EST, Concurso de Trabajos Estudiantiles

Algoritmo para la Sinopsis de Videos de Vigilancia

Resumen. Se presenta una técnica para generar una sinopsis de video (o video resumen) que permite realizar un análisis de forma económica y eficiente de las filmaciones provenientes de todas las cámaras de seguridad existentes en un sistema ya instalado, sin necesidad de alteración de dicho sistema. El algoritmo está basado en la estrategia Next-Fit del problema de Bin Packing: los objetos son autos, personas o animales que han sido filmados por las cámaras de vigilancia, y cada *bin* es un Quadtree que organiza espacio-temporalmente al subconjunto de objetos que no colisionan entre sí. Cuando ya no quedan más objetos por ubicar, se genera el resumen de video a partir de la secuencia de *bins*. En los experimentos realizados se logró resumir un video de vigilancia, cedido por el Municipio de Tandil (Buenos Aires), de 60 minutos, en el que transcurren 394 objetos, a sólo 8 minutos, con el objetivo de transmitir todas las actividades realizadas por dichos objetos al nuevo video acotado sin que interfieran entre sí y preservando el orden que aparecen en el video original.

Palabras claves: sinopsis de video de paneo fijo, optimización de vigilancia en urbes, soluciones basadas en el problema de Bin-Packing, detecciones de solapamiento en imágenes, descomposición del espacio con la estructura Quadtree.

1 Introducción

Este trabajo final, de la cátedra Análisis y Diseño de Algoritmos, es una etapa más del proyecto desarrollado en el Instituto PLADEMA (UNICEN). El proyecto surge como una necesidad de los Centros de Monitoreo de Seguridad Municipales, donde una cámara de vigilancia graba durante un tiempo prolongado un espacio, ya sea un cruce de calles, un hogar o un comercio. Los operadores del Centro deben observar todos los videos registrados por las distintas cámaras de vigilancia esparcidas en la ciudad para localizar eventos que hayan originado señales de alerta (como accidentes, búsquedas, entre otros). Es posible que esta tarea les resulte tediosa, especialmente por la gran cantidad de filmaciones, y además, de duración prolongada.

La sinopsis de video (o resumen de video) surge como solución alternativa al problema enunciado. El resumen del video compagina la presentación simultánea de los objetos y actividades que transcurrieron en tiempos diferentes. De este modo, el operador puede resolver en unos pocos minutos la revisión de horas de secuencia de video [1][2]. La fig. 1 esquematiza el antes y después de aplicar ésta técnica a una secuencia de video original. Una cámara fija captura tres pájaros en vuelo en distintos instantes (ver 1.a, 1.b y 1.c) y el resumen del video reúne sus vuelos en una única escena y muestra el transitar de todos ellos juntos (ver 1.d).

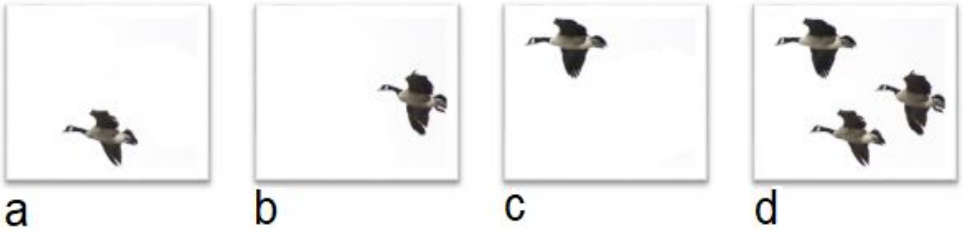

Fig. 1. Capturas de vuelos de pájaros en distintos instantes de tiempo de un video en a, b y c. Resultado de la sinopsis del video en d.

En este trabajo, se generó un video clip de corta duración que reúne a los objetos detectados de un video de vigilancia de larga duración. Primeramente, se realiza la detección, seguimiento y clasificación de los objetos que circulan en espacios públicos del Municipio de Tandil. Luego, los objetos y sus trayectorias se almacenan en una base de datos, distinguiendo a los objetos estáticos (el fondo) de los dinámicos (peatones, ciclistas, autos, etc.) que entran y salen de escena siguiendo direcciones particulares.

Básicamente, el videoclip se construyó usando una estrategia basada en la técnica Next-Fit, del problema de Bin-Packing [3][4], para organizar los objetos detectados en escenas. La organización espacial de los objetos se resolvió usando una estrategia basada en Quadtrees [5], a modo de clasificación geométrica del espacio, donde se desarrollan las escenas del video. Surgió una problemática adicional, relacionada al solapamiento

témporo-espacial de objetos (colisiones) que ocurre cuando comparten la misma ubicación espacial en la escena, y que se resolvió mediante estrategias de detección de colisiones [6][7][8].

A continuación, se describen los detalles de la implementación y los resultados obtenidos, para el caso de estudio, que consiste en un video de vigilancia cedido por el Municipio de Tandil.

