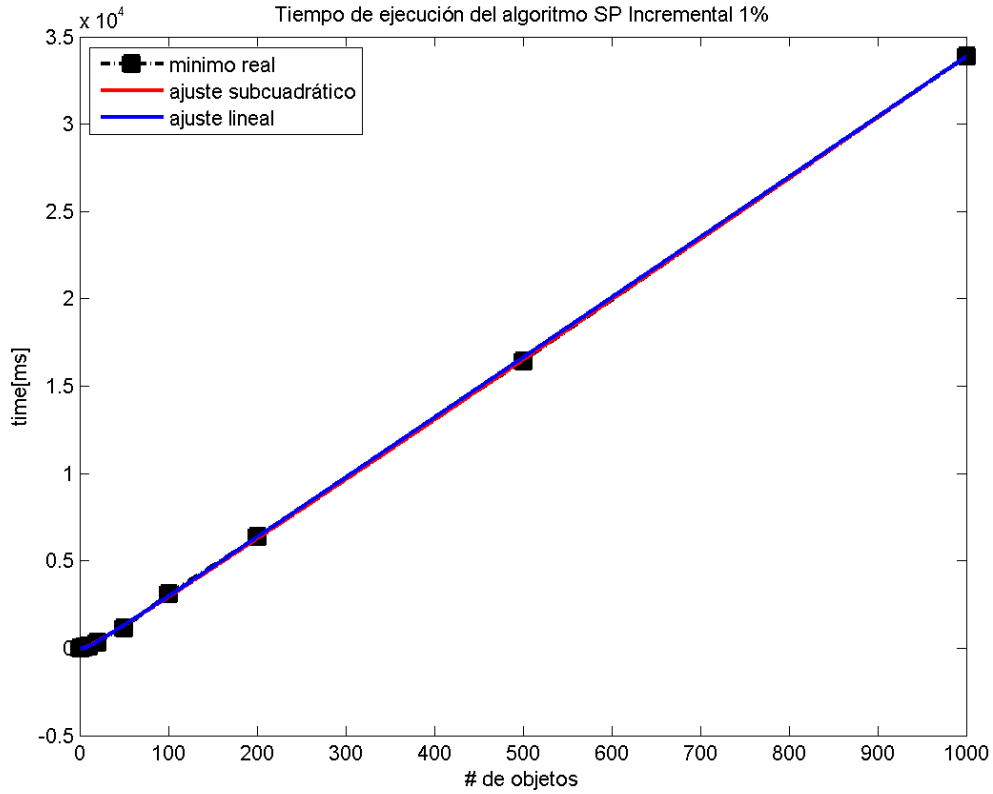
En las figuras 4.19 a 4.26 se tiene las curvas de tiempo de ejecución del mejor y peor desempeño de los algoritmos Sweep and Prune Incremental para la misma simulación, los mismos porcentajes de ocupación y cantidad de objetos de la sección 4.4.1. Cada figura incluye la mejor curva ajustada de acuerdo al mismo método de mínimos cuadrados no lineales de la sección 4.4.1.

Figura 4.19: Resultados de S&P incremental con ocupación del 1 % para curva del máximo en los experimentos %



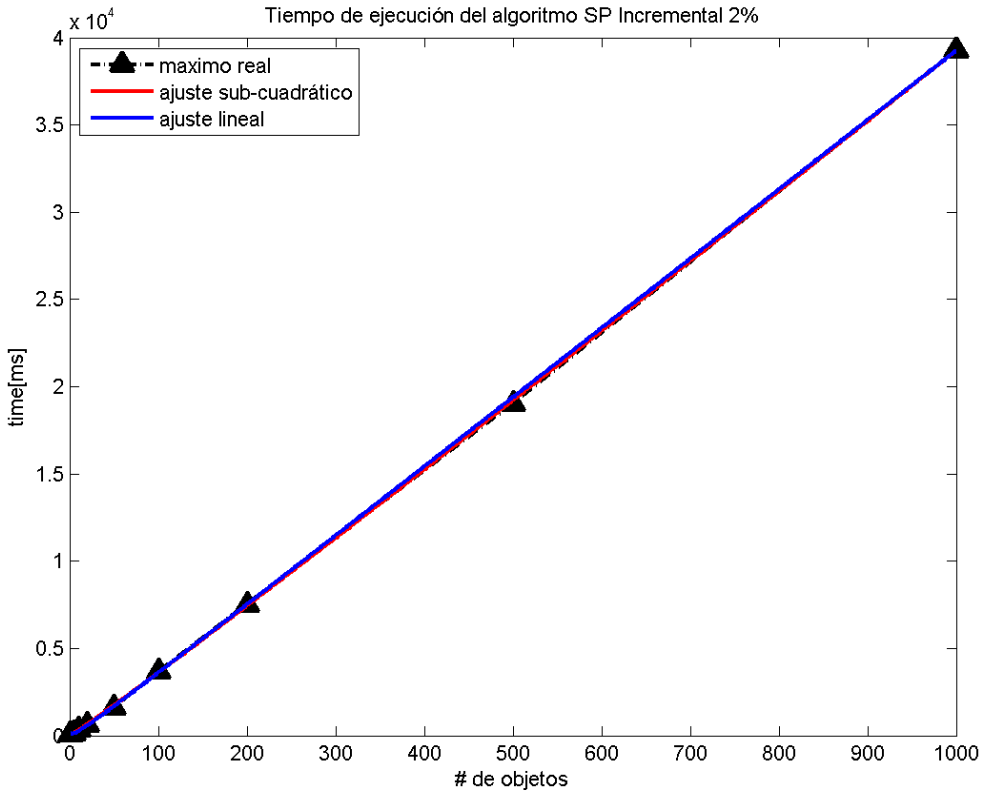
el ajuste subcuadrático corresponde a $0.2585x\log(x) + 11.2404x + 97.3676\log(x) - 12.9292$, con un $SSE = 4.2943 * 10^5$, $R^2 = 0.9974$, $dfe = 6$, $R^2_{adj} = 0.9961$ y $RMSE = 267.529$, mientras que el ajuste lineal es $13.1044x + 85.195\log(x) - 6.9128$, con un $SSE = 4.8993{*}10^5$, $R^2 = 0.9971$, $dfe = 7$, $R^2_{adj} = 0.9962$ y $RMSE = 264.5553$.

Figura 4.20: Resultados de S&P incremental con ocupación del 1 % para curva del mínimo en los experimentos %



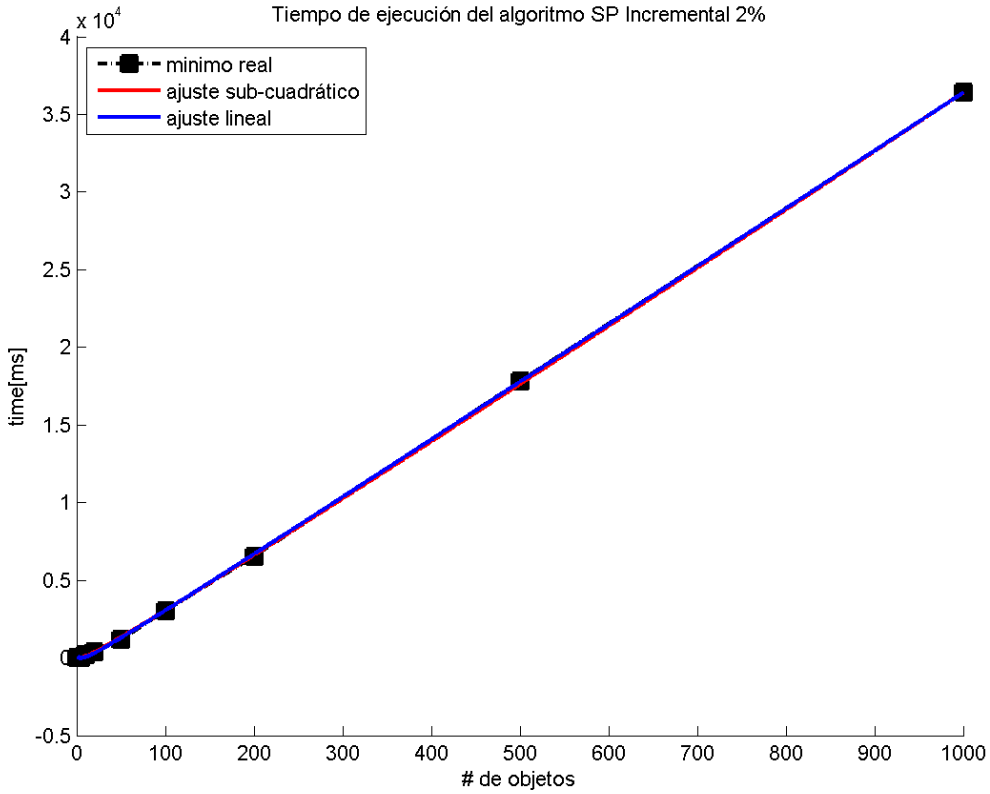
el ajuste subcuadrático corresponde a $0.2562x\log(x) + 10.6077x + 42.0309\log(x) - 50.7042$, con un $SSE = 2.6769 * 10^5$, $R^2 = 0.9981$, $dfe = 6$, $R^2_{adj} = 0.9972$ y $RMSE = 211.2219$, mientras que el ajuste lineal es $12.5345x + 14.755\log(x) - 19.41978$, con un $SSE = 3.8562 * 10^5$, $R^2 = 0.9973$, $dfe = 7$, $R^2_{adj} = 0.9965$ y $RMSE = 234.7106$.

Figura 4.21: Resultados de S&P incremental con ocupación del 2 % para curva del máximo en los experimentos %



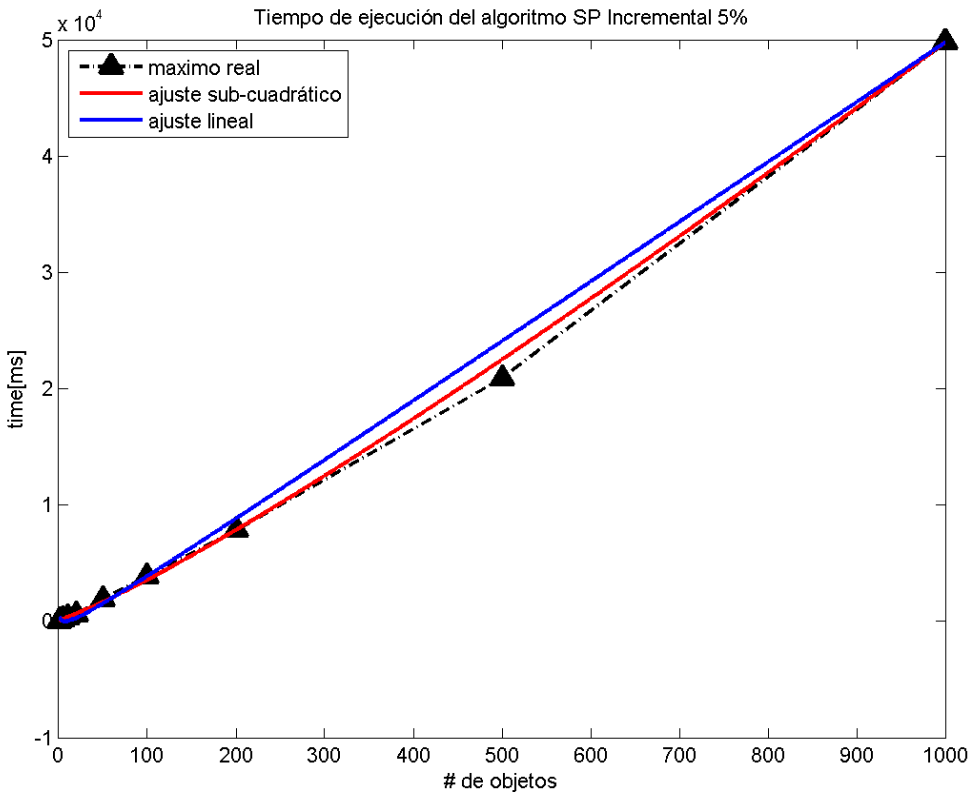
el ajuste subcuadrático corresponde a $1.0422x\log(x) + 32.2215x - 28.8233\log(x) + 38.2785$, con un $SSE = 6.8933 * 10^4$, $R^2 = 1.0$, $dfe = 6$, $R^2_{adj} = 0.9999$ y $RMSE = 107.1862$, mientras que el ajuste lineal es $39.8882x - 105.7867\log(x) + 102.5493$, con un $SSE = 1.6364 * 10^5$, $R^2 = 0.9999$, $dfe = 7$, $R^2_{adj} = 0.9999$ y $RMSE = 152.8949$.

Figura 4.22: Resultados de S&P incremental con ocupación del 2 % para curva del mínimo en los experimentos %



el ajuste subcuadrático corresponde a $0.8031x\log(x) + 31.5871x - 111.6443\log(x) + 27.5985$, con un $SSE = 6.923 * 10^4$, $R^2 = 0.9999$, $dfe = 6$, $R^2_{adj} = 0.9999$ y $RMSE = 107.4169$, mientras que el ajuste lineal es $37.4764x - 164.9238\log(x) + 53.8638$, con un $SSE = 6.8912 * 10^4$, $R^2 = 0.9999$, $dfe = 7$, $R^2_{adj} = 0.9999$ y $RMSE = 99.2197$.

Figura 4.23: Resultados de S&P incremental con ocupación del 5 % para curva del máximo en los experimentos %



el ajuste subcuadrático corresponde a $7.6289x\log(x) - 3.5362x + 82.1773\log(x) + 32.7354$, con un con un $SSE = 7.2086 * 10^5$, $R^2 = 0.9997$, $dfe = 6$, $R^2_{adj} = 0.9995$ y $RMSE = 346.6165$, mientras que el ajuste lineal es $51.8888x - 352.1811\log(x) + 306.9732$, con un $SSE = 3.9762 * 10^6$, $R^2 = 0.9982$, $dfe = 7$, $R^2_{adj} = 0.9977$ y $RMSE = 753.6765$.

Figura 4.24: Resultados de S&P incremental con ocupación del 5 % para curva del mínimo en los experimentos.

El ajuste subcuadrático corresponde a $5.9736x\log(x) + 2.2719x + 45.7061\log(x) - 44.5059$, con un $SSE = 6.0435 * 10^5$, $R^2 = 0.9997$, $dfe = 6$, $R^2_{adj} = 0.9995$ y $RMSE = 317.3721$, mientras que el ajuste lineal es $45.4477x - 255.782\log(x) + 126.1464$, con un $SSE = 2.5329 * 10^6$, $R^2 = 0.9986$, $dfe = 7$, $R^2_{adj} = 0.9988$ y $RMSE = 601.5296$.

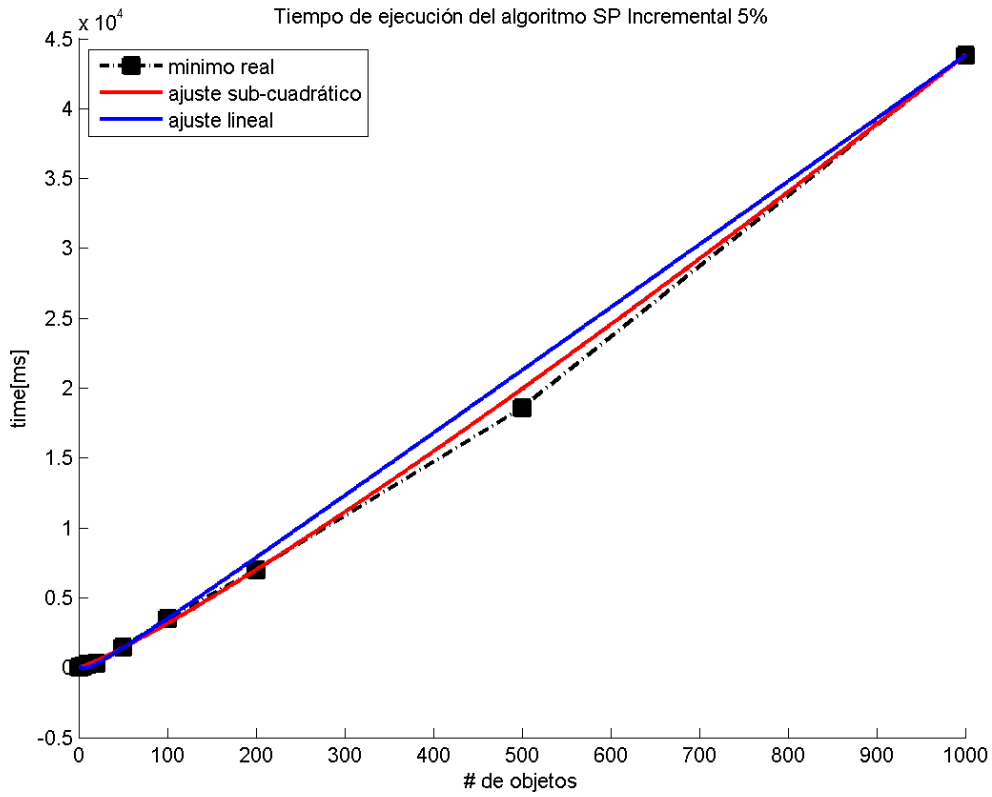


Figura 4.25: Resultados de S&P incremental con ocupación del 10 % para curva del máximo en los experimentos

El ajuste subcuadrático corresponde a $12.0604x\log(x) - 28.4675x + 178.0838\log(x) - 21.0226$, con un $SSE = 1.802 * 10^6$, $R^2 = 0.9994$, $dfe = 6$, $R^2_{adj} = 0.9990$ y $RMSE = 548.0329$, mientras que el ajuste lineal es $57.9995x - 316.8548\log(x) + 241.2166$, con un $SSE = 8.9448 * 10^6$, $R^2 = 0.9969$, $dfe = 7$, $R^2_{adj} = 0.9960$ y $RMSE = 1130.4$.

