DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Algoritmo de Prim - Multithreading

29 de octubre de 2019

Sistemas Operativos
2dp Cuatrimestre de 2019

Integrante	LU	Correo electrónico
Giudice, Carlos Rafael	694/15	carlosr.giudice@gmail.com
Olmedo, Dante	195/13	dante.10bit@gmail.com
Rosende, Federico Daniel	222/16	federico-rosende@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Palabras Clave

Árbol generador mínimo, Threads, Paralelismo, Mutex

Índice

1. Resumen

En este trabajo se buscará analizar la aplicación del modelo de ejecución multithreading sobre el algoritmo de Prim para obtener un árbol generador mínimo de un grafo.

Se describirá formalmente el problema, dando una breve noción del algoritmo de Prim. Posteriormente, se establecerá el método de aplicación de la técnica de multithreading, explicando las distintas estructuras que serán utilizadas para la representación interna, así como también los mecanismos de sincronización necesarios para el correcto funcionamiento del algoritmo.

Finalmente, mediante experimentación y análisis se buscará exponer los distintos comportamientos del algoritmo para distintos tamaños y tipos de grafos, así como también diferentes cantidades de threads.

2. Introducción

2.1. Formalización del problema

2.1.1. Algoritmo de Prim

Dado un grafo conexo[1] $G = (V, E)$, con V un conjunto de vértices de $|V|$ elementos y E un subconjunto de $V \times V \times \mathbb{N}$ llamado conjunto de ejes (con peso), se busca su árbol generador mínimo. Es decir, un grafo $\overline{G} = (V, \overline{E})$, que tiene el mismo conjunto de vértices que G, es conexo y tiene un subconjunto $\overline{E}$ de ejes de E, con $|V| - 1$ elementos y peso total mínimo.

El de Prim es un algoritmo goloso que parte de un vértice al azar y en cada paso busca el eje de menor peso que conecta el subconjunto de vértices que ya se tiene y el resto.

1. Se elige un vértice v al azar de todos los de V y se lo incorpora.
2. Mientras la cantidad de vértices agregados sea menor a $|V|$
   a) Elegir el eje e de menor peso incidente sobre algún vértice u tomado y que se enlace con otro $\overline{u}$ de afuera e incorporar tanto e como $\overline{u}$ al árbol.

2.1.2. Multithreading

La idea de este modelo es subdividir a los procesos en hilos (o threads) que se encargan de ejecutar distintas secciones del mismo de forma paralela. En el caso particular de nuestro problema, a partir de un proceso que realice el algoritmo de Prim, es posible subdividirlo en hilos de ejecución para que cada uno se engargue de explorar secciones distintas del grafo y así armar el árbol generador mínimo de forma paralelizada.

Cabe destacar, que a la hora de implementar esta técnica es indispensable tener presente la necesidad de generar exclusión mutua en ciertas partes del algoritmo con el fin de llegar a resultados correctos.

Se dice que un resultado es correcto cuando existe una ejecución de un algoritmo secuencial (no paralelizada) correcto para el problema que se busca modelar que lo devuelve.

[1] Un grafo se dice conexo cuando existe camino entre todo par de nodos del mismo.

3. Desarrollo

3.1. Implementación

A la hora de implementar el algoritmo utilizando el paradigma multithreading, se optó por la utilización de la API definida por el estándar POSIX. La misma permite el uso de semáforos, mutex, spinlocks y otras estructuras para lograr la exclusión mutua en secciones críticas del programa.

Más precisamente, se utilizaron:

Tipos

- *pthread_t* → utilizado como identificador de cada thread.
- *pthread_mutex_t* → como su nombre lo indica, define el tipo al que pertenecen los mutex.
- *pthread_barrier_t* → define el tipo al que pertenecen las barreras.

Funciones

- *pthread_mutex_init* → usada para inicializar los mutex.
- *pthread_barrier_init* → usada para inicializar la barrera, pasandole la cantidad de threads a lockear como parametro.
- *pthread_create* → que inicializa in thread, le asigna memoria compartida y una función en el código.
- *pthread_trylock* → cuya funcionalidad permite intentar tomar un mutex y controlar el flujo del programa basándose en el resultado ya que no es bloqueante.
- *pthread_lock* → contrario a *pthread_trylock*, esta función pide el recurso del mutex pero es bloqueante.
- *pthread_unlock* → utilizada para liberar el recurso de un mutex tomado por cualquiera de las dos funciones previas.
- *pthread_barrier_wait* → efectua el mismo comportamiento que pthread_lock, pero lo hace con los n threads definidos en pthread_barrier_init, unlockeandolos al llegar el n-esimo thread
- *pthread_self* que devuelve el thread id.
- *pthread_exit* → para indicarle a un thread que detenga su ejecución.

Estructuras Por otro lado, se definieron estructuras propias.