2 Implementación de la sinopsis de video

Se desarrolló una aplicación que resume los registros visuales de cada cámara de seguridad en un video nuevo que conserva, de la grabación, todos los objetos que transcurrieron (autos, peatones, ciclistas, entre otros) en el espacio físico determinado (un cruce de calles, para este caso). El video nuevo introduce parámetros que mejoran la performance de la solución propuesta: tiempo, cantidad de objetos y espacio ocupado por escena en el video resumido, entre otros. El parámetro tiempo, corresponde a la duración total por escena de reproducción del vídeo resumido, el cual se desea que sea mucho más corto que el tiempo total de la grabación original. También, para mejorar la vigilancia de todos los objetos en movimiento en el nuevo video, ya que al transcurrir varios en mismas franjas de tiempo, se puede perder mucha información si la cantidad de objetos en una misma escena no está limitada, por esto, es importante la función del parámetro del espacio ocupado por escena para distribuir los objetos y cada uno se pueda visualizar de la mejor forma posible.

2.1 Dataset de entrada

La solución se implementó en JavaScript [9]. Los objetos detectados en el video original son el principal insumo para generar el vídeo resumen y llegan en un archivo de formato JSON. Este tipo de objeto es esquematizado en la fig. 2, que corresponde a un dataset de 394 elementos que transcurrieron en un cruce de calles de la ciudad de Tandil, Buenos Aires, y fueron registrados por una cámara de vigilancia del lugar. El atributo *data* del objeto JSON, posee un tamaño de 394, que usa para almacenar todos los objetos detectados y clasificados, llamados Tracked Blob (TB). A su vez, los TB son objetos y poseen atributos con distintas funcionalidades, tal como, identificar a cada uno (*id*), la cámara que lo captura (*camara_id*), el tiempo de entrada y salida en el video original (*init* y *finish*), la ubicación de la imagen del objeto (*url_snapshot*) (que se accede anteponiendo la ruta *smartcam.pladema.net/img),* identificador de la imagen anterior (*blob_snapshot_id*), la clasificación del TB (color, tipo de objeto, identificado como “human”, “cyclist”, “car”, entre otros), datos porcentuales del TB a lo largo de la grabación original (*blob_snapshot*), lista de *Blobs* de dicho TB (*lightweight_blobs*), cada uno con su identificador, ubicación en coordenadas {x,y} y tiempos, y por último, parámetros del video original (*lightweigth_spritesheets*) como su resolución, en este caso de 800x480. En síntesis, cada TB está delimitado por una región rectangular que describe el ancho y alto del objeto, la coordenada {x,y} donde es detectado, el instante en que aparece en la escena y en el que la abandona. La trayectoria de cada TB es una lista de frames llamados *Blob*, los cuales lo representan en un tiempo y lugar determinado del espacio en el video original.

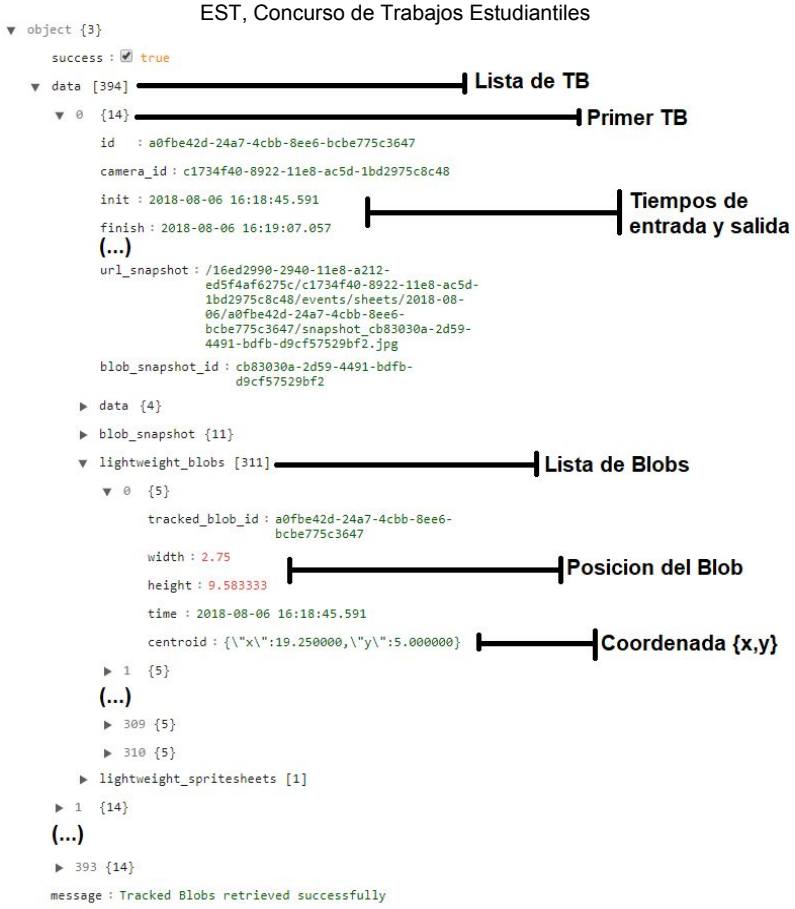

Fig. 2. Estructura del objeto JSON que guarda los datos de los 394 objetos detectados y clasificados del video de vigilancia.