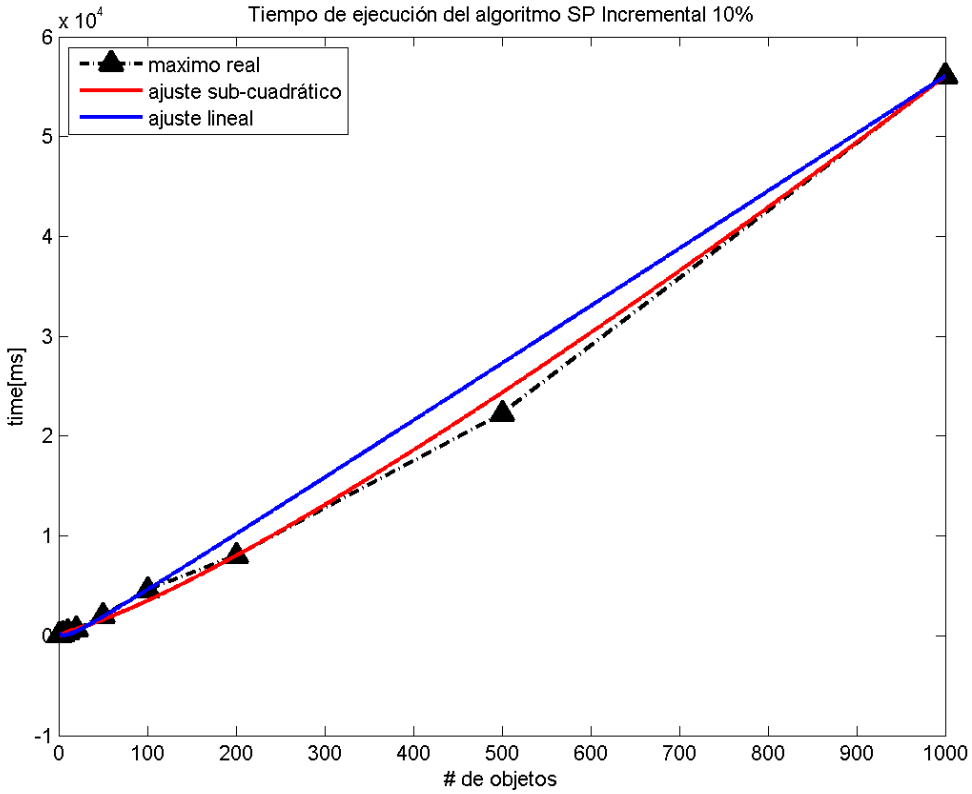
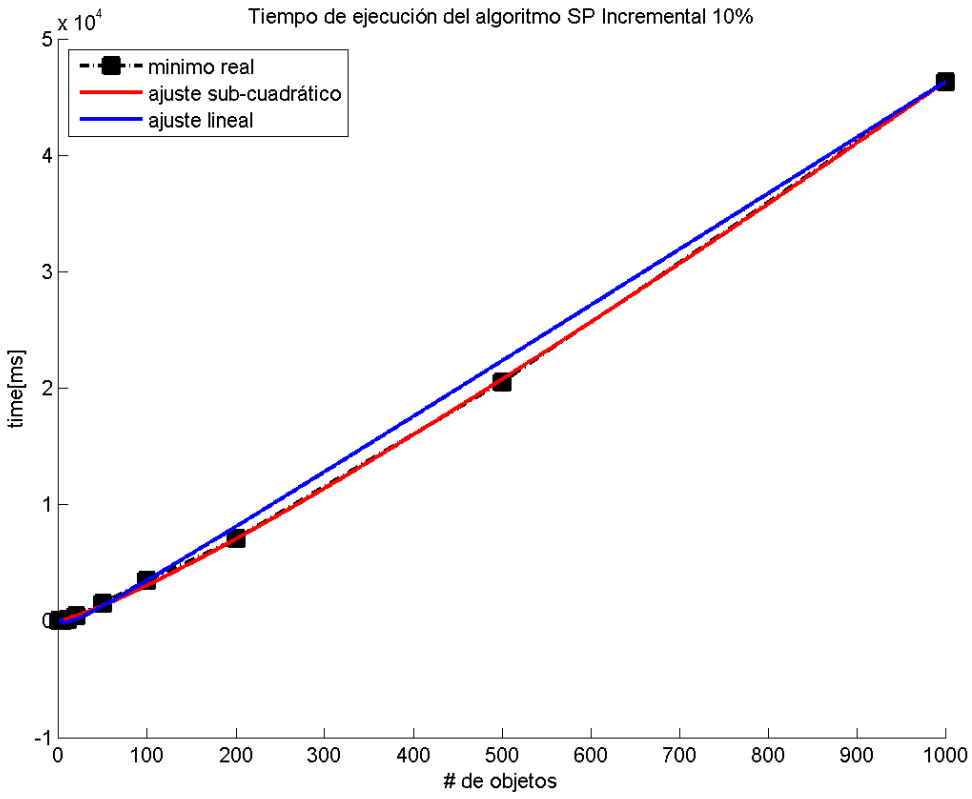


Figura 4.26: Resultados de S&P incremental con ocupación del 10 % para curva del mínimo en los experimentos.

El ajuste subcuadrático corresponde a $7.1834x\log(x) - 3.5055x + 31.2882\log(x) - 24.6346$, con un con un $SSE = 1.1391 * 10^5$, $R^2 = 0.9999$, $dfe = 6$, $R^2_{adj} = 0.9999$ y $RMSE = 137.7848$, mientras que el ajuste lineal es $48.4330x - 334.2907\log(x) + 183.1621$, con un $SSE = 1.587 * 10^6$, $R^2 = 0.9992$, $dfe = 7$, $R^2_{adj} = 0.9990$ y $RMSE = 476.1468$.



Todas las figuras anteriores muestran el ajuste para mejor y peor desempeño del algoritmo Sweep and Prune Incremental. El análisis para la curva promedio se ve en los anexos entre las tablas B.17 y B.32, haciendo el análisis de los métodos de ordenamiento y del cálculo de colisiones, y su influencia en el algoritmo final. De acuerdo a [34], el hecho de aprovechar la coherencia espacial permite que el algoritmo Sweep and Prune incremental mejore el desempeño en comparación al algoritmo Sweep and Prune directo. Esto puede notarse en la cantidad de veces que se debe ejecutar, en promedio, el algoritmo de cálculo de colisiones versus el total de los cuadros de la simulación, como se observa en la figura 4.27. Sin embargo, el desempeño del algoritmo general no está muy cerca de lo que se espera de su contraparte cinética. Esto se debe principalmente al cálculo de las listas de colisiones, como se observa en la figura 4.28.

Figura 4.27: Porcentaje de ejecuciones de cálculo de colisiones respecto del total de cuadros de la simulación para distintos porcentajes de ocupación.

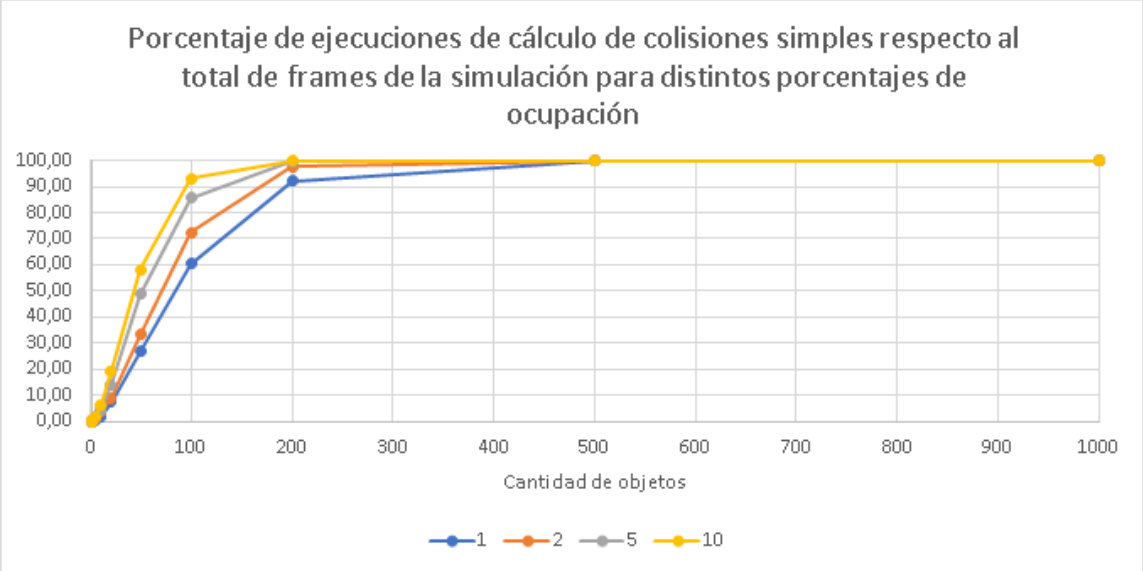
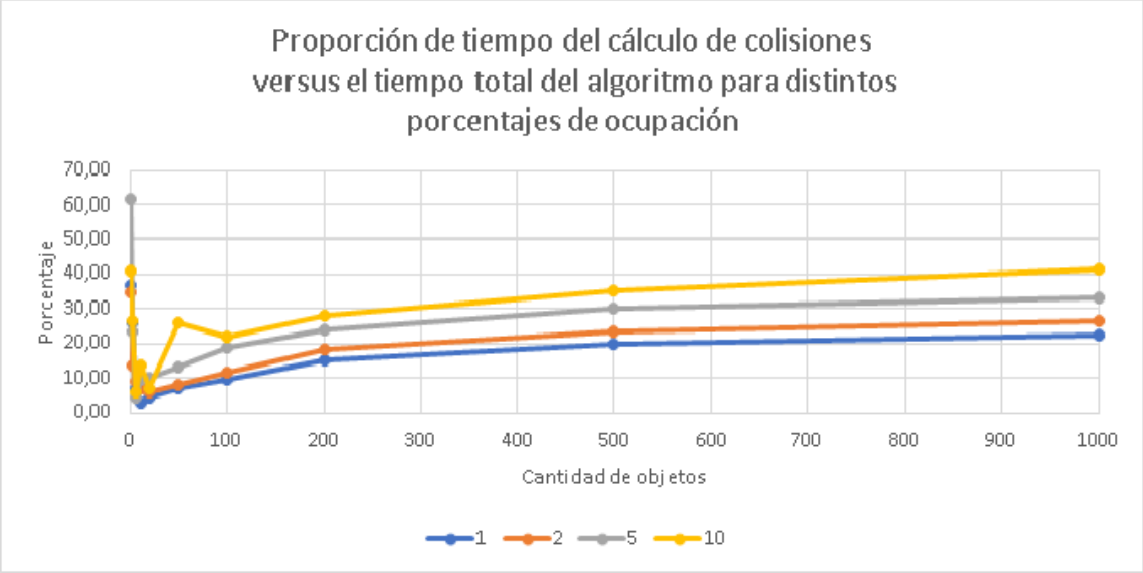


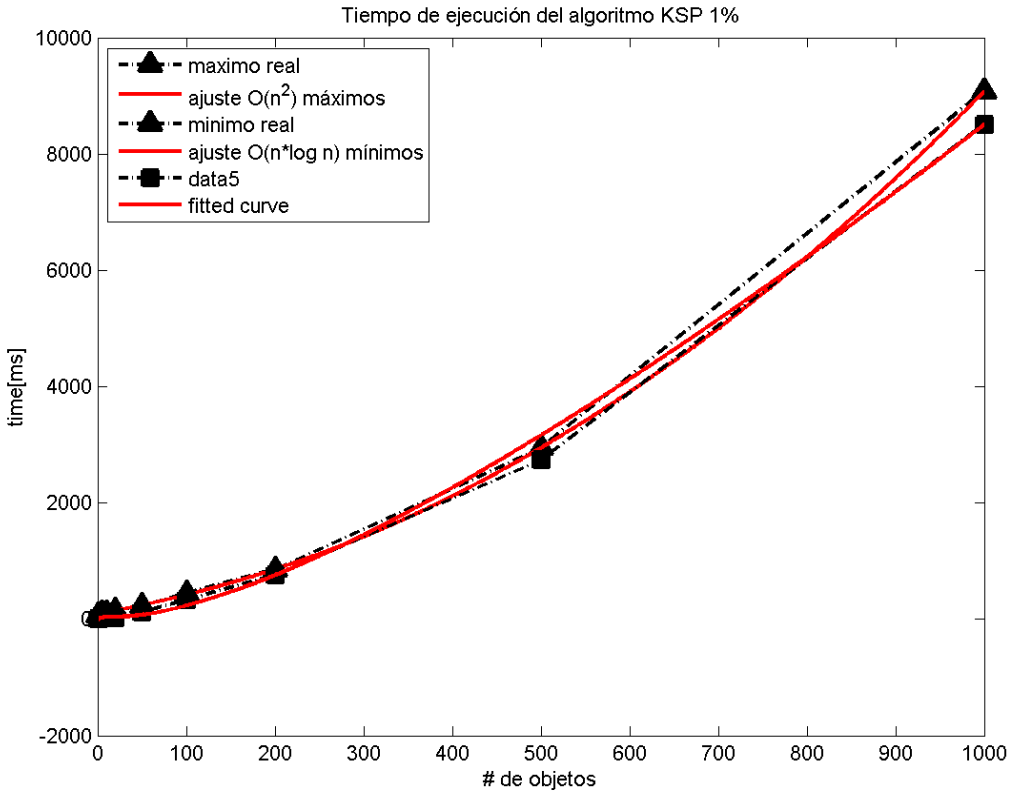
Figura 4.28: Porcentaje de tiempo de ejecución para el cálculo de colisiones respecto del total de cuadros de la simulación para distintos porcentajes de ocupación.



4.4.3. Sweep and Prune Cinético

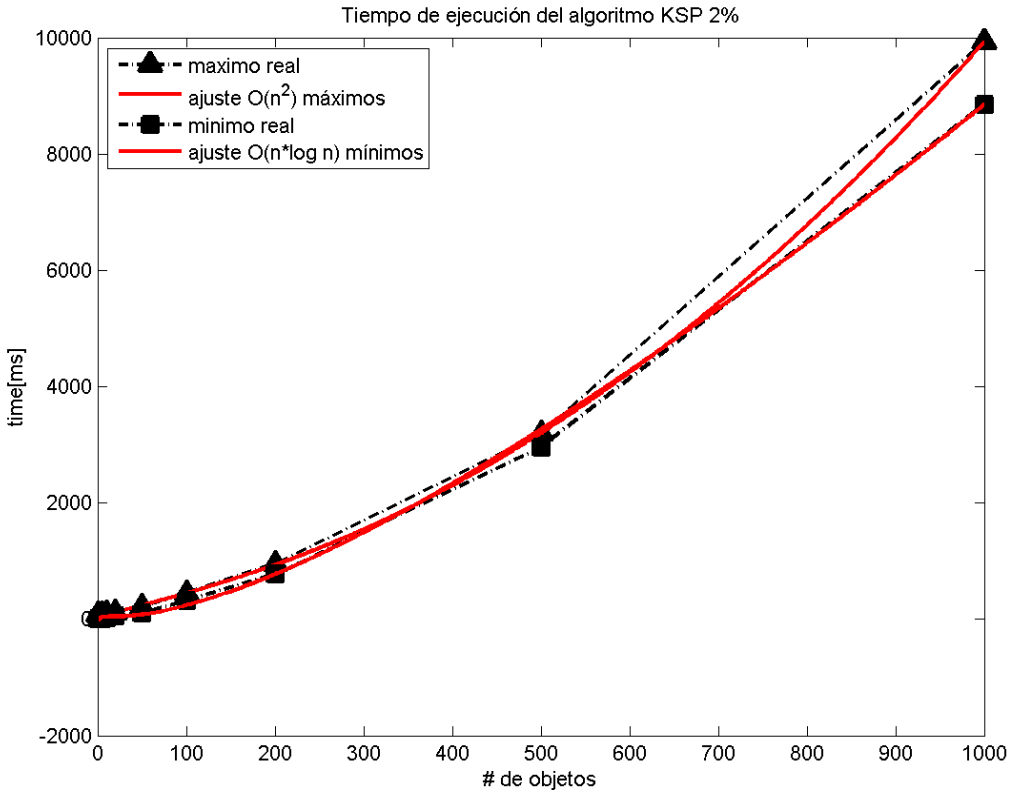
El análisis de optimalidad del algoritmo Sweep and Prune cinético se hace en base a la cantidad de eventos de swap dentro de las listas, cuyo peor caso es $O(n^2)$ [10]. Si bien no se implementaron todas las estructuras que utiliza [10], se busca comparar este método con sus versiones estáticas. Adicionalmente se desea observar el efecto de la detección de eventos múltiples en el desempeño del algoritmo. En las figuras 4.29 a 4.32 se tiene las curvas de tiempo de ejecución del mejor y peor desempeño de los algoritmos S&P-cinético para las mismas simulaciones de los casos anteriores, los mismos porcentajes de ocupación y cantidad de objetos de la sección 4.4.1. Cada figura incluye la mejor curva ajustada de acuerdo al mismo método de mínimos cuadrados no lineales de la sección 4.4.1.

Figura 4.29: Resultados de S&P Cinético con ocupación del 1 %



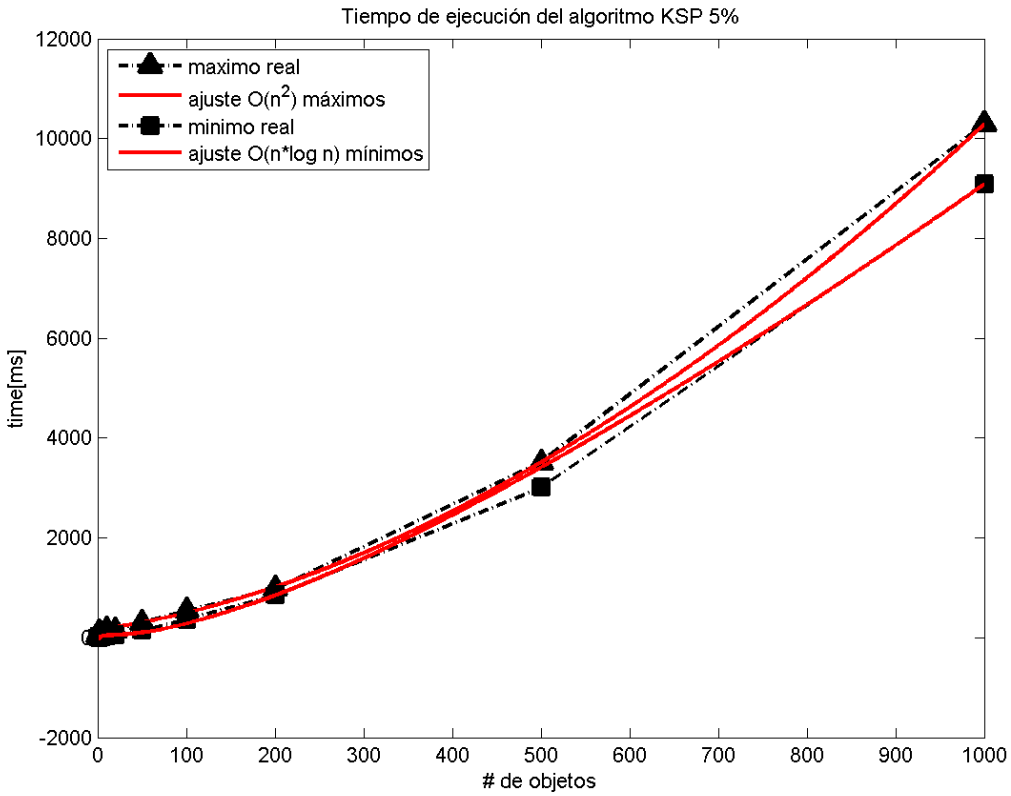
la curva superior obtenida corresponde a $0.0067x^2 - 0.0550x\log(x) + 2.6394x + 8.4357\log(x) + 73.2539$, con un **RMSE**=19.8313 y un R^2=1, mientras que el ajuste de la curva inferior da $3.5552x\log(x) - 16.4074x + 50.1543\log(x) + 6.4220$ con un **RMSE**=94.7696 y un R^2=0.9992.