1. En primer lugar, se cuenta con una clase $Grafo$, la que guarda el número de vértices, el de nodos y representa al grafo como una lista de adyacencias. La lista de adyacencias se encuentra implementada como un diccionario D de vectores. El vector correspondiente a la entrada k de D guarda los ejes incidentes en el nodo k con sus pesos correspondientes.
2. También se tiene una estructura Eje muy simple que, para el eje e, guarda los nodos incididos por el mismo, representados como *nodoOrigen* y *nodoDestino*, y su peso p.
3. Se definió otra estructura *sharedData*, la que se usa como wrapper de gran parte de lo que se quiera incluir en memoria compartida entre threads. En una instancia dada de la misma, se tiene: el grafo G al que se le quiere buscar el árbol generador mínimo; un diccionario de mutex para cada thread $threadsMutex$ cuyas keys están dadas por los thread id de cada uno; un vector de nodos libres para la inicialización; un vector con mutex para nodos y otro mutex para asegurar la contención en secciones críticas del código.
4. Finalmente, se tiene la clase $Thread$ que guarda toda la información relevante para la correcta ejecución de un hilo del proceso que corre el algoritmo de Prim. Se tienen:
   - Un grafo *mst* que representa el árbol que el thread tiene en un determinado momento de la ejecución.
   - El conjunto de ejes *mstEdges* candidatos a ser agregados en la siguiente iteración del algoritmo. Es decir, aquellos ejes que inciden en algún nodo de *mst* y en otro fuera del mismo. Los mismos se encuentran en una cola de prioridad ordenados por peso.

- El thread id obtenido al momento de crearse el thread, que sirve para indexar el diccionario de threads en la memoria compartida y como color para los nodos.
- La cola de pedidos de unión $request_queue$ y una variable *merged* que indica si el thread fue partícipe de una unión recientemente. Estos dos atributos tendrán mayor sentido cuando se especifique con más detalle el algoritmo final.
- Finalmente, se tiene una variable *die* que determina si un thread debe llamar a *pthread_exit*.

3.1.1. El algoritmo con multithreading en mayor detalle

Inicialmente, se crean todas los elementos necesarias para inicializar la estructura de memoria compartida ya mencionada previamente (*sharedData*). Sin embargo, como se evidenció al momento de la definición de esta última, no es la única estructura de memoria compartida que se tiene. Adicionalmente, se define un diccionario de *Threads* (objetos de la clase) cuyas keys, nuevamente (al igual que en el diccionario de mutex), se corresponden con los thread id. Todo esto forma parte de la memoria compartida entre threads.

Se cuenta con una función *mstParaleloThread* que se invoca por cada thread al momento de su creación mediante *pthread_create*. Esta función se encarga de inicializar los mutex del diccionario y los objetos *Thread*.

Al crearlos, utilizamos pthread_barrier_wait.La razón para esto es que aparecían condiciones de carrera de escrituras en los maps usados mientras que otros threads querían acceder, esto es esperado en las estructuras usadas, que no garantizan concurrencia. Como las creaciones están al inicio y no son comportamiento ciclicos podíamos poner una barrera sin perjudicar la performance.

Para inicializar los mutex del diccionario y los objetos *Thread* se cuenta con un método *initThread* de la clase *Thread* que obtiene un nodo libre para comenzar el algoritmo.

Algorithm 1 Init

```
1:  procedure initThread
2:      nodeFound ← false
3:      while ¬nodeFound do
4:          if |freeNodes| == 0 then
5:              die ← true
6:          end if
7:          node ← getFreeNode(freeNodes)
8:          mutexNodos[node].lock()
9:          if nodesColors[node] == −1 then
10:             nodeFound ← true
11:             eje = (−1, node, −1)
12:             processNode(eje)
13:         end if
14:         nodesMutex[node].unlock()
15:     end while
16: end procedure
```

En este punto se utilizó el mutex del nodo a procesar ya que de no hacerlo podría suceder que más de un thread obtuvieran el mismo nodo y lo procesaran entendiendo que tienen exclusividad sobre el mismo. Así, se caería en una condición de carrera, alterando el correcto funcionamiento del algoritmo, en la que uno de los threads continúa su procesamiento asumiendo que el nodo está en su *mst* cuando no es así.

Luego del llamado a *initThread*, se invoca a *processThread*. Este método dispara la funcionalidad principal del algoritmo en cuestión. La idea principal de este métoodo es la de buscar ejes a agregar mientras los haya y con los mismos proceder a determinar si es posible pintarlo directamente (porque nadie lo pintó) o si se debe entrar en proceso de unión con otro thread (porque lo tenía en su *mst*). Aquí resulta fundamental el uso adecuado de mutex y sus combinaciones con las funciones de la *API* para evitar deadlocks y/o livelocks.

Notar en el pseudocódigo que sigue, que a la hora de procesar un nuevo nodo, no se realiza un lock bloqueante. Esto es para permitir que el algoritmo siga avanzando aún cuando algún otro thread está pidiendo ese lock. Así, se puede atender la cola de pedidos de unión mientras el thread que pidió el lock realiza la tarea que necesite (más adelante mostraremos que pide el lock del otro cuando se ubica en la cola de pedidos de unión).

Algorithm 2 ProcessThread

```
1: procedure PROCESSTHREAD
2:     if |mstEdges| > 0 then
3:         eje ← getNextEdge()
4:         while |vertices(mst)| < |vertices(G)| do
5:             procesado ← true
6:             mutex_taken ← false
7:             if ¬nodesMutex[eje.nodoDestino].trylock() then
8:                 mutex_taken ← true
9:                 procesado ← processNode(eje)
10:                nodesMutex[eje.nodoDestino].unlock()
11:            end if
12:            if die then
13:                time_to_die()
14:            end if
15:            procesado ← procesado ∧ mutex_taken
16:            if |request_queue| > 0 then
17:                request ← get(request_queue)
18:                merge(request)
19:                if die then
20:                    time_to_die()
21:                end if
22:                eje ← getNextEdge()
23:                procesado ← false
24:            end if
25:            if procesado then
26:                eje ← getNextEdge()
27:            end if
28:        end while
29:    end if
30: end procedure
```

Observaciones

- El método *time_to_die* de *Thread*, es simplemente un wrapper para *pthread_exit*.
- El uso de las variables booleanas *procesado* y *mutex_taken* resultan fundamentales para la correcta exploración de los ejes a agregar. La primera indica si *processNode* fue exitoso a la hora de agregar un eje o realizar una unión con otro thread. La segunda adopta un valor verdadero solamente al tomar el mutex del nodo que se busca procesar (el nodo destino del eje de menor peso de *mstEdges*).

