Algoritmo de Kruskal y estructuras de partición.

Daniel Lamana García – Mochales.

Introducción.

El algoritmo de Kruskal hace uso de varias estructuras externas, como la estructura de partición o los grafos sobre los que trabaja. A continuación se expondrá con detalle la implementación de las mismas junto con el algoritmo.

Grafos y aristas. (graphwithedges.h)

Los grafos utilizados están implementados en una clase con listas de adyacencia. Para esto se ha optado por utilizar la estructura unordered_map, de la forma:
unordered_map<int, unordered_map<int, int>>
Esta elección se debe a que el coste amortizado para la operación at (acceso al elemento dada una clave) es costante (linear en el peor caso), como lo son emplace(args) y insert(int i) (operaciones de inserción). En esta clase también he añadido una variable _edges, que está implementada de esta forma: priority_queue<Edge, Vector<Edge>, greater<Edge>>. Esta es una cola de prioridad que, según se construye el grafo, recibe las aristas Edge (con tres valores enteros, origen, destino y valor) y las inserta ordenadamente en una cola de prioridad, de tal forma que no hay que reordenarlas antes ejecutar el algoritmo.

Para la prueba del programa se puede hacer uso de un constructor que, dados un número real p , un dos números enteros, maxValue y size, construye un grafo de size vértices, con una probabilidad p de que se forme una arista entre dos vértices de valor máximo maxValue. Existe una versión en la que se puede determinar si es el grafo dirgido o no (independientemente de las precondiciones del algoritmo) y otra en la que el grafo generado es asignado a uno existente.

Esta operación es de coste $O(n^2)$ debido a que realiza n iteraciones y en cada una de estas iteraciones considera, si es dirigido, los otros n – 1 vértices, y si no lo es, el bucle interno realiza $\sum_{i=0}^{n} i$ iteraciones, que equivale a $\frac{n*(n+1)}{2}$ (qyue es cuadrático) ya que al crear una arista no se consideran los pares de vértices ya considerados, es decir, en la primera iteración se consideran todos, en la segunda se consideran todos menos el primero, porque ese par ya se ha considerado, y así sucesivamente. Tampoco se restringen las aristas de un vértice a ese mismo vértice.

Clases de equivalencia. (eqclass.h)

Las clases de equivalencia se han implementado con dos variables: un puntero que apunta al padre _parent y un entero _heigth que es la altura.
El puntero _parent apunta a una clase que es equivalente a la que apunta, formando una estructura de árbol, en la cual la variable _heigth señala la altura a la que se encuentra el nodo.
Estas estructuras representan un único conjunto cociente y se utilizarán como parte de la estructura de partición. Si la clase de equivalencia es canónica, _parent apunta a si misma.

Estructura de partición (partition.h & partition.cpp).

Esta estructura es con la que representaremos los conjuntos cociente relacionados de cierta forma que es independiente a la implementación de esta estructura. En cuanto a los atributos hay un vector de clases de equivalencia. Los métodos son join, merge, find y findAux.

El constructor es sencillo, tiene un coste O(n), el método resize crea tantas clases de equivalencia en el vector como se le especifique en el argumento.

El método join une dos clases de equivalencia dadas las raíces de estas clases, los elementos identificativos, y tiene un coste constante, solo hace una asignación, una o dos comparaciones y una suma dependiendo del caso.

El método findAux busca el representante canónico de una clase dada esta clase de equivalencia. Este método es interesante porque aquí reside la razón para los buenos órdenes de esta estructura. Dada una clase de equivalencia eq, obtiene su padre, si este apunta a eq, hemos acabado y devuelve eq, si no, llama recursivamente a esta función pero con el padre de eq como argumento, y el valor que devuelve será el nuevo nodo al que apunte su campo _parent y luego es esta también la clase que retorna.
La idea de esto es recorrer el árbol hasta la raíz, donde está el representante canónico, que como se apunta a sí mismo, devuelve su clase, que es asignado al nodo que le ha llamado, y devuelve su padre, que es asignado al nodo que le ha llamado, y así sucesivamente, de tal manera que todos los nodos de esa clase de equivalencia están conectados directamente con el representante canónico (a esto lo llamaremos compresión de caminos), haciendo así que las operaciones de consulta de este sean constantes (dentro de un coste amortizado). Esta operación en el caso peor es O(n), sin embargo el coste amortizado es constante.

El método merge, dados dos enteros, llama a findAux para encontrar el representante canónico de sus clases de equivalencia, una vez obtenidos llama a join y los une. El coste amortizado de esta operación es constante por la combinación de los costes de la llamada y en el caso peor es lineal.

