Universidad del Bío-Bío, Chile

Facultad de Ciencias Empresariales

Departamento de Sistemas de Información

Implementación de un sistema Web y Móvil para obtener los K-pares de vecinos más cercanos entre dos conjuntos de puntos espaciales representados en la estructura de datos compacta k^2-tree

Proyecto de título presentado por Álvaro Fernando Nahuel Chandía
para obtener el título de Ingeniero Civil en Informática
Dirigido por Dra. Mónica Caniupán

2017

Resumen

En este proyecto se presenta el desarrollo de una aplicación Web y una aplicación para dispositivos Móviles que implementan la consulta de proximidad espacial de los K-pares de vecinos más cercanos ($KCPQ$) sobre dos conjuntos de puntos de interés R y S almacenados en la estructura de datos compacta k^2-tree. Las estructuras de datos compactas se caracterizan por tener la capacidad de poder representar muchos datos utilizando poco espacio de almacenamiento, permitiendo además el acceso a éstos en su forma compacta, es decir, sin tener que descompactarlos. Esto permite procesar grandes cantidades de datos directamente en memoria principal, que es mucho más rápida que la memoria secundaria.

Las estructuras de datos compactas se han utilizado en diferentes ámbitos, como por ejemplo, para representar documentos, para representar grafos de enlaces Web, etc. En este proyecto las estructuras de datos compactas son usadas para almacenar puntos espaciales, para esto, los sistemas interactúan con la API de Google Maps con el fin de obtener conjuntos de puntos de interés del mundo real, como por ejemplo, puntos correspondientes a hoteles, restaurantes, farmacias, bancos, etc. Los conjuntos obtenidos son almacenados en la estructura de datos compacta k^2-tree, sobre la cual se ejecuta la consulta $KCPQ$.

Mediante experimentos se demuestra que es posible utilizar la estructura de datos compacta k^2-tree para representar conjuntos de puntos de interés de forma eficiente en cuanto a espacio de almacenamiento y tiempos de ejecución de la consulta de proximidad $KCPQ$.

Keywords — Estructuras de datos compactas, k^2-tree, $KCPQ$, Consultas Espaciales, puntos espaciales.

Agradecimientos

Agradecer en primer lugar a mi familia, por su apoyo incondicional y paciencia. A mi padre, a quien dedico este logro y por entregarme las herramientas para poder desarrollarme como profesional. A mi profesora guía, Dra. Mónica Caniupán por darme la oportunidad de trabajar con ella y por sus enseñanzas. A Fernando Santolaya quién siempre estuvo dispuesto a resolver mis dudas y por su ayuda en general. A mis amigos más cercanos que nunca dejaron de motivarme.

Adicionalmente este proyecto fue financiado por el grupo de investigación Algoritmos y Bases de datos (ALBA) código GI160119-EF.

Índice general

Índice de figuras

Índice de tablas

Índice de Algoritmos

Capítulo 1

Introducción

A consecuencia de los avances tecnológicos de los últimos años la cantidad de información de interés que se genera diariamente a aumentado drásticamente, por lo que procesar y visualizar enormes cantidades de datos supone un gran desafío para las técnicas tradicionales. Un ejemplo de esto son los grafos utilizados para analizar la estructura de la Web, las relaciones de amistad entre usuarios de redes sociales, etc. Por lo general éstos grafos son enormes y no pueden ser almacenados completamente en memoria principal. En 2014, el grafo de la Web contenía 1.7 billones de nodos que representan páginas Web y alrededor de 64 billones de enlaces entre nodos (páginas)[1], el que sigue creciendo constantemente. Para procesar grafos tan grandes, éstos deben ser traidos en porciones a memoria, lo que relentiza el proceso y lo vuelve inefciente, por ejemplo, referenciar datos en memoria principal tarda al rededor de 100 nanosegundos, por otra parte, realizar operaciones de lectura y escritura en disco puede tardar hasta 5 milisegundos (Vallejos et al., 2017). En el artículo de (Brisaboa et al., 2009) se presenta una alternativa para representar de manera compacta grafos de la Web, esta alternativa está basada en el uso de la estructura de datos compacta k^2-tree. Las estructuras de datos compactas tienen la caracteristica de poder reducir el espacio de almacenamiento de los datos, permitiendo además el acceso y navegación de éstos en su forma compacta. Al reducir el espacio de almacenamiento de los datos, éstos pueden ser traídos a memoria principal completamente en un caso ideal, o en su defecto una gran parte de ellos. Esto nos permite procesar grandes cantidades de datos de forma más eficiente.

En la actualidad las estructuras de datos compactas han sido utilizadas en diferentes ámbitos, por ejemplo, para representar documentos (Navarro, 2014), para representar matrices de datos numéricos y realizar consultas *top-k* (Brisaboa et al., 2014), tales como "obtener los 4 productos más vendidos". La estructura de datos compacta k^2-tree además fue presentada en (Vallejos et al., 2017) para representar cubos de datos de Data Warehouses (DWs) y computar consultas de agregación con funciones de agregación `SUM` y `MAX`. Existen otras estructuras de datos compactas tales como Wavelet tree que son utilizadas en áreas como procesamiento de cadenas de texto, geometría computacional, etc,

[1] http://webdatacommons.org/hyperlinkgraph/2014-04/download.html

(Navarro, 2014).

En la tesis de postgrado (Santolaya, 2017) se implementan los algoritmos para resolver la consulta de los K-pares de vecinos más cercanos ($KCPQ$) sobre la estructura de datos compacta k^2-tree y se estudia la eficiencia y eficacia de estos algoritmos. Sin embargo, no se presentan sistemas que permitan visualizar los resultados de las consultas. En este sentido, este proyecto apoya el trabajo realizado en (Santolaya, 2017) entregando un contexto real de aplicación, generando una aplicación Web y una Móvil para ejecutar la consulta sobre dos conjuntos de puntos de interés R y S del mundo real, que son obtenidos a través de la utilización de las API de Google Maps, donde el usuario elige con que tipos de puntos desea trabajar, tales como hoteles, restaurantes, colegios, farmacias, gimnasios, etc. Una API ($ApplicationProgrammingInterface$) es un conjunto de subrutinas, funciones o métodos ofrecidos por ciertas bibliotecas, para ser utilizada por otro software.

La consulta $KCPQ$ se define como, dado dos conjuntos R y S, obtener los K-pares de vecinos (en este caso puntos) más cercanos, de la forma (r, s) tales que $r \in R$ y $s \in S$.

Ejemplo 1.1 La Figura 1.1 muestra un cuadro con dos conjuntos de puntos R y S, donde, los puntos que se encuentran en el lado izquierdo, pertenecen al conjunto R y los que se encuentran en el lado derecho, al conjunto S. Los puntos unidos con líneas corresponden a los $K = 3$ pares de puntos más cercanos encontrados. Cabe mencionar que un punto de $r \in R$ puede ser cercano a más de un punto $s \in S$, y viceversa, como en el caso de los pares 2 y 3 de la Figura 1.1. □

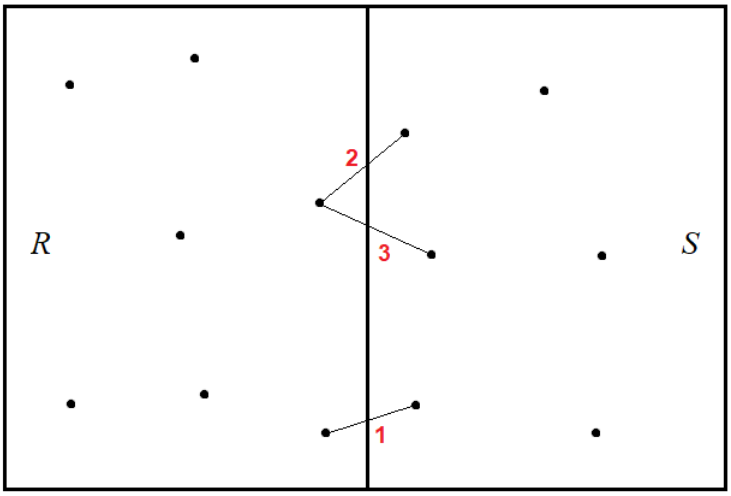


Figura 1.1: Visualización de los $K = 3$ pares de puntos más cercanos entre dos conjuntos de puntos R y S

Cabe señalar que para resolver la consulta $KCPQ$ es necesario trabajar con dos estructuras de datos compactas k^2-tree, encargadas de almacenar a cada uno de los conjuntos de puntos R y S por separado. Para esto es necesario traspasar los puntos de interés

encontrados en un mapa a dos matrices de adyacencia con las que se construyen dichas estructuras.

Este proyecto contempla un entorno de trabajo global, por ende, el usuario puede realizar la consulta propuesta en cualquier parte del mundo, todo esto gracias al uso de la API de Google Maps [2] y su librería de sitios Google Places API [3], que otorgan el acceso a la búsqueda de más de cien millones de sitios, empresas y lugares de interés. Los puntos de interés con los que trabajan las aplicaciones se presentan en la Tabla 2.1 del Capítulo 2.

Mediante la experimentación se demuestran los beneficios en cuanto al ahorro de espacio de almacenamiento de los datos y tiempos de ejecución de los algoritmos utilizados para resolver la consulta *KCPQ*, sobre las estructuras de datos compactas k^2-tree construidas, realizando diversas pruebas sobre distintos escenarios, donde se consideran factores como, la cantidad de puntos a procesar, las ubicaciones de éstos, etc.

El documento se organiza de la siguiente forma: En el Capítulo 2 se presentan los objetivos generales y específicos de este proyecto, además de los alcances y límites de este. En el Capítulo 3 se describen las consultas espaciales más comunes y en especial la consulta de proximidad *KCPQ*, también se define y se describe detalladamente el funcionamiento de la estructura de datos compacta k^2-tree y se presentan los algoritmos para computar la consulta *KCPQ* sobre la estructura. En el Capítulo 4 se presentan las aplicaciones implementadas y su arquitectura. En el Capítulo 5 se presentan las herramientas utilizadas para el desarrollo de las aplicaciones y se muestran las interfaces de usuario desarrolladas. En el Capítulo 6 se presentan los resultados de experimentación que muestran la eficiencia de las aplicaciones en cuanto a espacio de almacenamiento y tiempo de ejecución al resolver la consulta *KCPQ* sobre la estructura de datos compacta k^2-tree. Finalmente, en el Capítulo 7 se presentan las conclusiones obtenidas con la realización de este proyecto y los trabajos futuros.

[2] https://developers.google.com/maps/?hl=es-419
[3] https://developers.google.com/places/?hl=es-419

Capítulo 2

Objetivos

En este capítulo se presentan los objetivos generales y específicos asociados al proyecto, además de los alcances y límites de este.

2.1. Objetivo General

El objetivo general del proyecto es implementar una aplicación Web, además de una aplicación Móvil para la plataforma Android, con el fin de resolver la consulta de los K-pares de vecinos más cercanos ($KCPQ$), sobre dos conjuntos de puntos espaciales R y S que representan lugares de interés, los cuales se encuentran almacenados en la estructura de datos compacta k^2-tree.

2.2. Objetivos Específicos

Los objetivos específicos del proyecto son los siguientes:

1. Estudiar el Algoritmo $KCPQ$ propuesto en (Santolaya, 2017) para procesar datos almacenados en la estructura de datos compacta k^2-tree.
2. Recolectar las posiciones geográficas de los conjuntos de puntos de interés seleccionados por el usuario a través del uso de la API de Google maps.
3. Implementar matrices de conjuntos de puntos, que representan los lugares de interés obtenidos con el objetivo de crear matrices de adyacencia y las estructuras k^2-tree necesarias.
4. Implementar una aplicación Web para mostrar la consulta de los K-pares de vecinos más cercanos sobre los datos almacenados en las estructuras de datos compactas k^2-tree construidas.
5. Implementar una aplicación para la plataforma Android para mostrar la consulta de los K-pares de vecinos más cercanos sobre los datos almacenados en las estructuras de datos compactas k^2-tree construidas.

6. Realizar experimentación de la los algoritmos en términos de beneficios en reducción de espacio de almacenamiento y tiempo de ejecución de consultas.

2.3. Alcances y límites

Este proyecto contempla únicamente la representación de conjuntos de puntos de interés mediante la estructura de datos compacta k^2-tree.

Los conjuntos de puntos de interés corresponden a cualquier punto sobre un mapa que se encuentre dentro de la lista de sitios soportados por la API de Google Maps, estos se visualizan en la Tabla 2.1.

Puntos de interés incluidos en las aplicaciones.	
N°	Tipo de punto de interés
1	Universidades
2	Hoteles
3	Restaurantes
4	Hospitales
5	Aeropuertos
6	Bancos
7	Bares
8	Cementerios
9	Bomberos
10	Gasolineras
11	Gimnasios
12	Abogados
13	Botillerías
14	Estacionamientos
15	Farmacias
16	Carabineros
17	Colegios
18	Estaciones de Metro
19	Panaderías
20	Librerías
21	Iglesias
22	Delivery
23	Museos
24	Zapaterías

Tabla 2.1: Tipos de puntos de interés soportados por la API de Google Maps

Capítulo 3

Conceptos preliminares

En este capítulo se presentan las principales consultas espaciales, y en particular la consulta de proximidad espacial *KCPQ* que es implementada en este proyecto (Sección 3.1). Además, se presenta la estructura de datos compacta k^2-tree, describiendo en detalle cómo se construye y cómo se recorre (Sección 3.2). Finalmente se presentan los algoritmos utilizados para computar la consulta *KCPQ* sobre la estructura.

3.1. Consultas Espaciales

Un sistema de base de datos espaciales, es un sistema administrador de bases de datos, encargado principalmente de manejar datos de tipo espacial. Los datos espaciales se utilizan para representar de forma simplificada objetos y espacios físicos del mundo real, puesto que estos no pueden ser representados con tipos de datos convencionales. Algunos de los tipos de datos espaciales más comunes son: puntos, líneas y polígonos (Güting, 1994; Shelkhar y Chawla, 2013). En la Figura 3.1 se ilustran los tipos de datos espaciales mencionados sobre un mapa de Google Maps.

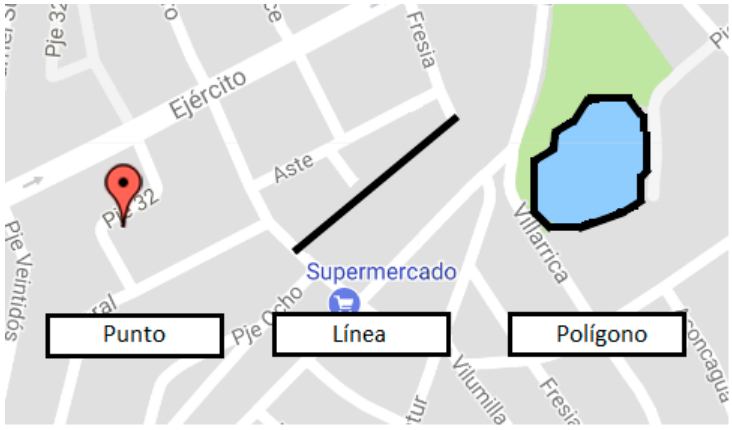


Figura 3.1: Tipos de puntos espaciales sobre un plano espacial de Google Maps

Un punto describe la forma de un objeto de dimensión cero, representado generalmente por coordenadas de la forma (latitud, longitud), por ejemplo, un punto de interés dentro de un mapa que representa la ubicación de un hotel (marcador rojo Figura 3.1). Una línea describe la forma de un objeto de una dimensión, una línea se conforma con la unión de dos o más puntos, por ejemplo, las calles en el mapa de una ciudad (línea de color negro Figura 3.1). Un polígono describe la forma de objetos de dos dimensiones, un polígono consiste en la unión de una serie de puntos o líneas que forman una figura cerrada, como por ejemplo, la laguna que se muestra en la Figura 3.1.

Existen muchas consultas espaciales que se aplican a estos tipos de datos, siendo las más comunes las siguientes (Shelkhar y Chawla, 2013):

1. Consulta de rango espacial (Window Query): Esta consulta retorna todos los objetos O que se encuentran dentro de un rango espacial determinado P. Por ejemplo: "Encontrar todas las gasolineras dentro de la ciudad de Santiago".
$$RQ(P) = \{O|O.G \bigcap P.G \neq \varnothing\}$$
Donde G es la geometría del objeto.

2. Consulta de join espacial: Esta consulta combina dos conjuntos de objetos espaciales R y S para formar un nuevo conjunto, con los objetos que satisfacen un predicado θ dado. Por ejemplo, la unión entre las parcelas que intersectan a un lago.
$$R \bowtie_\theta S = \{(o, o\prime)|o \in R, o\prime \in S, \theta(o.G, o\prime.G)\}$$
Donde G es la geometría del objeto.