Aquí resulta necesario detallar el método *processNode*, utilizado tanto en *initThread* como en *processThread*.

Como su nombre lo indica, es el encargado de pintar nodos y actualizar la cola de ejes candidatos, pero también determina si es necesario realizar un pedido de unión debido a que el nodo que se desea agregar no está libre (encolándose en la cola de pedidos del thread que lo había pintado).

Algorithm 3 ProcessNode

```
1: procedure processNode(eje)
2:     node_color ← nodesColors[eje.nodoDestino]
3:     if node_color == −1 then
4:         pintarNodo(eje)
5:         pintarVecinos(eje.nodoDestino)
6:         return ← true
7:     end if
8:     merge_solved ← false
9:     if ¬threadMutex[threadId].trylock() then
10:        if ¬threadMutex[node_color].trylock() then
11:            if node_color == nodesColors[eje.nodoDestino] then
12:                requestMerge(threadObjects[node_color], eje, node_color)
13:                while ¬merged do
14:                end while
15:                merged_solved ← true
16:            end if
17:            threadMutex[node_color].unlock()
18:        end if
19:        threadMutex[threadId].unlock()
20:        merged ← false
21:    end if
22:    return ← merged_solved
23: end procedure
```

Observaciones

- En el caso que el nodo que se desea incorporar esté pintado, se utilizan tanto el mutex del thread que realiza el pedido de unión como el del que lo recibe. Esto se debe a que no es deseado que ninguno reciba pedidos extra mientras se encuentra en proceso de unión con otro thread. Si se permitiera este comportamiento, podría darse una condición de carrera en la cola de pedidos y perderse alguno.
- Otra observación importante es que el thread que realiza el pedido, luego se queda esperando a que el otro le notifique la finalización del proceso de fusión mediante la variable *merged* que es atributo de *Thread*. Este busy waiting se realiza para efectivamente priorizar las uniones por sobre el procesamiento normal de nodos. Así, aquel que lo pide, no regresa a *processThread* y no sigue pintando otros nodos. Nuevamente, si esto sucediera, podría darse una condición de carrera sobre el nodo a incorporar por el thread que solicitó la unión.
- En el método *request_merge* simplemente se realiza el encolado en la cola de pedidos del thread dueño del nodo que se desea pintar.

Finalmente, es importante hablar sobre el proceso de unión de threads. En *processThread*, se menciona un método llamado *merge*. El mismo, es el encargado de realizar la comparación entre los *threadid* de ambos threads involucrados en la operación y determinar el que prevalece y el que es reiniciado. Para realizar este procedimiento, se cuenta con un método *fagocitar*.

La idea aquí, es que el thread que invoque a la función, será el de *threadid* más chico y, por lo tanto, aquel que se absorberá las estructuras del otro. En *fagocitar* simplemente se realizan las siguientes acciones:

1. Se agregan nodos y aristas del *mst* del otro al propio. Luego se agrega la arista que generó el llamado a *merge* para unir ambos subárboles.
2. Se modifica *nodesColors* coloreando todos los nodos con el *threadid* que absorbe al otro.
3. Se actualiza *mstEdges* con el nuevo *mst* para tener los ejes a explorar.
4. Se reinicia toda la estructura del otro thread como si recién hubiera sido creado y se procede a realizar el llamado a *initThread* para el mismo.

Notar que debido a que ambos threads quedan con sus mutex pedidos al momento del pedido de unión, no es necesario utilizar herramientas de sincronización aquí ya que ninguno sigue procesando nodos o recibiendo pedidos. Por esto último es que tampoco es necesario heredar los pedidos de la cola del otro thread (al momento de recibir o enviar uno, ambos threads se ponen a resolverlo antes de seguir).

De esta forma, quedan explicados las principales nociones de sincronización y las líneas de ejecución del algoritmo. Lo siguiente es exhibir los resultados obtenidos a la hora de experimentar con distintos valores de entrada y tipos de grafos.

4. Resultados

4.1. Objetivos

Buscando entender la performance del algoritmo desarrollado, fue evaluado su costo temporal. Para esto se realizaron corridas de dicho algoritmo, variando las condiciones de input. Aparte de variar la cantidad de threads, es posible elegir distintos grafos de entrada. Como se verá más adelante, no solo es relevante la cantidad de nodos y aristas de un grafo, sino también la morfología del mismo.

4.2. Familias de grafos

Para entender el comportamiento del algoritmo con distintas morfologías de grafos, se utilizaron dos familias de grafos propuestas por la cátedra, con la adición de tres familias extra.