Figura 4.30: Resultados de S&P Cinético con ocupación del 2 %



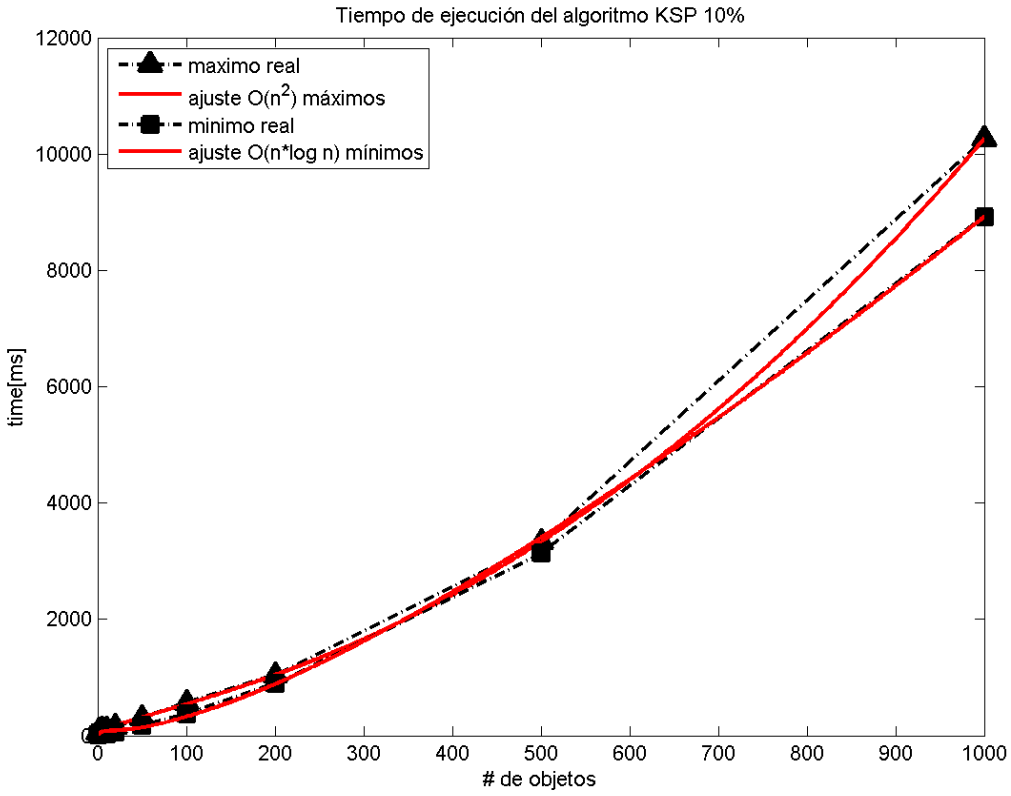
la curva superior obtenida corresponde a $0.0082x^2 - 0.8422x\log(x) + 7.6379x - 21.1691\log(x) + 83.8540$, con un **RMSE**=12.5595 y un R^2=1, mientras que el ajuste de la curva inferior da $3.8042x\log(x) - 17.8339x + 57.7473\log(x) + 3.6666$ con un **RMSE**=72.9016 y un R^2=0.9995.

Figura 4.31: Resultados de S&P Cinético con ocupación del 5 %



la curva superior obtenida corresponde a $0.0051x^2 + 1.5694x\log(x) - 6.1654x + 58.8120\log(x) + 66.3602$, con un **RMSE**=48.2287 y un R^2=0.9999, mientras que el ajuste de la curva inferior da $3.7450x\log(x) - 17.2075x + 63.1439\log(x) - 14.5447$ con un **RMSE**=91.7365 y un R^2=0.9993.

Figura 4.32: Resultados de S&P Cinético con ocupación del 10 %

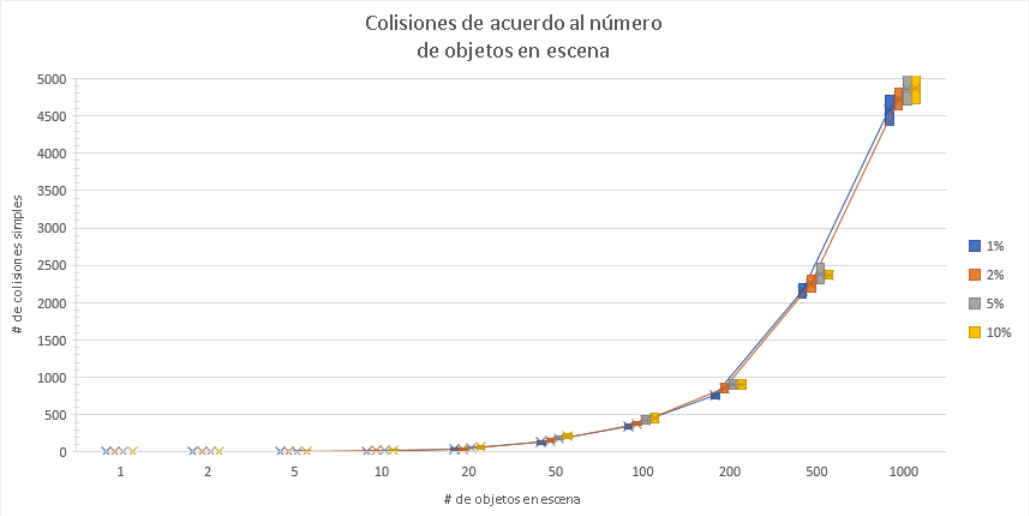


la curva superior obtenida corresponde a $0.0090x^2 - 1.2239x\log(x) + 9.6923x - 2.1984\log(x) + 50.9101$, con un
RMSE=18.6418 y un R^2=1, mientras que el ajuste de la curva inferior da $3.5367x\log(x) - 15.9239x + 55.4429\log(x) + 29.3239$ con un **RMSE**=117.0640 y un R^2=0.9989.

4.4.4. Eventos en función de la cantidad de objetos

Para este análisis sólo se utilizó los datos del algoritmo KSP, ya que ahí tenemos la cantidad de eventos simples y múltiples para cada experimento. Como puede verse en la figura 4.33, la cantidad de eventos aumenta en función del número de objetos de manera independiente del porcentaje de ocupación inicial del experimento. En el caso de la detección de eventos múltiples en los experimentos, en todos resultó ser 0. Esto indica, o que el evento de una colisión múltiple es extremadamente raro o la forma de detectarlo no fue la adecuada, ya que en la ejecución de los 400 experimentos no se observó ninguna sola vez.

Figura 4.33: Eventos simples en función de la cantidad de objetos para distintos valores de ocupación



Una de las razones para que la cantidad de eventos simples se vea inalterada puede deberse a que mientras la cantidad de objetos aumenta, su tamaño disminuye, con lo que también disminuye su velocidad de desplazamiento.

4.4.5. Discusión de los resultados

Al observar los resultados obtenidos con los experimentos (ver anexo B.5) y al contrastarlos con la teoría, se puede decir que efectivamente la implementación del algoritmo Sweep and Prune estático directo está ajustada a $O(n^2)$, como puede comprobarse en las figuras de la 4.12 a 4.17, donde las estadísticas de R^2 ajustado y RMSE muestran que las curvas se encuentran más ajustadas con una curva cuadrática que con una subcuadrática. Esto se debe a que la obtención de los pares fue la parte más costosa en conjunto con la ejecución del algoritmo inplace insertionSort. La obtención de los pares de objetos colisionando se hace obteniendo las listas de solapamiento por cada dimensión y se compara la lista más corta, elemento por elemento, con los elementos de las otras dimensiones. En el caso de los experimentos con el algoritmo Sweep and Prune estático incremental, [9] establece que el algoritmo tiene un tiempo de ejecución de $O(n + s)$, con n=cantidad de objetos y s=número de traslapes entre pares de objetos. Los resultados muestran que el desempeño no es el esperado, obteniendo una curva promedio de tipo cuadrático, pero con el coeficiente muy pequeño.

Comparativamente con el caso del algoritmo Sweep and Prune cinético, las constantes para el término cuadrático son comparables con los coeficientes de los algoritmos de ordenamiento y cálculo de colisiones de los métodos Sweep and Prune estáticos.

La densidad de los objetos no es una medida directa del desempeño de los algoritmos, ya que para diferentes cantidades de objetos se puede tener la misma densidad variando el tamaño de los mismos, pero si es una forma de relacionar que, en un ambiente con desplazamientos aleatorios, a medida que crece la densidad, las colisiones se hacen más probables.

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

De acuerdo a [34], el algoritmo Sweep and Prune Directo tiene un tiempo de ejecución lineal, asumiendo que se tiene ordenada la lista de proyecciones de los extremos de los objetos por cada eje coordenado. En las ejecuciones cuadro por cuadro no se ordenan los objetos iniciales, por lo que es necesario ordenar estos AABB. El tiempo de ejecución del algoritmo de ordenamiento insertionSort es de $\Theta(n^2)$. De acuerdo a [9], el algoritmo tiene un tiempo de ejecución de $O(n+m)$

En el caso de los algoritmos Sweep and Prune tradicionales, el trabajo realizado por el algoritmo depende de múltiples factores, dentro de los cuales se pudo manejar en este trabajo los siguientes:

- La cantidad de FPS que tiene la simulación, ya que ésta indicará la cantidad de veces que debe ser ejecutado el algoritmo DSP. En el caso del algoritmo ISP, una cantidad de FPS insuficiente hará que entre cada intervalo existan varios elementos que se encuentren en desorden, por lo que el algoritmo de ordenamiento se demorará más.
- La cantidad de objetos que se desplazan en la simulación influyen en cuantos elementos estarán desordenados en todas las listas, influyendo también en el tiempo que demora el algoritmo de ordenamiento.
- El tamaño de un objeto en la simulación y su velocidad de desplazamiento, ya que esto influye en la cantidad de colisiones posibles.

En el caso del algoritmo ISP, las simulaciones muestran que para una cantidad de objetos baja de hasta 16 objetos en escena, independiente de las densidades utilizadas, el algoritmo se comporta de manera similar a su versión cinética, mientras que a medida

que crece el número de objetos en la escena, los tiempos del algoritmo ISP se empiezan a acercar más a los del algoritmo Sweep and Prune directo (DSP). Esto se debe a que el algoritmo ISP debe certificar que la lista se mantiene ordenada en cada intervalo de tiempo, cosa que es cada vez más difícil de mantener con un número creciente de objetos.

Las mediciones realizadas en los experimentos de los algoritmos DSP e ISP muestran que su peor tiempo de ejecución está acotado asintóticamente a $O\left(n^2\right)$. Esto se debe a que el proceso de cálculo de las colisiones es el más demoroso, y para el cual en esta tesis no se implementó ningún método especial para ello, basandose solamente en los métodos *retainAll* y *addAll* de colecciones en Java.
El algoritmo ISP se comporta mejor para una baja cantidad de objetos debido a que no se ejecuta el cálculo de colisiones en cada frame. A medida que aumenta la cantidad de objetos y su densidad el cálculo de colisiones se degrada, ejecutando el método en cada frame. Se esperaba que el mejor tiempo de ejecución del algoritmo *Sweep and Prune* Incremental estuviera cercano al tiempo de ejecución del algoritmo KSP. Para poder descartar que este resultado sea independiente del porcentaje de ocupación de los objetos se debe validar con otro tipo de experimentos que deben demostrar que no existe correlación. En estos experimentos se debe evaluar cantidades crecientes de objetos con un área por objeto fija, con una velocidad de desplazamiento cuyo rango no varíe entre experimentos. Las mediciones realizadas en el capítulo 4 muestran que el método Sweep and Prune cinético (KSP) se comporta mejor que los algoritmos Sweep and Prune tradicionales. Esto se a debe a dos hechos:

- En KSP la lista de colisiones inicial está precalculada, al igual que el algoritmo Sweep and Prune Incremental (ISP), pero sucede que los eventos en el algoritmo KSP mayoritariamente involucran sólo a un par de objetos, es decir son eventos simples, cosa que no se sabe para el algoritmo ISP en el paso de un intervalo de tiempo ($\frac{1}{1000}$ de segundo en el caso de estas simulaciones), por lo que en cada intervalo de tiempo se debe certificar que la lista de objetos aún mantiene el orden.

- En KSP los eventos estaban precalculados de antemano, donde un evento simple influye en sólo un par de elementos de la lista cinética, los que son inmediatamente reordenados y generan a lo más dos eventos nuevos.

En la ejecución de los experimentos con el algoritmo KSP, la cantidad de eventos múltiples fue 0. Esto implica que el evento es extremadamente raro o que la forma de obtenerlo no es la adecuada, ya que existen dos maneras de saber si existen eventos de este tipo en una simulación: revisar en el tiempo que sucede un evento, si más de dos vértices de AABBs coinciden en un mismo punto; o, extraer de la cola de eventos dos eventos consecutivos y revisar si estos eventos suceden al mismo tiempo, en la misma dimensión

y en la misma posición. En cuanto a la medida de densidad de objetos, no es concluyente su uso mientras no se establezca una comparación similar en tiempos entre los algoritmos Sweep and Prune estático y cinético.

Para la verificación de los resultados o un posterior desarrollo se incluye:

- El código fuente de los métodos, con los test unitarios incluídos.
- Un archivo de texto para cada experimento, con los polígonos iniciales y sus velocidades de desplazamiento.
- El análisis de los resultados en varios archivos excel.
- Los script de MATLAB®para la ejecución de los análisis de ajuste de curvas.

estos están disponibles a través de GitLab en las siguientes direcciones:

- https://gitlab.com/hmoraga/codigo-tesis.git
- https://gitlab.com/hmoraga/experimentos-tesis.git

5.2. Trabajo futuro

Se puede plantear muchas líneas de trabajo futuro de la detección de colisiones. Partiendo desde los desarrollos realizados, se podría:

- Expandir la lista de volúmenes envolventes disponibles a esferas, *OBBs*, *k-DOPs*, cerradura convexa, etc., y con ello evaluar sus desempeños.
- Implementar algún método de detección de colisiones de fase angosta, para poder tener el proceso de detección completo. Un ejemplo puede ser el AABB Tree estático, que representa a un objeto como un árbol de *aabbs*, partiendo desde las aristas como raíces del árbol y construyendo a partir de ellas, una jerarquía de *AABBs*. Lo interesante de este método particular es que existe su contraparte cinética, como el trabajo desarrollado por [38], el que podría implementarse tanto en su versión en Java como en su versión en C++.
- Expandir el soporte de ambas bibliotecas a 3D. Esto implica actualizar las versiones de puntos, aristas, caras (o planos) y los métodos Sweep and Prune estático y cinético. Con la finalidad de utilizarlo en escenarios reales, como por ejemplo miscroscopía masiva o simulaciones en 3D.

- Se podría mejorar la forma de obtención de los pares de objetos que colisionan, ya que es el proceso que más demora en los algoritmos estáticos de Sweep and Prune.

Con respecto a las comparaciones de desempeños de los algoritmos de detección de colisiones se podría implementar el uso de AABBTree para el descarte rápido de objetos que no se encuentran en colisión. También se puede evaluar el uso de los spanners geométricos de Gao *et al.* [16] en la manteción del par más cercano como una forma de detectar colisiones.

Se puede implementar una manera de observar el desempeño de los algoritmos descritos pero en función del espacio de memoria utilizado, lo cual implica no solo mantener las referencias a objetos, que es lo que mantiene el profiler de Java, sino que llevar un catastro efectivo del tamaño de las estructuras usadas en todo instante.

Glosario

R^2 Coeficiente de determinacón. es un número que indica la proporción de la varianza en la variable dependiente que es predecible por la(s) variable(s) independiente(s). R^2 es una estadística que dará alguna información sobre la bondad de ajuste del modelo. En una regresión, el coeficiente de determinación R^2 es una medida estadística de qué tan bien la regresión se aproxima a los puntos de datos reales. Un R^2 de 1 indica que la regresión calza perfectamente con los datos. Matemáticamente se define como la proporción entre la suma de los cuadrados de la regresión (SSR) y la suma total de los cuadrados (SST).

$$SSR = \sum_{i=1}^{n} w_i \left(\hat{y}_i - \overline{y}\right)^2$$

$$SST = \sum_{i=1}^{n} w_i \left(y_i - \overline{y}\right)^2$$

$$SST = SSR + SSE$$

$$R^2 = \frac{SSR}{SST} = 1 - \frac{SSE}{SST}$$

R^2 **ajustado** El coeficiente de determinación ajustado es el mejor indicador de la calidad del ajuste cuando se comparan dos modelos que están anidados, es decir, una serie de modelos cada uno de los cuales agrega coeficientes adicionales al modelo previo. Se define como:

$$R^2_{adj} = 1 - \frac{SSE\,(n-1)}{SST\,(v)}$$

El R^2 ajustado puede tener cualquier valor menor o igual a 1, con un valor cercano a 1 indicando un mejor ajuste. Valores negativos pueden ocurrir cuando el modelo contiene términos que no ayudan a predecir la respuesta.

AABBTree Tipo de jerarquía de volúmenes envolventes, donde cada nodo corresponde a un AABB.

boundingVolume Volúmenes envolventes, en general cualquier geometría más simple que el objeto original. Ésta puede ser desde el uso de una esfera (en $\mathbb{R}_n$) hasta el uso de una cerradura convexa.

BSPTree estructura de datos de partición espacial que subdivide el espacio de manera recursiva en dos hasta algún criterio para formar las hojas. La partición binaria del espacio de un BSP-tree puede ser mediante un hiperplano en cualquier dirección, comparado con los árboles octree o kd-tree, cuya división está dada por un hiperplano alineado con los ejes.

coherencia temporal Sucede cuando un ambiente dinámico cambia de forma suave a través del tiempo, p.e., un punto de vista y/o movimiento pequeño de un objeto.

coherenciaEspacial Describe homogeneidades espaciales. Son una consecuencia de la lenta o constante variación de relaciones en la disposición espacial de objetos o datos.

detección de colisiones de fase amplia métodos de detección de colisiones cuya finalidad es reducir los pares de posibles objetos en colisión

detección de colisiones de fase angosta métodos de detección de colisiones cuya finalidad es, dados dos objetos que posiblemente colisionan, encontrar si efectivamente lo hacen y entre qué elementos que componen los objetos

evento múltiple situación que se da en una simulación cuando se tienen tres o mas elementos en la misma posición al mismo tiempo. Por ejemplo, cuando tres aristas de tres objetos distintos se ubican en la misma posición en un instante de tiempo determinado

Grados de libertad de R^2 ajustado Esta estadística usa la definición de R^2, y se ajusta en base a los grados de libertad residuales. Se define grados de libertad residuales como el número de valores de la respuesta n menos el número m de coeficientes ajustados estimados desde los valores originales.

$$v = n - m$$

v indica el número de piezas independientes de información que involucran a los n puntos de datos que son requeridos para calcular la suma de los cuadrados. Notar que si los parámetros son acotados y una o más de las estimaciones son sus cotas, luego esas estimaciones son consideradas fijas. El número de grados de libertad se incrementa por la cantidad de aquellos parámetros.

Kd-tree estructura de datos de partición espacial que organiza puntos en un espacio k-dimensional. Corresponden a un caso especial de los árboles de partición binaria del espacio.

octree estructura de datos de partición espacial que divide el espacio en ocho octantes, lográndose una división por dos de cada dimensión (del mismo tamaño).

SSE Esta estadística mide la desviación total entre la diferencia de los valores reales con la obtenida por la curva de calce. Es también llamada la suma cuadrática de los residuales y es usualmente etiquetada como *SSE*.

$$SSE = \sum_{i=1}^{n} w_i \left(y_i - \hat{y}_i\right)^2$$

Un valor más cercano a 0 indica que el modelo tiene una componente de error aleatorio pequeña, y que el calce será más útil para predicciones.

stack pila o montón

Sweep and Prune algoritmo de fase amplia que utiliza un barrido con los elementos iniciales y finales del AABB de una lista de objetos para detectar si existen posibles colisiones entre los volúmenes envolventes, mecanismo que se realiza en todas las dimensiones del objeto.

Abreviaciones

AABB	Axes Aligned Bounding Box
bv	Volumen Envolvente
bvh	Jerarquía de Volúmenes Envolventes
k-DOP	k-Discrete Oriented Polytope
KDS	Estructura de datos Cinética
OBB	Oriented Bounding Box
RMSE	Error cuadrático medio

Bibliografía

[1] Compare fits programmatically. `https://la.mathworks.com/help/curvefit/compare-fits-programmatically.html`. Accessed: 2018-11-11.

[2] ABAM, M. A., AGARWAL, P. K., DE BERG, M., AND YU, H. Out-of-order event processing in kinetic data structures. In *Algorithms – ESA 2006: 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings*, Y. Azar and T. Erlebach, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 624–635.

[3] ALI ABAM, M., AND DE BERG, M. Kinetic sorting and kinetic convex hulls. *Comput. Geom. Theory Appl. 37*, 1 (May 2007), 16–26.