3. Consulta de los K vecinos más cercanos (KNN): Esta consulta retorna los K puntos más cercanos a un punto q dado. Por ejemplo: "Encontrar los $K = 3$ hospitales más cercanos al lugar de un accidente q".

4. Consulta de los K-pares de vecinos más cercanos sobre dos conjuntos de datos de puntos ($KCPQ$): La consulta de los K pares de vecinos más cercanos consiste en, dado dos conjuntos de puntos R y S, encontrar los K pares más cercanos, de la forma (r, s) tal que, $r \in R$ y $s \in S$. Por ejemplo: "Encontrar los hoteles (R) más cercanos a aeropuertos (S)".

En este proyecto nos concentramos en la consulta de proximidad espacial $KCPQ$, donde los puntos son agrupados en conjuntos según el tipo de punto de interés, que como ya hemos mencionado pueden ser, hoteles, restaurantes, aeropuertos, gimnasios, farmacias, etc, (ver Tabla 2.1 del Capítulo 2). Estos conjuntos de puntos son obtenidos y mostrados en un mapa a través de la API de Google Maps y posteriormente son representados en matrices de adyacencia de orden $n \times n$ teniendo en cuenta las distancias Euclidianas entre puntos, con estas matrices se construyen las estructuras de datos compactas k^2-tree para procesar la consulta $KCPQ$.

Ejemplo 3.1 La Figura 3.2 ilustra el resultado de la consulta de proximidad espacial $KCPQ$ sobre puntos de interés ubicados en un mapa de la ciudad de Concepción, donde los

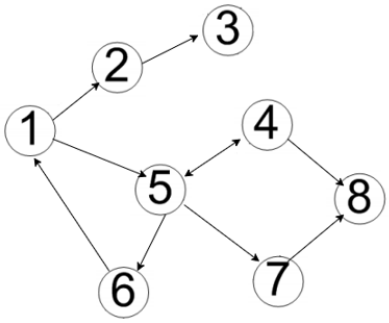
puntos de color rojo corresponden a hoteles y los de color azul corresponden a restaurantes. Los puntos unidos con líneas verdes representan a los $K = 4$ pares de puntos más cercanos. □

Figura 3.2: Visualización de los $K = 4$ pares de puntos más cercanos sobre un mapa de Google Maps

3.2. Estructura de datos compacta k^2-tree

Esta estructura de datos compacta fue presentada en (Brisaboa et al., 2009) para representar grafos de la Web, donde los nodos del grafo representan páginas Web y las aristas representan enlaces entre las páginas. En este escenario, la estructura compacta k^2-tree permite representar la relación binaria de enlace entre nodos del grafo, mediante una matriz de adyacencia. De esta forma, una celda (i, j) de la matriz de adyacencia contiene un 1 si existe una arista entre el nodo i y el nodo j en el grafo Web, en caso contrario, contiene un 0. En la Figura 3.3 se muestra un ejemplo de grafo Web y su representación mediante una matriz de adyacencia.

Ejemplo 3.2 La Figura 3.3(a) muestra un grafo con un total de 8 nodos, por lo tanto, se genera una matriz de 8×8, la que se muestra en la Figura 3.3(b). Como ilustración, la matriz se construye de la siguiente manera: El nodo etiquetado con un 1 en la Figura 3.3(a), se conecta con el nodo 2 y con el nodo 5, por lo tanto, en las celdas (1,2) y (1,5) se



(a)

	1	2	3	4	5	6	7	8
1	0	1	0	0	1	0	0	0
2	0	0	1	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0	1
5	0	0	0	1	0	1	1	0
6	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0

(b)

Figura 3.3: Grafo de enlaces Web y matriz de adyacencia

registra un 1, indicando que existe un enlace entre el nodo 1 y el nodo 2 y entre el nodo 1 y el nodo 5, el resto de las columnas queda con un 0 porque no hay un enlace entre el nodo 1 y el resto de los nodos. Así sucesivamente con los demás nodos. □

En este proyecto, en vez de trabajar con nodos de la Web, la estructura de datos compacta k^2-tree se utiliza para trabajar sobre puntos de interés dentro de un mapa. Para esto, se utiliza un método sencillo de traspaso de puntos desde un mapa a matrices de adyacencia, con las que se construyen las estructuras que representan a cada conjunto de puntos R y S. Primero se debe construir una única matriz que englobe a ambos conjuntos de puntos, esto con el fin de evitar inconsistencias, puesto que las matrices que representan a los conjuntos deben ser de igual tamaño y deben representar una misma área del mapa a la hora de procesar la consulta *KCPQ*, ya que por ejemplo, no tiene sentido realizar la consulta con hoteles que pertenecen a la ciudad de Concepción y con restaurantes que pertenecen a la ciudad de Santiago. Además de que no puede existir más de un punto de interés dentro de una misma celda, de lo contrario se perderían datos, ya que solo se toma en cuenta a uno de los puntos. Esto se ilustra en la Figura 3.4.

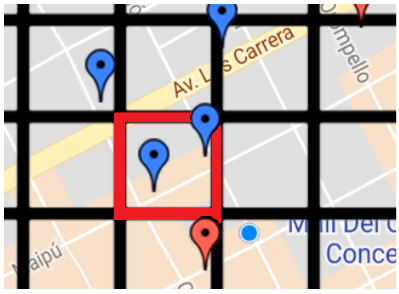
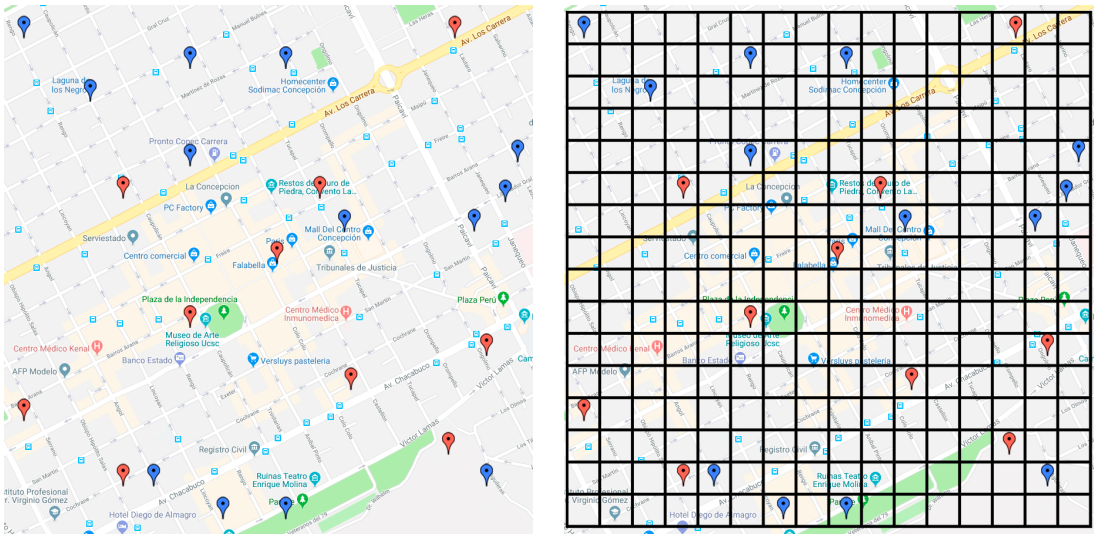


Figura 3.4: Ilustración de iconsistencia al tener más de un punto en una misma cuadrícula.



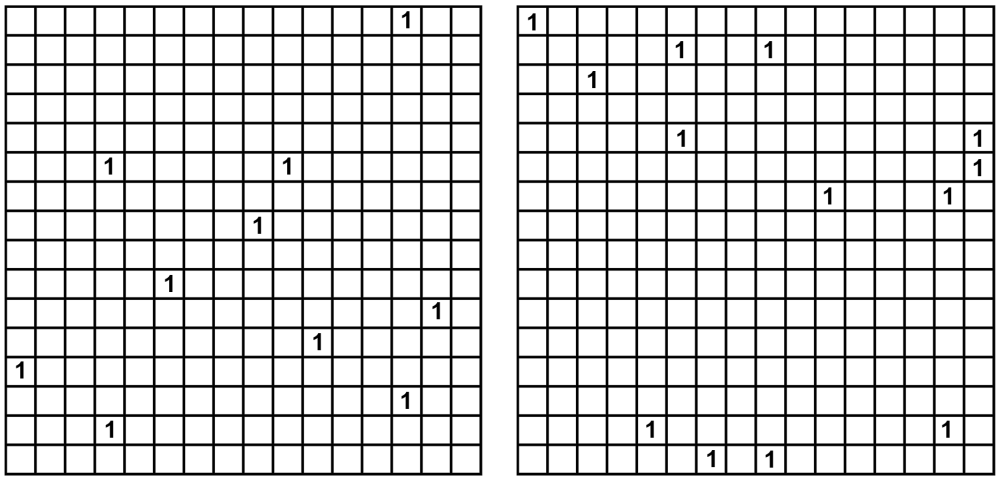
(a) (b)

Figura 3.5: Traspaso de puntos de interés a matriz de adyacencia

Ejemplo 3.3 La Figura 3.5(a) muestra el mapa de un área de la ciudad de Concepción con varios puntos de interés, donde los puntos de color rojo corresponden a hoteles y representan al conjunto R, por otra parte, los puntos de color azul corresponden a restaurantes y representan al conjunto S. La Figura 3.5(b) muestra la matriz que representa a todos los puntos de interés de la Figura 3.5(a), evitando inconsistencias como que exista más de un punto en una de las celdas. □

Cabe señalar que no podemos trabajar directamente sobre la matriz obtenida, ya que esta contiene más de un tipo de puntos de interés, que en este caso, son los puntos correspondientes a los conjuntos R y S. Al ser esta estructura una representación binaria solo podemos representar con un 1 la existencia de un elemento y con 0 lo contrario, por lo que no tendríamos como diferenciar que puntos pertenecen a un conjunto u otro. Entonces, a partir de esta matriz, obtenemos dos matrices de adyacencia, una que representa los puntos del conjunto R (color rojo) y otra que representa los puntos del conjunto S (color azul). Esto se aprecia en el siguiente ejemplo.

Ejemplo 3.4 La Figura 3.5(b) muestra un mapa con dos conjuntos de puntos ubicados en una matriz, donde cada punto se posiciona dentro de una única celda. Las Figura 3.6(a) y Figura 3.6(b) muestran el resultado de la separación por conjuntos de dichos puntos, donde la Figura 3.6(a) corresponde a la matriz de adyacendcia generada para representar a los puntos de interés del conjunto R (color rojo) y la Figura 3.6(b) corresponde a la matriz de adyacencia generada para representar a los puntos de interés del conjunto S (color azul). Se asume que los espacios vacíos estan rellenados con 0's. □



(a) Matriz de adyacencia conjunto R. (b) Matriz de adyacencia conjunto S.

Figura 3.6: Matrices de adyacencia correspondientes a los conjuntos de puntos R y S de la Figura 3.5

3.2.1. Construcción de la estructura de datos compacta k^2-tree

Como hemos mencionado, a partir de una matriz de adyacencia se puede construir la estructura de datos compacta k^2-tree, de la cual se crea un árbol k^2-ario que se calcula mediante un proceso de divisiones recursivas de la matriz de adyacencia. Para calcular el primer nivel del árbol se subdivide la matriz en k^2-submatrices cuadradas y del mismo tamaño. Cada una de estas submatrices se representan en el árbol con un 1, si contiene al menos un elemento y con un 0 en caso contrario. Para los siguientes niveles, cada submatriz que contenga al menos un elemento (las que están representadas con un 1) serán representadas con k^2-hijos en el siguiente nivel, los que se calculan subdividiendo dichas submatrices en k^2 submatrices nuevamente utilizando el mismo procedimiento. Este proceso recursivo continúa hasta que cada elemento del último nivel del árbol corresponde a una celda de la matriz de adyacencia. Cabe destacar que, en este proyecto utilizamos un valor de $k = 2$ por lo que dividimos en 4 submatrices en cada iteración.

Ejemplo 3.5 En la Figura 3.7 se presenta el k^2-tree de altura $h = [log_k n]$ correspondiente a la matriz de adyacencia del conjunto de puntos S de la Figura 3.6(b), donde n es la cantidad de filas y/o columnas de la matriz, y k es la constante binaria de la cual se descompone la matriz para generar la estructura k^2-tree, en este caso $k = 2$ y $n = 16$. Por lo tanto, la altura del k^2-tree correspondiente al ejemplo de la Figura 3.6(b) queda como $h = [log_2 8] = 4$. Dado que $k = 2$ se realizan 4 particiones a la matriz de adyacencia, lo que genera 4 hijos de la raíz del árbol (nivel 1). El primer elemento representa a la submatriz A, el segundo a la submatriz B, el tercero a la submatriz C y el cuarto a la submatriz D. Como A contiene al menos un valor 1, el primer elemento toma el valor 1 (lo mismo para el caso de B, C y D). Los siguientes niveles se calcularon de igual manera, particionando

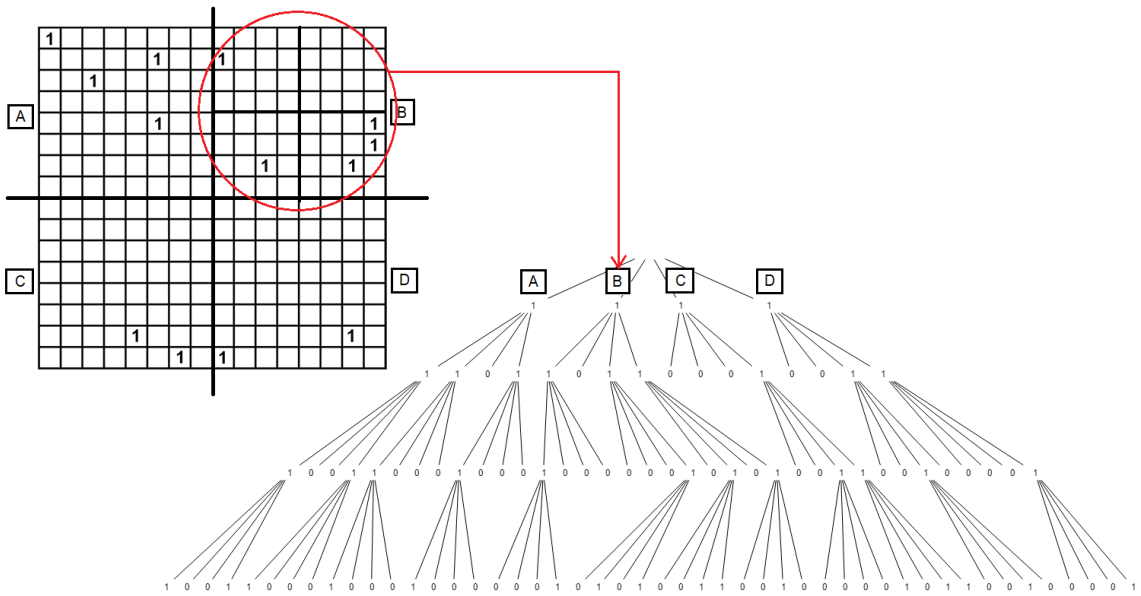


Figura 3.7: Construcción de árbol k^2-ario a partir de matriz de adyacencia correspondiente al conjunto de puntos S de la Figura 3.6(b)

nuevamente las submatrices hasta llegar a descomponer en celdas. □

El árbol obtenido sólo se utiliza como una representación visual de la estructura, siendo lo realmente relevante los bitmaps A y H que se generan a partir del recorrido en anchura de este, los cuales representan a los nodos internos y a los nodos hojas del árbol. Un bitmap es un arreglo que solo contiene 0's y 1's. Almacenar los datos en forma de bitmaps permite compactar el árbol generado, ya que solo se guardan los datos relevantes (representados con un 1), dejando fuera los datos vacíos (representados con 0). Esta representación está diseñada para compactar grandes matrices, en donde existen muchas áreas de 0's (Brisaboa et al., 2009). Cabe destacar que, en la implementación, A y H se implementan como un único bitmap que engloba a los dos $(A + H)$, el cual denominamos T.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	...
T	1	1	1	1	1	1	0	1	1	0	1	1	0	0	0	1	0	0	1	1	1	0	...

22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	...
0	1	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0	1	0	...

Figura 3.8: Bitmap T generado a partir del recorrido en anchura del árbol de la Figura 3.7

3.2.2. Navegación sobre la estructura de datos compacta k^2-tree

Para recorrer la estructura, es necesario utilizar dos operaciones sobre el bitmap T, estas son: *Rank* y *Select* (Fariña et al., 2009), (González et al., 2005). Dado un bitmap T y una posición p, la $Rank_1(T, p)$ cuenta la cantidad de 1's (o 0's), hasta una posición p en T. La operación $Select_1(T, j)$ retorna la posición de la j-ésima ocurrencia de 1 (o 0) en T. Para obtener el hijo de un nodo, se utiliza la función $child_i(x) = Rank(A, x) \times k^2 + i$ que permite obtener el i-ésimo hijo del nodo x en el k^2-tree.