Familias

- Completos: son grafos en los cuales todo nodo tiene una arista apuntando a cada nodo del grafo. Aquí, cada nodo tiene grado[2] $|V| - 1$ (donde V es el conjunto de nodos del grafo).
- Árboles: son grafos conexos acíclicos.
- Listas enlazadas: son un caso particular de arbol, donde solo dos nodos tienen grado uno y el resto tienen grado dos.
- Grafos estrellas: son un caso particular de árbol, donde hay un nodo central c tal que para todo nodo v distinto de c, existe un único eje que incide sobre él y lo une con c. Es decir, el nodo c tiene grado $|V| - 1$ y cualquier otro nodo tiene grado uno.
- Ralos: son grafos aleatorios, donde tan solo un 10 % de los pares de nodos estan unidos por aristas. Notar que esto podría darnos un grafo no conexo por lo que no tendría sentido plantear la búsqueda de un árbol generador mínimo. En este caso, se busca un bosque generador mínimo[3] en el que cada componente conexa cubre aquella del grafo con la que comparte nodos. Debido a la forma en que fue realizado el algoritmo, es lo suficientemente robusto como para soportar la búsqueda de este bosque generador mínimo y por ello es posible realizar la comparación con los grafos previamente mencionados.

El propósito de agregar la lista enlazada y el grafo estrella, es contar con grafos que tienen la misma cantidad de ejes y vertices, pero una morfología muy distinta. En particular podemos observar que para un grafo estrella, no hay dos nodos con mas de un grado de separación, mientras que para una lista enlazada el maximo grado de separación es n - 1.

4.3. Experimentos

A la hora de experimentar sobre el algoritmo, se encaró el análisis desde distintas perspectivas. Para ello, se graficaron diferentes tipos de relaciones para cada familia propuesta en el apartado previo.

Tipos

1. La primera perspectiva nace a raíz de pensar en la relación entre tiempo de ejecución y cantidad de nodos. Surge naturalmente y probablemente sea la más intuitiva a la hora de pensar en algoritmos sobre grafos.
2. En segundo lugar, se tiene otra comparación que resulta natural al pensar en grafos y es la de tiempo de ejecución en función de cantidad de aristas.
3. Para el tercer y cuarto análisis se pensó más en el aspecto multithreading del algoritmo propuesto. Así surgió la idea de observar la relación entre tiempo y cantidad de threads, y tiempo y cantidad de uniones.

[2] El grado de un nodo es igual al número de aristas que inciden sobre él.
[3] Se llama bosque a un conjunto de árboles.

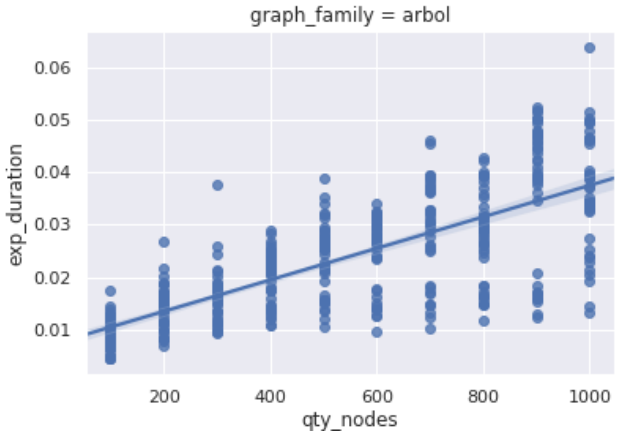
Aclaraciones generales

- Se realizaron diez ejecuciones de cada parametrización (conjunto de valores de entrada) del algoritmo.
- Se guardaron todos los resultados obtenidos y, a la hora de realizar los gráficos, se agruparon en conjuntos que dependieron de la elección de la variable independiente y se realizó una regresión lineal sobre los mismos.
- Los tiempos se miden en segundos en todos los gráficos y todas las otras variables no tienen unidad.

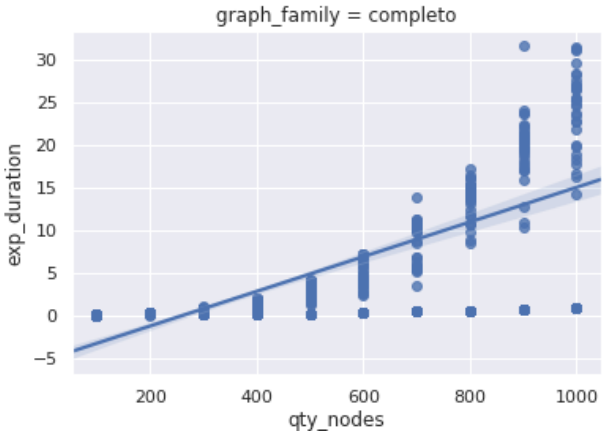
4.3.1. Tiempos en función de la cantidad de nodos

Como fue mencionado previamente, esta experimentación resulta natural y, por ese motivo, no requiere de mayor introducción. El resultado esperado era que el tiempo de ejecución aumente conforme se incrementa la cantidad de nodos.

Los resultados para cada familia fueron los siguientes.

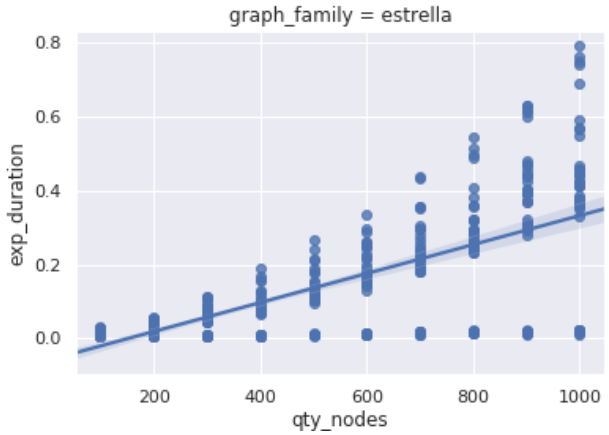


(a) Tiempo en función de la cantidad de nodos para grafo árbol.