Se usó la librería JQuery [10] para acceder al dataset de entrada. Como resultado, se permitió que en el archivo JavaScript, destinado a la implementación de esta sección, utilizando el método getJSON propio de la librería agregada, se pudiera acceder al archivo .json contenido como un query string de manera local. Este método recibe un parámetro destinado a la ruta del archivo .json, si no hubo errores en la solicitud, se accede a una función anónima que recibe como parámetro un objeto de tipo JSON [11], el cual contiene los datos requeridos.

Un problema que se puede encontrar en la solución que utiliza getJSON es que el objeto JSON, dado por la función anónima y que contiene los datos requeridos, solo se puede utilizar dentro de la misma función, lo que implica desorganización en cuanto a código. Esto se debe a que la llamada al método getJSON es asincrónica, por lo que el script que utiliza el método, no puede emplear sus resultados inmediatamente. Además, como los procesos de inserción son asincrónicos, la lista de Blobs de un TB no se encuentra ordenada, por lo tanto, es importante ordenarla antes de trabajar con los Blobs, aunque este mecanismo de preprocesamiento no se considere en las mediciones de complejidad. A continuación, se presenta el pseudocódigo correspondiente.

```
getJSON('ruta del archivo');
      done(function(response))
      {     Obtener datos de la resolución del video;
            sortBlob()
            {     for(cada TrackedBlob del dataset)
                        Ordenar todos los Blobs de cada TrackedBlob de
                        forma ascendente según el tiempo de entrada en el
                        video;
                  Ordenar todos los TrackedBlobs entre sí de forma
                  ascendente por el tiempo de entrada al video;
            }(...)}
```

2.2 Principales abstracciones implementadas

Entre las más importantes abstracciones implementadas se puede mencionar al TDA Tuple, TDA Frame, TDA TrackedBlob, TDA Rectangle, TDA Quadtree y el TDA Scene. La figura 3 corresponde al diagrama de clases en UML [12], el cual esquematiza la estructura del sistema para la solución implementada, mostrando las clases del mismo, sus atributos, operaciones y relaciones entre objetos.

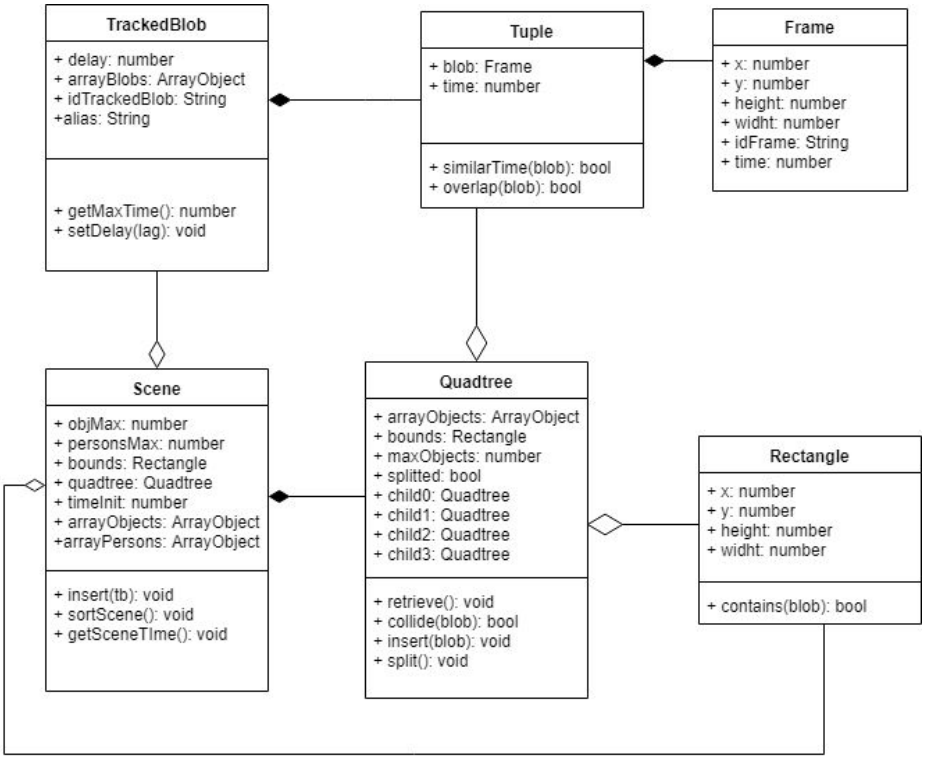

Fig. 3. Diagrama de clases que se utilizó para la implementación de la solución.