El siguiente ejemplo ilustra la aplicación de las operaciones *Rank*, *Select* y *Child* sobre el bitmap de la Figura 3.8.

Ejemplo 3.6 Considerando el k^2-tree de la Figura 3.7 y su respectivo bitmap T (Figura 3.8). La operación $Rank(T, 15) = 11$ nos indica que existen 11 1's hasta la posición 15 del bitmap T (desde la posición 0). La operación $Select(T, 19) = 39$ nos indica que el decimonoveno 1 está en la posición 39 del bitmap T.

Si queremos obtener el tercer hijo (las posiciones parten de 0, por lo tanto $i = 2$) del segundo hijo de la raíz ($x = 1$) del k^2-tree, debemos aplicar $child_2(1) = Rank(T, 1) \times k^2 + 2 = 2 \times 2^2 + 2 = 10$, lo que indica que el tercer hijo del segundo hijo de la raíz se encuentra en la posición 10 del bitmap T. Si queremos obtener todos los hijos del tercer hijo del segundo hijo de la raíz, es decir, los hijos del nodo 10, debemos obtener debemos obtener la posición del primer hijo de dicho nodo, por lo tanto, aplicamos: $child_0(10) = Rank(T, 10) \times k^2 + 0 = 9 \times 2^2 + 0 = 36$, lo que indica que el primer hijo del nodo 10 se encuentra en la posición 36 del bitmap. Entonces, el resto de hijos se encontrarían en las tres siguientes posiciones. □

3.3. Algoritmos utilizados

3.3.1. Algoritmo para construir matrices de adyacencia desde un mapa de Google Maps

Como se mencionó anteriormente, en este proyecto aplicamos un sencillo método de traspaso de puntos desde un mapa a matrices de adyacencia, que se ilustra en la Figura 3.5. Para realizar esta labor, utilizamos los algoritmos $obtenerParametros()$ y $obtenerMatriz()$ implementados en (Maldonado, 2017).

El Algoritmo 1 se encarga de obtener las ubicaciones de los puntos obtenidos a través de la API de Google Places en formato *Marker* y almacenarlas en los arreglos que son procesados por los algoritmos encargados de construir las matrices de adyacencia $obtenerParametros()$ y $obtenerMatriz()$ en el formato $(latitud, longitud)$.

En las Líneas 1-6 se declaran las variables, que contendrán las latitudes y longitudes de los conjuntos de puntos retornados por la API. Los arreglos $latR$ y $lngR$ almacenan las latitudes y longitudes de los puntos correspondientes al conjunto R. Los arreglos $latS$ y $lngS$ almacenan las latitudes y longitudes de los puntos correspondientes al conjunto S.

Algoritmo 1: Obtener coordenadas de puntos para construir matrices de adyacencia

```
   Input: Conjuntos de puntos R[] y S[] en formato Marker
   Output: Conjunto P[] y T[] con las posiciones de los puntos en la matriz de adyacencia
 1 latR[] ← createArrayList(); 
 2 lngR[] ← createArrayList(); /*latitudes y longitudes conjunto R*/
 3 latS[] ← createArrayList(); 
 4 lngS[] ← createArrayList(); /*latitudes y longitudes conjunto S*/
 5 latitudes[] ← createArrayList(); 
 6 longitudes[] ← createArrayList(); /*latitudes y longitudes de ambos conjuntos para obtener
     parámetros*/
 7 for i = 0; i < R.length; i++ do
 8     latR.push(R[i].getLatitude());
 9     lngR.push(R[i].getLongitude());
10     latitudes.push(R[i].getLatitude());
11     longitudes.push(R[i].getLongitude());
12 for j = 0; j < S.length; j++ do
13     latS.push(S[j].getLatitude());
14     lngS.push(S[j].getLongitude());
15     latitudes.push(S[j].getLatitude());
16     longitudes.push(S[j].getLongitude());
17 obtenerParametros(latitudes[], longitudes[]);
18 P[] ← construirMatriz(latR[], lngR[]);
19 T[] ← construirMatriz(latS[], lngS[]);
```

Los arreglos *latitudes* y *longitudes* se encargan de almacenar las latitudes y longitudes de los puntos de ambos conjuntos.

En las Líneas 7-11 comienza el ciclo **for** que recorre el arreglo que almacena los puntos del conjunto R en formato *Marker* obteniendo las latitudes y longitudes de cada punto, que son agregadas a los arreglos $latR$, $lngR$, *latitudes* y *longitudes*.

En las Líneas 12-16 se declara otro ciclo **for** que recorre el arreglo que almacena los puntos del conjunto S obteniendo las latitudes y longitudes de cada uno de sus puntos, agregándolas a los arreglos $latS$, $lngS$, *latitudes* y *longitudes*.

En la Línea 17 se llama al algoritmo *obtenerParametros*() que recibe las latitudes y longitudes de ambos conjuntos de puntos (arreglos *latitudes* y *longitudes*) y se obtienen los parámetros necesarios para construir las matrices de adyacencia (mayor y menor distancia entre puntos).

Finalmente, se llama al algoritmo *obtenerMatriz*() dos veces, una con las latitudes y longitudes del conjunto R (arreglos $latR$ y $lngR$) y la otra con las latitudes y longitudes del conjunto S (arreglos $latS$ y $lngS$). Con esto obtenemos las posiciones de los puntos ubicados en sus respectivas matrices de adyacencia para los conjuntos de puntos R y S (arreglos P y T respectivamente).

El Algoritmo *obtenerParámetros*() recibe como entradas los arreglos de *latitudes*[] y *longitudes*[] (que contienen las posiciones de los puntos de interés de ambos conjuntos) y genera los parámetros que se utilizarán para construir las matrices de adyacencia, siendo los más relevantes la mayor y menor distancia entre puntos. Con la menor distancia, se obtiene la distancia mínima que abarca cada cuadrícula respecto al mapa en la matriz de adyacencia, y con esto se evita que dos o más puntos se encuentren en la misma (ver

Figura 3.4). Por otra parte, sabiendo la mayor distancia entre dos latitudes o dos longitudes (puntos que se encuentran en los extremos), podemos obtener la cantidad máxima de cuadrículas para las filas y columnas, dividiendo la mayor medida entre la menor (X e Y máximos del plano Cartesiano).

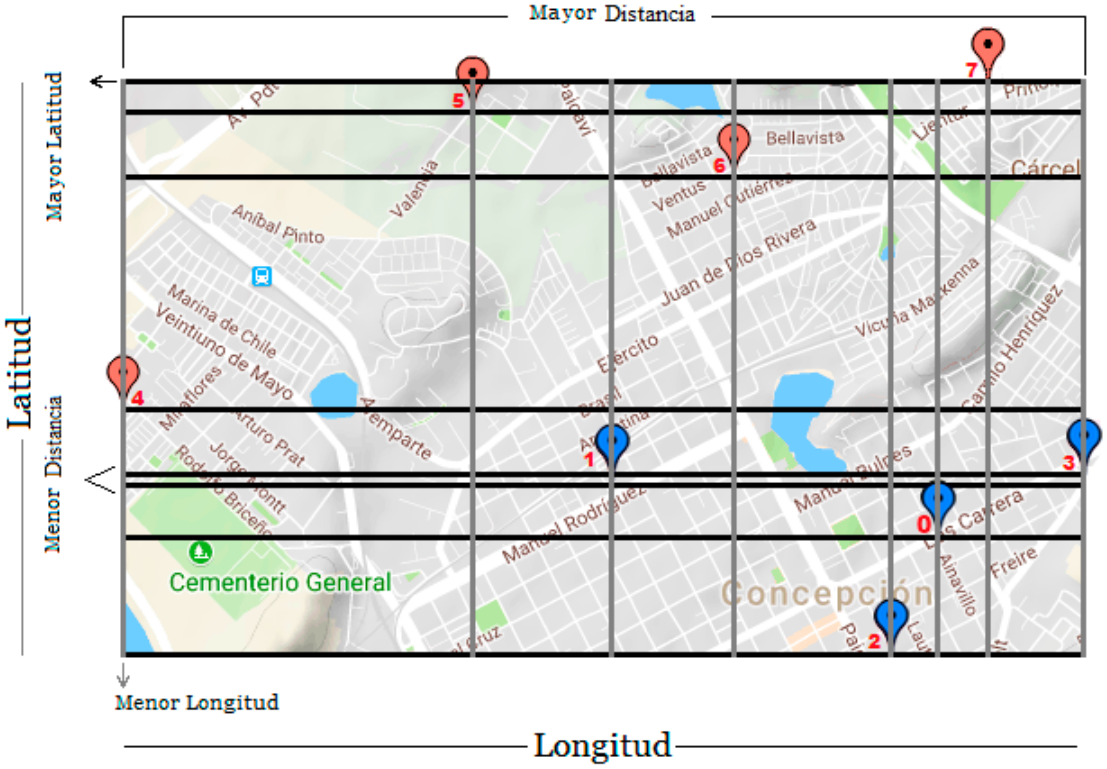


Figura 3.9: Parámetros obtenidos por algoritmo *obtenerParametros*() sobre el mapa del Ejemplo 3.7

Punto	Latitud	Longitud
0	-36.81715691684944	-73.03874015808105
1	-36.81598451063416	-73.05521965026855
2	-36.82292420683277	-73.04140090942383
3	-36.81547917040088	-73.03221702575684
4	-36.81261750064258	-73.07719230651855
5	-36.80189705401094	-73.06139945983887
6	-36.80423368952764	-73.04895401000977
7	-36.80017889417579	-73.03659439086914

Tabla 3.1: Coordenadas de los puntos de la Figura 3.9

Ejemplo 3.7 La Figura 3.9 muestra un mapa de la comuna de Concepción con ocho puntos, donde los puntos de color rojo representan al conjunto R y los de color azul al conjunto S, cuyas coordenadas se visualizan en la Tabla 3.1 (que corresponden a las coordenadas almacenadas en *latitudes*[] y *longitudes*[]. Estas coordenadas son precisas en cuanto a decimales, ya que, se necesita la mayor precisión posible en cuanto a la ubicación de los puntos dentro del mapa, al ser este de una escala pequeña, ya que aproximar coordenadas cambiaría considerablemente la posición de los puntos. El algoritmo compara la diferencia de las distintas coordenadas entre todos los puntos en el mapa, obteniendo como resultados: La menor distancia = 0.000505340233277739, que se presenta entre las latitudes del Punto 3 y el Punto 1 (latitudes -36.81547917040088 y -36.81598451063416 respectivamente), la mayor medida = 0.04497528076171875 entre las longitudes más alejadas correspondientes al Punto 4 y Punto 3 (longitudes -73.07719230651855 y -73.03221702575684 respectivamente), la mayor latitud = -36.80017889417579 en el Punto 7 (punto ubicado más arriba) y la menor longitud = -73.07719230651855 en el Punto 4 (punto ubicado más a la izquierda). Con estas dos últimas medidas se obtiene el punto del mapa donde iniciaría la matriz de adyacencia para la estructura compacta de datos k^2-tree. En cuanto a la cantidad de cuadrículas de la matriz, obtenemos el resultado aproximado 0.04497528076171875/0.000505340233277739 = 89. Por último, el tamaño geográfico de cada cuadrícula corresponde a la menor distancia (0.000505340233277739). En la práctica, cada cuadrícula abarca un área de $0.000505340233277739 \times 0.000505340233277739$ dentro del mapa. □

Ahora bien, para ubicar los puntos del mapa dentro de la matriz de adyacencia, se utiliza el algoritmo $obtenerMatriz()$, que devuelve un conjunto de puntos que almacena las coordenadas cartesianas (X, Y) de las cuadrículas, donde los puntos del mapa corresponden a 1's dentro de la matriz de adyacencia. Este arreglo es utilizado posteriormente para generar la estructura de datos compacta k^2-tree. Las posiciones de las cuadrículas que no están presentes en el arreglo se rellenan con 0's en la matriz. El formato del arreglo es el siguiente: *[X punto 0, Y punto 0, X punto 1, Y punto 1, ... ,X punto n, Y punto n]*.

Ejemplo 3.8 La Figura 3.10 muestra una representación de la matriz de adyacencia para la estructura de datos compacta k^2-tree que resultaría respecto a los parámetros obtenidos en el Ejemplo 3.7 sobre el mapa de la Figura 3.10, en que se visualiza una matriz de adyacencia de 53×89, esto para ahorrar espacio en el informe frente al tamaño real (89×89), donde las cuadrículas amarillas, representan 1's dentro de esta. Cabe destacar que la matriz de adyacencia, en la implementación, es una matriz de tamaño 128×128, al ser este valor la potencia de 2 más cercana capaz de comprender completamente la matriz. Este tamaño es relevante al momento de construir el k^2-tree puesto que la cantidad de particiones que se realizan a la matriz también es un valor potencia de 2.

Ahora bien, para obtener las matrices de adyacencia correspondientes a ambos conjuntos de puntos R y S (puntos rojos y puntos azules respectivamente) debemos llamar al algoritmo $obtenerMatriz()$ dos veces: Una solo con los puntos del conjunto R y otra solo con los puntos del conjunto S, además de los parámetros obtenidos por el algoritmo

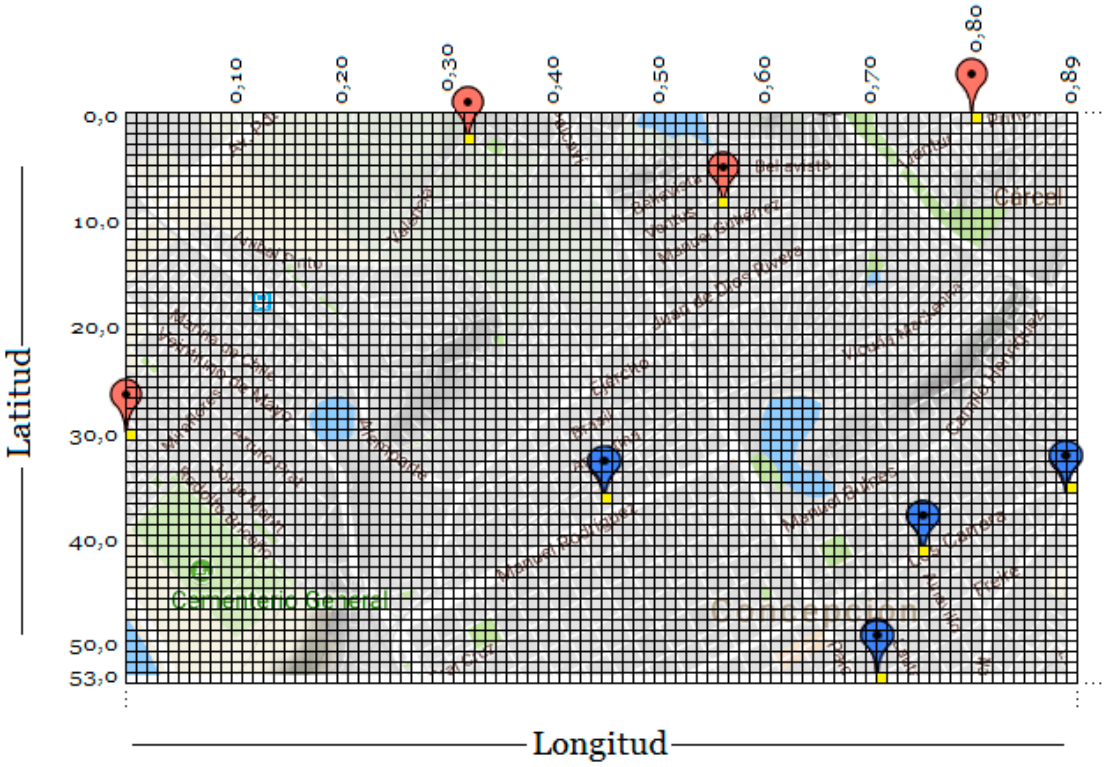


Figura 3.10: Representación del algoritmo *obtenerMatriz*() aplicado al Ejemplo 3.7

obtenerParametros() . En consecuencia obtenemos dos arreglos de coordenadas cartesianas que almacenan las posiciones de las cuadrículas donde se ubica cada punto de interés del mapa, en su respectiva matriz de adyacencia, los que denominamos P y T. Recordemos que estos algoritmos devuelven un conjunto con las posiciones de los puntos ubicados en una matriz de adyacencia y no la matriz como tal.