(b) Tiempo en función de la cantidad de nodos para grafo completo.

Figura 1



(a) Tiempo en función de la cantidad de nodos para grafo estrella.

(b) Tiempo en función de la cantidad de nodos para grafo lista enlazad.

Figura 2

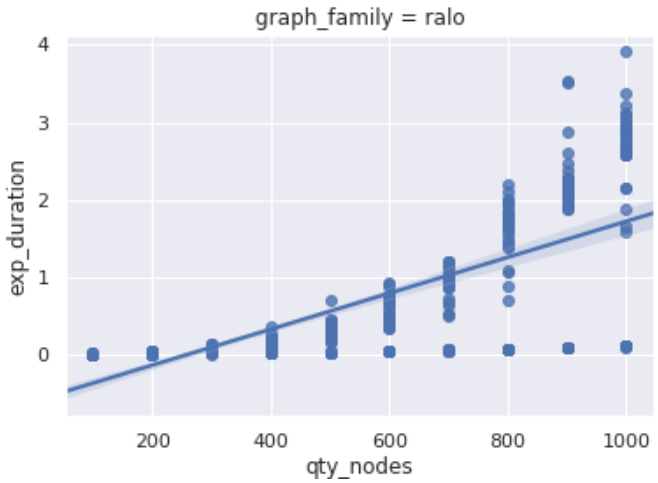
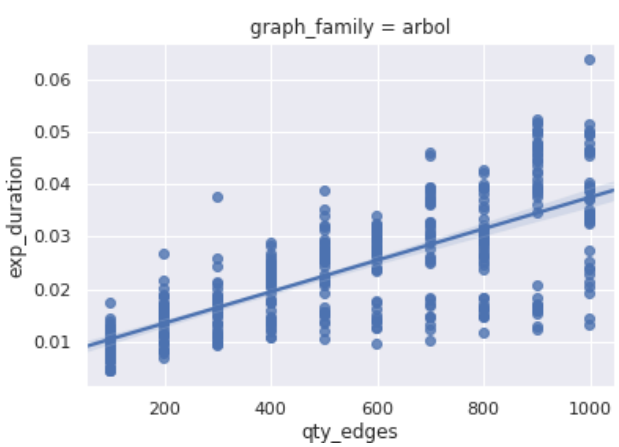


Figura 3: Tiempo en función de la cantidad de nodos para grafo ralo.

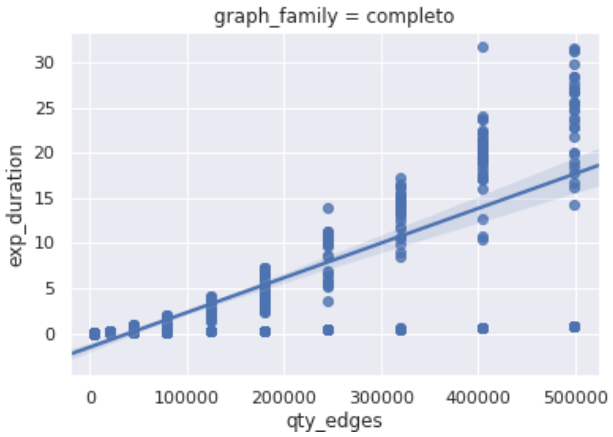
Se observa que el comportamiento es el esperado previamente y los tiempos crecen junto con el tamaño del conjunto de nodos. Notar que la inclusión de multithreading en el algoritmo no puede significar una eliminación de la relación entre tiempos y cantidad de nodos ya que la complejidad del algoritmo de Prim[4] depende de ello.

4.3.2. Tiempos en función de la cantidad de ejes

Nuevamente, no es necesario extenderse mucho en explicar nociones de este caso de experimentación. Sin embargo, cabe destacar que se espera un resultado similar al anterior. Esta idea encuentra su justificación, al igual que en el experimento previo, en la complejidad del algoritmo de Prim.



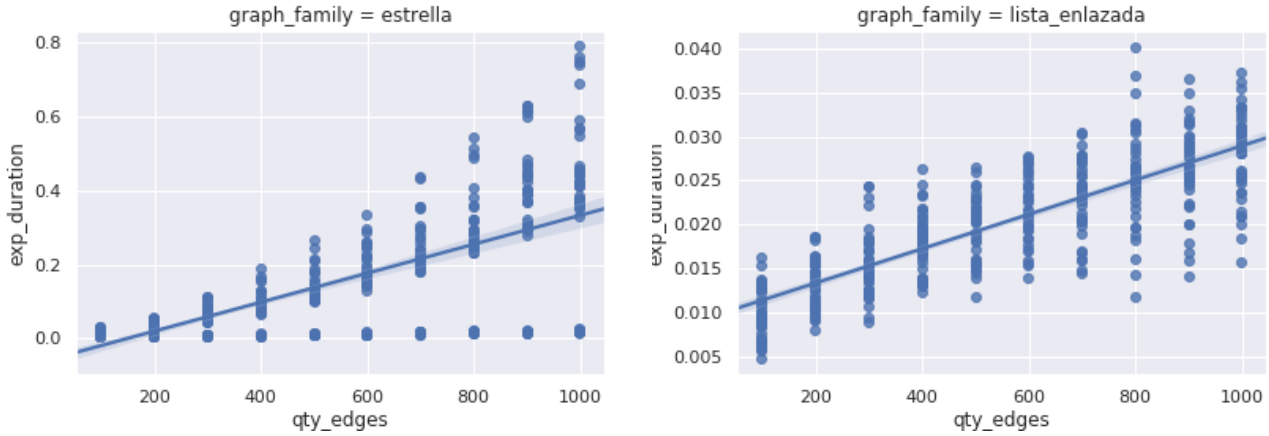
(a) Tiempo en función de la cantidad de ejes para grafo árbol.