[4] BARAFF, D., AND WITKIN, A. Dynamic simulation of non-penetrating flexible bodies. *SIGGRAPH Comput. Graph. 26*, 2 (July 1992), 303–308.

[5] BASCH, J., AND DEPT, S. U. C. S. *Kinetic data structures*. Stanford University, 1999.

[6] BASCH, J., GUIBAS, L. J., AND HERSHBERGER, J. Data structures for mobile data. In *JOURNAL OF ALGORITHMS* (1997), pp. 747–756.

[7] CHANG, A. Y. A survey of geometric data structures for ray tracing. *Polytechnic University (New York), Tech. Rep. TR-CIS-2001-06* (2001).

[8] CHANG, V., SAAVEDRA, J. M., CASTAÑEDA, V., SARABIA, L., HITSCHFELD, N., AND HÄRTEL, S. Gold-standard and improved framework for sperm head segmentation. *Computer Methods and Programs in Biomedicine 117*, 2 (2014), 225–237.

[9] COHEN, J. D., LIN, M. C., MANOCHA, D., AND PONAMGI, M. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics* (New York, NY, USA, 1995), I3D '95, ACM, pp. 189–ff.

[10] COMING, D. S., AND STAADT, O. G. Kinetic sweep and prune for collision detection. In *Proc. Workshop on Virtual Reality Interactions and Physical Simulations* (2005), pp. 81–90.

[11] COMING, D. S., AND STAADT, O. G. Virtual reality interaction and physical simulation: Kinetic sweep and prune for multi-body continuous motion. *Comput. Graph. 30*, 3 (June 2006), 439–449.

[12] ERICSON, C. *Real-time collision detection*. Taylor & Francis US, 2005.

[13] ERWIN COUMANS, E. A. Bullet physics library. `http://bulletphysics.org/wordpress/`, Nov. 2012 (accessed 2016-April-02).

[14] FISCHER, K., GÄRTNER, B., HERRMANN, T., HOFFMANN, M., AND SCHÖNHERR, S. Bounding volumes. `http://doc.cgal.org/4.3/Manual/packages.html{#}PkgBoundingVolumesSummary`, 2013.

[15] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[16] GAO, J., GUIBAS, L. J., AND NGUYEN, A. Deformable spanners and applications. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry* (New York, NY, USA, 2004), SCG '04, ACM, pp. 190–199.

[17] GOODMAN, J., AND O'ROURKE, J. *Handbook of Discrete and Computational Geometry, Second Edition*. Discrete Mathematics and Its Applications. Taylor & Francis, 2004.

[18] GOTTSCHALK, S., LIN, M. C., AND MANOCHA, D. Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 171–180.

[19] GUIBAS, L. J. Kinetic data structures: A state of the art report. In *Proceedings of the Third Workshop on the Algorithmic Foundations of Robotics on Robotics : The Algorithmic Perspective: The Algorithmic Perspective* (Natick, MA, USA, 1998), WAFR '98, A. K. Peters, Ltd., pp. 191–209.

[20] IVEZIC, Z., TYSON, J., ACOSTA, E., ALLSMAN, R., ANDERSON, S., ANDREW, J., ANGEL, R., AXELROD, T., BARR, J., BECKER, A., ET AL. Lsst: from science drivers to reference design and anticipated data products. *arXiv preprint arXiv:0805.2366* (2008).

[21] JIMÉNEZ, P., THOMAS, F., AND TORRAS, C. 3d collision detection: a survey. *Computers & Graphics 25*, 2 (2001), 269–285.

[22] JOHNSON, D. S. A theoretician's guide to the experimental analysis of algorithms. *Data structures, near neighbor searches, and methodology: fifth and sixth DIMACS implementation challenges 59* (2002), 215–250.

[23] KELLER, P. J., SCHMIDT, A. D., WITTBRODT, J., AND STELZER, E. H. Reconstruction of zebrafish early embryonic development by scanned light sheet microscopy. *Science 322*, 5904 (2008), 1065–1069.

[24] KOCKARA, S., HALIC, T., IQBAL, K., BAYRAK, C., AND ROWE, R. Collision detection: A survey. In *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*. IEEE, 2007, pp. 4046–4051.

[25] MADERA, F. An introduction to the collision detection algorithms. *Abstraction and Application*, 5 (2011), 7–18.

[26] MORET, B. M. Towards a discipline of experimental algorithmics. *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges 59* (2002), 197–213.

[27] NGUYEN, A. *Implicit bounding volumes and bounding volume hierarchies*. PhD thesis, Citeseer, 2006.

[28] PELFREY, B. A simple, optimized bounding volume hierarchy for ray/object intersection testing. `https://github.com/brandonpelfrey/Fast-BVH`. Accesed: 2013-12-10.

[29] RAHMATI, Z., KING, V., AND WHITESIDES, S. Kinetic data structures for all nearest neighbors and closest pair in the plane. In *Proceedings of the Twenty-ninth Annual Symposium on Computational Geometry* (New York, NY, USA, 2013), SoCG '13, ACM, pp. 137–144.

[30] RIVERA, J.-C. A non-recursive javascript implementation of an n-dimensional bounding-volume hierarchy. `http://github.com/imbcmdth/jsBVH`. Accesed: 2013-12-10.

[31] SANDERS, P. Presenting data from experiments in algorithmics. In *Experimental algorithmics*. Springer, 2002, pp. 181–196.

[32] SZALAY, A. S., KUNSZT, P. Z., THAKAR, A., GRAY, J., SLUTZ, D., AND BRUNNER, R. J. Designing and mining multi-terabyte astronomy archives: The sloan digital sky survey. *SIGMOD Rec. 29*, 2 (May 2000), 451–462.

[33] TESCHNER, M., KIMMERLE, S., HEIDELBERGER, B., ZACHMANN, G., RAGHUPATHI, L., FUHRMANN, A., CANI, M.-P., FAURE, F., MAGNENAT-THALMANN, N., STRASSER, W., ET AL. Collision detection for deformable objects. In *Computer Graphics Forum* (2005), vol. 24, Wiley Online Library, pp. 61–81.

[34] TRACY, D. J., BUSS, S. R., AND WOODS, B. M. Efficient large-scale sweep and prune methods with aabb insertion and removal. In *2009 IEEE Virtual Reality Conference* (2009), IEEE, pp. 191–198.

[35] WELLER, R. Kinetic data structures for collision detection. In *New Geometric Data Structures for Collision Detection and Haptics*. Springer, 2013.

[36] WELZL, E. Smallest enclosing disks (balls and ellipsoids). *Lecture Notes in Computer Science 555* (1991), 359–370.

[37] WOLLMAN, R., AND STUURMAN, N. High throughput microscopy: from raw images to discoveries. *Journal of Cell Science 120*, 21 (2007), 3715–3722.

[38] ZACHMANN, G., AND WELLER, R. Kinetic bounding volume hierarchies for deformable objects. In *Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and Its Applications* (New York, NY, USA, 2006), VRCIA '06, ACM, pp. 189–196.

Anexos

Anexo A

Configuración de la computadora

Los resultados fueron obtenidos en un computador Intel®Core™ i5-3230M de 2.6 GHz con 8 GB de RAM. El sistema operativo utilizado fue un Windows 8 de 64 bits cargando solamente los drivers del computador. Como IDE se usó Netbeans 8.1 de 64 bits para ejecutar las simulaciones y realizar las mediciones de desempeño. La versión de Java utilizada es la:

```
java version "1.8.0_141"
Java(TM) SE Runtime Environment (build 1.8.0_141-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.141-b15, mixed mode)
```

La configuración de la máquina virtual de Java tiene el parámetro *InitialHeapSize := 134 217 728* y un *MaxHeapSize := 2 118 123 520*, disponible gracias al comando **java -XX:+PrintFlagsFinal -version | findstr /i "HeapSize PermSize ThreadStackSize"**.

Anexo B

Implementaciones de los Algoritmos

Para la implementación de los algoritmos se dividió la aplicación en cuatro proyectos que están relacionados, como lo muestra la figura B.1. El primero de ellos, llamado **StaticLibrary**, implementa los dos algoritmos de detección de colisiones de detección de colisiones de fase amplia estáticos: sweep and prune estático directo e incremental. El segundo proyecto, llamado **KineticLibrary**, posee lo necesario para la implementación del algoritmo sweep and prune cinético. El tercer proyecto, llamado **Scenarios**, posee los modelos de prueba y las clases necesarias para generar objetos de manera aleatoria, cargar datos desde archivos y grabar los resultados. Por último, el proyecto **ShowModel** que me permite generar las imágenes de un proyecto y mostrar la secuencia de la ejecución, pero no en tiempo real.

B.1. La biblioteca StaticLibrary

Esta biblioteca tiene las implementaciones de todas las primitivas usadas, además de las implementaciones de los algoritmos sweep and prune estaáticos en su versión directo e incremental. En la figura B.2 se puede ver la lista de paquetes y algunas de sus clases componentes.

Dentro de las primitivas implementadas se encuentra los puntos, aristas y cajas o AABB en 2D. Nodos y árboles de volúmenes envolventes se implementaron como parte de una Jerarquía de Volúmenes Envolventes, para resolver la necesidad de tener un métodos de detección de colisiones de detección de colisiones de fase angosta para saber qué primitivas (aristas) entre dos objetos son las que colisionan, como se ve en la figura B.3. Clases que pueden ser usadas en el futuro.

El paquete sap (figura B.4) contiene las clases necesarias para la ejecución de los algoritmos sweep and prune estaáticos:

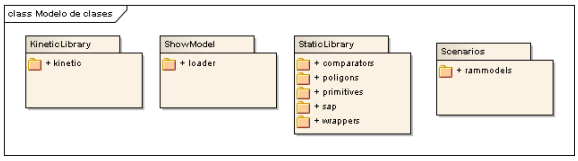


Figura B.1: Diagrama paquetes de los cuatro proyectos

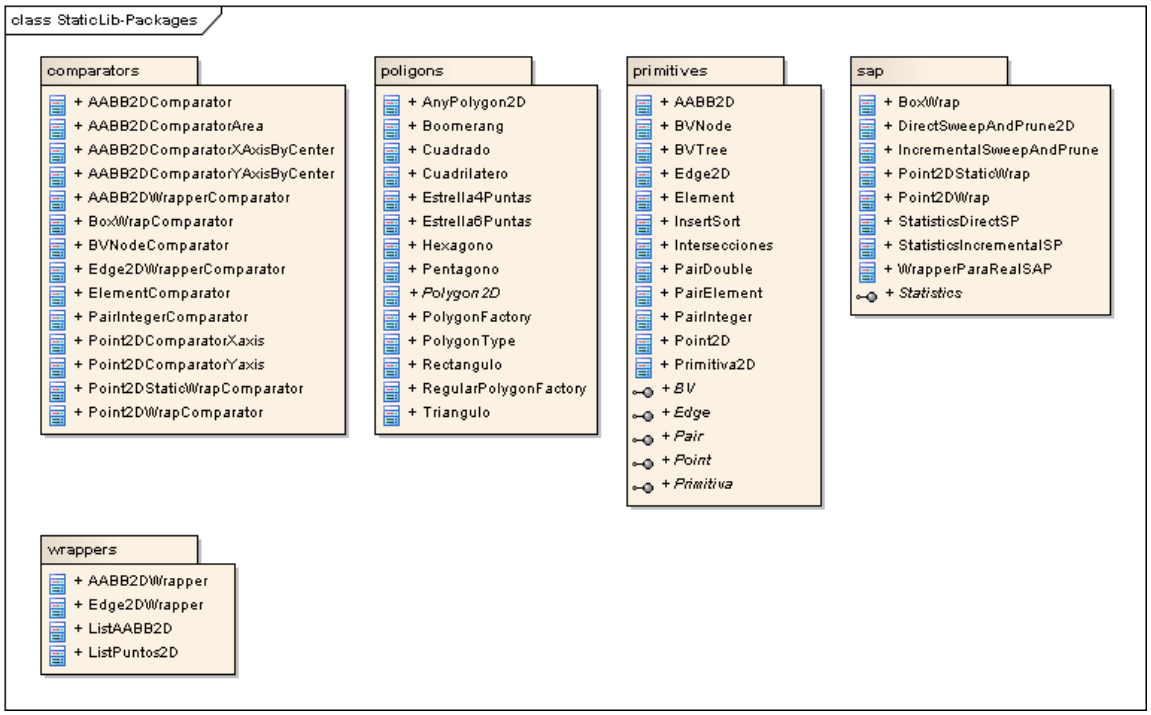


Figura B.2: Diagrama de paquetes del proyecto StaticLibrary

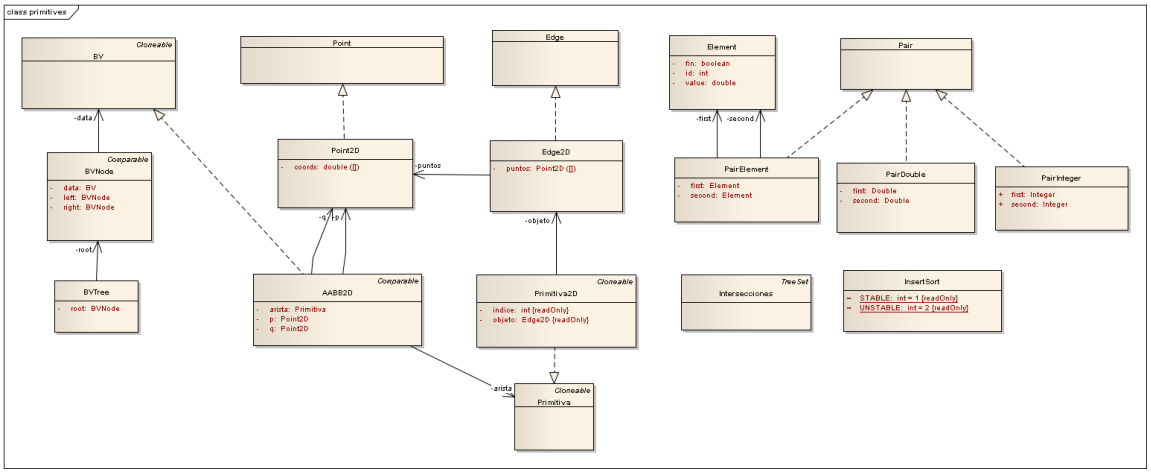


Figura B.3: Diagrama de clases de las primitivas en el proyecto staticLib

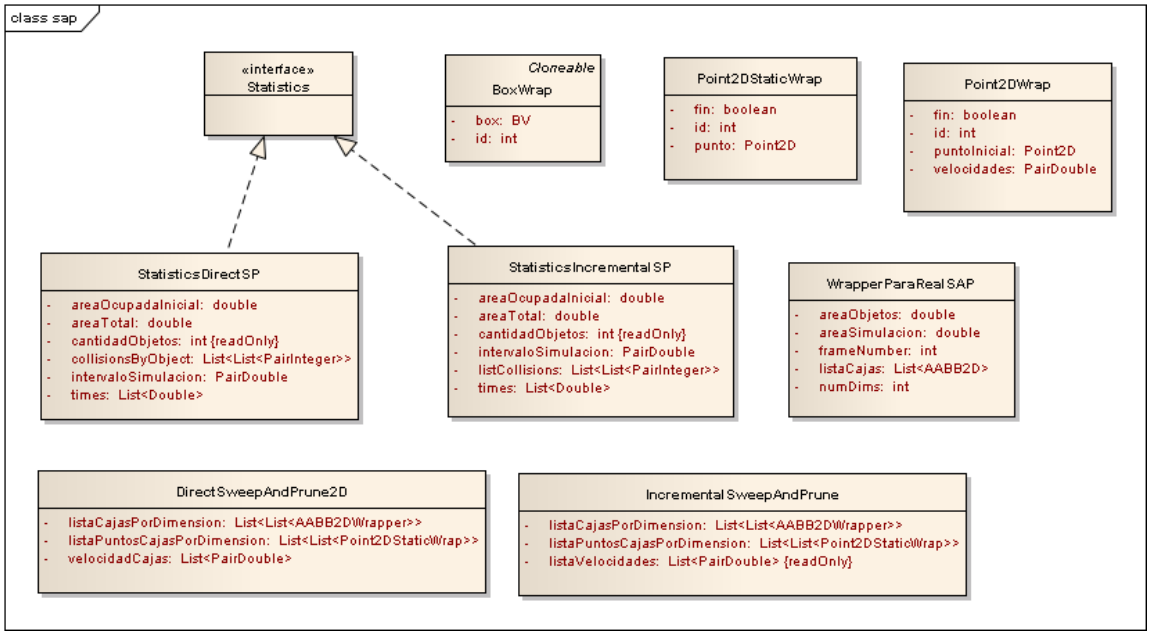


Figura B.4: Diagrama de clases del paquete sap, el que contiene los algoritmos estáticos de Sweep and Prune

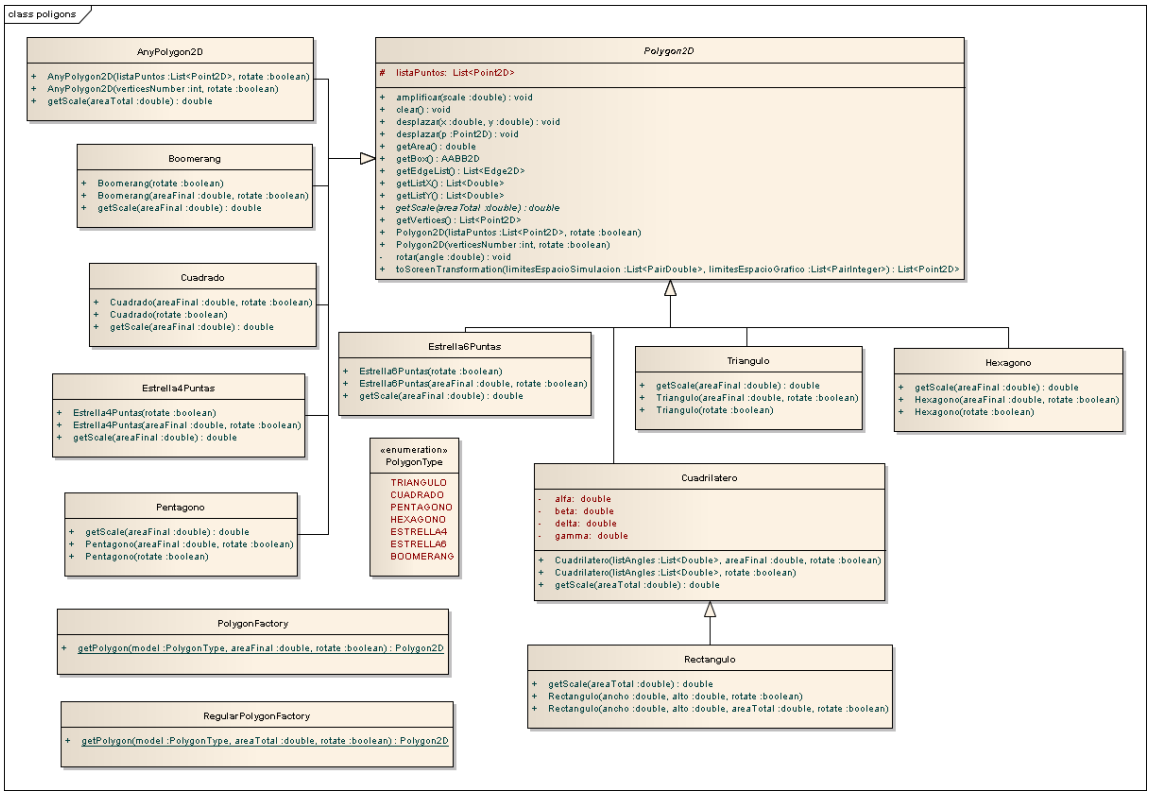


Figura B.5: Diagrama de clases de la fábrica de polígonos 2D

El algoritmo DSP (3) consiste en la mantención de listas de los puntos extremos de cada polígono, las que deben ser barridas en todas sus coordenadas para obtener los pares de objetos que colisionan. Existe una lista por coordenada. En esta implementación se obtienen las cajas (AABB) de cada objeto, a las que en cada intervalo de tiempo se le actualiza su posición de acuerdo a la velocidad de desplazamiento. Cada lista de cajas es creada en cada intervalo de tiempo, donde se mantiene ordenada de manera ascendente en base al valor inferior de la caja que rodea a cada polígono, revisar el algoritmo 1. Luego, cada lista de puntos es barrida para así obtener la lista de colisiones de acuerdo a lo descrito en 2. La figura B.6 muestra el diagrama de secuencia de la ejecución del algoritmo DSP.