La clase Frame representa a cada Blob encontrado en el objeto JSON y guarda sus datos originales más importantes para la implementación propuesta, es meramente una clase contenedora. Mientras que la clase Tuple, se la utilizó principalmente para contener, además de la instancia de Frame correspondiente, un atributo *time,* que hace referencia al tiempo que transcurre el Blob en el video nuevo, como un número entero.

A continuación, se presenta el algoritmo en pseudocódigo que se implementó para pasar el contenido del objeto JSON a instancias de las clases anteriormente mencionadas.

```
loadBlob(TrackedBlob){
      inicializa el arreglo de Blobs correspondientes al TB pasado por
      parámetro;
      for (cada Blob del arreglo de Blobs)
      {     if (posicion del Blob es par || es el último del arreglo)
                  Obtener datos;
            inicializar una instancia de la clase Tuple con el tiempo
            correspondiente al video resumen;
            Guardar la instancia en el arreglo de Blobs;}
      return arreglo de Blobs;}
```

Para mejorar la performance general del sistema se consideró que no eran necesarios todos los *Blobs* de todos los TB en cuestión, sino únicamente tomar la mitad de ellos, haciendo un filtrado para que solo se considere un Blob de por medio en cada TB, por esto, como se ve en el pseudocódigo, solo se extraen los datos de los Blobs pares y del último, para no perder la información del momento en que el TB sale de escena en el video original. Además, al menos en la implementación hecha en JavaScript, dado que, al no tener control sobre la memoria

secundaria y al tener grandes volúmenes de Blobs en el dataset original, el navegador no es capaz de procesar toda la información y enciende el flag de “Stack Overflow”.

2.3 Empaquetado de los objetos

Para almacenar los TB se implementó un algoritmo basado en la estrategia Next Fit que resuelve el problema de empaquetado (Bin Packing), que postula organizar n objetos en el menor número de contenedores posibles. La estrategia implementada coloca cada objeto en el ultimo *bin* utilizado, si posee espacio para contenerlo, de lo contrario, se crea un nuevo *bin* y se inserta el objeto en cuestión. Cada vez que se desee empaquetar un nuevo elemento, se comprobará si puede contenerlo el ultimo *bin*. La figura 4 esquematiza el funcionamiento de ésta técnica utilizada.

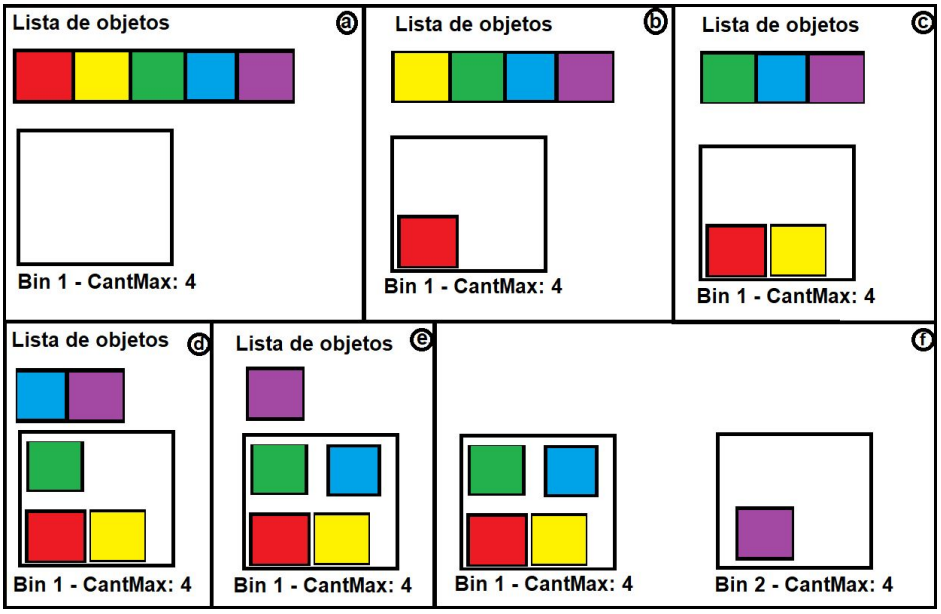

Fig. 4. Esquema de la técnica de empaquetado utilizada. Lista de cinco elementos a empaquetar y un primer *bin* vacío, con una capacidad para cuatro objetos empaquetados en (a). Extracción de a un elemento de la lista y empaquetado en el último *bin* utilizado y que posee espacio suficiente (b, c, d y e). Creación de nuevo *bin*, ya que el último no posee capacidad suficiente para almacenar los objetos restantes (f).