(b) Tiempo en función de la cantidad de ejes para grafo completo.

Figura 4

[4]El algoritmo de Prim tiene complejidad $\mathcal{O}(Elog(V))$ o $\mathcal{O}(E + log(V))$ usando Fibonacci heaps (donde E es el número de aristas y V el de nodos)



(a) Tiempo en función de la cantidad de ejes para grafo estrella.

(b) Tiempo en función de la cantidad de ejes para grafo lista enlazad.

Figura 5

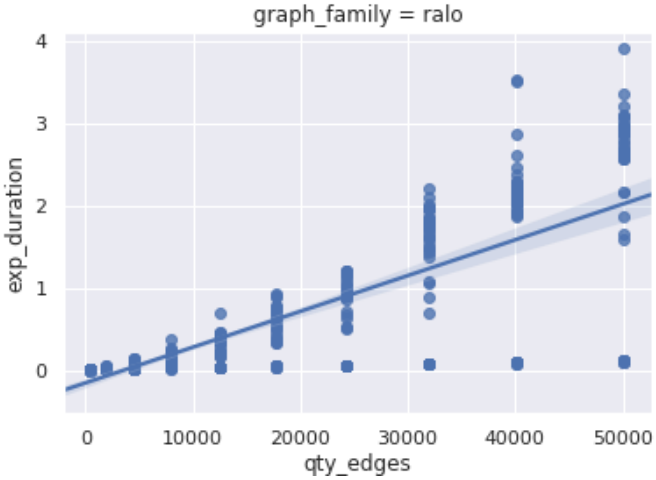


Figura 6: Tiempo en función de la cantidad de ejes para grafo ralo.

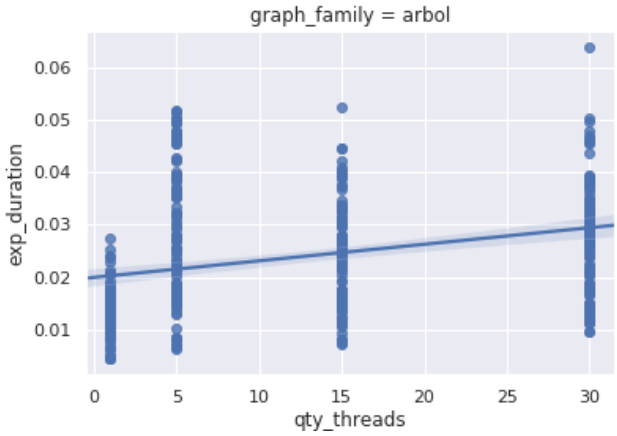
Similarmente a lo observado para la variación de nodos, el resultado es el esperado y se debe, como fue mencionado, a la complejidad de base que tiene el algoritmo.

4.3.3. Tiempos en función de la cantidad de threads

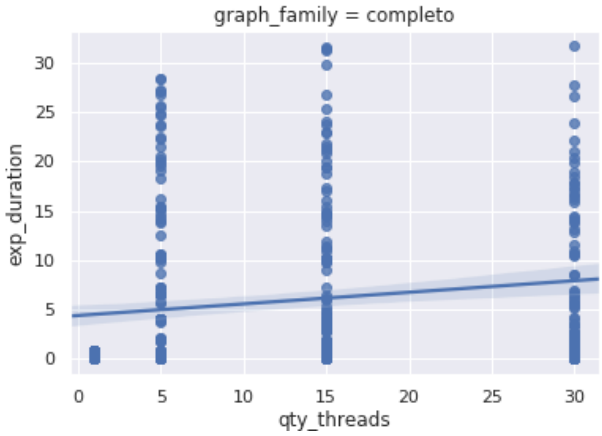
Aquí comienza la etapa de experimentación más interesante ya que se empiezan a introducir nociones propias de multithreading.

En este caso particular, se busca la relación entre cantidad de threads y el tiempo de ejecución. Se espera que el tiempo baje al aumentar los threads, ya que al paralelizar el procesamiento debería llegar a encontrar el árbol con mayor rapidez.

Los resultados fueron los exhibidos a continuación.

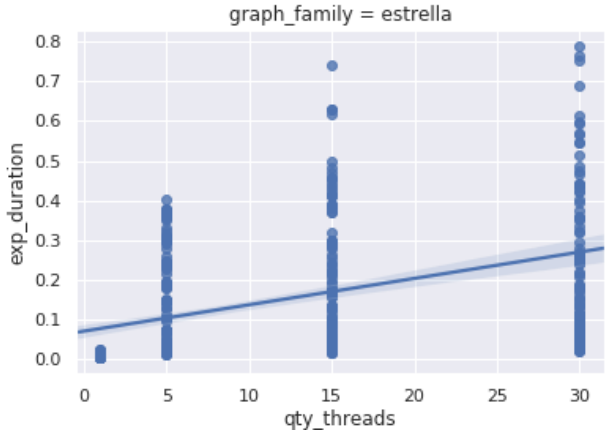


(a) Tiempo en función de la cantidad de threads para grafo árbol.