En el caso del algoritmo incremental se necesita que las listas de puntos iniciales, una por coordenada, sean mantenidas a lo largo de toda la simulación, y en cada paso temporal se debe evaluar si se debe actualizar alguna de las listas. Un detalle del diagrama de secuencia de la inicialización del algoritmo se muestra en la figura B.7.

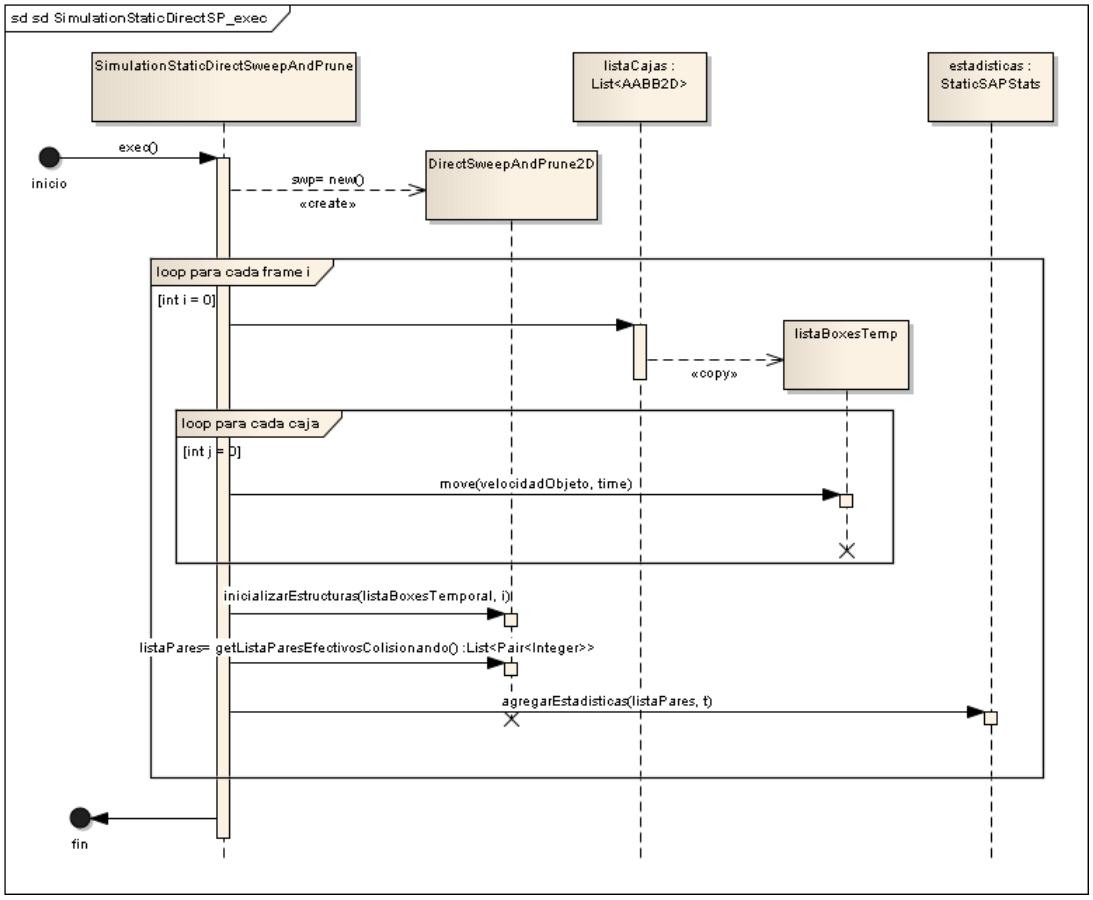


Figura B.6: Ejecución del algoritmo sweep and prune directo

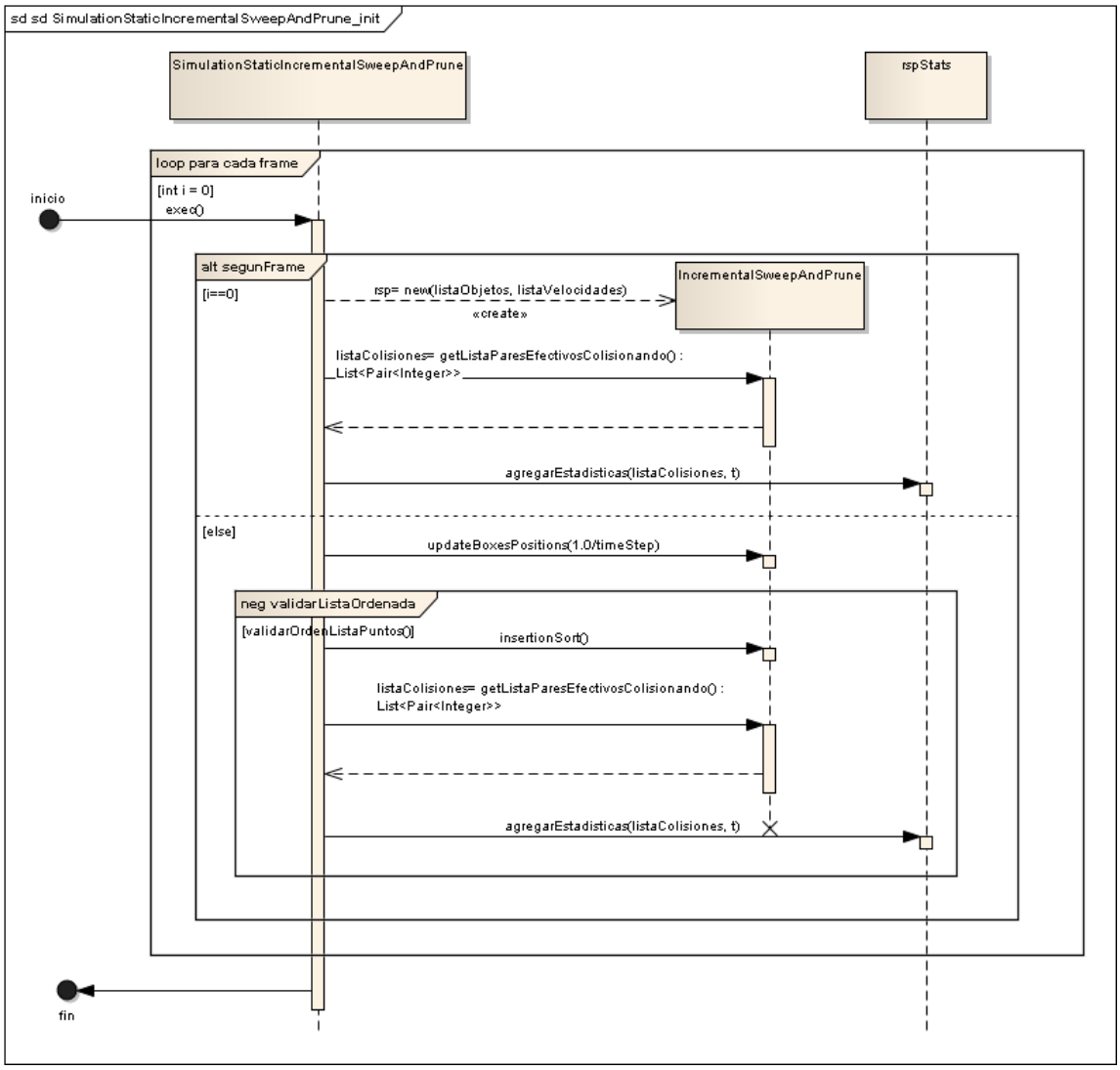


Figura B.7: Inicialización del algoritmo Sweep and Prune Incremental

B.2. Implementación del método Sweep and Prune cinético

El método Sweep and Prune cinético para su implementación necesita de la biblioteca *StaticLibray y*. Está compuesto de los paquetes actuadores, eventos y comparadores, como lo muestra la figura B.8. El diagrama de clases asociado a la biblioteca se puede ver en la figura B.9. El algoritmo KSP, está dividido en dos partes: la inicialización de todas las estructuras y el procesamiento de los eventos por parte de la cola de eventos, parte de lo que se puede ver en el diagrama de clases B.10. Estructuralmente el método consiste en la interacción de un arreglo de listas cinéticas (una por dimensión), una cola de eventos (implementada usando la clase PriorityQueue de Java), una lista de colisiones y un archivo encargado de registrar las estadísticas. Por medio de una clase ObservadorEventos que es el encargado de redireccionar hacia una clase Actuador cuando se realiza uno de los eventos definidos: INICIALIZARESTRUCTURAS, EVENTOSIMPLE, EVENTOMULTIPLE. Esta idea está basada en el patrón **Observer**, dado que el procesamiento de la cola de eventos es el gatillante de los distintos eventos y se requería independencia de responsabilidades entre las distintas componentes del algoritmo. El Actuador se encarga de ejecutar las diferentes acciones que requiere cada evento.

B.2.1. Tipos de eventos en el algoritmo KSP

El primer evento a realizarse es la inicialización de las diferentes estructuras, que están representadas en la figura B.11. Una vez inicializadas las estructuras, y llenadas la lista de colisiones y la lista de eventos iniciales, se procede a la ejecución del algoritmo, representado por el diagrama de la figura B.12.

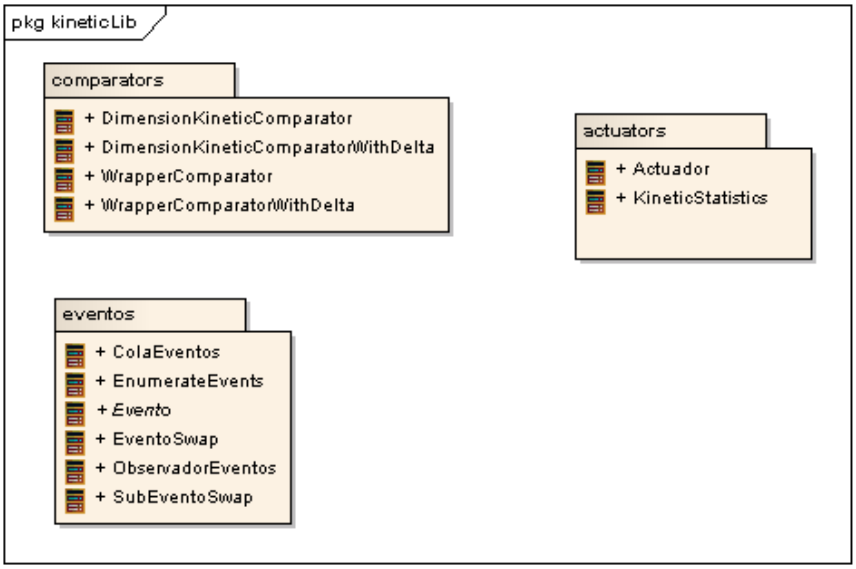


Figura B.8: Diagrama de paquetes de la biblioteca kineticLib

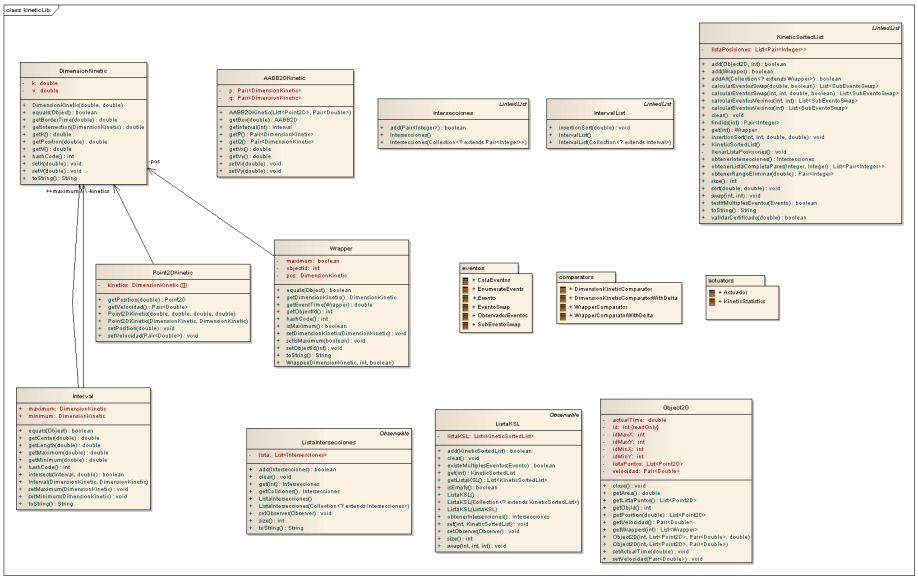


Figura B.9: Diagrama de clases del paquete kineticLib

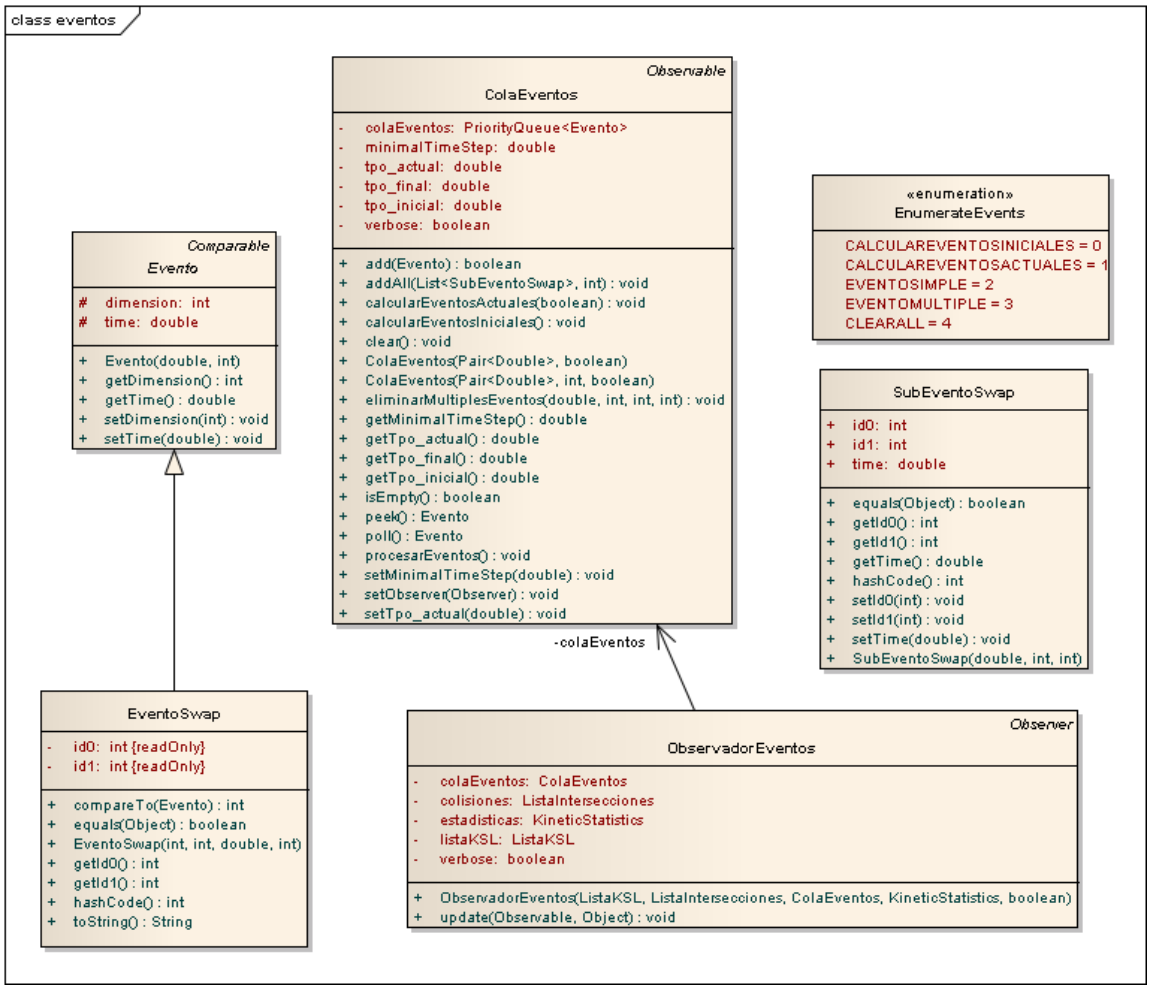


Figura B.10: Diagrama de clase asociados a los eventos

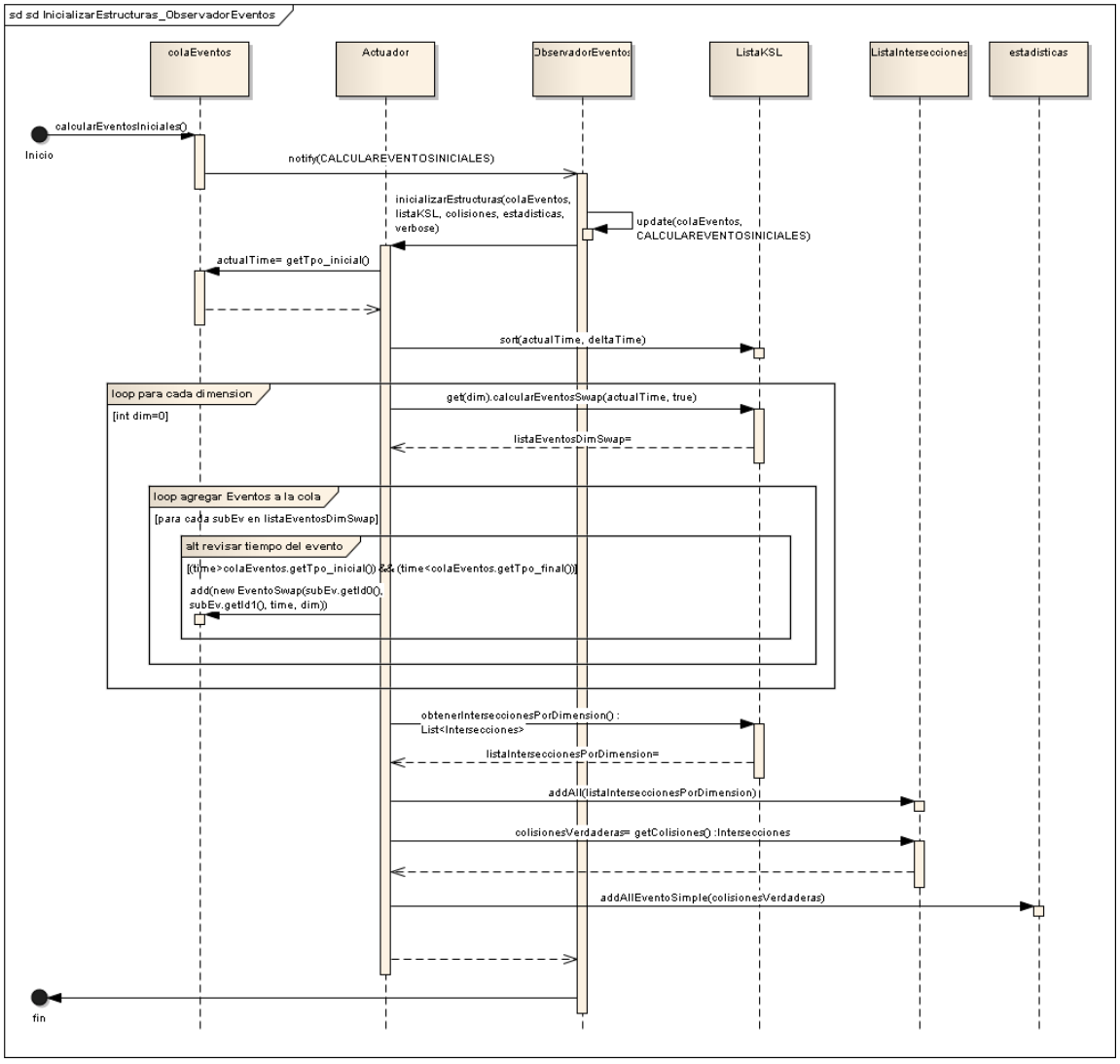


Figura B.11: Diagrama de secuencia del observador de eventos para inicializar todas las estructuras