Por la estrategia de empaquetado utilizada, los objetos son los TB y los contenedores o *bins* son instancias de clases llamadas “Scene”, que representan las escenas del video resumen. Cada objeto de esta clase contendrá tantos TB como su capacidad lo permita. Luego, se crea un contenedor nuevo, destinatario de los TB restantes y se repite el procedimiento. Así, ajusta parámetros de performance en cada escena como la máxima cantidad de vehículos (objMax), la máxima cantidad de personas o ciclistas (personsMax), la resolución del video original (widthMax y hightMax) y la demora temporal de inicio de la escena (timeInit). Esta demora inicial de la escena está supeditada al término del último Blob del TB más lento de la escena anterior. Por lo tanto, a medida que crezca el número de *bins*, mayor será el tiempo de espera hasta aparecer el próximo TB. El siguiente pseudocódigo corresponde al algoritmo encargado de esto.

```
loadScenes()

{     inicializar arreglo de Scene (sceneList);

      for(cada TB del dataset original)

      {     Obtener datos del TB;
            inicializar arreglo de Blobs;
            arregloBlobs = loadBlob(TB);
            inicializar instancia de TrackedBlob con los datos obtenidos de
            TB y arregloBlobs;
            if (sceneList.empty())
                  {inicializar una instancia de Scene con el tiempo seteado
                  en 0;
```

```
                scene.insert(TrackedBlob);
                sceneList.push(scene);
        }else
                if(sceneList[sceneList.length - 1].insert(TrackedBlob))
                        inserta TrackedBlob en la última escena;
                else{
                        timeInit=sceneList[sceneList.length-1].getSceneTime
                        ();
                        inicializar una instancia de Scene con timeInit ya
                        calculado;
                        scene.insert(TrackedBlob);
                        sceneList.push(scene);} }}
```

La complejidad temporal de este algoritmo es O($nm^2 log \frac{m}{2}$) en notación Big-Oh, la cual toma el peor caso, *n* corresponde a la cantidad de TB del dataset original, ya que recorre uno a uno hasta que todos se empaquetan en alguna escena, y *m* corresponde a todos los Blobs de todos los TB que se inserten. Como de estos últimos se toma la mitad, se divide el valor de *m* por dos. La complejidad temporal es logarítmica, ya que para la inserción se utiliza el atributo *quadtree,* correspondiente a la instancia Scene, el cual es un objeto de la clase Quadtree donde se insertarán todos los Blobs de cada TB contenido en la escena.

Otro atributo de la clase Scene es *bounds,* que es un objeto de la clase Rectangle, que posee atributos que contienen los valores de las coordenadas {x,y}, y el ancho y alto que delimitan la escena.

2.4 Resolviendo las colisiones espacio-temporales

Como se mencionó anteriormente, para la inserción de cada TB, se toma cada uno por vez y se los inserta en arrayObjects (para los objetos identificados en general por "car") o arrayPersons (para los clasificados como "human" o "cyclist") según corresponda la clasificación, hasta completar la instancia de Scene en cuestión. Es decir, hasta que los valores de objMax o personsMax sean excedidos por la cantidad de objetos en respectivos arreglos. Además, en cada escena sólo se agregan los TB que cumplan que sus Blobs no provocan colisiones con los que ya están almacenados en la región Quadtree, asociada a la instancia de Scene. Si se detecta que un Blob ocasiona colisión entre los demás ya ubicados en la escena, se retrasa el tiempo de entrada a escena del TB que lo contiene. En la figura 5 se puede observar cómo mejora la visualización de los objetos en el video resumen luego de aplicar detección de colecciones, que es dado cuando dos o más frames de objetos diferentes comparten una misma posición en el mismo intervalo de tiempo, agregando retardo al tiempo de entrada al video de los TB colisionados.

Fig. 5. Video resumido generado con colisiones (izquierda) y el mismo video luego de aplicar detección de colisiones (derecha).

La inserción de los TB en escenas de video, se modeló mediante el método *insert,* el cual agrega una nueva instancia de la clase Tuple al Quadtree, si no hay otros que colisionan con la misma. Tal verificación se realiza mediante la operación *collide,* propia de la clase Quadtree, detallado a continuación:

```
collide(blob){
      if(contains(blob)) {return false;}
      else  if( #objetos(arrayObjects) <= cantMax && !splited)
            {     for(cada Blob del arreglo de objetos){
```

```
                        Compara cada blob del arreglo con el blob pasado
                        por parámetro;
                        if(       poseen       distinto       ID       &&
                        !overlap(arrayObjects[blob])  &&  comparten  mismo
                        intervalo de tiempo)
                        {return true;}
                  }return false;
            }else{if(splited){
                        Chequea cual de sus divisiones pueden contener al
                        blob;
                        Si alguno lo puede contener: return true;
                  }return false;}}
```