(b) Tiempo en función de la cantidad de threads para grafo completo.

Figura 7



(a) Tiempo en función de la cantidad de threads para grafo estrella.

(b) Tiempo en función de la cantidad de threads para grafo lista enlazada.

Figura 8

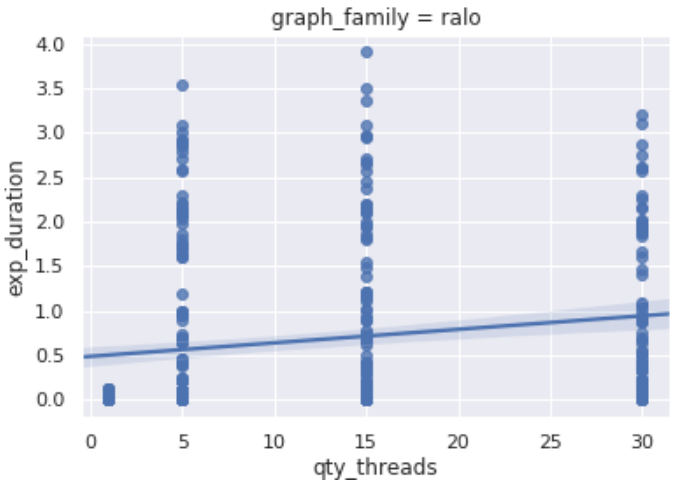


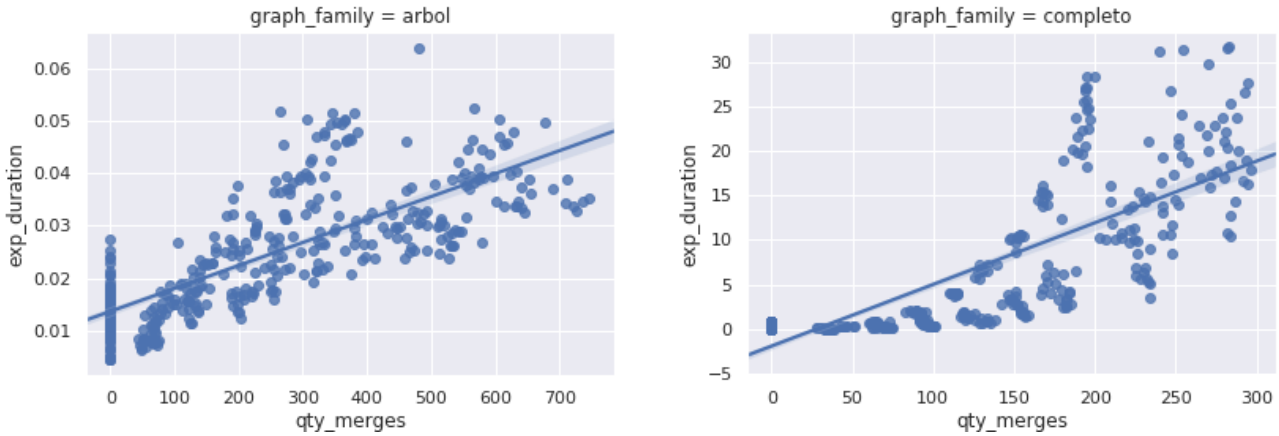
Figura 9: Tiempo en función de la cantidad de threads para grafo ralo.

Contrario a lo que se esperaba, no hay una reducción de los tiempos. Al contrario, se observa que se mantienen dentro de los mismos valores o crecen levemente.

Al observar el código, se ve que cada unión entre threads es costosa. Esto introduce un overhead en el algoritmo que parece anular los beneficios de la paralelización dada por el multithreading. A raíz de esto, surgió la idea del siguiente experimento. Es decir, si se cree que las uniones son tan costosas, sería útil observar la relación entre la cantidad que se producen y el tiempo requerido.

4.3.4. Tiempos en función de la cantidad de uniones

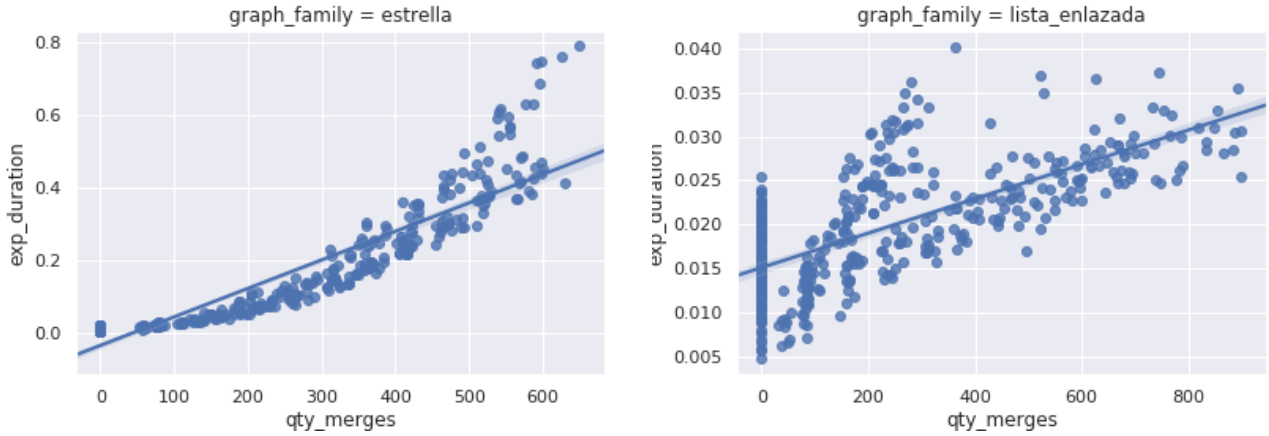
En este caso, el objetivo es determinar si efectivamente existe una relación entre la cantidad de uniones y los tiempos. Para ello, se agregó una forma de contar las mismas en el algoritmo y se obtuvieron gráficos de tiempo en función de uniones.



