75.04/95.12 Algoritmos y Programación II
Trabajo práctico 1: algoritmos y estructuras de datos

Universidad de Buenos Aires - FIUBA
Segundo cuatrimestre de 2017

1. Objetivos

Ejercitar conceptos relacionados con estructuras de datos, diseño y análisis de algoritmos. Escribir un programa en C++ (y su correspondiente documentación) que resuelva el problema que presentaremos más abajo.

2. Alcance

Este trabajo práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

3. Requisitos

El trabajo deberá será entregado personalmente, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes, un informe impreso de acuerdo con lo que mencionaremos en la sección 5, y con una copia digital de los archivos fuente necesarios para compilar el trabajo.

4. Descripción

El comandante Kane se ha mostrado sumamente satisfecho por las pruebas de concepto que le fueron entregadas. De esta forma, superada la etapa exploratoria en nuestra colaboración con la hermandad de Nod, nuestra nueva y más ambiciosa misión es la de desarrolar un software más eficiente que le permita a Kane y sus dirigidos reducir de forma drástica los tiempos de consulta.

El propósito de este trabajo, entonces, es el de analizar e implementar estructuras de datos eficientes para resolver el mismo problema introducido en la instancia anterior. En definitiva, los programas del TP 1 deberán comportarse exactamente igual que los del TP 0 desde la perspectiva de un usuario, recibiendo una base de datos de coordenadas de reservas de tiberium y luego procesando una cantidad arbitraria de consultas sobre potenciales coordenadas de bases con el objetivo de encontrar la coordenada de la reserva que esté más próxima (tomando como métrica la distancia euclideana). A diferencia del caso anterior, y por urgencias impostergables, el comandante Kane solicitó explícitamente enfocar el problema en el plano y no en dimensiones arbitrariamente grandes, aunque prometió un beneficio adicional si también podemos ofrecerle soporte para consultas en el espacio.

La problemática será abordada en esta instancia a través de la implementación de árboles k-d y tres heurísticas de división, que pasaremos a detallar en la siguiente sección. Además de implementar lo solicitado, cada grupo deberá:

- Calcular la complejidad total de resolver en el TP 0 k consultas frente a una base de datos de n puntos y contrastarla frente la complejidad de resolver la misma cantidad de consultas utilizando

un árbol k-d. Para esto último, tomar tanto una estimación de peor caso como otra promedio, asumiendo una buena distribución de los datos. Tener en cuenta que no es necesario un argumento formal para justificar la complejidad promedio.

- Comparar empíricamente las tres heurísticas de división, sugiriendo escenarios favorables y desfavorables para cada una de ellas y discutiendo cómo impacta esto en la complejidad de las operaciones.

4.1. Árboles k-d

Un **árbol k-d**[1] es una estructura de datos que permite agrupar y organizar puntos en un espacio k-dimensional de forma tal de soportar diversas operaciones sobre ellos de manera eficiente, como por ejemplo el conteo de puntos en cierto rango o la búsqueda (exacta o aproximada) de vecinos cercanos a un punto dado. Justamente, en este trabajo práctico nos apoyaremos en esta estructura para poder resolver consultas de vecino más cercano sin necesidad de recorrer por completo toda la base de datos de coordenadas por cada consulta que se desee resolver.

De forma más precisa, un árbol k-d es un árbol binario en el que cada nodo interno indica la división del espacio en dos porciones mediante un hiperplano fijado en cierta coordenada. Los puntos propiamente dichos quedan almacenados en las hojas del árbol. Al trabajar en el plano, por ejemplo, cada nodo interno dividirá el espacio en dos partes por medio de una recta fijada en la coordenada x o en la coordenada y. Idealmente, se desea computar sucesivamente divisiones en partes iguales con el objeto de construir un árbol perfectamente balanceado, aunque esto no suele ser siempre posible. Al momento de hacer esta partición, se utiliza una heurística de *splitting* (o división) que indica cómo agrupar los puntos de uno u otro lado del hiperplano.