Ejecutando el algoritmo *obtenerMatriz*() con ambos conjuntos de puntos por separado obtenemos un arreglo $P = [41, 75, 36, 45, 53, 71, 35, 89]$ correspondiente a las posiciones de los puntos 0 al 3 en su matriz de adyacencia y un arreglo $T = [30, 0, 2, 32, 8, 56, 0, 80]$ correspondiente a las posiciones de los puntos 4 al 7 en su matriz de adyacencia.

□

3.3.2. Algoritmo para computar la consulta *KCPQ*

El algoritmo 2 implementa la consulta para obtener los K-pares de vecinos más cercanos ($KCPQ$), este algoritmo fue implementado por (Santolaya, 2017) y se encuentra alojado en un Servidor Web (Web Service) al que se conectan las aplicaciones. El algoritmo recibe como parámetros de entrada el valor de K y dos estructuras de datos compactas k^2-tree que almacenan los conjuntos de puntos de interés R y S sobre los que se aplica la consulta,

Algoritmo 2: Obtener los K-pares de vecinos más cercanos sobre dos estructuras compactas k^2-tree

```
   Input: Dos estructuras de datos compactas k^2-tree para R y S
          Número de K-pares más cercanos a obtener
   Output: Un conjunto de K tuplas de la forma (r, s)
 1     /*Inicialmente el heap con los pares de candidatos Cand se encuentra vacío*/
 2   C ← CreateMaxHeap();
 3     /*pQueue es un heap de valores mínimos para almacenar datos de la forma (Q1, Q2, md)*/
 4     /*crea el primer elemento de pQueue*/
 5   e ← CreateElementQueue(R.quad(), S.quad(), minDist(R.quad(), S.quad()));
 6   pQueue ← CreateMinHeap();
 7   pQueue.insert(e);
 8   while (pQueue ≠ 0) do
 9       e ← pQueue.delete();
10       if |Cand| = K AND e[3] ≥ Cand.Max() then
11           return Cand;
12       if e[1] AND e[2] are leaves then
13           e_2 ← CreateEntry(e[1].coord(), e[2].coord, e[3]);
14           if |Cand| < K then
15               Cand.insert(e_2);
16           else
17               if e[3] < Cand.Max() then
18                   Cand.delete();
19                   Cand.insert(e_2);
20       else
21           if (e[1] OR e[2] have children) then
22               forall pair of (child h_1 ∈ e[1] and h_2 ∈ e[2]) do
23                   minD ← MinDist(h_1.quad(), h_2.quad());
24                   if (|Cand| < K) OR (minD < Cand.Max()) then
25                       e_2 ← CreateElementQueue(h_1.quad(), h_2.quad(), minD);
26                       pQueue.insert(e_2);
27   return Cand;
```

para esto se implementan dos heap, uno llamado *pQueue* que corresponde a un heap de valores mínimos que almacena las distancias entre los pares de cuadrantes R y S ordenadas de menor a mayor, de esta forma, el primer elemento e de *pQueue* (elemento en el tope del heap) corresponde a los pares de cuadrantes que están más cercanos entre sí. La idea es ordenar los cuadrantes de las matrices de adyacencia (representadas en las estructuras de datos compactas k^2-tree) en *pQueue*, respecto a sus distancias mínimas, de manera de evitar visitar aquellos cuadrantes que están más lejanos, dado que en estos últimos no existe probabilidad de encontrar mejores candidatos.

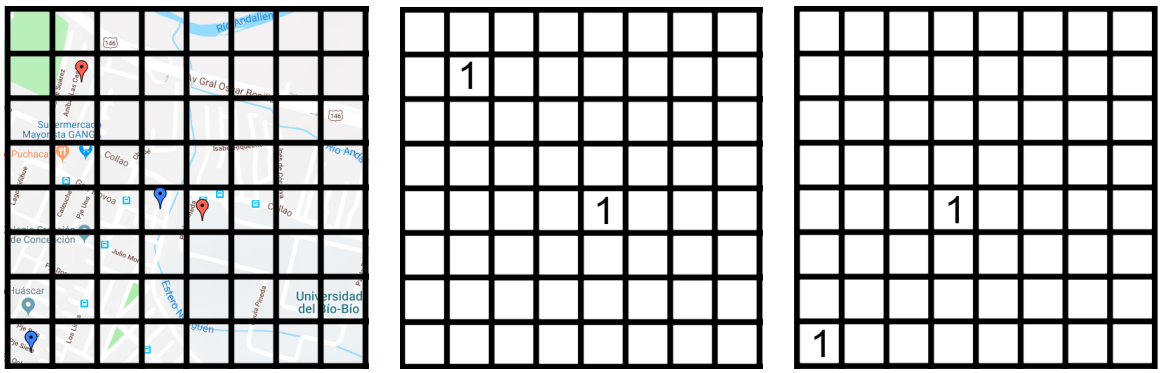
El otro heap se denomina *Cand* y corresponde a un heap de valores máximos que almacena los K potenciales pares de puntos más cercanos (pares de puntos candidatos), de esta forma, el elemento en el tope del heap *Cand* corresponde a los pares de puntos que está más alejados entre sí. De esta forma, si en algun momento se encontrase un mejor par de puntos candidatos con respecto al que se encuentra en el tope *Cand*, este se elimina directamente y se reemplaza por el par de puntos con menor distancia, así evitamos recorrer el heap completamente.

Primero se crea el heap $Cand$ que contendrá, al final del proceso, los K-pares de puntos más cercanos entre R y S (Línea 2 del Algoritmo 2). En la Línea 5, se crea el primer elemento e de $pQueue$ que almacena los pares de cuadrantes que serán visitados. En las Líneas 6-7 se crea el heap $pQueue$, al que en la primera iteración se le inserta el elemento e con las raíces de ambas estructuras, y en cada una de estas será actualizado considerando las mínimas distancias entre los cuadrantes de R y S. Luego, mientras el heap $pQueue$ no esté vacío, el algoritmo busca los K-pares de puntos más cercanos entre R y S (Líneas 8-26). Cada vez que se obtiene un nuevo elemento e desde $pQueue$ existen 3 posibilidades:

1. El número de candidatos K ha sido alcanzado y el elemento e en el tope de $pQueue$ tiene una distancia mayor que el elemento en el tope de $Cand$. En este caso el algoritmo finaliza (Líneas 10-11).

2. El elemento e es un par de hojas (con valor 1 en la estructura), en tal caso, el algoritmo crea un elemento e_2 para ser insertado en el heap $Cand$. Si el numero actual de candidatos es menor que K, entonces e_2 se inserta en $Cand$. Por el contrario, si la distancia del elemento e es menor que la distancia del elemento en el tope de $Cand$, entonces el elemento tope de $Cand$ se borra y el nuevo elemento e_2 se inserta en $Cand$ (Líneas 12-19).

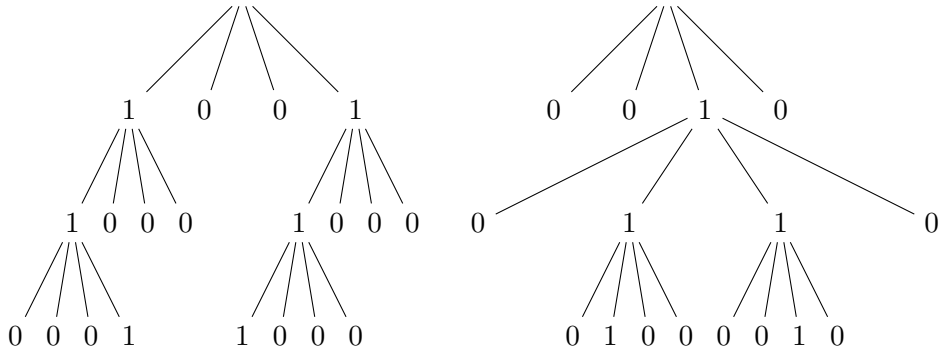
3. El elemento e contiene un nodo con hijos. Entonces para cada hijo $h_1 \in e[1]$ y $h_2 \in e[2]$, el algoritmo obtiene la mínima distancia entre h_1 y h_2. Si el número de candidatos actuales es menor que K o la distancia mínima del actual par de hijos es menor que el elemento tope de $Cand$, entonces, es posible que un nuevo candidato pueda ser encontrado en el par de cuadrantes, por lo tanto, un nuevo elemento e_2 es añadido en $pQueue$ (Líneas 21-26).



(a) (b) (c)

Figura 3.11: Puntos de interés en un mapa y sus respectivas matrices de adyacencia R y S

Ejemplo 3.9 Consideremos la Figura 3.11(a) que muestra un mapa con 4 puntos de interés pertenecientes a dos conjuntos según su color y la Figura 3.11(b) y Figura 3.11(c) que



T: 1001-1000-1000-0001-1000 (a)

T: 0010-0110-0100-0010 (b)

Figura 3.12: k^2-tree y bitmaps correspondientes a matrices de adyacencia de la Figura 3.11(b) y Figura 3.11(c)

muestran las respectivas matrices de adyacencia con las que se construyen los k^2-tree utilizados por el algoritmo para resolver la consulta *KCPQ*. Consideremos que se desean encontrar los $K = 2$ pares de puntos (r, s) más cercanos entre sí.

1. Inicialmente *pQueue* almacena al elemento e que al principio contiene los pares de cuadrantes iniciales de los k^2-tree de R y S, que corresponden a las raíces de éstos, como el número de candidatos K no ha sido alcanzado (Líneas 10-11), ni los elementos $e[1]$ y $e[2]$ no son hojas (o celdas, Líneas 12-19), se insertan en *pQeue*, todos los hijos de las raíces que contengan hijos, en este caso los pares de cuadrantes $(R_A, S_C, 4)$ y $(R_D, S_C, 4)$ con sus respectivas distancias (4 en ambos casos). Estos elementos son actualizados y ordenados en el heap *pQueue* que es un heap de valores mínimos, donde el primer elemento o el elemento en el tope del heap, es el que posee la distancia más corta (ver Figura 3.13(a)).

2. Luego, el algoritmo retorna a la Línea 9 donde toma al elemento en el tope de *pQueue*, el par de cuadrantes $(R_A, S_C, 4)$ que posee la misma distancia que el otro elemento, como ambos elementos poseen la misma distancia, ambos son analizados con el objetivo de encontrar los mejores candidatos, como el par de cuadrantes $(R_A, S_C, 4)$ poseen hijos, todos son agregados a *pQueue* (ver Figura 3.13(b))

3. El algoritmo vuelve a la Línea 9 y toma el elemento en el tope de *pQueue* que contiene los pares de cuadrantes $(R_D, S_C, 4)$, que al parecer contiene mejores candidatos que el elemento analizado en la iteración anterior, ya que las distancias entre estos son menores, como estos pares de cuadrantes poseen hijos, estos también son agregados a *pQueue*, los que son ordenados de acuerdo a la mínima distancia entre subcuadrantes (ver Figura 3.13(c)).

4. El algoritmo sigue el proceso de subdivisiones puesto que aún no se obtienen los K-pares de puntos candidatos (elementos en $Cand$) ni el elemento en el tope de $pQueue$ es un par de hojas, por lo que se toma al elemento con la siguiente menor distancia, correspondiente a los pares de cuadrantes $(R_{D1}, S_{C2}, 2)$ que nuevamente tienen hijos, los que deben ser insertados en $pQueue$ (ver Figura 3.13(d)), $pQueue$ es actualizado ubicando al elemento con la menor distancia entre pares de cuadrantes, correspondiente a las hojas $(R_{D11}, S_{C22}, 1)$ con la menor distancia igual a 1.

5. Nuevamente se repite el proceso, volviendo a la Línea 9 donde el algoritmo toma al primer elemento de $pQueue$ que corresponde a un par de hojas, y como $Cand$ aún se encuentra vacío, el algoritmo ejecuta las Líneas 12-15, donde las coordenadas del primer y segundo elemento $e[1]$ y $e[2]$ son insertadas en el heap $Cand$, junto con su distancia (en este caso 1), obteniendo así, el primer par de puntos más cercanos entre los conjuntos R y S (ver Figura 3.13(e)).

6. Como aún no tenemos los K-pares de puntos solicitados, el algoritmo continúa el proceso, tomando el elemento en el tope de $pQueue$ (Línea 9), el par de cuadrantes $(R_{D1}, S_{C3}, 4.472)$ que poseen hijos, correspondientes al par de hojas $(R_{D11}, S_{C33}, 5)$ que también son insertadas en $pQueue$ y ordenadas de acuerdo a sus distancias (ver Figura3.13(f)).

7. El algoritmo vuelve a la Línea 9 y toma el elemento en el tope de $pQueue$ que contiene los pares de cuadrantes $(R_{A1}, S_{C2}, 4.472)$ que también poseen hijos, el par de hojas $(R_{A14}, S_{C22}, 3.605)$, este elemento es insertado en $pQueue$ y ubicado en el tope del heap, ya que posee la menor distancia (ver Figura 3.13(g)).

8. Una vez más el algoritmo retorna a la Línea 9 y toma nuevamente al elemento en el tope de $pQueue$, como este corresponde al par de hojas $(R_{A14}, S_{C22}, 3.605)$ se ejecutan las Líneas 12-19, donde las coordenadas del primer y segundo elemento $e[1]$ y $e[2]$ son insertadas en el heap $Cand$, obteniendo así, el segundo par de puntos más cercanos (ver Figura 3.13(h)).

9. Como ya tenemos los $K = 2$ pares de puntos más cercanos solicitados, y la distancia del elemento en el tope de $pQueue$ es mayor a la distancia del elemento en el tope del heap $Cand$, el algoritmo retorna $Cand$ y finaliza (Líneas 10-11).

□

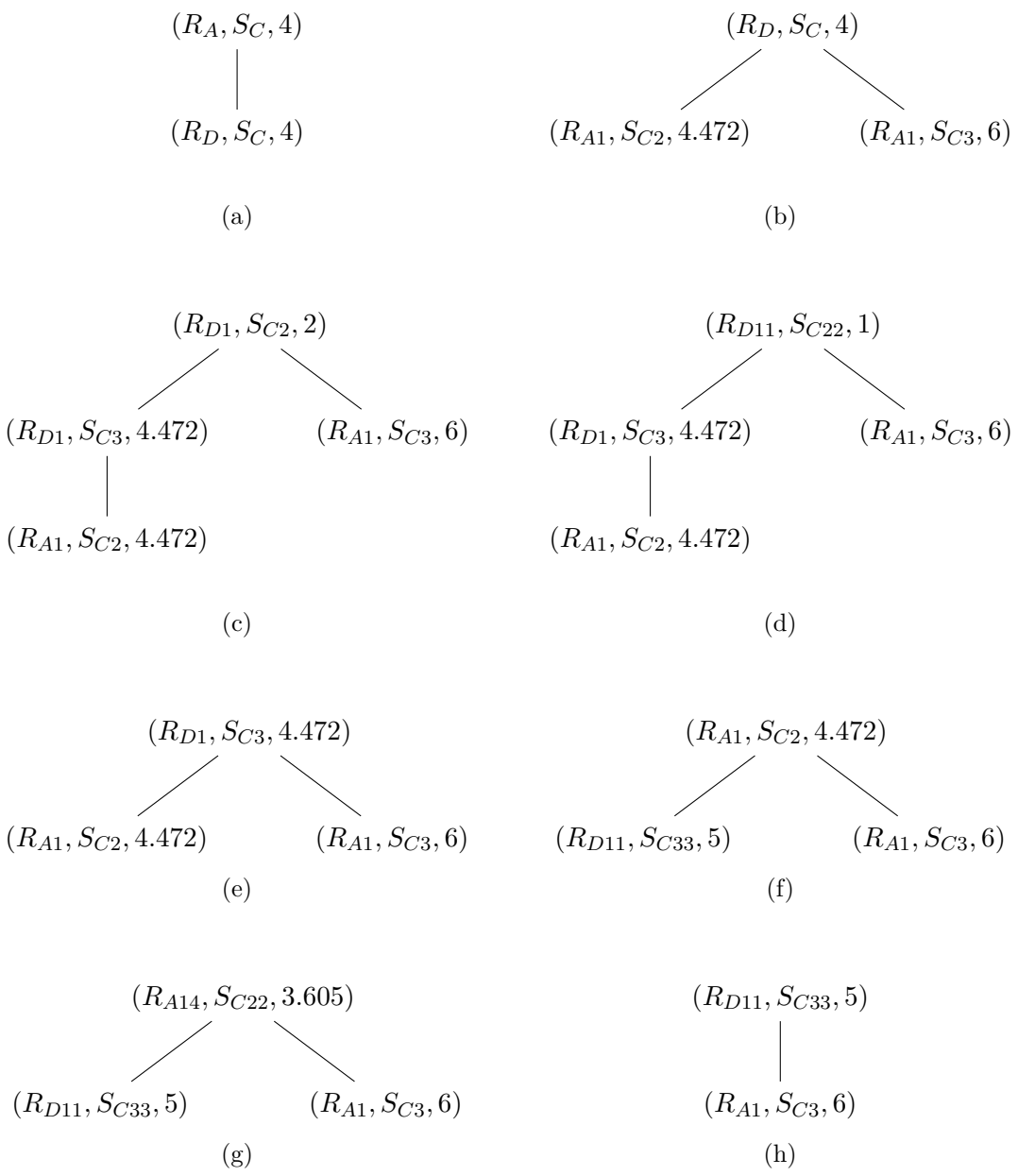


Figura 3.13: Heap de valores mínimos *pQueue* para el Ejemplo 3.9

Capítulo 4

Sistema Web y Móvil *KCPQ*

En este capítulo se presentan las aplicaciones que se implementaron para resolver la consulta de los K-pares de vecinos más cercanos, en su versión Web y Móvil respectivamente. En la sección 4.1 se presenta la arquitectura diseñada para ambas aplicaciones y sus correspondientes módulos.