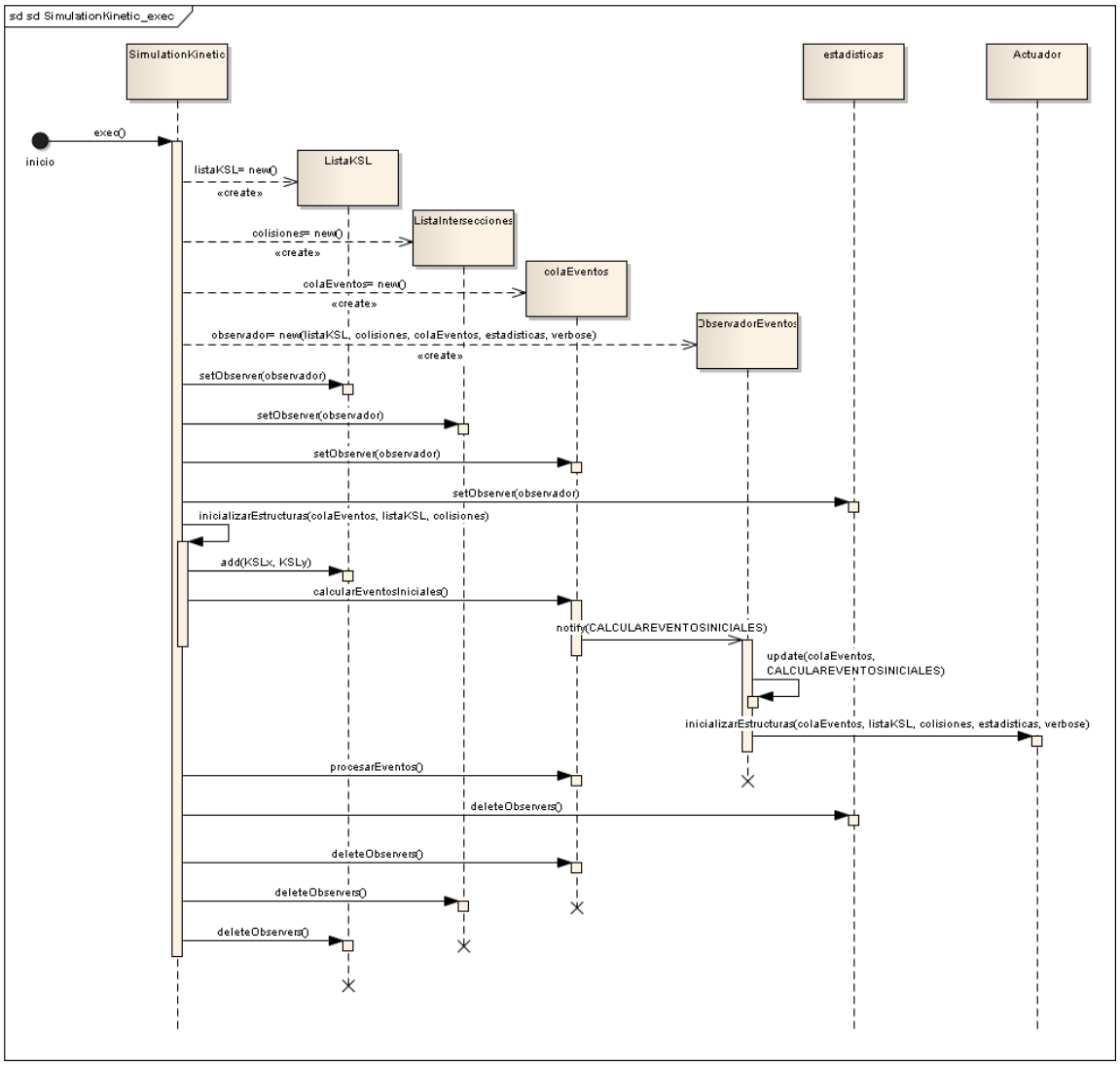


Figura B.12: Ejecución del algoritmo cinético

Cuando se envía un evento, lo primero que se debe establecer es si este evento es múltiple. Debido a que esto no es comentado en [10], pero que puede tener consecuencias irremediables en el orden de la lista cinética de acuerdo a lo establecido por [2]. Existen dos formas de evaluar si un evento simple forma parte de un evento múltiple: extrayendo dos eventos consecutivos de la cola de eventos y revisar que estos eventos suceden en el mismo instante de tiempo, en la misma dimensión y uno de los índices de los puntos del evento, se repite en el otro evento o, suponer que el evento es simple y comparar en la vecindad de lista cinética donde ocurre el evento si tres o más elementos se encuentran en la misma coordenada. En este caso, se evalúa si en el tiempo del evento existen más de dos puntos en la misma coordenada en la misma dimensión del evento. Si el evento es simple, se procesa de acuerdo a lo representado en la figura B.13.

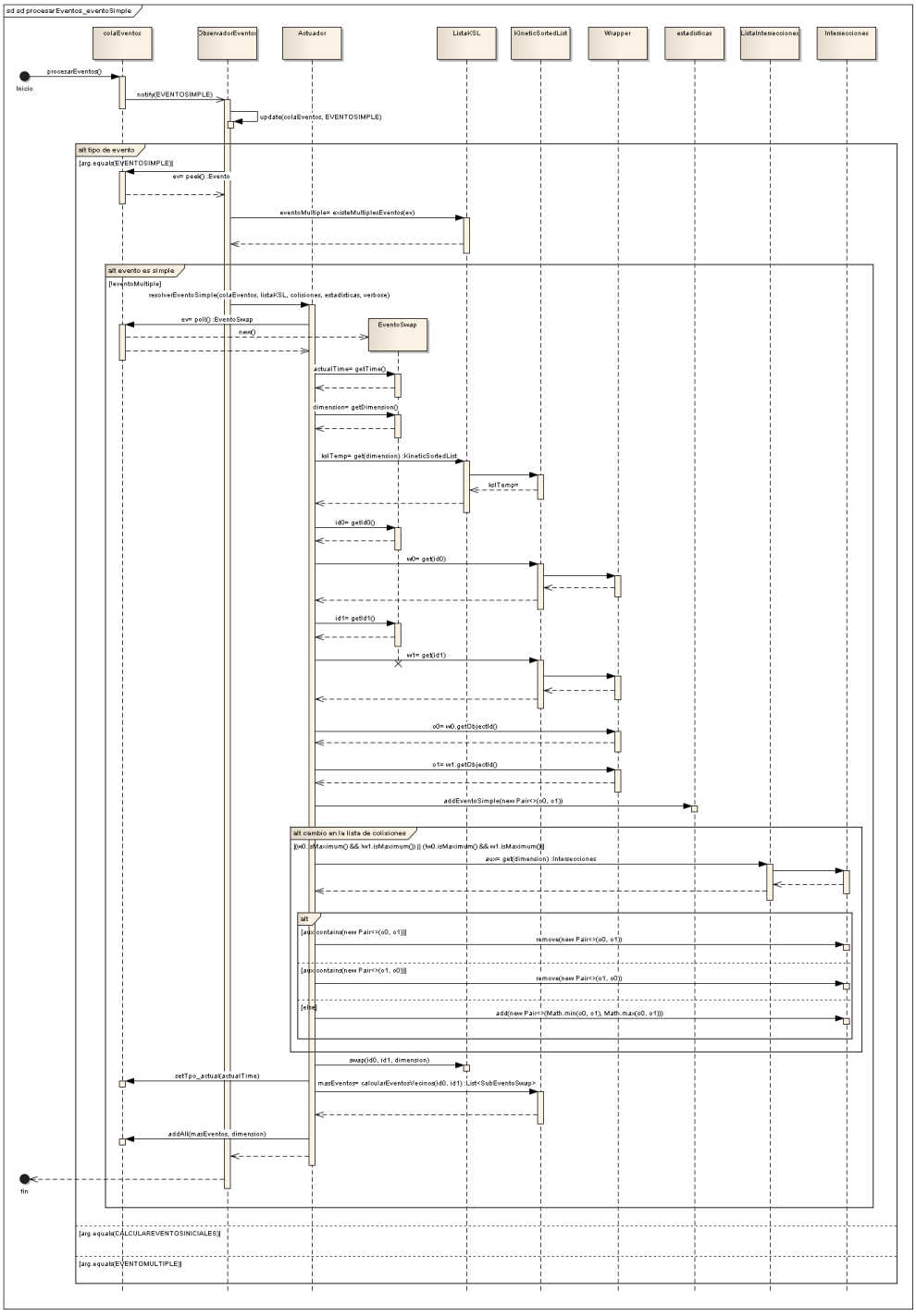


Figura B.13: Procesar Evento simple en el algoritmo cinético

El procesamiento de un evento simple consta de los pasos siguientes:

- Extracción del evento de la cola de eventos.
- Invalidar todos los eventos de la cola cuyos objetos están involucrados en el evento que está siendo procesado. En el caso de las simulaciones realizadas en este estudio, debido a las trayectorias lineales de los objetos no existen otros eventos en la cola asociados a los mismos objetos, al menos en la misma dimensión.
- Ejecutar el intercambio de posiciones en la lista cinética respectiva, dada por la dimensión donde sucede el evento.
- Calcular los nuevos eventos generados con los nuevos vecinos, asociados al paso anterior, los cuales son agregados en la cola de eventos. Cabe mencionar que como máximo son dos los eventos posibles a ser agregados.

En el caso de un evento múltiple, es fundamental que se mantenga el orden correcto de los elementos en las listas cinéticas y se haya eliminado todos los eventos simples asociados a este evento múltiple. Para tal efecto, se definen los siguientes pasos, que se pueden ver en la figura B.14:

- se invalida la lista cinética asociada (de acuerdo a la dimensión donde ocurre el evento).
- Se extrae los elementos de esta lista involucrados en el evento múltiple.
- Elimino de la cola de eventos los futuros eventos asociados a los objetos involucrados en la colisión múltiple.
- Se resuelve el orden final de los elementos en la lista con un pequeño $\delta(t)$ y se reinsertan en la lista cinética respectiva.
- Se calculan los futuros eventos asociados al nuevo orden de los elementos y son agregados a la cola de eventos.

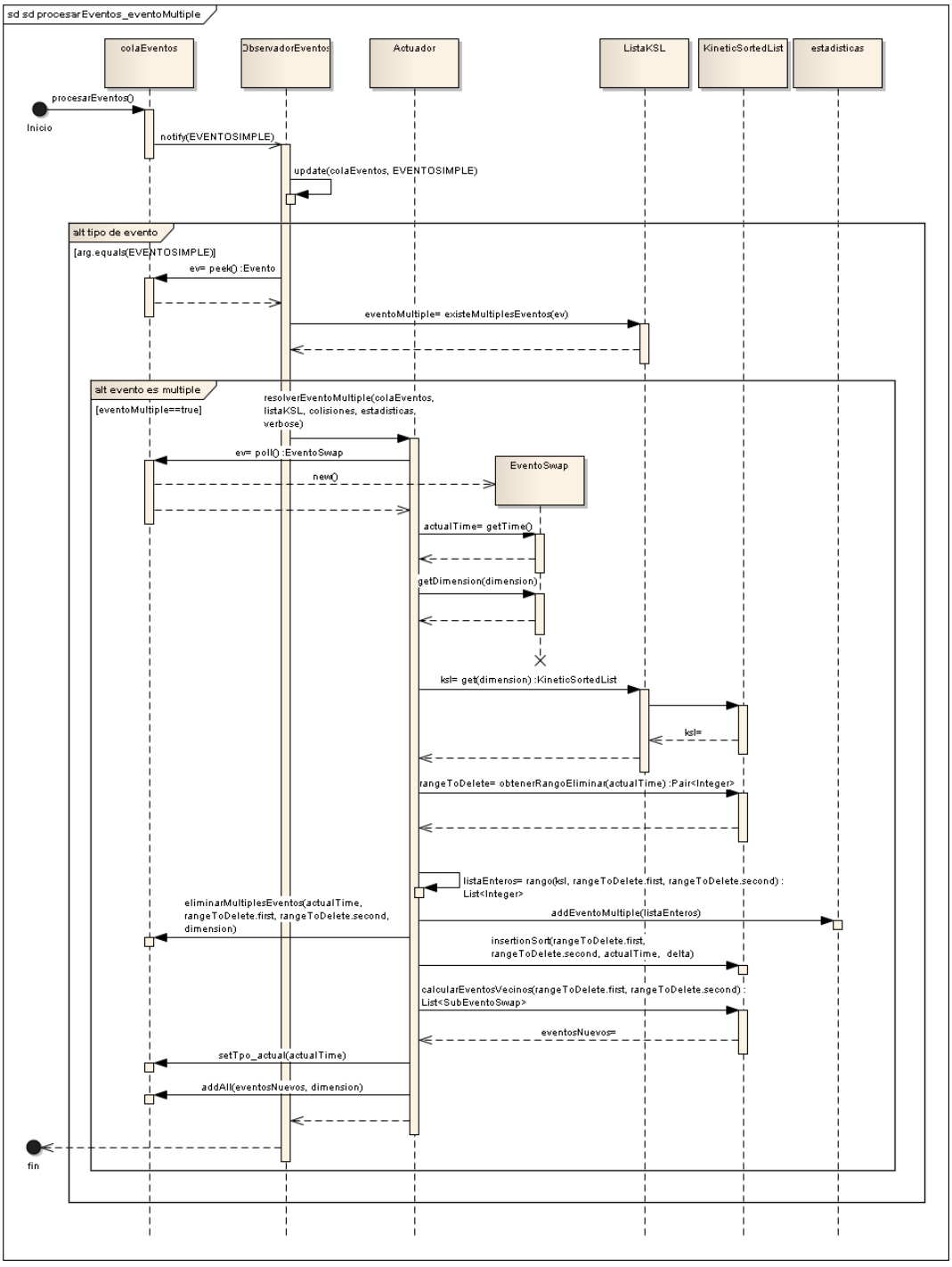


Figura B.14: Procesar evento múltiple en el algoritmo cinético

B.3. Los modelos

Como se describió en la sección 3.5, existen 4 modelos para probar distintos aspectos de la aplicación. El modelo *TestIncrementalSP* sirve para probar como una cantidad de rombos concéntricos se desplazan alternadamente a la izquierda o a la derecha y con ello probar las colisiones que se producirán en un intervalo de 1 segundo con un 10 FPS. El modelo *TestEquidistribucion* genera una cantidad de $n * n$ triángulos del mismo tamaño, los que se desplazan en direcciones aleatorias, este modelo se usa con el algoritmo SPI para probar la generación de las listas de objetos colisionando. El modelo *TresObjetos* se utilizó para corroborar el correcto funcionamiento de los tres algoritmos a evaluar, el que se encuentra detallado en la sección 4.2. El modelo *TestAleatorio* fue el utilizado para la medición de desempeño de los tres algoritmos. En este modelo se generan de manera aleatoria objetos de distintas formas que están inscritos dentro del círculo unitario, los que luego son escalados según el área que ocupan dentro del círculo unitario. Por ejemplo, para un triángulo, su área dentro del círculo unitario es $Area_{\triangle} = \frac{3*\sqrt{3}}{4}$ y para poder calcular el factor de escala, dado por R, obtenemos:

$$Area_{deseada} = \frac{3 * \sqrt{3}}{4} R^2 = Area_{\triangle} R^2$$

y despejando R se obtiene que $R = \sqrt{\frac{Area_{deseada}}{Area_{\triangle}}}$. Para cada tipo de objeto existe un procedimiento similar.
Los objetos así creados son posicionados dentro del área inicial de la simulación de manera aleatoria y la magnitud de su velocidad generada al azar en base al rango $[0.5R, 3.5R]$, con dirección aleatoria. La figura B.15 muestra un diagrama general de este modelo. En él se muestra la creación de tres objetos que heredan de la interfaz *Simulation*, hecha para la ejecución de cada una de las simulaciones. Los objetos creados son: un objeto de la clase *SimulationKinetic* que se encarga de la ejecución del método Sweep and Prune cinético, un objeto de la clase *SimulationStaticDirectSweepAndPrune* que se encarga de la ejecución del método estático sweep and prune directo, y un objeto de la clase *SimulationStaticIncrementalSweepAndPrune* que se encarga de la ejecución del método estático sweep and prune incremental. En cualquiera de los tres casos, el constructor se encarga de dejar todos los objetos creados y listos para la ejecución del algoritmo, que se realiza mediante la ejecución del método **exec**.

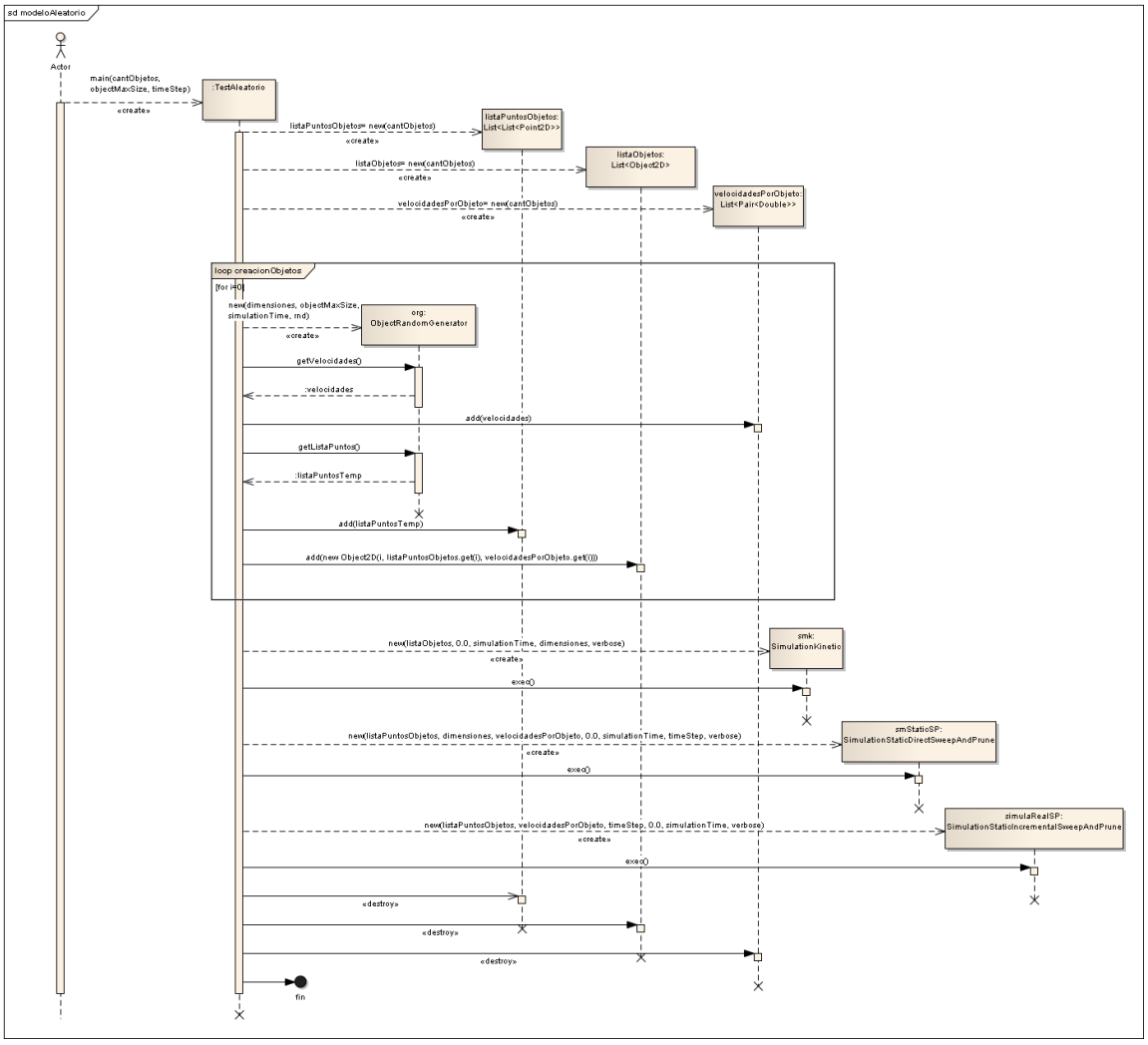


Figura B.15: Diagrama de secuencia de la ejecución del modelo de simulación aleatoria

B.4. Los modelos de pruebas unitarias

Dentro de la aplicación existen test unitarios para la gran mayoría de los métodos de las clases en los proyectos *StaticLibrary* y *KineticLibray*, que en algunos casos se prueban para varios escenarios. En total son 556 tests unitarios distribuidos en: 388 en la biblioteca estática, 168 en la biblioteca cinética.