El método find encuentra la clase de equivalencia canónica de un número entero dado dicho entero, esto lo hace llamando a findAux con la clase de equivalencia del entero como argumento. Al igual que find, el coste amortizado es constante, encontrar la clase de equivalencia es un acceso de coste constante y findAux tiene un coste amortizado constante, en el caso peor es lineal.

Algoritmo de Kruskal. (kruskal.cpp)

El algoritmo de Kruskal es un método voraz que, dado un grafo, va construyendo un conjunto E de aristas. Este conjunto empieza vacio y se le añaden las aristas de una en una sin formar ningún ciclo con las que están ya en E (esto se consigue con la estructura de partición). Durante la ejecución, E no es ni conexo ni un ARM (árbol de recubrimiento mínimo), pero cuando termina, E es conexo, es ARM y recubre todo G, también contienen n – 1 aristas, con n el número de vértices. Este algoritmo termina cuando E alcanza las n – 1 aristas que debe de tener. Si el grafo no tiene n – 1 aristas, el algoritmo no terminaría, pero es precondición que el grafo sea conexo. La estrategia que se ha implementado es, teniendo un grafo y una cola de sus aristas ordenadas según su valor en orden creciente (la llamaremos pendientes), ambos pertenecientes a la clase GraphWithEdges:

Primero se declaran todas las variables. Esto tiene cierta importancia ya que se crea una cola de prioridad para las aristas y se copia la recibida por argumentos, esto se hace por si la susodicha lista ha de ser inmutable, lo cual tiene un coste O(n). Después de esto se entra un un bucle.

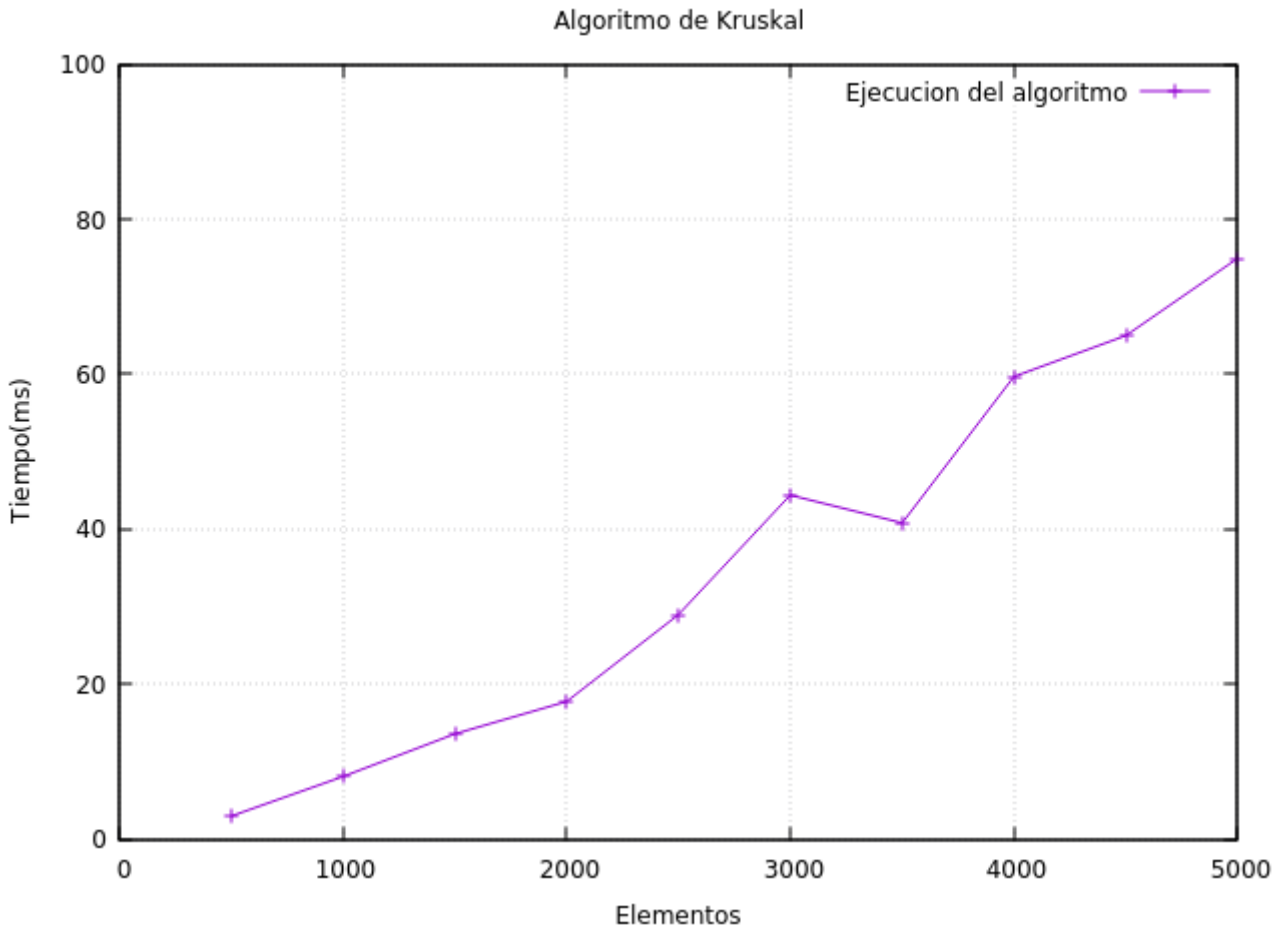
-Mientras pendientes no esté vacío y el contador i no sea menor que el número de vértices:

1) Se buscan las clases de equivalencia de los vértices de la arista a la cabeza de pendientes. Esta operación tiene un coste amortizado constante gracias a la estructura de partición.

2) Se comparan las clases de equivalencia, si son distintas se unen y se añade al conjunto E la arista que une a estos vértices. El coste amortizado de este conjunto de operaciones es constante por la estructura de partición, que además nos permite comprobar si al añadir la arista a E se formaría un ciclo, si los extremos de la arista pertenecen a la misma clase, se formaría.

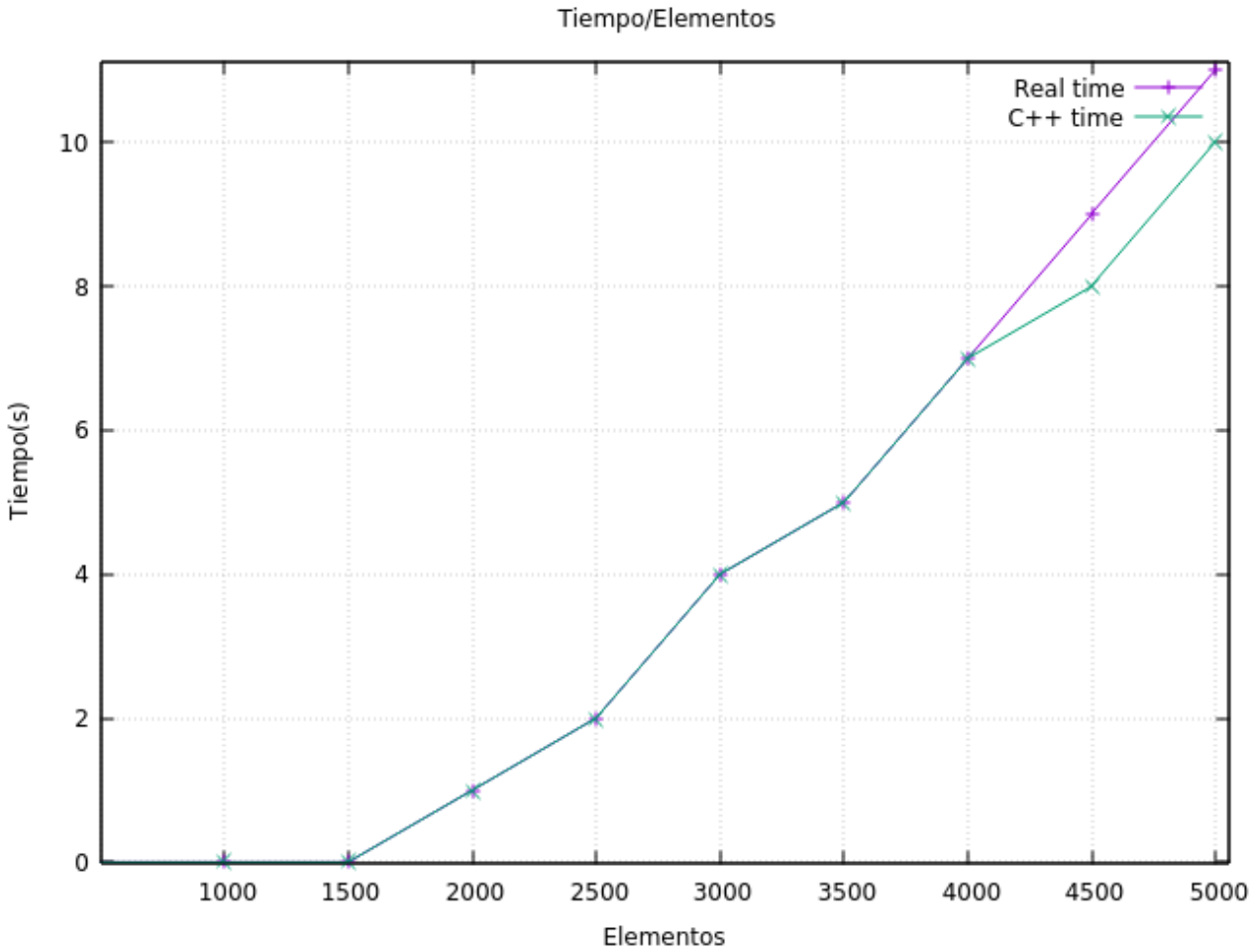
3) Se elimina de pendientes la arista considerada, se añada o no. Esta operación tambien es constante.