4.1. Arquitectura

En este proyecto se implementaron dos aplicaciones, la primera para la plataforma Web y la segunda para la plataforma Android, si bien son plataformas distintas, la arquitectura que se utilizó es exactamente la misma (ver Figura 4.1), donde R y S corresponden al primer y segundo conjunto de puntos de interés que son seleccionados por el usuario, q es el punto en el mapa donde se ubica el usuario, rd es el radio en metros en donde se buscarán los puntos de interés previamente seleccionados por el usuario (respecto a la posicion actual q), k es la constante a utilizar en la estructura compacta k^2-tree (debe ser múltiplo de 2), $R[0, n]$ y $S[0, m]$ son los conjuntos de puntos de interés encontrados. $P[0, n]$ y $T[o, m]$ son las coordenadas cartesianas de los conjuntos de puntos de interés R y S ubicados en sus respectivas matrices de adyacencia para construir los k^2-tree, *cuad* es la cantidad de filas y columnas de las matrices de adyacencia a construir, K es la cantidad de pares de vecinos más cercanos que el usuario desea encontrar y $V[K]$ es el conjunto que contiene la respuesta a la consulta *KCPQ* con los K-pares de vecinos (puntos) más cercanos, entregada por el Web Service que es mostrada en las interfaces de las aplicaciones.

4.1.1. Módulo 1: Seleccionar conjuntos de puntos de interés a ingresar en el mapa

En este módulo se generan los conjuntos de puntos de interés con los que se va a construir la estructura k^2-tree y así resolver la consulta propuesta en este proyecto. Para esto, el usuario debe elegir en que lugar del mapa se desea posicionar (punto q), por defecto el sistema ubica al usuario en su posición actual (utilizando el servicio de Geolocalización del dispositivo), sin embargo, si se requiere, este puede modificar dicha posición a gusto

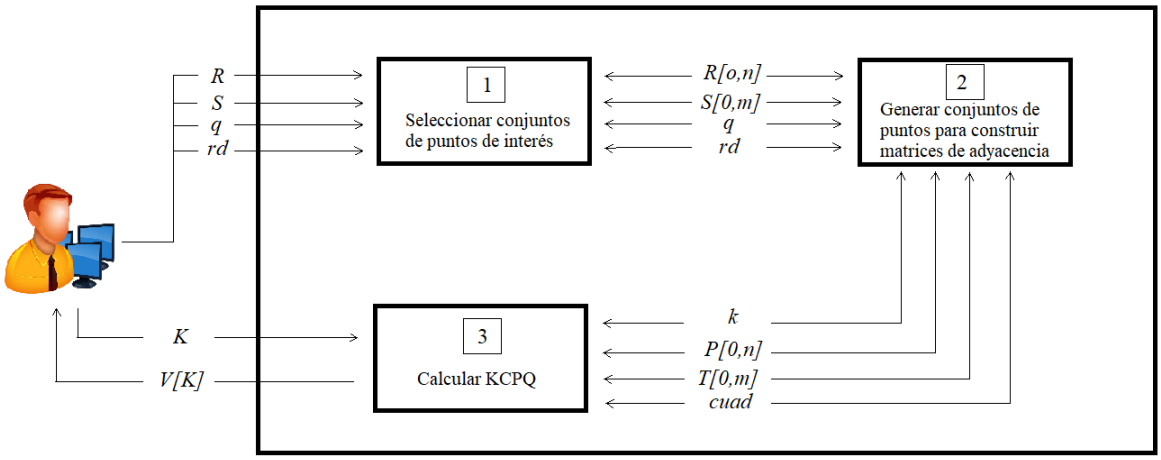


Figura 4.1: Arquitectura del Sistema *KCPQ*

ubicandola en cualquier parte del mapa. Luego, el usuario debe seleccionar, a través de dos listas desplegables, los tipos de puntos de interés con los que va a trabajar, correspondientes a los conjuntos R y S respectivamente (ver Tabla 2.1 del Capítulo 2). Una vez seleccionados los conjuntos, se procede a obtener los puntos correspondientes, para esto, se dió uso a las bibliotecas que provee la API de Google Maps, en específico la Google Places API, que permite encontrar los puntos de interés seleccionados (R y S) dentro de un radio rd respecto a la posición seleccionada por el usuario q. Cabe señalar que la aplicación guarda los puntos obtenidos desde la API en dos arreglos distintos (uno para cada conjunto) en el formato (latitud, longitud), aparte del formato predeterminado propuesto por la API, que devuelve objetos de la clase $Marker$, esto con el objetivo de trabajar solamente con los datos relevantes de cada punto (en este caso su ubicación). Estos arreglos son: $latitudesR[0,n]$ y $longitudesR[0,n]$ que contienen las latitudes y longitudes de los puntos encontrados para el conjunto R. Los arreglos, $latitudesS[0,m]$ y $longitudesS[0,m]$ que contienen las latitudes y longitudes de los puntos encontrados para el conjunto S.

4.1.2. Módulo 2: Construir matrices de adyacencia para R y S

Este módulo es el encargado de construir las matrices de adyacencia en las que se ubicarán los puntos de los conjuntos R y S obtenidos en el módulo anterior. Para esto, hay que tener en cuenta dos cosas. Primero, ambas matrices de adyacencia, tienen que ser del mismo tamaño y deben representar a un mismo plano dentro del mapa, como se mencionó en el Capítulo 3. Segundo, en cada celda de la matriz, no se puede representar más de un punto de interés a la vez (independientemente del conjunto al que pertenezcan), de lo contrario se generarán inconsistencias al momento de construir la estructura (ver Figura 3.4 del Capítulo 3). Para esto, este módulo llama al Algoritmo 1 con los arreglos $latitudes[0,n+m]$ y $longitudes[0,n+m]$ que almacenan las latitudes y longitudes de ambos conjuntos de puntos $R[0,n]$ y $S[0,m]$ combinados, de esta manera logramos construir

una sola matriz general que represente el mismo plano dentro del mapa y contenga a todos los puntos de R y S, evitando que existan más de uno dentro de una misma celda. De esta forma el Algoritmo 1 genera una serie de parámetros relevantes para la construcción de las matrices de adyacencia, como la cantidad de cuadrículas (*cuad*) que tendrán nuestras matrices, el tamaño de cada una de estas, entre otros. Estos valores nos sirven para obtener las coordenadas cartesianas de cada punto a ubicar en la matrices de adyacencia correspondientes.

Luego el algoritmo construye dos matrices de adyacencia con los parámetros obtenidos, una con los puntos del primer conjunto $latitudes[0, n]$ y $longitudes[o, n]$ correspondientes al conjunto R y la otra con las posiciones de los puntos del segundo conjunto $latitudes[0, m]$ y $longitudes[0, m]$ correspondientes al conjunto S.

Finalmente obtenemos dos arreglos $P[]$ y $T[]$ con las posiciones de los puntos ubicados en sus matrices de adyacencia, donde $P[]$ almacena los puntos del conjunto R y $T[]$ almacena los puntos del conjunto S. Estos arreglos son utilizados en el siguiente módulo para construir las estructuras de datos compactas k^2-tree y así aplicar la consulta de proximidad propuesta en este proyecto (*KCPQ*).

4.1.3. Módulo 3: Calcular K-pares de vecinos más cercanos (*KCPQ*)

Este módulo toma como entradas: los conjuntos P y T con las coordenadas de los puntos ubicados en sus matrices de adyacencia correspondientes (de tamaño potencia de 2), una constante k elevada al cuadrado (para ambos sistemas, se utiliza un $k = 2$), la cantidad de cuadrículas que contendrán las matrices *cuad* y el número de pares vecinos K que se desean obtener. Como salida se entrega un conjunto $V[]$, el cual almacena los K-pares de vecinos más cercanos entre los conjuntos R y S de la forma (r, s) representados en $P[]$ y $T[]$ además de sus distancias. Estos últimos serán los que se utilizarán para mostrar el resultado de la consulta en el mapa, uniendo los pares más cercanos solicitados por el usuario con líneas de color verde.

En este módulo se llama al Algoritmo 2, que fue adaptado e implementado en lenguaje JAVA por (Santolaya, 2017) y se encuentra almacenado en un servicio Web (o $WebService$), al cual los Sistemas acceden en dos ocasiones: La primera para construir las estructuras de datos compactas k^2-tree, en base a las coordenadas de los puntos de R y S ubicados en sus respectivas matrices de adyacencia $P[]$ y $T[]$, y la segunda para obtener el conjunto $V[K]$ con los K-pares de vecinos más cercanos desde los k^2-tree construidos. Ver Figura 4.2

Para llevar a cabo la primera interacción, es necesario pasar ciertos datos relevantes para la construir los k^2-tree (estos son: K, P, T y k) a un arreglo en formato de texto JSON, el cuál es enviado al Web Service, el cual se encarga de procesarlo y con estos datos construir las estructuras de datos compactas k^2-tree. Finalmente, en la segunda interacción, el Web Service retorna un nuevo arreglo en formato JSON que contiene los K-pares de puntos más cercanos entre las estructuras de datos compactas k^2-tree construidas en el paso anterior, que representan a los conjuntos de puntos R y S propuestos. Con los pares de puntos más cercanos devueltos por el Web Service estos se muestran en las interfaces de las aplicaciones, específicamente en el mapa, uniendo los pares de puntos obtenidos por

el Algoritmo 2 con líneas de color verde.

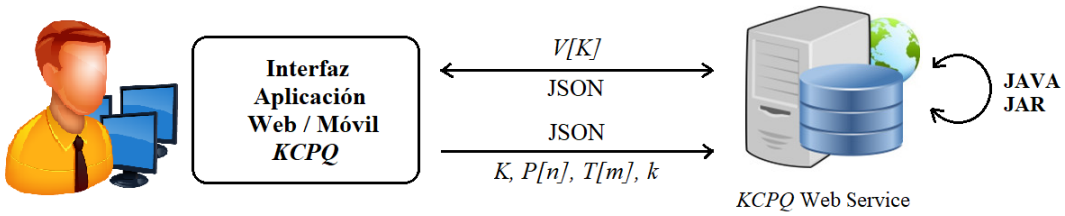


Figura 4.2: Representación del funcionamiento del Web Service utilizado

Capítulo 5

Desarrollo e Interfaz de los sistemas Web y Móvil *KCPQ*

En este capítulo se presentan y describen las herramientas utilizadas para el desarrollo tanto de la aplicación Web como de la aplicación Móvil, Además se ilustra el funcionamiento de estas con capturas de pantalla correspondientes a cada aplicación.

5.1. Herramientas utilizadas

Las herramientas utilizadas para el desarrollo de las aplicaciones Web y Móvil son:

5.1.1. PHP y HTML

PHP, de las siglas recursivas en inglés *Hypertext Preprocessor (procesador de hipertexto)*, es un lenguaje de programación de código abierto creado por Rasmus Lerdorf en el año 1995, este es uno de los lenguajes más populares, utilizado principalmente para el desarrollo Web.[1].

Este lenguaje se suele procesar directamente en un servidor, el cual genera código HTML que puede ser enviado a una aplicación cliente.

HTML por su parte, del inglés *HyperText Markup Language*[2], es el elemento de construcción más básico de una página Web. Es necesario resaltar que no es un lenguaje de programación como tal, ya que, no cuenta con funciones aritméticas, variables, o estructuras de control, por lo que sólo determina el contenido de la página, pero no su funcionalidad. Por esto, HTML se suele combinar con diferentes tecnologías que son usadas para describir, por ejemplo, la apariencia de una página Web (CSS) o su funcionalidad (Javascript).

Hiper Texto se refiere a los enlaces que conectan una página Web con otra, ya sean dentro del mismo sitio, o con sitios diferentes.

[1] http://php.net/manual/es/intro-whatis.php
[2] https://developer.mozilla.org/es/docs/Web/HTML

El lenguaje de marcado de HTML utiliza "markups"para anotar textos, imágenes entre otros elementos que se muestran en el Navegador web tales como <head>, <title>, <body>, <header>, <section>, , <p>, <img>, entre muchos otros más. Su estructura básica la podemos visualizar en la Figura 5.1.

Figura 5.1: Estructura básica de HTML interpretada por un Navegador Web

5.1.2. AJAX

Para definir AJAX es necesario definir *JavaScript* y *XML* dado que AJAX usas funciones de *JavaScript* y genera código *XML*. *JavaScript*, comúnmente llamado JS, es un lenguaje de programación orientado a objetos implementado como parte de un navegador Web (en la actualidad, todo navegador moderno interpreta código *JavaScript* integrado en las páginas Web). Por su parte, *XML* (del inglés *eXtensible Markup Language*) es un lenguaje estándar para la estructuración e intercambio de datos a través de la Web. Por lo general se usa para ofrecer información acerca de la estructura y el significado de datos.

AJAX corresponde a las siglas *Asynchronous JavaScript And XML*[3], es una tecnología que permite a una página Web actualizarse de forma dinámica sin la necesidad de recargarse completamente. *JavaScript* es el encargado de comunicarse con el servidor enviando y recibiendo datos desde la página Web en la cual se está trabajando, en el servidor la solicitud es procesada, y por último se envía una respuesta en formato *XML* (en la mayoría de los casos) que es interpretada de nuevo por *JavaScript* en la página Web.

El esquema de funcionamiento de una página Web con AJAX puede ser visto en la Figura 5.2.

5.1.3. JSON

JSON (*JavaScript Object Notation*), es un formato de texto más ligero y con notación más simple de objetos *JavaScript* [4].

El formato JSON se presenta de dos formas diferentes:

[3]http://www.uco.es/ lr1maalm/manualdeajax.pdf
[4]http://www.json.org/json-es.html

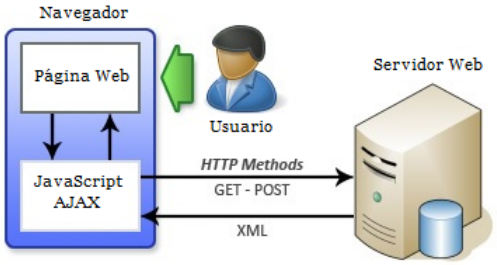


Figura 5.2: Modelo de página Web utilizando AJAX

- Un objeto (*object*), que es un conjunto desordenado de pares nombre/valor. Un objeto comienza con una llave de apertura y termina con una llave de cierre. Cada nombre es seguido por dos puntos (:) y los pares nombre/valor están separados por una coma.
- Un arreglo (*array*) es una colección de valores u objetos. Este comienza con un corchete izquierdo y termina con corchete derecho. Los valores u objetos son separados por una coma.

```
{"k":2,
"cantidad":7,
"xMax":8192,
"yMax":8192,
"puntos":[{"x":2482,"y":3052,"id":1,"correlative":1},
        {"x":1053,"y":1868,"id":2,"correlative":2},
        {"x":1956,"y":1,"id":3,"correlative":3},
        {"x":3198,"y":5388,"id":4,"correlative":4},
        {"x":249,"y":1923,"id":5,"correlative":5},
        {"x":1100,"y":1255,"id":6,"correlative":6},
        {"x":207,"y":1929,"id":7,"correlative":7}]
}
```

Figura 5.3: Arreglo en formato JSON utilizado por las aplicaciones

La Figura 5.3 muestra un ejemplo de la notación utilizada para almacenar los datos de una matriz de adyacencia y las coordenadas de sus puntos en formato JSON, este tipo de arreglos es utilizado por las aplicaciones para la construcción de las matrices de adyacencia.

5.1.4. Google Maps APIs

Una API (*Application Programming Interface*) es un conjunto de subrutinas, funciones o procedimientos (métodos en programación orientada a objetyos) ofrecidos por ciertas bibliotecas, para ser utilizadas por otros software.