B.5. Tablas resumen de ajustes de curvas para experimentos

B.5.1. Sweep And Prune Directo

Cálculo de Colisiones

Las siguientes tablas muestra el comportamiento del algoritmo que cálculo de colisiones que forma parte del algoritmo Sweep And Prune Directo, para diferentes valores de densidad de objetos (1, 2, 5 y 10 %). Se mostrará el proceso de selección de la curva de mejor ajuste para las dos primeras tablas, de acuerdo al criterio establecido en [1]. El proceso es idéntico para el resto de los análisis, por lo que solamente se indicará el resultado en aquellos casos.

Mediciones	**sublineal**[1]	**lineal**[2]	**subcuadrática**[3]	**cuadrática**[4]
SSE	$6.8385 * 10^9$	$4.1190 * 10^7$	$5.0249 * 10^6$	$1.3678 * 10^6$
R^2	0.8594	0.9915	0.9999	1.0000
dfe	8	7	6	5
R^2_{adj}	0.8419	0.9891	0.9998	0.9999
RMSE	$2.9237 * 10^4$	$7.6709 * 10^3$	915.1439	523.0307

Tabla B.1: Estadísticas de ajuste para el método de cálculo de colisiones con un 1 % de densidad de objetos.

En la tabla B.1 se puede observar que en la medida que se aumenta el grado de la curva, el valor de R^2_{adj} va acercándose al valor 1.0 y el valor de $RMSE$ se va reduciendo. Si en alguna curva el valor de R^2_{adj} no aumenta y el valor del $RMSE$ disminuye marginalmente o crece, es señal que la curva de mayor grado no aporta al ajuste.

Para asegurar que, entre dos curvas de distinto grado, la de mayor grado aporta al calce de la curva, se debe observar que los valores de los coeficientes superiores no tienen un intervalo que cruce por cero, ya que esto no asegura que el valor de esos coeficientes sean 0 en algún ajuste. Si eso sucede con alguno de los coeficientes superiores, implica que se estaría haciendo un sobre-ajuste de la curva. De acuerdo a la tabla B.2, desde la curva lineal hacia arriba, el primer coeficiente (a) tiene un rango de valores que son del mismo

[1] ecuación del tipo $a\log(x) + b$
[2] ecuación del tipo $ax + b\log(x) + c$
[3] ecuación del tipo $ax\log(x) + bx + c\log(x) + d$
[4] ecuación del tipo $ax^2 + bx\log(x) + cx + d\log(x) + e$

Tipo Curva	Coeficientes
Logarítmica	$a = 6607\ (-3027, 1.624 * 10^4)$ $b = -1.037 * 10^4\ (-4.989 * 10^4, 2.915 * 10^4)$
Lineal	$a = 253.4\ (223.7, 283.1)$ $b = -3774\ (-7905, 357.6)$ $c = 2811\ (-9160, 1.478 * 10^4)$
Subcuadrática	$a = 80.17\ (69.85, 90.48)$ $b = -331.5\ (-406.8, -256.2)$ $c = 1165\ (230.7, 2099)$ $d = -187.2\ (-1823, 1449)$
Cuadrática	$a = 0.0512\ (0.02083, 0.08157)$ $b = 38.16\ (12.99, 63.33)$ $c = -86.4\ (-236.2, 63.36)$ $d = 245.6\ (-634.1, 1125)$ $e = 17.84\ (-1001, 1036)$

Tabla B.2: Coeficientes de ajuste para distintas curvas del método de cálculo de colisiones junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 1 %

signo, indicando que incluso la curva cuadrática aporta al ajuste de los datos originales, lo que se traduce en que el mejor ajuste es de tipo cuadrático.

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$1.2382 * 10^{10}$	$5.7283 * 10^{8}$	$1.3502 * 10^{6}$	$1.3400 * 10^{6}$
R^2	0.8600	0.9935	1.0000	1.0000
dfe	8	7	6	5
R^2_{adj}	0.8426	0.9917	1.0000	1.0000
RMSE	$3.9341 * 10^{4}$	$9.0461 * 10^{3}$	474.3681	517.6965

Tabla B.3: Estadísticas de ajuste para el método de cálculo de colisiones con un 2 % de densidad de objetos.

Tipo Curva	Coeficientes
Logarítmica	$a = 9155\ (-3809, 2.212 * 10^4)$ $b = -1.442 * 10^4\ (-6.76 * 10^4, 3.876 * 10^4)$
Lineal	$a = 343.1\ (308.1, 378.2)$ $b = -5472\ (-1.034 * 10^4, -599.3)$ $c = 4051\ (-1.007 * 10^4, 1.817 * 10^4)$
Subcuadrática	$a = 99.47\ (94.12, 104.8)$ $b = -385.5\ (-424.5, -346.5)$ $c = 1126\ (641.4, 1610)$ $d = -8.705\ (-856.7, 839.3)$
Cuadrática	$a = 0.0185\ (-0.012, 0.04811)$ $b = 83.95\ (59.04, 108.9)$ $c = -294.1\ (-442.4, -145.9)$ $d = 800.2\ (-70.59, 1671)$ $e = 55.61\ (-952.6, 1064)$

Tabla B.4: Coeficientes de ajuste para distintas curvas del método de cálculo de colisiones junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 2 %

En la tabla B.3 se aprecia que entre las curvas subcuadrática y cuadrática el valor del RMSE aumenta, aunque los valores de R^2_{adj} son en ambos casos 1.0000. Al observar la tabla B.4, el coeficiente a de la curva cuadrática tiene un intervalo de confianza con valores con signo contrario, indicando que la curva más ajustada es subcuadrática.

Mediciones	**sublineal**	**lineal**	**subcuadrática**	**cuadrática**
SSE	$3.0685 * 10^{10}$	$2.4334 * 10^{9}$	$3.2420 * 10^{7}$	$8.3414 * 10^{4}$
R^2	0.8563	0.9886	0.9998	1.0000
dfe	8	7	6	5
R^2_{adj}	0.8384	0.9854	0.9998	1.0000
RMSE	$6.1933 * 10^{4}$	$1.8654 * 10^{4}$	$2.3245 * 10^{3}$	128.1241

Tabla B.5: Estadísticas de ajuste para el método de cálculo de colisiones con un 5 % de densidad de objetos.

Tipo Curva	**Coeficientes**
Logarítmica	$a = 1.359 * 10^4 \ (-6813, 3.4 * 10^4)$ $b = -2.141 * 10^4 \ (-1.051 * 10^4, 6.23 * 10^4)$
Lineal	$a = 538.6 \ (466.3, 610.8)$ $b = -8903 \ (-1.894 * 10^4, 1139)$ $c = 6768 \ (-2.233 * 10^4, 3.587 * 10^4)$
Subcuadrática	$a = 169.8 \ (143.6, 196)$ $b = -704.3 \ (-895.6, -513.1)$ $c = 2229 \ (-143.8, 4602)$ $d = -276.8 \ (-4432, 3879)$
Cuadrática	$a = 0.1658 \ (0.1584, 0.1733)$ $b = 65.53 \ (59.37, 71.7)$ $c = -137.1 \ (-173.7, -100.4)$ $d = 322.4 \ (106.9, 537.9)$ $e = 151.3 \ (98.24, 400.8)$

Tabla B.6: Coeficientes de ajuste para distintas curvas del método de cálculo de colisiones junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 5 %

De acuerdo a lo que indica la tabla B.5, y que corroboran los valores de los coeficientes en la tabla B.6, es que el mejor ajuste para la curva del cálculo de colisiones de 5 % de densidad es la curva cuadrática.

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$5.3021 * 10^{10}$	$3.1367 * 10^{9}$	$2.4401 * 10^{7}$	$4.1273 * 10^{6}$
R^2	0.8581	0.9916	0.9999	1.0000
dfe	8	7	6	5
R^2_{adj}	0.8404	0.9892	0.9999	1.0000
RMSE	$8.1410 * 10^{4}$	$2.1168 * 10^{4}$	$2.0166 * 10^{3}$	908.5464

Tabla B.7: Estadísticas de ajuste para el método de cálculo de colisiones con un 10 % de densidad de objetos.

Tipo Curva	Coeficientes
Logarítmica	$a = 1.791 * 10^4 \ (-8918, 4.473 * 10^4)$ $b = -2.822 * 10^4 \ (-1.383 * 10^5, 8.181 * 10^4)$
Lineal	$a = 708.1 \ (626.1, 790.1)$ $b = -1.149 * 10^4 \ (-2.289 * 10^4, -84.58)$ $c = 8520 \ (-2.452 * 10^4, 4.155 * 10^4)$
Subcuadrática	$a = 221.9 \ (199.1, 244.6)$ $b = -914.2 \ (-1080, -748.3)$ $c = 2761 \ (702.2, 4820)$ $d = -254.7 \ (-3860, 3350)$
Cuadrática	$a = 0.1075 \ (0.05477, 0.1603)$ $b = 139.5 \ (95.77, 183.2)$ $c = -441.7 \ (-701.8, -181.5)$ $d = 1142 \ (-386.1, 2670)$ $e = 36.19 \ (-1733, 1806)$

Tabla B.8: Coeficientes de ajuste para distintas curvas del método de cálculo de colisiones junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 10 %

De acuerdo a lo que indica la tabla B.7, y que corroboran los valores de los coeficientes en la tabla B.8, es que el mejor ajuste para la curva del cálculo de colisiones de 10 % de densidad es la curva cuadrática.

Ordenamiento

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$1.9320 * 10^9$	$1.9552 * 10^6$	$1.7291 * 10^6$	$6.8823 * 10^5$
R^2	0.8887	0.9999	0.9999	1.0000
dfe	8	7	6	5
R^2_{adj}	0.8748	0.9999	0.9998	0.9999
RMSE	$1.554 * 10^4$	528.5058	536.8324	371.0077

Tabla B.9: Estadísticas de ajuste para el método de ordenamiento con un 1 % de densidad de objetos.

Tipo Curva	Coeficientes
Logarítmica	$a = 6998\ (1877, 1.212 * 10^4)$ $b = -1.046 * 10^4\ (-3.146 * 10^4, 1.055 * 10^4)$
Lineal	$a = 134.5\ (132.4, 136.5)$ $b = 44.6\ (-240.1, 329.2)$ $c = 60.93\ (-763.9, 885.7)$
Subcuadrática	$a = -2.706\ (-8.757, 3.345)$ $b = 153.7\ (109.6, 197.9)$ $c = -36.98\ (-585, 511.1)$ $d = 82.75\ (-876.9, 1042)$
Cuadrática	$a = -0.02045\ (-0.04199, 0.00109)$ $b = 9.453\ (-8.4, 27.31)$ $c = 88.87\ (-17.37, 195.1)$ $d = 155.9\ (-468.1, 780)$ $e = 61.96\ (-660.6, 784.5)$

Tabla B.10: Coeficientes de ajuste para distintas curvas del método de ordenamiento junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 1 %

Puede observarse en la tabla B.9 que los valores de R^2_{adj} no varían a partir del ajuste lineal. Corroborando la situación con la tabla de coeficientes B.10, se observa que los primeros parámetros para las curvas subcuadrática y cuadrática tienen rangos de valores que cruzan por 0, lo que no da garantía que esos coeficientes aporten al ajuste de la curva. Por lo que la curva lineal es la de mejor ajuste.

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$1.9546 * 10^9$	$6.6373 * 10^6$	$3.7063 * 10^6$	$6.7425 * 10^5$
R^2	0.8902	0.9996	0.9998	1.0000
dfe	8	7	6	5
R^2_{adj}	0.8765	0.9995	0.9997	0.9999
RMSE	$1.5631 * 10^4$	973.7483	785.9545	367.2179

Tabla B.11: Estadísticas de ajuste para el método de ordenamiento con un 2 % de densidad de objetos.

Tipo Curva	Coeficientes
Logarítmica	$a = 7295\ (2145, 1.245 * 10^4)$ $b = -1.097 * 10^4\ (-3.209 * 10^4, 1.016 * 10^4)$
Lineal	$a = 135.3\ (131.6, 139.1)$ $b = 46.65\ (-477.8, 571.1)$ $c = 52.46\ (-1467, 1572)$
Subcuadrática	$a = -1.806\ (-10.66, 7.054)$ $b = 148.3\ (83.63, 213)$ $c = -28.33\ (-830.7, 774)$ $d = 91.05\ (-1314, 1496)$
Cuadrática	$a = -0.04816\ (-0.06948, -0.02684)$ $b = 24.95\ (7.279, 42.62)$ $c = 8.653\ (-96.49, 113.8)$ $d = 412.7\ (-205, 1030)$ $e = 27.74\ (-687.4, 742.9)$

Tabla B.12: Coeficientes de ajuste para distintas curvas del método de ordenamiento junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 2 %

Puede observarse en la tabla B.11 que el valor de R^2_{adj} crece en la medida que la curva de ajuste aumenta de grado, además de disminuir el valor del $RMSE$. Esto implica que el mejor ajuste está dado por una curva cuadrática. Esto se puede corroborar con la tabla de coeficientes B.12, dado que entre la curva lineal y la cuadrática no cruza por cero, excepto en la curva subcuadrática.

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$2.0673 * 10^9$	$1.2803 * 10^5$	$1.6012 * 10^5$	$1.8509 * 10^5$
R^2	0.8873	1.0000	1.0000	1.0000
dfe	8	7	6	5
R^2_{adj}	0.8732	1.0000	1.0000	1.0000
RMSE	$1.6075 * 10^4$	135.2381	163.3592	192.4016

Tabla B.13: Estadísticas de ajuste para el método de ordenamiento con un 5 % de densidad de objetos.

Tipo Curva	Coeficientes
Logarítmica	$a = 7408\ (2111, 1.27 * 10^4)$ $b = -1.116 * 10^4\ (-3.289 * 10^4, 1.057 * 10^4)$
Lineal	$a = 139.2\ (138.7, 139.7)$ $b = 10.24\ (-62.59, 83.08)$ $c = 57.18\ (-153.9, 268.2)$
Subcuadrática	$a = 0.9746\ (-0.8668, 2.816)$ $b = 132.2\ (118.7, 145.6)$ $c = 55.67\ (-111.1, 222.4)$ $d = 38.55\ (-253.5, 330.6)$
Cuadrática	$a = 3.462 * 10^{-4}\ (-1.083 * 10^{-2}, 1.152 * 10^{-2})$ $b = 0.662\ (-8.597, 9.92)$ $c = 134\ (78.93, 189.1)$ $d = 48.12\ (-275.5, 371.7)$ $e = 36.68\ (-338, 411.4)$

Tabla B.14: Coeficientes de ajuste para distintas curvas del método de ordenamiento junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 5 %

Siguiendo los mismos análisis anteriores, la curva que mejor ajusta para el 5 % es la curva lineal, ya que desde la curva lineal a las de orden superior, su R^2_{adj} no varía, dando valores de $RMSE$ mayores, en la medida que aumenta el orden de la curva, lo que indica que en esos casos se estaría haciendo un sobreajuste. Puede corroborarse con los datos de la tabla de coeficientes B.14 donde los rangos del primer coeficiente tanto para la curva subcuadrática como para la cuadrática, poseen valores que cruzan por cero.

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$1.7598 * 10^9$	$1.3569 * 10^6$	$8.8764 * 10^5$	$5.2461 * 10^5$
R^2	0.8903	0.9999	0.9999	1.0000
dfe	8	7	6	5
R^2_{adj}	0.8766	0.9999	0.9999	0.9999
RMSE	$1.4832 * 10^4$	440.2701	384.6305	323.9152

Tabla B.15: Estadísticas de ajuste para el método de ordenamiento con un 10 % de densidad de objetos.

Tipo Curva	Coeficientes
Logarítmica	$a = 6944\ (2057, 1.183 * 10^4)$ $b = -1.036 * 10^4\ (-3.041 * 10^4, 9689)$
Lineal	$a = 128.5\ (126.7, 130.2)$ $b = 196.3\ (-40.83, 433.4)$ $c = -36.87\ (-723.9, 650.2)$
Subcuadrática	$a = -5.947\ (-10.28, -1.611)$ $b = 171.3\ (139.7, 203)$ $c = -80.02\ (-472.7, 312.6)$ $d = 77.15\ (-610.4, 764.7)$
Cuadrática	$a = -0.008205\ (-0.02701, 0.0106)$ $b = 0.4062\ (-15.99, 15.18)$ $c = 140.4\ (47.65, 233.1)$ $d = 51.23\ (-493.6, 596.1)$ $e = 36.55\ (-594.3, 667.4)$

Tabla B.16: Coeficientes de ajuste para distintas curvas del método de ordenamiento junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 10 %

Las tablas B.15 y B.16, muestran que la mejor curva de ajuste está dada por la curva subcuadrática.

B.5.2. Sweep And Prune Incremental

Colisiones

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$7.9458 * 10^6$	$1.41 * 10^5$	$2.1642 * 10^3$	623.4262
R^2	0.8637	0.9976	1.0000	1.0000
dfe	8	7	6	5
R^2_{adj}	0.8467	0.9969	0.9999	1.0000
RMSE	996.6094	141.9252	18.9921	11.1663

Tabla B.17: Estadísticas de ajuste para el método de cálculo de colisiones con un 1 % de densidad de objetos.