La cantidad de objetos en una misma escena es un problema para el resumen de video. Por ello, en cada instancia de la clase Scene se restringe la cantidad de objetos en la misma. Así, se comprueba la colisión de los Blobs, sólo si hubiera capacidad suficiente de objetos o personas y, solo si se cumple, se pasa a insertar en el arreglo correspondiente de la instancia de Scene. A continuación, se presenta el pseudocódigo del método insert proveniente de esta clase, el cual utiliza el metodo setDelay, propia de la instancia de la clase TrackedBlob, que se obtiene por parámetro, que se encarga de agregar retardo al tiempo de entrada de dicho TB e implica setear también a cada Blob el mismo delay. Este retardo es dado por el tiempo de inicio de escena, y además, un retardo fijo que se adiciona cada vez que se detecta una colisión.

```
insert(TB){
      if (arrayObjects.length < objMax || arrayPersons.length < persMax)
      {     TB.setDelay(timeInic);
            for(cada Blob del TB)
            {     TB.setDelay(timeInic);
                  if (quadtree.collide(Blob)){
                        Recorre los blobs recorridos anteriormente y les
                        setea el delay anterior;
                        TB.setDelay(100);}}
            if(!(TB.alias   ==   undefined)   &&   arrayPersons.length   <
            arrayPersonsMax  &&  (TB.alias  ==  "human"  ||  TB.alias  ==
            "cyclist")){
                  for (cada Blob del TB) {
                        quadtree.insert(Blob);}
                  arrayPersons.push(TB);
                  return true;
            }else
                  if((TB.alias  ==  undefined)  &&  (arrayObjects.length  <
                  objMax)   ||   (arrayObjects.length   <   objMax)   ||
                  !(arrayPersons.length < persMax) ) {
                        for (cada Blob del TB) {
                              quadtree.insert(Blob);  }
                        arrayObjects.push(tb);
                        return true;
                  }return false;}}
```

La complejidad del algoritmo es de O(m^2 *log* $\frac{m}{2}$) con m la cantidad de Blobs en un TB y es logarítmica ya que la inserción de cada uno se realiza en la estructura Quadtree asociada a la instancia de Scene específica.

2.5 La organización espacial de los objetos en la escena

En la clase Quadtree se organizan las instancias de Tuple, que representan a los Blobs de cada objeto TB, en su estructura interna que es un árbol cuaternario. El nodo raíz representa al rectángulo que circunscribe al dominio entero. Cada nodo divide sucesivamente al espacio en cuatro cuadrantes iguales, hasta que la región contenga una cantidad de objetos máximos (parámetro) y se transforme en un nodo terminal (o nodo hoja) del

Quadtree. Dichos nodos heredan los objetos, en este caso Blobs, provenientes de su padre. En la figura 6, se puede observar esta división espacial, cómo se corresponden los cuadrantes del espacio utilizado con los nodos del árbol cuaternario y la ubicación de los objetos en los nodos terminales.

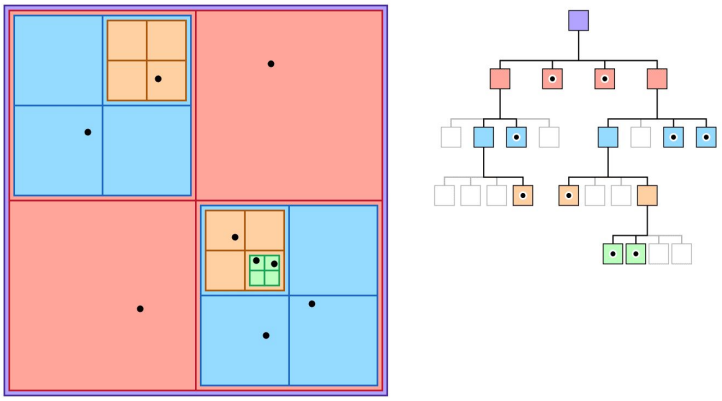

Fig. 6. Esquema del Quadtree. División espacial en cuatro regiones (izquierda) y organización jerárquica de la misma (derecha). Los puntos negros representan a los objetos.

Mediante el atributo bounds, objeto de la clase Rectangle, se delimita la región Quadtree propia de cada cuadrante (nodo del árbol) por medio de coordenadas (*x* e *y*), ancho y alto (*heigth* y *width*).