El uso de las APIs ofrecidas por Google fueron fundamentales para este proyecto. Google Maps API nos otorga el acceso a los servicios de creación de mapas de Google dentro de nuestras aplicaciones, permitiendonos agregar marcadores personalizados, ventanas de información, polilíneas, entre otras caracteristicas.[5]

[5]https://enterprise.google.com/intl/es-419/maps/products/mapsapi.html

Google Places API nos permite acceder a las mismas bases de datos Google Maps y Google+, por lo que nos permite obtener información detallada de mas de 100 millones de sitios, negocios o puntos de interés de todo el mundo. Con esto pudimos incluir la búsqueda de puntos de interés (por tipo) en un área definida del mapa creado en nuestras aplicaciones.

Cabe señalar que, estas APIs son ofrecidas para diferentes plataformas, en este caso utilizamos *Google Maps Javascript API* y *Google Places API Web Service* para la aplicación Web, para la aplicación móvil utilizamos *Google Maps Android API* y *Google Places API for Android*.[6]

5.1.5. Sistema operativo Android

Si bien en la actualidad existe una diversa cantidad de sistemas operativos móviles, se escogió Android para almacenar la aplicación móvil de este proyecto, al ser éste el dominante actual del mercado[7], además de poseer un lenguaje (de nombre homónimo) de código abierto. En cuanto al entorno de desarrollo (*IDE*), utilizamos el software oficial para la plataforma Android llamado Android Studio (En su versión 3.0.1.), el cual trabaja en lenguaje *Java*. Para el desarrollo de la aplicación móvil, se contempló su funcionabilidad para versiones de Android 4.0.3 (Ice Cream Sandwich) en adelante. Esto ya que a partir de esta versión se asegura el acceso a prácticamente el cien por ciento de los usuarios del mercado android.

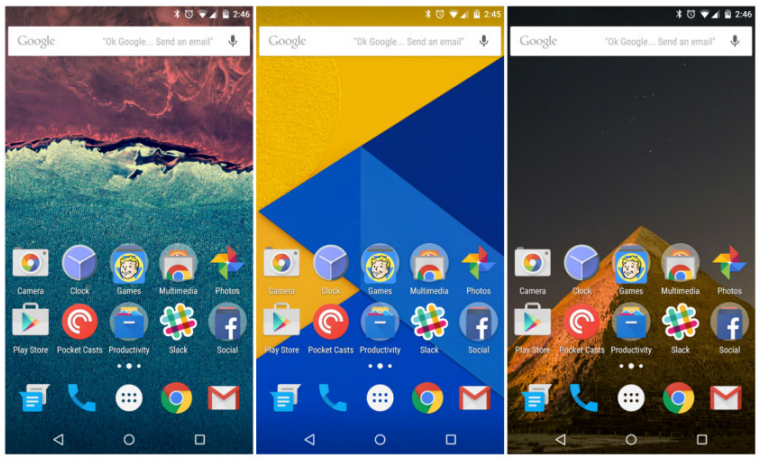


Figura 5.4: Interfaz para versiones de dispositivos móviles con Android 6.0

[6] https://developers.google.com/maps/?hl=es-419
[7] http://www.gartner.com/newsroom/id/3725117

5.2. Interfaz de las aplicaciones

Como se mencionó en el capítulo anterior, ambas aplicaciones desarrolladas poseen la misma arquitectura, por lo que las interfaces y el modo de uso para cada una de estas, es bastante similar. Tanto la aplicación Web como la aplicación Móvil, inician con la visualización de un mapa predeterminado, obtenido desde la API de Google Maps y centrado en la ubicación actual del usuario, sin embargo, dicho punto puede ser modificado según se requiera, para dar la libertad de realizar la consulta en cualquier parte del mapa. También se muestran los inputs necesarios con los que deberá interactuar el usuario para así relover la consulta propuesta. Estas interfaces pueden ser visualizadas en la Figura 5.5 para la aplicación Web y en la Figura 5.6 para la aplicación Móvil.

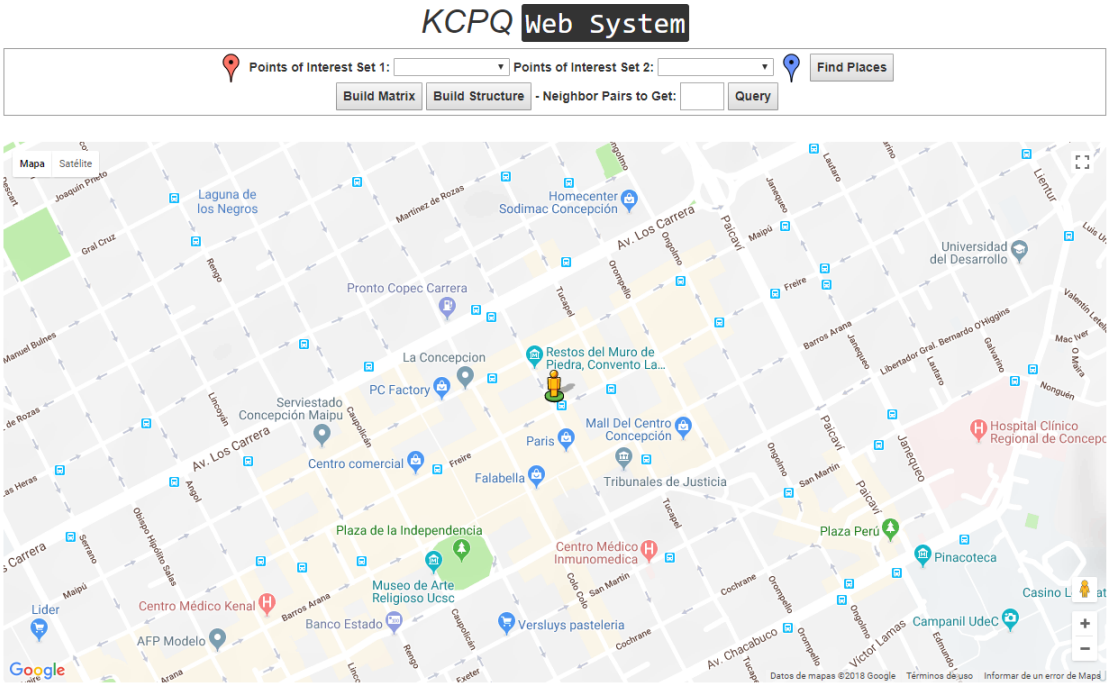


Figura 5.5: Pantalla principal del Sistema Web *KCPQ*

Una vez que el usuario fija la ubicación dentro del mapa, donde realizar la consulta, debe elegir los dos conjuntos de puntos de interés con los que va a trabajar, seleccionandolos por su tipo, mediante dos listas desplegables disponibles en la parte superior de la pantalla. Los puntos de interés seleccionados de la lista de la izquierda (Points of interest Set 1) se muestran en el mapa con marcadores de color rojo, por el contrario, los puntos de interés seleccionados de la lista derecha (Points of interest Set 2), se muestran con marcadores azules. Una vez seleccionados, el usuario debe presionar el botón "Find Places", el que desencadena la llamada a una serie de métodos que interactúan con la API de Google Places, la que devuelve dos arreglos (uno para cada conjunto) con la información de los sitios más cercanos a la posición seleccionada por el usuario dentro de un radio que defi-

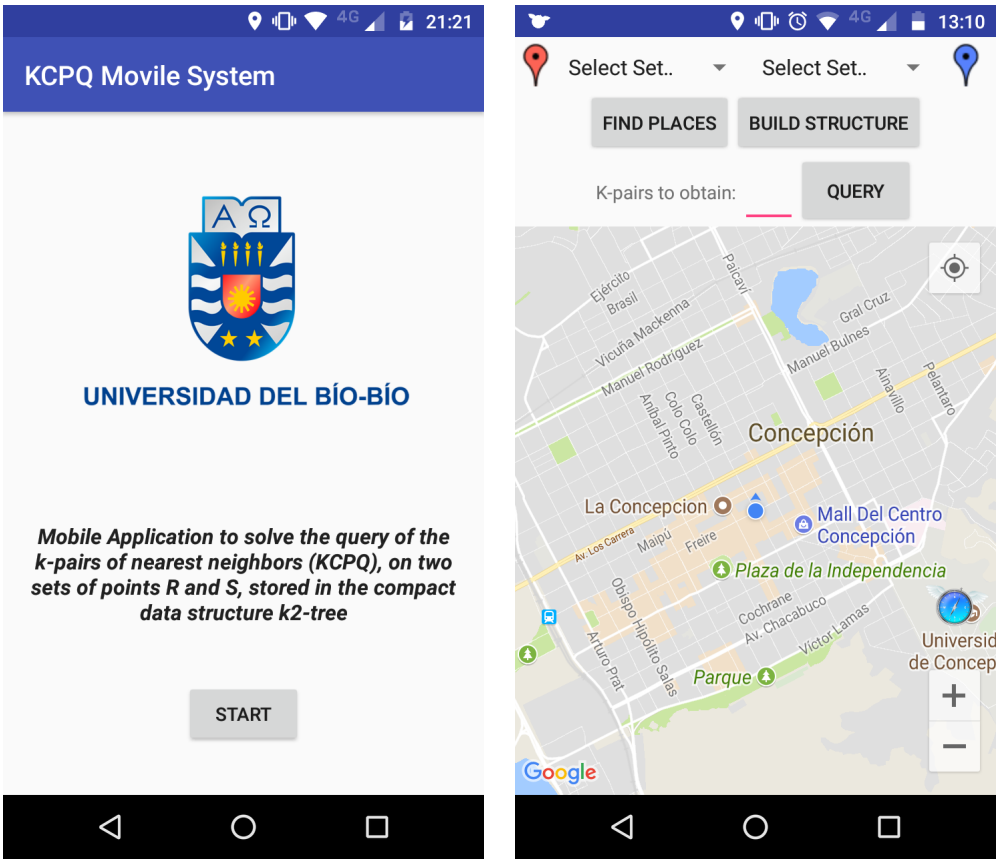


Figura 5.6: Pantallas principales del Sistema Móvil *KCPQ*

nimos en 600 metros, principalmente para evitar obtener matrices demasiado grandes que pueden provocar retrasos en los tiempos de ejecución. Estos arreglos, son representados gráficamente en el mapa con marcadores rojos y azules según corresponda. La Figura 5.7 muestra una ejemplo de búsqueda de los gimnasios y gasolineras más cercanas al punto del usuario, situado en el centro de la comuna de Concepción en su versión Web. Para versión Móvil, ver Figura 5.8.

A partir de los puntos de interés obtenidos, se construyen las matrices de adyacencia correspondientes a cada conjunto, las que son enviadas al Web Service implementado por (Santolaya, 2017), que se encarga de construir y almacenar las estructuras de datos compactas k^2-tree, sobre las cuales se realiza la consulta de los K-pares de vecinos más cercanos. Para esto el usuario debe presionar el boton "Build Structure"que llama a los métodos necesarios para realizar esta labor (Algoritmo 1).

Una vez enviadas las matrices al Web Service y construidos exitosamente los k^2-tree, las aplicaciones se encuentran en condiciones de mostrar la respuesta a la consulta *KCPQ*, sobre las estructuras. Para esto debe ser ingresada la cantidad de K-pares a obtener, la que es enviada al Web Service, que se encarga de aplicar el Algoritmo 2 sobre las estructuras

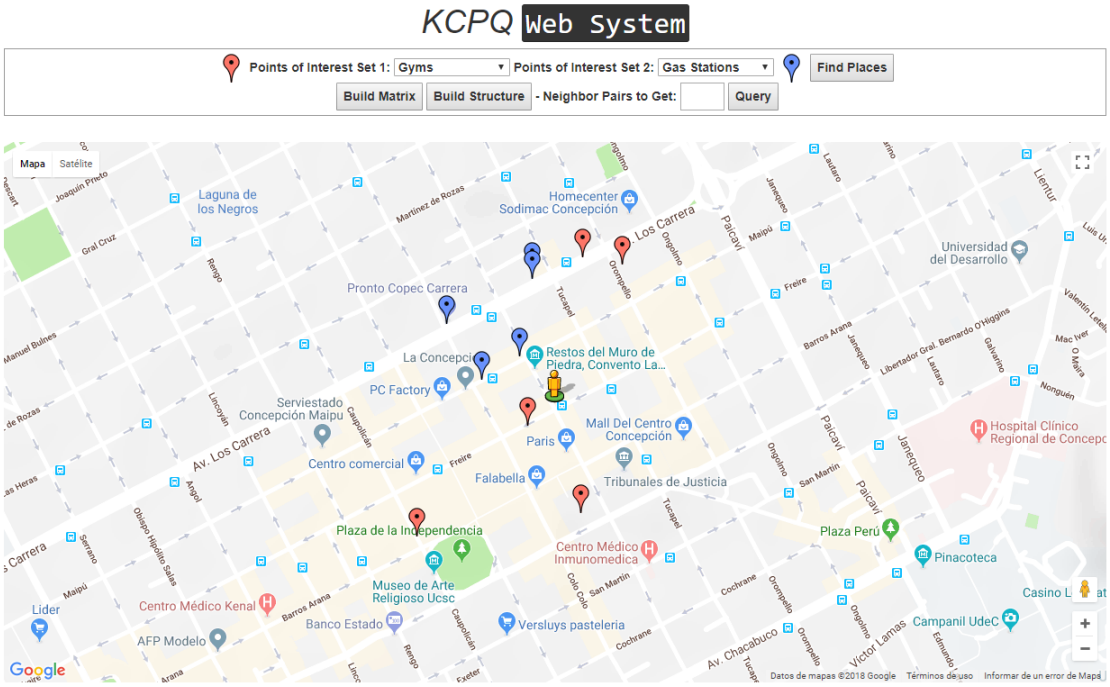


Figura 5.7: Búsqueda de los gimnasios y gasolineras más cercanas Aplicación Web

almacenadas, devolviendo a las aplicaciones, las coordenadas de los K-pares de vecinos más cercanos entre los dos conjuntos de puntos seleccionados por el usuario. Para mostrar esta respuesta de forma visual, los pares de puntos obtenidos son unidos con líneas (polylineas en datos espaciales) de color verde.

Ejemplo 5.1 Consideremos el mapa de la Figura 5.7 (Aplicación Web) y 5.8 (Aplicación Móvil) y un $K = 4$, lo que indica que el usuario desea obtener los 4 pares de vecinos más cercanos a su posición. La Figura 5.9 muestra los resultados obtenidos para la consulta *KCPQ* con $K = 4$ en la aplicación Web y la Figura 5.10 muestra el resultado de la consulta para la aplicación Móvil con el mismo valor de K. □

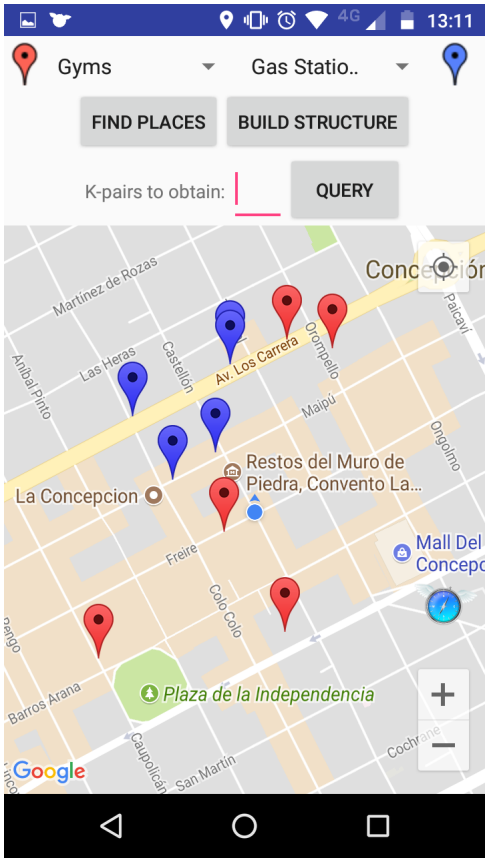


Figura 5.8: Búsqueda de los gimnasios y gasolineras más cercanas Aplicación Móvil

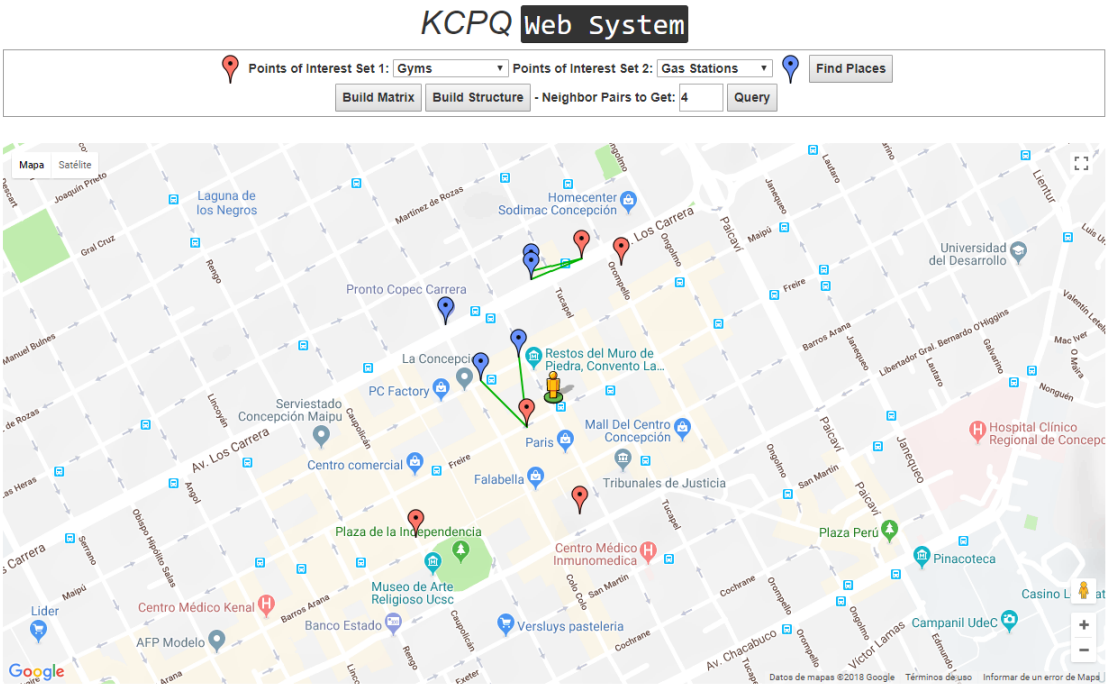


Figura 5.9: Resultado de la consulta *KCPQ* Aplicación Web

Figura 5.10: Resultado de la consulta *KCPQ* en Aplicación Móvil

Capítulo 6

Experimentación

Este capítulo comprende el hardware utilizado en los distintos dispositivos, tanto Web como Móvil, así como los resultados de la experimentación a partir de las pruebas aplicadas con diferentes parámetros, con el fin de comparar eficiencia en cuanto a tiempos de ejecución y espacio de almacenamiento.