Tipo Curva	Coeficientes
Logarítmica	$a = 268\ (-60.39, 596.4)$ $b = -423\ (-1770, 924)$
Lineal	$a = 8.752\ (8.202, 9.301)$ $b = -141.8\ (-218.2, -65.34)$ $c = 104\ (-117.5, 325.5)$
Subcuadrática	$a = 1.768\ (1.554, 1.982)$ $b = -4.313\ (-5.876, -2.751)$ $c = -6.568\ (-25.96, 12.82)$ $d = 20.68\ (-13.27, 54.63)$
Cuadrática	$a = -0.001491\ (-0.00214, -0.000843)$ $b = 2.82\ (2.283, 3.358)$ $c = -10.2\ (-13.39, -6.999)$ $d = 8.473\ (-10.31, 27.25)$ $e = 22.97\ (1.222, 44.71)$

Tabla B.18: Coeficientes de ajuste para distintas curvas del método de cálculo de colisiones junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 1 %

Al observar la tabla B.17, el valor del R^2_{adj} aumenta y $RMSE$ disminuye cuando mayor sea el grado de la curva de ajuste. Al verificar los coeficientes en la tabla B.18, se tiene que tanto para las curvas subcuadrática como cuadrática el coeficiente mayor no posee cruce por 0, lo que implica que la mejor curva de ajuste es cuadrática.

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$1.2929 * 10^7$	$2.3407 * 10^5$	$1.4128 * 10^4$	$2.2445 * 10^3$
R^2	0.8638	0.9975	0.9999	1.0000
dfe	8	7	6	5
R^2_{adj}	0.8468	0.9968	0.9998	1.0000
RMSE	$1.2713 * 10^3$	182.8629	48.5241	21.1873

Tabla B.19: Estadísticas de ajuste para el método de cálculo de colisiones con un 2 % de densidad de objetos.

Tipo Curva	Coeficientes
Logarítmica	$a = 343.7\ (-75.17, 762.6)$ $b = -543.1\ (-2261, 1175)$
Lineal	$a = 11.19\ (10.48, 11.9)$ $b = -184.6\ (-283.1, -86.12)$ $c = 130.4\ (-154.9, 415.8)$
Subcuadrática	$a = 2.252\ (1.705, 2.799)$ $b = -5.492\ (-9.485, -1.5)$ $c = -5.005\ (-54.54, 4453)$ $d = 17.03\ (-69.71, 103.8)$
Cuadrática	$a = -0.002283\ (-0.003513, -0.001053)$ $b = 3.897\ (2.878, 4.917)$ $c = -14.73\ (-20.8, -8.666)$ $d = 16.29\ (-19.35, 51.92)$ $e = 26.27\ (-14.99, 67.54)$

Tabla B.20: Coeficientes de ajuste para distintas curvas del método de cálculo de colisiones junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 2 %

Al igual que el caso anterior, viendo las tablas B.19 y B.20, se tiene que tanto para las curvas subcuadrática como cuadrática el coeficiente mayor no posee cruce por 0, lo que implica que la mejor curva de ajuste es cuadrática.

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$2.9871 * 10^7$	$1.5646 * 10^6$	$2.1364 * 10^4$	$1.6705 * 10^3$
R^2	0.8601	0.9927	0.9999	1.0000
dfe	8	7	6	5
R^2_{adj}	0.8426	0.9906	0.9998	1.0000
RMSE	$1.9323 * 10^3$	472.7795	59.6707	18.2781

Tabla B.21: Estadísticas de ajuste para el método de cálculo de colisiones con un 5 % de densidad de objetos.

Tipo Curva	Coeficientes
Logarítmica	$a = 481.8\ (-155, 1118)$ $b = -769.8\ (-3382, 1842)$
Lineal	$a = 16.82\ (14.98, 18.65)$ $b = -257.1\ (-511.8, -2.484)$ $c = 191.6\ (-546.2, 929.4)$
Subcuadrática	$a = 4.255\ (3.582, 4.927)$ $b = -14.34\ (-19.25, -9.429)$ $c = 23.67\ (-37.25, 84.58)$ $d = 16.59\ (-90.07, 123.3)$
Cuadrática	$a = 0.004142\ (0.003081, 0.005204)$ $b = 1.737\ (0.8573, 2.616)$ $c = -0.7864\ (-6.02, 4.447)$ $d = -21.18\ (-51.93, -9.56)$ $e = 24.05\ (-11.54, 59.65)$

Tabla B.22: Coeficientes de ajuste para distintas curvas del método de cálculo de colisiones junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 5 %

Viendo las tablas B.21 y B.22, se tiene que tanto para las curvas subcuadrática como cuadrática el coeficiente mayor no posee cruce por 0, lo que implica que la mejor curva de ajuste es cuadrática.

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$5.5580 * 10^7$	$4.7790 * 10^6$	$9.5397 * 10^4$	$2.1685 * 10^4$
R^2	0.8563	0.9876	0.9998	0.9999
dfe	8	7	6	5
R^2_{adj}	0.8383	0.9841	0.9996	0.9999
RMSE	$2.6358 * 10^3$	826.2615	126.0932	65.8556

Tabla B.23: Estadísticas de ajuste para el método de cálculo de colisiones con un 10 % de densidad de objetos.

Tipo Curva	Coeficientes
Logarítmica	$a = 569.6\ (-298.9, 1438)$ $b = -908.6\ (-4471, 2654)$
Lineal	$a = 22.88\ (19.68, 26.08)$ $b = -370.7\ (-815.7, 74.35)$ $c = 286.1\ (-1003, 1576)$
Subcuadrática	$a = 7.019\ (5.598, 8.44)$ $b = -28.31\ (-38.68, -17.94)$ $c = 60.1\ (-68.63, 188.8)$ $d = 15.14\ (-210.3, 240.5)$
Cuadrática	$a = 0.008466\ (0.004642, 0.01229)$ $b = 1.763\ (-1.406, 4.932)$ $c = 0.01541\ (-18.84, 18.87)$ $d = -9.075\ (-119.8, 101.7)$ $e = 11.05\ (-117.2, 139.3)$

Tabla B.24: Coeficientes de ajuste para distintas curvas del método de cálculo de colisiones junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 10 %

Al observar las tablas B.23 y B.24, se tiene que tanto para las curvas subcuadrática como cuadrática el coeficiente mayor no posee cruce por 0, lo que implica que la mejor curva de ajuste es cuadrática.

Ordenamiento

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$5.5337 * 10^7$	$2.8863 * 10^5$	$5.7166 * 10^5$	$3.5769 * 10^4$
R^2	0.8796	0.9994	0.9988	0.9999
dfe	8	7	6	5
R^2_{adj}	0.8646	0.9992	0.9981	0.9999
RMSE	$2.63 * 10^3$	203.0574	308.6679	84.5804

Tabla B.25: Estadísticas de ajuste para el método de ordenamiento con un 1 % de densidad de objetos.

Tipo Curva	Coeficientes
Logarítmica	$a = 1042\ (175, 1908)$ $b = -1673\ (-5228, 1882)$
Lineal	$a = 22.42\ (21.63, 23.21)$ $b = -129.3\ (-238.6, -19.91)$ $c = 46.32\ (-270.6, 363.2)$
Subcuadrática	$a = -0.5136\ (-3.993, 2.966)$ $b = 26.27\ (0.8757, 51.67)$ $c = -176.8\ (-491.9, 138.3)$ $d = 72.26\ (-479.5, 624)$
Cuadrática	$a = -0.01281\ (-0.01772, -0.007898)$ $b = 8.08\ (4.01, 12.15)$ $c = -21.34\ (-45.56, 2.878)$ $d = -20.46\ (-162.7, 121.8)$ $e = 47.26\ (-117.5, 212)$

Tabla B.26: Coeficientes de ajuste para distintas curvas del método de ordenamiento junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 1 %

De acuerdo a lo observado en la tabla B.25, entre la curva lineal y la subcuadrática, el R^2_{adj} disminuye su valor y el $RMSE$ aumenta, sin embargo, entre las curvas subcuadrática y cuadrática, el valor de R^2_{adj} aumenta y el de $RMSE$ disminuye. Al corroborar esta información con la tabla B.26, se observa que la curva de mejor ajuste es la cuadrática.

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$5.8918 * 10^7$	$1.2622 * 10^6$	$5.7303 * 10^5$	$7.3926 * 10^4$
R^2	0.8827	0.9975	0.9989	0.9999
dfe	8	7	6	5
R^2_{adj}	0.8681	0.9968	0.9983	0.9997
RMSE	$2.7138 * 10^3$	424.626	309.0383	121.5947

Tabla B.27: Estadísticas de ajuste para el método de ordenamiento con un 2 % de densidad de objetos.

Tipo Curva	Coeficientes
Logarítmica	$a = 1168\ (273.8, 2062)$ $b = -1878\ (-5547, 1790)$
Lineal	$a = 23.19\ (21.54, 24.83)$ $b = -109.3\ (-338, 119.4)$ $c = 28.45\ (-634.2, 691.1)$
Subcuadrática	$a = -2.044\ (-5.527, 1.44)$ $b = 38.09\ (12.67, 63.52)$ $c = -232.2\ (-547.7, 83.26)$ $d = 87.62\ (-464.8, 640.1)$
Cuadrática	$a = -0.01307\ (-0.02013, -0.006013)$ $b = 7.007\ (1.156, 12.86)$ $c - 12.49 =\ \ (-47.31, 22.32)$ $d = -63.05\ (-267.6, 141.5)$ $e = 59.03\ (-177.8, 295.8)$

Tabla B.28: Coeficientes de ajuste para distintas curvas del método de ordenamiento junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 2 %

De acuerdo a lo observado en la tabla B.27, al aumentar el grado de la curva, el R^2_{adj} aumenta su valor y el $RMSE$ disminuye, lo que indica que la curva de mejor ajuste es la cuadrática. Y no es necesario revisar la otra tabla.

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$7.2178 * 10^7$	$1.8955 * 10^5$	$1.2471 * 10^5$	$7.7483 * 10^4$
R^2	0.8792	0.9997	0.9998	0.9999
dfe	8	7	6	5
R^2_{adj}	0.8641	0.9996	0.9997	0.9998
RMSE	$3.0037 * 10^3$	164.5551	144.1722	124.4851

Tabla B.29: Estadísticas de ajuste para el método de ordenamiento con un 5 % de densidad de objetos.

Tipo Curva	Coeficientes
Logarítmica	$a = 1232\ (242.6, 2222)$ $b = -1979\ (-6039, 2081)$
Lineal	$a = 26.01\ (25.37, 26.65)$ $b = -145.7\ (-234.4, -57.1)$ $c = 50.54\ (-206.3, 307.3)$
Subcuadrática	$a = 1.276\ (-0.3487, 2.902)$ $b = 16.76\ (4.9, 28.62)$ $c = -73.57\ (-220.8, 73.61)$ $d = -16.66\ (-274.4, 241.1)$
Cuadrática	$a = 0.005155\ (-0.002072, 0.01238)$ $b = -3.221\ (-9.211, 2.7699)$ $c = 43.61\ (7.965, 79.25)$ $d = -221.7\ (-431.1, -12.33)$ $e = 72.46\ (-170, 314.9)$

Tabla B.30: Coeficientes de ajuste para distintas curvas del método de ordenamiento junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 5 %

De acuerdo a lo observado en la tabla B.29, al aumentar el grado de la curva, el R^2_{adj} aumenta su valor y el $RMSE$ disminuye. Sin embargo, el valor de mejora de ambas estadísticas no es mucho desde la curva lineal. Si se compara la tabla de coeficientes B.30, muestra que tanto la curva subcuadrática como la cuadrática tienen sus coeficientes mayores con intervalos de confianza que cruzan por 0, indicando que ambas curvas serían sobreajustes. La curva de mejor ajuste, por lo tanto, es la lineal.

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$6.6301 * 10^7$	$5.8472 * 10^4$	$5.8197 * 10^4$	$5.2701 * 10^4$
R^2	0.8821	0.9999	0.9999	0.9999
dfe	8	7	6	5
R^2_{adj}	0.8674	0.9999	0.9998	0.9998
RMSE	$2.8788 * 10^3$	91.3951	98.4861	102.6654

Tabla B.31: Estadísticas de ajuste para el método de ordenamiento con un 10 % de densidad de objetos.

Tipo Curva	Coeficientes
Logarítmica	$a = 1204\ (255.7, 2153)$ $b = -1934\ (-5825, 1957)$
Lineal	$a = 24.98\ (24.62, 25.33)$ $b = -106.7\ (-155.9, -57.44)$ $c = 16.11\ (-126.5, 158.7)$
Subcuadrática	$a = 0.2236\ (-0.8866, 1.334)$ $b = 23.32\ (15.22, 31.42)$ $c = -84.61\ (-185.2, 15.93)$ $d = -23.25\ (-199.3, 152.8)$
Cuadrática	$a = 0.001221\ (-0.00474, 0.007182)$ $b = -0.766\ (-5.706, 4.174)$ $c = 29.14\ (-0.2526, 58.54)$ $d = -122\ (-294.7, 50.66)$ $e = 27.33\ (-172.6, 227.3)$

Tabla B.32: Coeficientes de ajuste para distintas curvas del método de ordenamiento junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad de objetos del 10 %

De acuerdo a lo observado en la tabla B.31, al aumentar el grado de la curva, el R^2_{adj} disminuye su valor y el $RMSE$ aumenta, siendo una clara muestra de que las curvas superiores a la lineal son un sobreajuste (que se puede corroborar con los coeficientes de B.32).

B.5.3. Sweep And Prune Cinético

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$1.0348 * 10^7$	$1.6019 * 10^6$	$4.9476 * 10^4$	$1.0492 * 10^3$
R^2	0.8492	0.9767	0.9993	1.0000
dfe	8	7	6	5
R^2_{adj}	0.8303	0.9700	0.9989	1.0000
RMSE	$1.1373 * 10^3$	478.3772	90.8071	14.4859

Tabla B.33: Estadísticas de ajuste para el algoritmo Sweep and Prune Cinético con un 1 % de densidad.

Tipo Curva	Coeficientes
Logarítmica	$a = 204.2\ (-170.6, 578.9)$ $b = -259.4\ (-1797, 1278)$
Lineal	$a = 9.776\ (7.923, 11.63)$ $b = -168.3\ (-425.9, 89.39)$ $c = 175.1\ (-571.5, 183.8)$
Subcuadrática	$a = 3.759\ (2.735, 4.782)$ $b = -17.65\ (-25.12, -10.18)$ $c = 65.69\ (-27.01, 158.4)$ $d = 21.49\ (-140.8, 183.8)$
Cuadrática	$a = 6.498 * 10^{-3}\ (5.657 * 10^{-3}, 7.339 * 10^{-3})$ $b = -0.3508\ (-1.048, 0.3463)$ $c = 4.747\ (0.599, 8.895)$ $d = -11.01\ (-35.37, 13.36)$ $e = 43.05\ (14.84, 71.26)$

Tabla B.34: Coeficientes de ajuste para distintas curvas del algoritmo Sweep and Prune Cinético junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad del 1 %

De acuerdo al análisis de las tablas B.33 y B.34, la curva de mejor ajuste es la cuadrática.

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$1.1252 * 10^7$	$1.5846 * 10^6$	$4.3287 * 10^4$	584.6982
R^2	0.8502	0.9790	0.9994	1.0000
dfe	8	7	6	5
R^2_{adj}	0.8322	0.9730	0.9991	1.0000
RMSE	$1.1859 * 10^3$	475.7883	84.9381	10.8139

Tabla B.35: Estadísticas de ajuste para el algoritmo Sweep and Prune Cinético con un 2 % de densidad.

Tipo Curva	Coeficientes
Logarítmica	$a = 222.7\ (-168.1, 613.4)$ $b = -306.7\ (-1010, 1296)$
Lineal	$a = 10.21\ (8.363, 12.05)$ $b = -171.4\ (-427.7, 84.85)$ $c = 173.5\ (-569, 916)$
Subcuadrática	$a = 3.789\ (2.832, 4.747)$ $b = -17.41\ (-24.4, -10.43)$ $c = 59.28\ (-27.43, 146)$ $d = 26.65\ (-125.2, 178.5)$
Cuadrática	$a = 5.799 * 10^{-3}\ (5.171 * 10^{-3}, 6.427 * 10^{-3})$ $b = 0.2169\ (-0.3035, 0.7373)$ $c = 1.859\ (-1.238, 4.954)$ $d = 0.7515\ (-17.44, 18.94)$ $e = 35.72\ (14.66, 56.78)$

Tabla B.36: Coeficientes de ajuste para distintas curvas del algoritmo Sweep and Prune Cinético junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad del 2 %

De acuerdo al análisis de las tablas B.35 y B.36, la curva de mejor ajuste es la cuadrática.

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$1.2714 * 10^7$	$1.9668 * 10^6$	$6.3020 * 10^4$	$1.7429 * 10^3$
R^2	0.8504	0.9768	0.9993	1.0000
dfe	8	7	6	5
R^2_{adj}	0.8316	0.9702	0.9989	1.0000
RMSE	$.12606 * 10^3$	530.0706	102.4854	18.6705

Tabla B.37: Estadísticas de ajuste para el algoritmo Sweep and Prune Cinético con un 5 % de densidad.

Tipo Curva	Coeficientes
Logarítmica	$a = 237.5\ (-177.9, 652.9)$ $b = -330.9\ (-2035, 1373)$
Lineal	$a = 10.82\ (8.765, 12.87)$ $b = -177.5\ (-463, 108)$ $c = 185.9\ (-641.3, 1013)$
Subcuadrática	$a = 4.085\ (2.93, 5.241)$ $b = -18.94\ (-27.37, -10.51)$ $c = 67.25\ (-37.38, 171.9)$ $d = 29.56\ (-153.6, 212.8)$
Cuadrática	$a = 6.976 * 10^{-3}\ (5.892 * 10^{-3}, 8.06 * 10^{-3})$ $b = -0.2459\ (-1.144, 0.6525)$ $c = 4.501\ (-0.8446, 9.847)$ $d = -5.9\ (-37.31, 25.5)$ $e = 40.15\ (3.791, 76.51)$

Tabla B.38: Coeficientes de ajuste para distintas curvas del algoritmo Sweep and Prune Cinético junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad del 5 %

De acuerdo al análisis de las tablas B.37 y B.38, la curva de mejor ajuste es la cuadrática.

Mediciones	sublineal	lineal	subcuadrática	cuadrática
SSE	$1.1324 * 10^7$	$1.4675 * 10^6$	$4.742 * 10^4$	$1.3788 * 10^3$
R^2	0.8535	0.9810	0.9994	1.0000
dfe	8	7	6	5
R^2_{adj}	0.8352	0.9756	0.9991	1.0000
RMSE	$1.1898 * 10^3$	457.8711	88.9499	16.6059

Tabla B.39: Estadísticas de ajuste para el algoritmo Sweep and Prune Cinético con un 10 % de densidad.

Tipo Curva	Coeficientes
Logarítmica	$a = 245.6\ (-146.5, 637.6)$ $b = -336.5\ (-1945, 1272)$
Lineal	$a = 10.24\ (8.469, 12.02)$ $b = -160\ (-406.6, 86.6)$ $c = 176.9\ (-537.6, 891.5)$
Subcuadrática	$a = 3.51\ (2.508, 4.513)$ $b = -15.32\ (-22.64, -8.002)$ $c = 48.3\ (-42.51, 139.1)$ $d = 52.99\ (-106, 212)$
Cuadrática	$a = 6.01 * 10^{-3}\ (5.046 * 10^{-3}, 6.974 * 10^{-3})$ $b = -0.1345\ (-0.9336, 0.6646)$ $c = 4.201\ (-0.5535, 8.956)$ $d = -0.1517\ (-28.08, 27.78)$ $e = 34.01\ (1.667, 66.35)$

Tabla B.40: Coeficientes de ajuste para distintas curvas del algoritmo Sweep and Prune Cinético junto con los valores de bondad de ajuste para un intervalo de confianza del 95 %, para una densidad del 10 %

De acuerdo al análisis de las tablas B.39 y B.40, la curva de mejor ajuste es la cuadrática.