4.1.1. Armado

Consideremos un conjunto de puntos S que pretendemos organizar en forma de árbol k-d. De lo introducido más arriba, surgen distintos interrogantes, entre los que se destacan cómo tomar la decisión de qué coordenada utilizar para las divisiones y cuándo se decide parar la división para agrupar los puntos de cierto subespacio en una hoja del árbol. En lo que sigue delinearemos el algoritmo de armado y responderemos estos interrogantes:

- Comenzando por la primera coordenada, x_1, tomar la heurística de splitting H que se desea utilizar y computar $x = H(S, x_1)$. Este valor será usado para particionar S en dos subconjuntos S_1 y S_2 de forma tal que todo $s \in S_1$ verifique $s[x_1] \leq x$ y todo $s \in S_2$ verifique $s[x_1] > x$ (notando $p[x_i]$ a la proyección de la i-ésima coordenada del punto p).
- Generar un nuevo nodo interno y almacenar en él x_1, x y la *región* de S: una caja que engloba de la forma más justa posible los puntos de S (observar la Fig. 1 para un ejemplo en el plano).
- Si $|S_1| > N$, siendo N una constante predefinida que indica la máxima cantidad de puntos permitida para un nodo hoja, construir recursivamente un árbol k-d sobre S_1 que ubicaremos como subárbol izquierdo de nuestro nodo. En este caso, no obstante, utilizar la siguiente coordenada, x_2, para hacer la división –más generalmente, ciclaremos coordenada tras coordenada en cada llamada recursiva.
- Si $|S_1| \leq N$, generar un nodo hoja con los puntos de S_1 y ponerlo como subárbol izquierdo.
- Repetir los pasos anteriores para S_2 (usando también x_2 como coordenada de división puesto que está al mismo nivel que S_1) y poner el resultado como subárbol derecho.

La Fig. 1 muestra un conjunto de puntos en el plano junto con las sucesivas divisiones hechas al construir un árbol k-d sobre ellos. Por otro lado, en la Fig. 2 se puede ver la estructura final del árbol para este mismo conjunto de puntos.

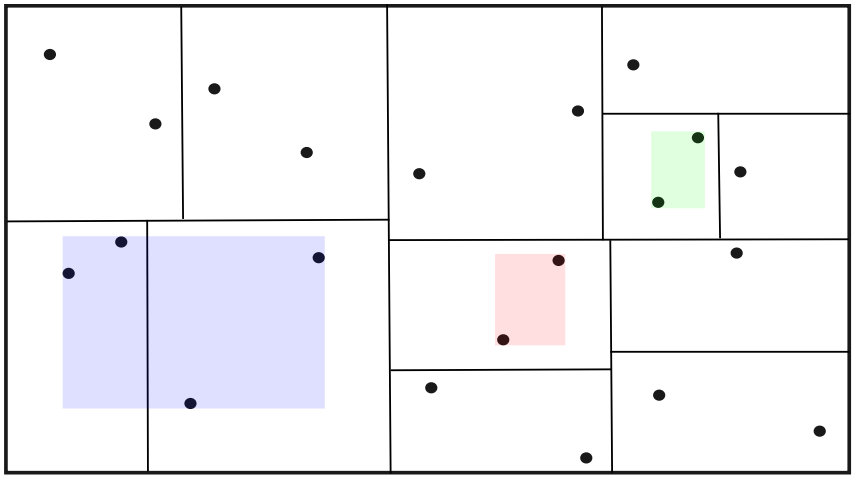

Figura 1: Espacio sucesivamente dividido y tres regiones de puntos destacadas

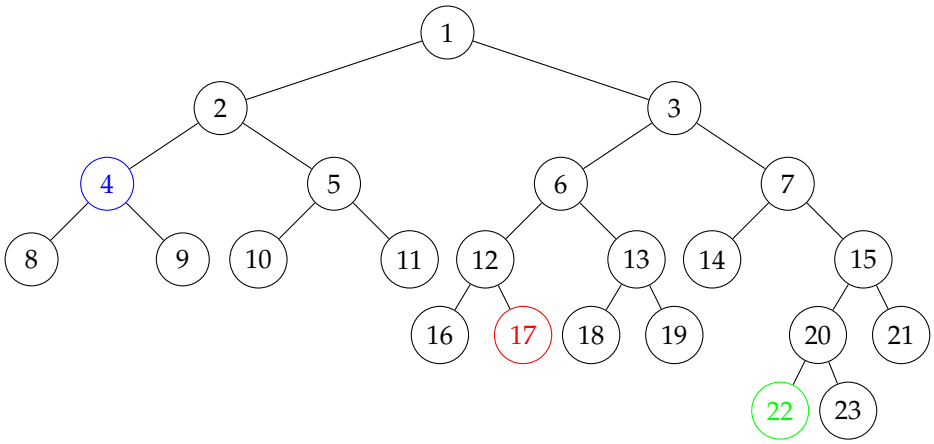

Figura 2: Estructura del árbol k-d para el plano de la Fig. 1

4.1.2. Heurísticas de splitting

En este trabajo práctico aplicaremos una de tres heurísticas posibles de división, configurable por línea de comando. Para ilustrar cada una con un ejemplo concreto, utilizaremos el conjunto de puntos $S = \{(2,8),(4,1),(50,3),(10,10),(5,-5)\}$.