6.1. Hardware

Para específicar el hardware a utilizar, se tomaron en cuenta distintos elementos a considerar: El tipo procesador o CPU que poseen los dispositivos, la cantidad y el tipo de memoria RAM, la tarjeta de video integrada, el espacio de almacenamiento disponible, y el sistema operativo utilizado. Estos elementos Estos elemento se visualizan a continuación. Para las pruebas realizadas a la aplicación Web, utilizamos un hardware con un procesador (CPU) Intel Core i5-8400 de 2.8 Ghz, memoria RAM de 8Gb DDR4 a 3000 Mhz, almacenamiento de 1 Tb HDD de 7200 RPM y sistema operativo Windows 10 de 64 bits. Para las pruebas realizadas a la aplicación móvil, utilizamos un dispositivo Android Motorola Moto G3, con un procesador (CPU) Snapdragon 410 de 1.4 Ghz, memoria RAM de 1 Gb, Almacenamiento de 8 Gb interno y sistema operativo Android 6.0.1 Marshmallow.

6.2. Descripción de los datos de prueba

Para realizar los experimentos utilizamos una mezcla de datos reales obtenidos por las aplicaciones y datos ficticios que fueron agregados manualmente y de forma arbitraria sobre éstas. Esto debido a que la única forma de controlar la cantidad de puntos a mostrar en el mapa es incrementando el radio de búsqueda *rd* que indicamos a la API de Google Places, la que devuelve todos los puntos del tipo solicitado dentro del radio. El problema de incrementar mucho el radio de búsqueda es que se nos escapa de las manos el control sobre el tamaño de las matrices de adyacencia que se generan a partir de los puntos obtenidos, esto es un problema a la hora de ubicar las posiciones de los puntos en sus respectivas celdas, ya que al tener matrices muy grandes (del orden de miles de filas y columnas), relentizan

el desempeño de las aplicaciones. Debido a esto decidimos fijar el radio de búsqueda en 600 metros, lo que nos permite encontrar conjuntos de puntos de interés dentro de un área aproximada de 1 kilómetro, distancia que consideramos más que aceptable para el tipo de consulta, en este caso *KCPQ*, que busca encontrar lugares cercanos a otros.

6.3. Pruebas para medir el espacio de almacenamiento

Para medir la eficiencia en cuanto al espacio de almacenamiento ocupado (expresado en bytes), se realizaron experimentos con diferentes cantidades de puntos de interés utilizados en las aplicaciones, comparando el tamaño de éstos en el formato devuelto por las APIs de Google Places (formato llamado *Marker*), frente al tamaño de los mismos puntos transformados en los arreglos de puntos P y T a partir del Algoritmo 1, correspondiente a los puntos ubicados en sus respectivas matrices de adyacencia.

Como la ubicación de los puntos y el tiempo de construcción de las matrices es irrelevante al momento de realizar las pruebas de almacenamiento, estas se realizaron utilizando puntos en un escenario real de la ciudad de Concepción, en el que se aumentó el radio de búsqueda de puntos a 5000 metros, obteniendo como resultado al rededor de 120 puntos de interés reales, los que fueron procesados y almacenados a sus respectivas matrices de adyacencia. Se realizó la comparación teniendo en cuenta la cantidad de puntos obtenidos n, desde $n = 20$ hasta $n = 120$. Cabe señalar que, al tratarse del mismo formato en que estos son guardados, estas pruebas son válidas tanto para la Aplicación Web como para la Aplicación Móvil. Los resultados se muestran en la Tabla 6.1 y en la Figura 6.1 se presenta su correspondiente gráfico.

Eficiencia en espacio de almacenamiento		
n	Puntos en formato *Marker*	Puntos en P y T para matrices de adyacencia
20	6552 bytes	234 bytes
40	13132 bytes	469 bytes
60	19684 bytes	703 bytes
80	26236 bytes	937 bytes
100	32816 bytes	1172 bytes
120	39396 bytes	1407 bytes

Tabla 6.1: Espacio de almacenamiento utilizado por puntos en formato *Marker* respecto al formato usado en los arreglos P y T de las matrices de adyacencia (coordenadas cartesianas)

De estas pruebas pudimos observar que, el espacio de almacenamiento que utilizan los puntos de las matrices de adyacencia almacenados en P y T es aproximadamente 28 veces menor al espacio que ocupan los mismos puntos en su formato de origen (formato *Marker* con el que trabajan las APIs de Google Maps), demostrando la gran ventaja en cuanto ahorro de espacio de almacenamiento que supone el uso de estructuras de datos compactas y en concreto la estructura de datos compacta k^2-tree. Cabe mencionar que el tamaño de los puntos en formato *Marker* pudo ser ligeramente menor, debido a que éstos, además de

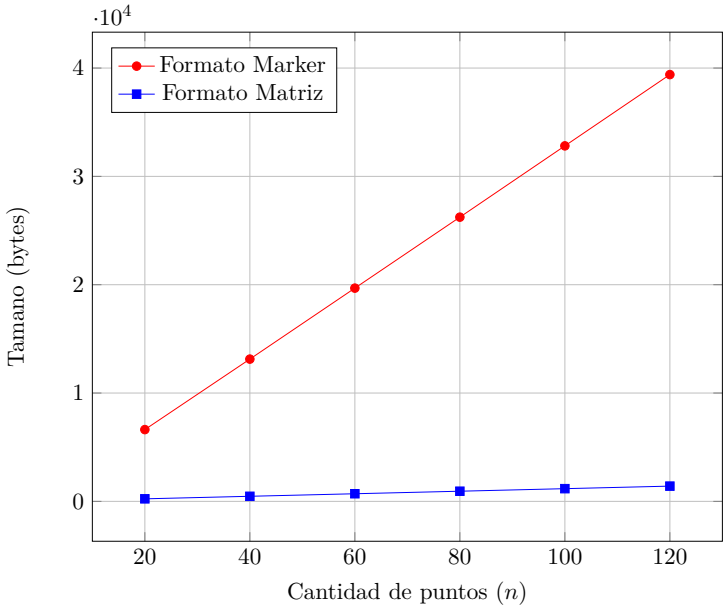


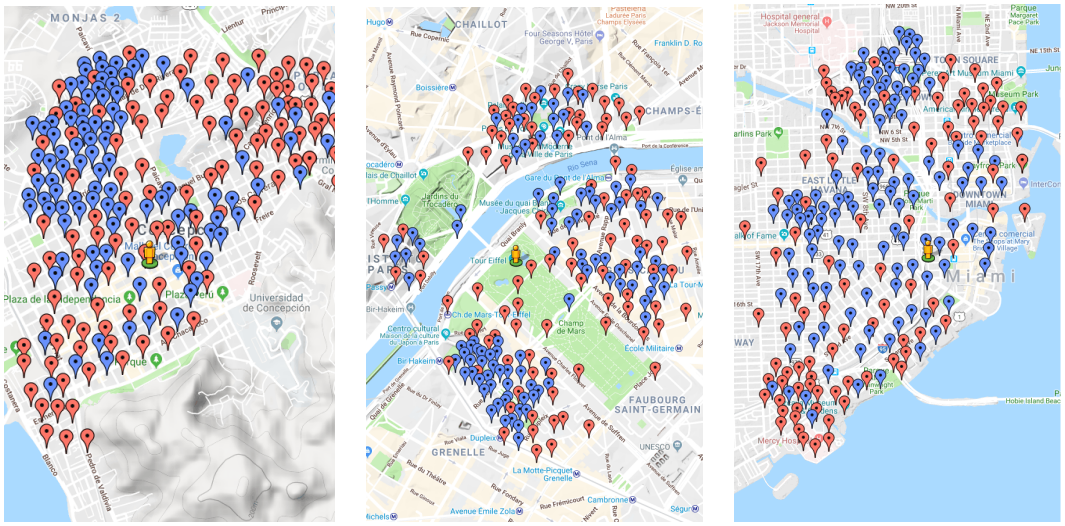
Figura 6.1: Gráfico de espacio de almacenamiento de puntos en formato *Marker* respecto al formato utilizado en matrices de adyacencia

contener las coordenadas (latitudes y longitudes) contienen otros valores de utilidad, como el nombre del sitio, dirección, entre otros, mientras que el formato del arreglo de puntos P y T de las matrices de adyacencia solo contienen las coordenadas cartesianas de sus ubicaciones representadas en dichas matrices.

6.4. Experimentación para medir tiempos de ejecución

Para medir los tiempos de ejecución de los sistemas Web y Móvil se realizaron diferentes pruebas, las que estan divididas en dos partes. (i) Tiempos de ejecución de los algoritmos encargados de la construcción de matrices de adyacencia (Algoritmo 1). (ii) Tiempo de ejecución del algoritmo *KCPQ* sobre dos estructuras de datos compactas k^2-tree ya construídas (Algoritmo 2). Para cada una de las pruebas se consideraron diferentes cantidades de puntos de interés a procesar y tres distintas ciudades del mundo en áreas donde existe gran aglomeración de puntos. Las ciudades en donde se realizaron las pruebas fueron: Concepción (Chile), París (Francia) y Miami (EEUU). En cuanto a los puntos utilizados, se usaron conjuntos de puntos de interés reales obtenidos desde la API de Google Places, los que fueron rellenados con puntos ficticios para así poder realizar pruebas cada vez con una mayor cantidad de puntos y mantener al mismo tiempo cierto grado de realismo, puesto que al construir las matrices para representar a los puntos, las distancias que existen entre estos influye directamente en los tiempos de ejecución, ya de la distancias entre estos depende directamente el tamaño de la matriz a generar. Los conjuntos de puntos de prueba

se visualizan en la Figura 6.2(a), Figura 6.2(b) y Figura 6.2(c) para Concepción, París y Miami respectivamente, donde se probó con un máximo de 100 puntos por conjunto en cada una de las ciudades, obteniendo un máximo de 200 puntos a evaluar.



(a) Concepción (b) París (c) Miami

Figura 6.2: Conjuntos de puntos de prueba para medir tiempos de ejecución

6.4.1. Tiempos de ejecución para la construcción de matrices de adyacencia

En estos experimentos se busca medir cuanto tardan en promedio los sistemas en generar los conjuntos de P y T puntos ubicados en sus matrices de adyacencia, respecto a una cantidad de puntos determinada. Para esto se realizaron varios experimentos en los que se construyeron matrices de adyacencia con n puntos de forma incremental, desde $n = 10$ hasta $n = 100$, además de distintas ubicaciones de puntos, para cada una de las ciudades mencionadas anteriormente. Los resultados de las pruebas aplicadas a la aplicación Web se muestran a continuación en la Tabla 6.2, junto con su respectivo gráfico en la Figura 6.3.

En la Tabla 6.2 y Figura 6.3 se puede observar que los tiempos de ejecución obtenidos en cada prueba son muy similares para cada ciudad analizada, la diferencia entre los tiempos de cada una se debe principalmente a la posición de los puntos dentro del mapa. Esto se debe al tamaño de las matrices de adyacencia obtenidas a partir de los parámetros otorgados por el Algoritmo 1, el algoritmo obtiene este valor a partir de la menor distancia entre las latitudes o longitudes entre dos puntos, para saber el tamaño de las cuadrículas, y de la distancia entre los puntos más alejados entre sí (mayor distancia), para así obtener la cantidad de cuadrículas para las filas y columnas de la matriz de adyacencia (*cuad*).

En la Tabla 6.5 se muestran los resultados de las pruebas aplicadas a la aplicación Móvil y en la Figura 6.4 se presenta su respectivo gráfico. Cabe destacar que por las condiciones

n	Concepción	París	Miami
10	96 ms	31 ms	52 ms
20	97 ms	91 ms	94 ms
30	103 ms	109 ms	103 ms
40	675 ms	785 ms	771 ms
50	8725 ms	5145 ms	7026 ms
60	35007 ms	22060 ms	27003 ms
70	57642 ms	51984 ms	53287 ms
80	66309 ms	59351 ms	65122 ms
90	83017 ms	78176 ms	81083 ms
100	106798 ms	94166 ms	97102 ms

Tabla 6.2: Tiempos de ejecución para la construcción de matrices de adyacencia - Aplicación Web

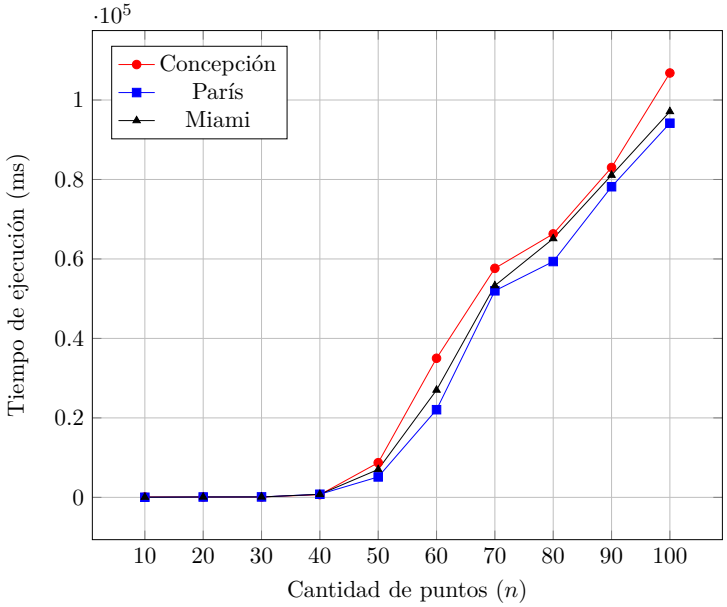


Figura 6.3: Gráfico de tiempos de ejecución para la construcción de matrices de adyacencia - Aplicación Web

del hardware del dispositivo Móvil (que es mucho menos potente que un computador) los tiempos de ejecución fueron mayores en comparación a la pruebas realizadas en la aplicación Web, debido a esto no se pudieron realizar todas las pruebas en relación a la cantidad de puntos utilizados en la aplicación Web, por lo que solo se muestran con un máximo de $n = 50$. Es necesario destacar que, los valores obtenidos son un promedio de 8 pruebas realizadas, utilizando distintas ubicaciones de puntos dentro de una misma área,

n	Concepción	París	Miami
10	1249 ms	1340 ms	1347 ms
20	4685 ms	4206 ms	4385 ms
30	22046 ms	15021 ms	17612 ms
40	92485 ms	80468 ms	99430 ms
50	130364 ms	118707 ms	121258 ms

Tabla 6.3: Tiempos de ejecución para la construcción de matrices de adyacencia - Aplicación Móvil

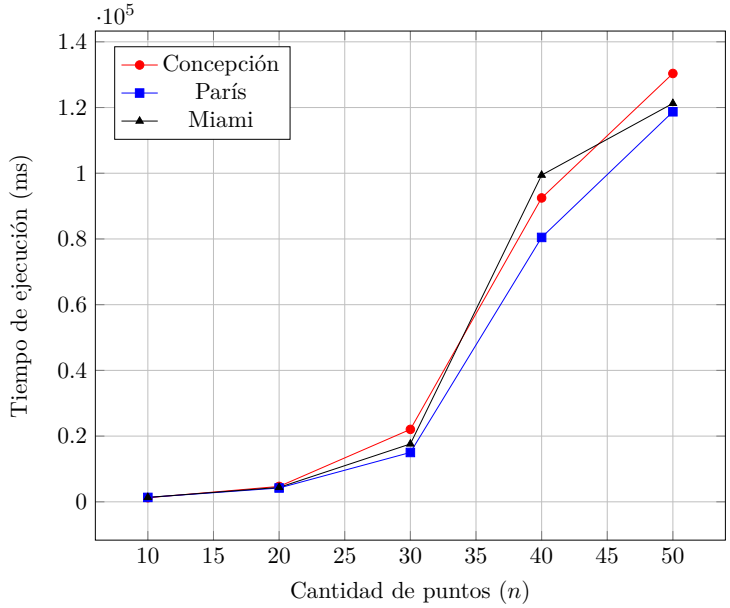


Figura 6.4: Gráfico de tiempos de ejecución para la construcción de matrices de adyacencia - Aplicación Móvil

que en este caso son las ciudades Concepción, París y Miami. Estas pruebas son el resultado de la construcción de una sola matriz de adyacencia con diferentes cantidades de puntos n, como en este proyecto buscamos almacenar dos conjuntos de puntos de interés en dos estructuras de datos compactas k^2-tree, los tiempos de ejecución finales dependerán de la cantidad de puntos del primer conjunto sumada a la cantidad de puntos del segundo. Por ejemplo, si tenemos un conjunto con 10 puntos y otro con 20 puntos ubicados en la ciudad de Miami, el tiempo total de construcción de ambas matrices sería de aproximadamente unos 146 milisegundos (52 ms + 94 ms), según la Tabla 6.2. Lo mismo para el caso de la aplicación Móvil.