A continuación, se presenta el método insert de la clase Quadtree, el cual inserta cada blob pasado por parámetro, y si cumple que su posición (dada por las coordenadas (*x* e *y*), *heigth* y *weidth*) se encuentra dentro de los límites de la instancia Rectangle (mediante el método *contains*), asociada a dicho objeto Quadtree, y además que la cantidad máxima de objetos en dicha instancia Quadtree no sea superada (*maxObjects*), lo agrega, de lo contrario divide la estructura y distribuye los objetos del mismo en las nuevas divisiones, mediante los métodos *split* y *retrieve* respectivamente. Esto último se realiza solo si el Quadtree no se encuentra dividido anteriormente, lo cual lo verifica con el atributo *splited*, propio de la instancia, de ser verdadero esto, pasa a chequear si alguna de sus divisiones (*child0, child1, child2* o *child3*) pueden contener al blob específico.

```
insert(blob)
{if(contains(blob))
      return false;
  else if( #objetos(arrayObjects) <= maxObjects && !splited ){
            insertar blob en arrayObjects;
            return true;}
      else{if(!splited){
                  split();
                  retrieve();}
            if (child0.insert(blob)) {return true;}
            else if (child1.insert(blob)) {return true;  }
            else if (this.child2.insert(blob)) {return true;}
            else if (this.child3.insert(blob)) {return true;}}
```

3 Resultados

Se evaluó la solución propuesta en un único caso de estudio: un dataset de 394 objetos en movimiento de 60 minutos de duración. Antes de que esto pueda ser visualizado, existe una etapa de captura, la cual se encarga de procesar el video requerido, separando entre objetos sin movimiento, los cuales pertenecen al fondo, y los que sí poseen movimiento. La figura 7 esquematiza el antes y el después de aplicar nuestra técnica al video original, respectivamente. La primera, la captura instantánea del caso de estudio cuyo video fue generado por una cámara de vigilancia durante 60 minutos, ubicada en el cruce de calles de San Lorenzo y Gral. Belgrano de la ciudad de Tandil (Buenos Aires). A la derecha, una captura instantánea del vídeo resumido, generado mediante la técnica presentada en este trabajo, cuya duración es de 8 minutos. Las escenas del resumen de video incorporan una mayor cantidad de objetos, evaluando siempre que no se produzcan colisiones entre ellos. De esta manera, se

optimiza el uso espacial de las regiones que no eran circuladas en el video original. No obstante, los objetos incorporados al resumen mantienen asociada la hora en la que transcurrieron por el lugar (rectángulo negro sobre los mismos). Una reproducción digital del video resumen está disponible en el sitio web [13].

Fig. 7. Capturas instantáneas del antes y después de aplicar la técnica de sinopsis de video o video resumen.

La salida es un documento en formato CSV, que contiene cada TB con su respectivo tiempo de entrada y de salida de escena en el video resumido. Las figuras 8 y 9 comparan los tiempos de todos los TB que transcurren en el video, antes y después de aplicado el algoritmo propuesto, respectivamente. En ambos gráficos, en el eje *y* están discretizados todos los TB y en el eje *x* se indica el tiempo transcurrido entre el comienzo y el fin de la reproducción de los videos. Por tanto, cada segmento rojo representa el tiempo de vida de un TB en escena.

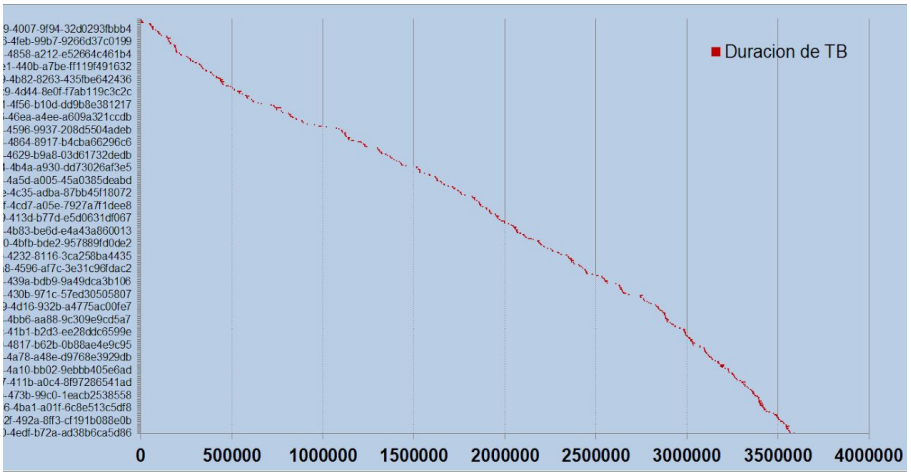

Fig. 8. Distribución de los TB en el video de estudio.