- **Mediana**, que consiste en computar la mediana del conjunto de datos a través del algoritmo QuickSelect[3, 2][1]. En este caso, $H(S, x_1) = 5$ puesto que la mediana de las primeras coordenadas de los puntos es 5.
- **Mitad**, que se basa en tomar el rango completo de los datos y calcular su punto medio exacto. Usando esta heurística, $H(S, x_1) = 26$ siendo 26 el punto medio entre 2 y 50, los extremos del rango.
- **Promedio**, que toma tres puntos aleatorios del conjunto y promedia sus coordenadas. Para el ejemplo dado, asumiendo que se eligen los tres primeros puntos de S, $H(S, x_1) = \frac{56}{3}$.

[1]Este algoritmo será brevemente explicado en el pizarrón durante la clase de presentación del trabajo práctico.

4.1.3. Vecino más cercano

Una vez construido el árbol, podremos resolver una cantidad arbitraria de consultas de proximidad sin necesidad de recorrer íntegramente la base de datos cada vez (a excepción, en verdad, de situaciones sumamente desafortunadas). Dado un punto de consulta q, el algoritmo para encontrar el punto de S más próximo a q es el siguiente:

- Encontrar en el árbol k-d de S la hoja que mejor se ajusta a q. Para esto, deberemos comenzar en la raíz e ir descendiendo según lo indicado por la comparación entre los valores de las coordenadas de q y los valores computados por H al armar el árbol. Por ejemplo, si en la raíz tenemos almacenado el valor x para la coordenada x_1, descenderemos al subárbol izquierdo si $q[x_1] \leq x$ y al subárbol derecho en caso contrario.
- Una vez en dicha hoja, computar las distancias $d_1, \ldots, d_n$ de q a cada punto $p_1, \ldots, p_n$, $1 \leq n \leq N$, y guardar en d la mínima de ellas y en p el punto en cuestión. El objetivo ahora será refinar la búsqueda pero tomando a p como mejor estimación actual del vecino más próximo a q. Observar, pues, que no será necesario revisar regiones que tengan intersección vacía con una bola de radio d centrada en q.
- En el nodo hermano de esta hoja (es decir, el nodo que se encuentra ascendiendo y luego descendiendo por la rama opuesta), tomar la caja que delimita su región y computar su distancia d' a q.
- Si $d' \geq d$, podemos omitir por completo todo el subárbol originado en este nodo, de manera que ascenderemos en el árbol y volveremos a repetir el paso anterior en el padre.
- Si $d' < d$, tendremos que continuar explorando este subárbol puesto que podría haber puntos que mejoren nuestra estimación del vecino más próximo. Entonces, computaremos las distancias de las dos subregiones a q y comenzaremos explorando la más próxima recursivamente. Si, al regresar, la distancia de la otra subregión a q es mayor que d, podremos omitir la llamada y ascender. De lo contrario, también tendremos que descender por dicha rama.
- El proceso anterior termina cuando nos encontremos en la raíz del árbol de S y ascendamos por haber agotado las posibilidades.

La Fig. 3 muestra, para el plano de la Fig. 1, una consulta de un punto q, los puntos p_1 y p_2 de la hoja que mejor se ajusta a q, junto con sus respectivas distancias, y un ejemplo de distancia a una región dada.

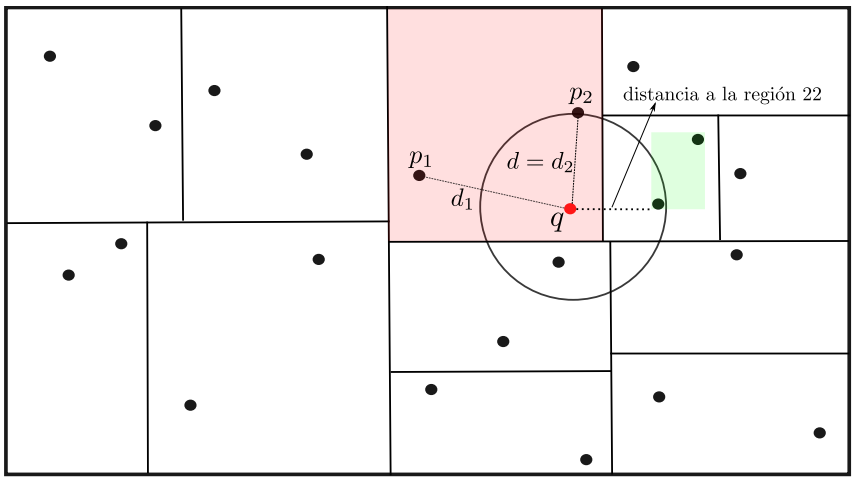

Figura 3: Consulta por el vecino más cercano al punto q

4.2. Interfaz

La forma de interactuar con nuestros programas es a través de la línea de comando, tal como ocurrió en el TP 0. Las opciones a implementar en este TP son las mismas que las utilizadas anteriormente más una adicional para indicar qué heurística de splitting se debe utilizar para operar con el árbol:

- `-p`, o `--points`, indica el archivo desde donde se leerá la base de datos de coordenadas de las reservas de tiberium. Esta opción es obligatoria.
- `-s`, o `--split`, indica la heurística de splitting que debe emplearse. Los valores posibles para esta opción son tres cadenas de caracteres: `mediana`, `mitad` o `promedio`, sin mayúsculas. En caso de no estar presente, el valor por defecto deberá ser `mediana`.
- `-i`, o `--input`, permite controlar el stream de entrada de las consultas. El programa deberá recibir las consultas a partir del archivo con el nombre pasado como argumento. Si dicho argumento es "`-`", el programa leerá de la entrada standard, `std::cin`.
- `-o`, o `--output`, permite direccionar la salida al archivo pasado como argumento o, de manera similar a la anterior, a la salida standard -`std::cout`- si el argumento es "`-`".

4.3. Formatos de entrada y salida

Al igual que en el primer trabajo práctico, la base de datos de coordenadas se especifica mediante un archivo cuya primea línea debe contener únicamente un número entero positivo: la dimensión de las coordenadas, d. A continuación deben seguir las coordenadas propiamente dichas, ocupando cada una una única línea. Cada coordenada viene dada por una secuencia de números reales $x_1\ x_2\ \ldots\ x_d$ separados por uno o más espacios o tabs, indistintamente.

Las consultas pueden ingresarse por teclado o bien por archivo. Se espera que el *stream* de consultas provea una única consulta por línea. Las coordenadas deben seguir los mismos lineamientos que los mencionados en el párrafo anterior.

En cuanto al stream de salida, se espera que las coordenadas encontradas se muestren cada una en líneas distintas y con sus componentes separadas por un único espacio.

4.4. Ejemplos

Para los ejemplos mostrados en esta sección utilizaremos la siguiente base de datos de coordenadas:

```
$ cat db.txt
2
1 1
-1.5 1.5
0 0
10 10
0 -10
```

El ejemplo más simple consiste en una entrada vacía. Observar que la salida es también vacía:

```
$ touch entrada1.txt
$ ./tp1 -i entrada1.txt -o salida1.txt -p db.txt
$ cat salida1.txt
$
```

El siguiente ejemplo muestra una ejecución con dos consultas donde la salida se muestra por pantalla:

```
$ cat entrada2.txt
0  0
2 -9.5
$ ./tp1 -p db.txt -i entrada2.txt
0 0
0 -10
```

Observar que, en caso de especificar una heurística de splitting específica, la salida debería seguir siendo equivalente:

```
$ cat entrada2.txt
0  0
2 -9.5
$ ./tp1 -p db.txt -i entrada2.txt -s mediana
0 0
0 -10
```

4.5. Portabilidad

Es deseable que la implementación desarrollada provea un grado mínimo de portabilidad. Sugerimos verificar nuestros programas en alguna versión reciente de UNIX: BSD o Linux.

5. Informe

El informe deberá incluir, como mínimo:

- Una carátula que incluya los nombres de los integrantes y el listado de todas las entregas realizadas hasta ese momento, con sus respectivas fechas.
- Documentación relevante al diseño e implementación del programa.
- Documentación relevante a los algoritmos y estructuras de datos involucrados en la solución del trabajo.
- El análisis de las complejidades solicitado en la sección 4.
- La comparación entre las heurísticas de división solicitada en la sección 4, incluyendo ejemplos de escenarios favorables y desfavorables para cada una.
- Documentación relevante al proceso de compilación: cómo obtener el ejecutable a partir de los archivos fuente.
- Las corridas de prueba, con los comentarios pertinentes.
- El código fuente, en lenguaje C++ (en dos formatos, digital e impreso).
- Este enunciado.

6. Fechas

La última fecha de entrega es el **jueves 19 de octubre**.

Referencias

[1] Wikipedia, "k-d tree." `https://en.wikipedia.org/wiki/K-d_tree`.

[2] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001. pp. 185-189.

[3] Wikipedia, "Quickselect." `https://en.wikipedia.org/wiki/Quickselect`.