6.4.2. Tiempos de ejecución para obtener los *K*-pares de puntos más cercanos *KCPQ*

Con estas pruebas buscamos medir cuanto tardan en promedio los sistemas Web y Móvil en consultar al algoritmo *KCPQ* (Algoritmo 2) que retorna los K-pares de puntos más cercanos entre dos conjuntos de puntos, que se encuentran almacenados en dos estructuras de datos compactas k^2-tree. Para esto utilizamos los mismos datos de prueba de la subsección anterior (ver Figura 6.2), y evaluamos el desempeño de esta consulta con el mismo número de puntos n incremental, desde $n = 10$ hasta $n = 100$ para el caso de la aplicación Web, en el caso de la aplicación Móvil evaluamos hasta un máximo de $n = 50$. La constante k (potencia de 2), como se mencionó en el Capítulo 3, es de 2 para la cantidad de subdivisiones realizadas a la estructura compacta k^2-tree. La cantidad de pares de puntos a obtener que contemplamos para estas pruebas es de $K = 10$, también probamos con otros valores, desde $K = 10$ hasta $K = 100$, donde nos dimos cuenta que los resultados variaron muy poco, esto debido a que las cantidades mencionadas son ínfimas con respecto a la real capacidad del algoritmo *KCPQ* trabajando sobre estructuras de datos compactas k^2-tree, donde a partir de miles de datos se comenzarían a ver diferencias notorias (ver Tabla 6.4).

Tiempos de ejecución Algoritmo *KCPQ*	
K	Tiempo de ejecución (ms)
10	38 ms
20	39 ms
30	42 ms
40	42 ms
50	44 ms
60	43 ms
70	67 ms
80	73 ms
90	70 ms
100	74 ms

Tabla 6.4: Tiempos de ejecución para obtener los K-pares de vecinos más cercanos con $n = 200$

En la Tabla 6.4 se visualizan los resultados de las pruebas realizadas al algoritmo *KCPQ* para diferentes valores de K, sobre sets de datos con $n = 200$ puntos (100 por cada conjunto) y en la Figura 6.5 se muestra su correspondiente gráfico. Los resultados arrojados son los promedios de 10 experimentos realizados para cada valor de K incremental (desde $K = 10$ hasta $K = 100$), donde los tiempos fluctuan entre los 38 y 118 milisegundos.

Para las siguientes pruebas consideramos un valor de $K = 10$ en cada una de las pruebas, ya que es poco probable que se busquen más de 10 lugares cercanos a otros en un escenario real.

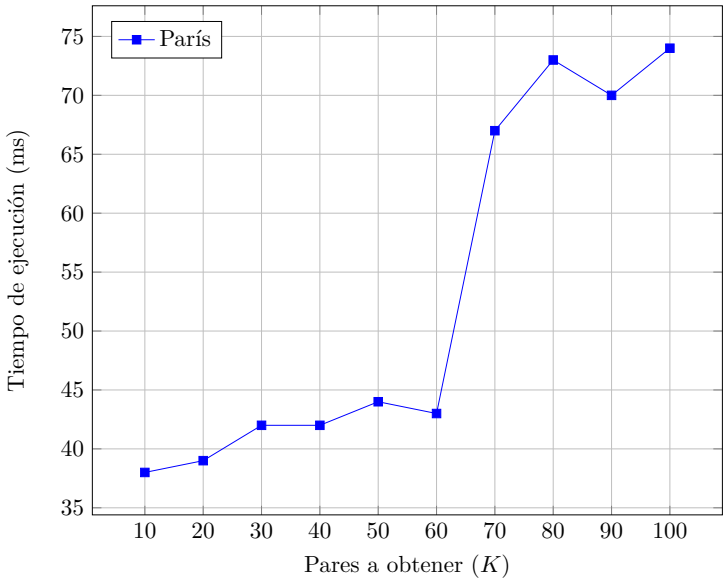


Figura 6.5: Gráfico de tiempos de ejecución para obtener los K-pares de vecinos más cercanos con $n = 200$

Tiempos de Ejecución para la Aplicación Web

Los resultados de estas pruebas, se muestran a continuación en la Tabla 6.5 junto con su respectivo gráfico en la Figura 6.6.

Tiempos de ejecución Algoritmo *KCPQ*			
n	Concepción	París	Miami
10	50 ms	52 ms	56 ms
20	58 ms	57 ms	60 ms
30	66 ms	67 ms	67 ms
40	71 ms	75 ms	73 ms
50	85 ms	88 ms	85 ms
60	94 ms	91 ms	96 ms
70	105 ms	98 ms	97 ms
80	120 ms	132 ms	125 ms
90	126 ms	135 ms	135 ms
100	137 ms	142 ms	141 ms

Tabla 6.5: Tiempos de ejecución para obtener los $K = 10$ pares de puntos más cercanos - Aplicación Web

Como podemos apreciar, los tiempos de ejecución para obtener los $K = 10$ pares de puntos más cercanos en la aplicación Web son muy similares para las mismas cantidades

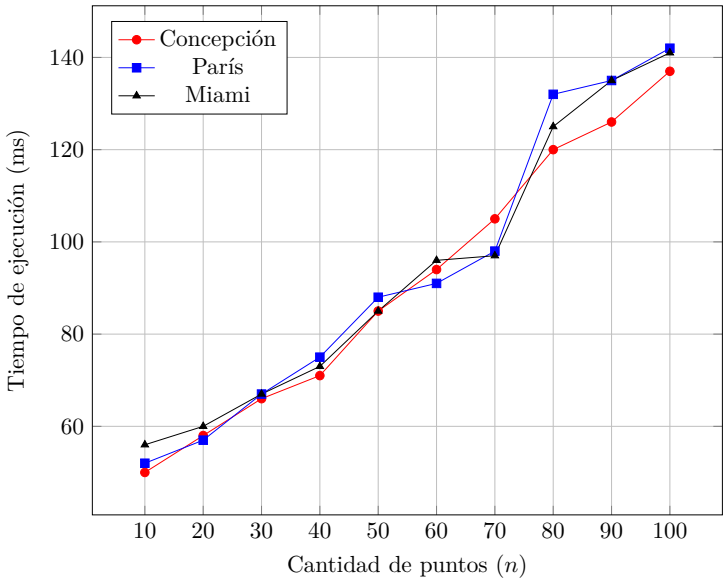


Figura 6.6: Gráfico de tiempos de ejecución para obtener los $K = 10$ pares de puntos más cercanos - Aplicación Web

de puntos y para las tres ciudades en las que se aplicó la consulta, donde los tiempos fluctuan entre los 50 y 56 milisegundos para la menor cantidad de puntos $n = 10$, y entre los 141 y 142 milisegundos para la mayor cantidad de puntos $n = 100$. Como se mencionó anteriormente, las diferencias entre los tiempos de ejecución para distintos valores de K y ahora para las diferentes cantidades de puntos n con que se probó el algoritmo, refuerza el hecho de que estos valores (n y K) son ínfimos en comparación a la real capacidad del algoritmo al aplicarlo sobre la estructura compacta, por lo que para poder medir realmente la eficiencia de este es necesario trabajar sobre una gran cantidad de datos (del orden de miles).

Tiempos de Ejecución para la Aplicación Móvil

Los resultados de las pruebas realizadas en la aplicación Móvil, se muestran a continuación en la Tabla 6.6 junto con su respectivo gráfico en la Figura 6.7.

Como podemos observar, los tiempos de ejecución para la consulta *KCPQ* en la aplicación Móvil, si bien son un relativamente mayores con respecto a los de la aplicación Web, siguen la misma línea en cuando a la similitud de los tiempos para las tres ciudades evaluadas con la misma cantidad de puntos. En este caso los tiempos de ejecución fluctuan entre los 63 y 64 milisegundos para la menor cantidad de puntos $n = 10$, y entre los 103 y 109 milisegundos para la mayor cantidad de puntos $n = 50$.

Tiempos de ejecución Algoritmo *KCPQ*			
n	Concepción	París	Miami
10	64 ms	63 ms	63 ms
20	72 ms	68 ms	69 ms
30	86 ms	81 ms	81 ms
40	91 ms	90 ms	91 ms
50	109 ms	105 ms	103 ms

Tabla 6.6: Tiempos de ejecución para obtener los $K = 10$ pares de puntos más cercanos - Aplicación Móvil

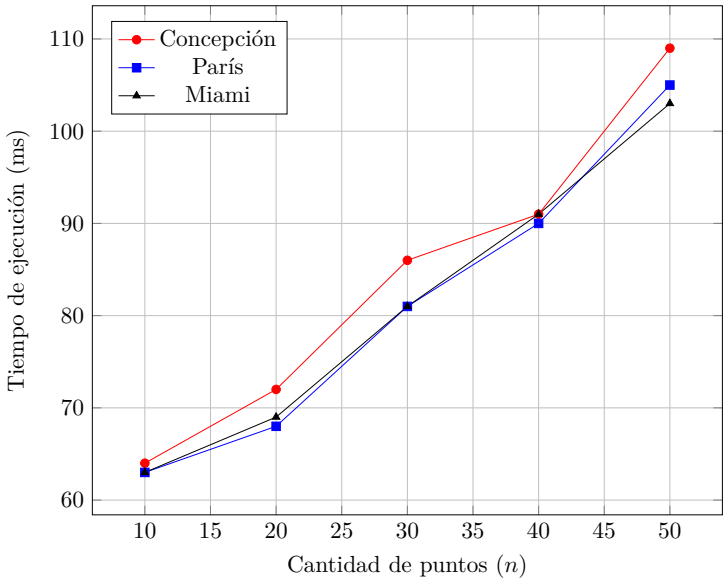


Figura 6.7: Gráfico de tiempos de ejecución para obtener los $K = 10$ pares de puntos más cercanos - Aplicación Móvil

Capítulo 7

Conclusiones y Trabajos Futuros

En este proyecto se presentaron dos aplicaciones, en sus versiones Web y Móvil (la última para la plataforma Android) que permiten resolver y visualizar de forma gráfica, la consulta de proximidad espacial de los K-pares de vecinos más cercanos ($KCPQ$) entre dos conjuntos de puntos R y S, que representan lugares de interés del mundo real, los que son almacenados en dos estructuras de datos compactas k^2-tree, sobre las cuales se procesa la consulta. Estas aplicaciónes fueron llevadas a un escenario global, en el que se obtienen conjuntos de puntos de interés que son a elección del usuario de forma autónoma desde un mapa, esto fue posible mediante el uso de las APIs de Google Maps, los conjuntos de puntos son adaptados, para así poder almacenarlos en las estructuras de datos compacta k^2-tree.

Mediante la experimentación podemos concluir la gran ventaja que supone el uso de estructuras de datos compactas y en particular de la estructura de datos compacta k^2-tree que es en la que nos enfocamos, donde nos dimos cuenta de la gran ventaja en cuanto a ahorro de espacio de almacenamiento y tiempos de ejecución al poder realizar consultas sobre esta sin tener que descomprimir los datos. A través de la experimentación también concluimos que los tiempos de ejecución al traspasar los datos, en este caso los conjuntos de puntos, desde el mapa a las matrices de adyacencia necesarias para la construcción de las estructuras de datos compactas, depende mucho de la ubicación de los puntos. Esto se debe a que, para la creación de la estructura de datos compacta k^2-tree a partir de una matriz, se necesita disponer de la distancia más corta y la más lejana entre los puntos, para así calcular el tamaño de cada celda de la matriz y la cantidad de filas y columnas. Por ejemplo, es muy diferente tener 10 puntos ubicados a una distancia mínima de 100 kilómetros y máxima de 1000 kilómetros entre sí, a tener 10 puntos ubicados a una distancia mínima de 1 kilómetro y máxima de 1000 kilómetros. La matriz de adyacencia a utilizar para el primer caso sería de 10×10 con 100 cuadrículas a analizar, mientras que para el segundo caso, se necesitaría una matriz de 1000×1000 con 1000000 cuadrículas analizar (Maldonado, 2017). Los tiempos de ejecución para la construcción de matrices, en casos como el segundo, aumenta drásticamente. Como solución a este problema se restringió el rango de búsqueda de puntos de las aplicaciones a 600 metros, con respecto a la posición actual del usuario. Para el tipo de consulta ($KCPQ$) esto no es un mal rango, puesto que

la idea es encontrar puntos cercanos a otros.

Con la realización de este proyecto, además pudimos demostrar que el uso de estructuras de datos compactas como el k^2-tree, pueden ser utilizadas sin ningún problema en escenarios prácticos, como el de consultas de proximidad espacial sobre datos de tipo punto. En este sentido proponemos como trabajo futuro el uso de otra estructura de datos compacta para resolver la consulta *KCPQ*, tal como la estructura de datos compacta *Wavelet tree* (Navarro, 2014) y evaluar su desempeño con respecto al k^2-tree.

Referencias

Nieves Brisaboa, Guillermo Bernardo, Roberto Konow, y Gonzalo Navarro. K2-treaps: Range top-k queries in compact space. En *Proceedings of the 21st International Symposium on String Processing and Information Retrieval*, SPIRE 2014, págs. 215–226. 2014.

Nieves Brisaboa, Susana Ladra, y Gonzalo Navarro. k2-trees for compact web graph representation. págs. 18–30. 2009.

Antonio Fariña, Susana Ladra, Oscar Pedreira, y Ángeles S. Places. Rank and select for succinct data structures. *Electronic Notes Theoretical Computer Science*, 236:131–145, 2009.

Rodrigo González, Szymon Grabowski, Veli Mäkinen, y Gonzalo Navarro. Practical implementation of rank and select queries. En *In Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms*, págs. 27–38. 2005.

Ralf Güting. An introduction to spatial database systems. *The VLDB Journal*, 3(4):357–399, 1994.

Ignacio Maldonado. Aplicación web y móvil para implementar la consulta de los k vecinos más cercanos en el contexto de centros de salud de la provincia de concepción, sobre datos espaciales almacenados en la estructura compacta k2-tree. 2017.

Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

Fernando Santolaya. Procesamiento de consultas de proximidad espacial sobre datos almacenados en la estructura de datos compacta k2-tree. 2017.

Shashi Shelkhar y Sanjay. Chawla. *Spatial Databases a Tour*. Prentice Hall, 2013.

Cristian Vallejos, Mónica Caniupán, y Gilberto Gutiérrez. K2-treaps to represent and query data warehouses into main memory. En *Proceedings of the 36th International Conference of the Chilean Computer Society*, SCCC 2017, págs.–. 2017.