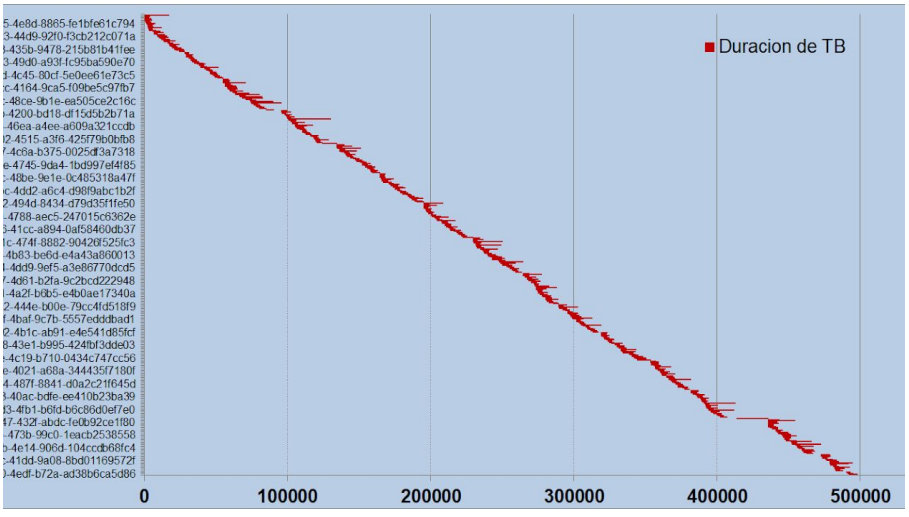

Fig. 9. Esquema de resultados. Distribución de los TB en el nuevo video, luego de aplicada la estrategia propuesta.

Comparando ambos gráficos, en la fig. 8 se observa una curva delgada de tiempo de vida (segmento rojo), que corresponde a lapsos temporales donde unos pocos elementos transcurren en la escena. Es decir, en la circulación de los objetos, no se utiliza todo el espacio disponible que brinda cada escena con la resolución original. Mientras que, en la fig. 9, se observa una mayor utilización del espacio en cada instante del video. La solución propuesta en este trabajo es conveniente, aunque la duración de la escena está condicionada al TB más prolongado. Además, dada la conversión de los últimos valores de tiempo, de la comparación, puede estimarse que la duración de 60 minutos (primer esquema) se reduce a un resumen de 8 minutos (segundo).

4. Conclusiones y trabajos futuros

La sinopsis del video se propuso como un enfoque para condensar las actividades captadas en una filmación, en un período de tiempo acotado. Esta representación concentrada puede permitir un acceso eficiente a dichas actividades en secuencias de video.

Se resumió un video de 60 minutos con 394 objetos en movimiento, captado por una cámara de vigilancia de la ciudad de Tandil, a un nuevo video de 8 minutos, de duración máxima, preservando el orden en el que aparecieron dichos objetos. Lo importante de este resultado, es que el video resumen reproduce la misma cantidad de objetos en unos pocos minutos, y todo, evitando las colisiones espacio-temporales entre los elementos dentro de una misma escena.

Como trabajo futuro se estudiará una limitación a la solución propuesta que está relacionada al retardo del TB más lento, ya que condiciona el tiempo final de cada escena, por ende el tiempo total del video resumen. Además, de la reducción del número de escenas utilizadas para empaquetar los objetos, haciendo foco en mejorar la distribución de los mismos, reduciendo el tiempo total de video resumido. Por otro lado, se evaluará la herramienta desarrollada para diferentes casos de estudio y en diferentes contextos.

Referencias

1. Alex Rav-Acha, Yael Pritch, Shmuel Peleg, (2006), Making a Long Video Short: Dynamic Video Synopsis. School of Computer Science and Engineering, The Hebrew University of Jerusalem.
2. BriefCam, https://www.briefcam.com/
3. Pérez-Ortega, Castillo-Zacatelco, Vilariño-Ayala, Mexicano-Santoyo, Zavala-Díaz, Martínez-Rebollar, Estrada-Esquivel, (2015). Una nueva estrategia heurística para el problema de Bin Packing - A New Heuristic Strategy for the Bin Packing Problem. Ingeniería Investigación y Tecnología, volumen XVII (número 2).
4. Miyazawa, Wakabayashi, (2003). Parametric On-Line Algorithms for Packing Rectangles and Boxes. Universidad Estatal de Campinas, Universidad de Sao Paulo, Brasil.
5. Hanan Samet, (1984). The Quadtree and Related Hierarchical Data Structures. Computer Science Department, University of Maryland, College Park, Maryland 20742.
6. Juan Jose Jimenez Delgado, (2006). Detección de Colisiones mediante Recubrimientos Simpliciales. Tesis Doctoral. Editorial de la Universidad de Granada.
7. Diego Ortego Hernandez (2013), Detección de objetos estáticos de primer plano en escenarios altamente concurridos de video-seguridad. Universidad autónoma de Madrid, escuela politécnica superior.
8. Cormen, T.; Lieserson, C.; Rivest, R. Introduction to Algorithms. Ed. The MIT Press. 2009.
9. JavaScript, https://www.javascript.com/
10. JQuery, https://api.jquery.com
11. JSON, https://www.json.org/json-es.html
12. G. Booch, J. Rumbaugh, and I. Jacobson, (1999), The Unified Modeling Language. User Guide, Addison-Wesley.
13. Video resumido, https://youtu.be/wdQ_SN_Jhxk