Como podemos observar en la teoría, el coste amortizado del bucle es O(n), realiza n veces unas operaciones costantes y hace una copia inicia de un argumento, que también es O(n), en total O(2n) que sigue dentro del orden de n ahora veremos con ayuda de unas gráficas si esto se cumple.



En el gráfico de arriba está el tiempo de ejecución del algoritmo en milisegundos según el número de elementos. Se han realizado tres mediciones por cada caso, y los casos empiezan en quinientos elementos y acaban en los cinco mil. Las mediciones e han efectuado cada quinientos elementos. Parece que sigue un crecimiento lineal como lo esperado, excepto en los 3500 elementos que, a pesar de haber sido medido tres veces, ha habido cierta fluctuación. Aun así el tiempo está dentro de las expectativas teóricas, que es el crecimiento lineal.

El tiempo de ejecución del algoritmo ha sido medido con la librería chrono y utilizando las facilidades de system_clock, siendo una opción más cercana al reloj del sistema que a un cálculo virtual.



Para una comprobación de que el tiempo es bastante preciso dentro de lo que cabe, este es el tiempo de ejecución del programa kurskal.cpp completo medido con la herramienta time de linux y con librerías de c++ que llaman al reloj del sistema, que tiene un crecimiento que se puede llegar a asemejar con el inicio de una curva que crece de manera exponencial, esto se debe a que las pruebas se han llevado a cabo con una generación aleatoria de grafos la cual es cuadrática y es donde el programa hace uso de la mayoría de tiempo de cómputo. Sin embargo, el tiempo real y el estimado por c++ son muy parecidos hasta los cuatro mil elementos, donde ya se observan ciertas diferencias.

Con estas dos gráficas podemos sacar varias conclusiones, para empezar el algoritmo se ejecuta dentro de las expectativas teóricas, con más o menos fluctiaciones que pueden derivarse de causas como fallos en la memoria caché. También destaca la diferencia de cómputo necesario para crear los grafos y otras operaciones externas al algoritmo con el cómputo del propio algoritmo (la gráfica de tiempo total está en segundos y la de ejecución del algoritmo está en milisegundos, la diferencia es grande)

Como conlcusión, a destacar la utilidad y eficiencia de la estructura de partición, con un coste amortizado costante en todas las operaciones menos en las de creación. El algoritmo puede llegar a ser útil en algunos casos pero destaca la eficiencia conseguida gracias a la estructura de partición.

Bibliografía.

Los órdenes de las funciones de librerías se pueden encontrar en https://es.cppreference.com
El trabajo ha sido guiado y complementado con el libro:
Algoritmos y estructuras de datos Con programas verificados en Dafny
de Ricardo Peña Marí.

Notas.

Para la compilación del los archivos existe un makefile, con el comando make all se compila, con make clear el archivo compilado se borra, ha sido probado en gnu/linux (ubuntu 18.04) y es necesario el compilador g++.

El programa puede tomar argumentos, estos son r p maxValue vertexs.
- r es un flag que indica al programa la generación de un grafo aleatorio, en caso de no indicarse, ejecuta el algoritmo con un grafo predefinido en el programa (el mismo que sale en la página 234 del libro mencionado en la bibliografía)
- p es la probabilidad que tienen los vértices de ser conectados mediante una arista.
- maxValue es el valor máximo de las aristas.
- vertex es el número de vértices del grafo

Para tomar los tiempos se ha creado un script que ejecuta tres veces el programa mostrando los tiempos tomados por el programa y por el sistema operativo, el cual recibe los mismos argumentos de forma opcional:

bash medirtiempos.sh r p maxValue vertexs