(a) Tiempo en función de la cantidad de uniones para grafo árbol.

(b) Tiempo en función de la cantidad de uniones para grafo completo.

Figura 10



(a) Tiempo en función de la cantidad de uniones para grafo estrella.

(b) Tiempo en función de la cantidad de uniones para grafo lista enlazada.

Figura 11

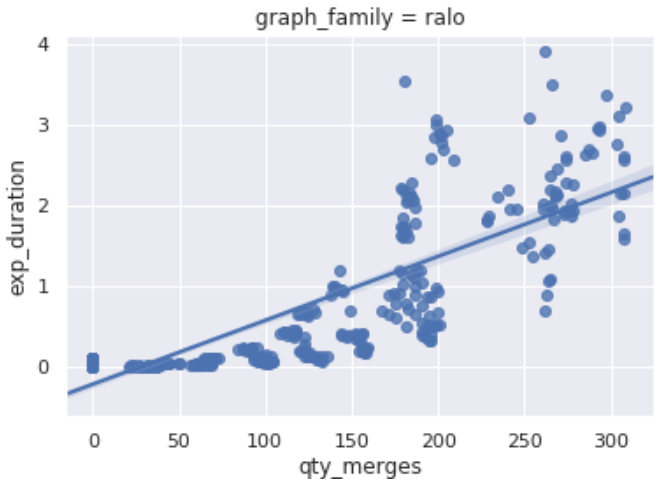


Figura 12: Tiempo en función de la cantidad de uniones para grafo ralo.

Acá se observa una clara relación entre la cantidad de uniones y los tiempos. Esto otorga evidencia empírica para respaldar la idea del overhead aportado por las mismas.

Un interrogante que puede resultar de interés a partir de esto es si existe una clase de grafos que sea mejor en torno a esta métrica.

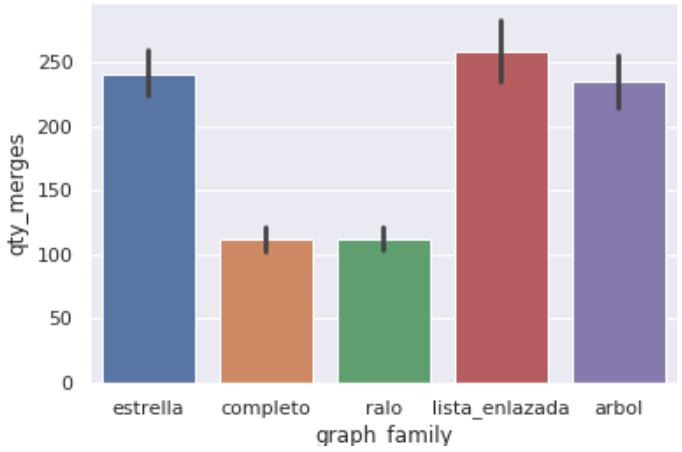


Figura 13: Uniones por familia de grafo.

Se ve que la mayor cantidad de uniones se producen en los grafos pertenecientes a la familia de los árboles (tanto los estrella como las listas enlazadas son casos particulares de la misma). Esto está directamente vinculado con la cantidad de ejes que tienen.

Debido a que un árbol T tiene $|V| - 1$ ejes esto aumenta las colisiones entre threads. Esto deriva en pedidos de unión, eso lleva a destinar mucho tiempo de cómputo en uniones y no en procesamiento del árbol generador mínimo.

Por otro lado, se aprecia que tanto los grafos completos como los ralos, tienen una cantidad de uniones mucho menor a los árboles. En estos casos, las justificaciones son diferentes para cada familia.

En el caso de los completos, se tiene una gran cantidad de ejes a explorar y eso disminuye la probabilidad de caer en colisiones. Al reducir eso, también se reduce el overhead aportado por las uniones.

En el caso de los ralos, es posible que tenga ver con el hecho que muchos de los generados pueden no ser conexos. Esto genera que muchos threads no lleguen a ejecutar muchas iteraciones del ciclo principal del algoritmo.

Algunos ejemplos de lo mencionado son los siguientes:

- Si un thread tomara un nodo aislado, simplemente terminaría la ejecución al ver que no posee ejes para explorar.
- Si cayera en una componente conexa y fuera el único, sería imposible que tuviera una colisión.

Notar que esta familia no cae en las consideradas al construir el algoritmo pero sirve para realizar un punto de comparación.

5. Conclusiones

5.1. Conclusiones generales

Se realizó una implementación paralela del algoritmo de Prim, mediante el uso de la tecnología de multithreading. Para esto fue necesario trabajar con estructuras de datos compartidas por distintos threads, siempre prestando atención al problema del uso concurrente de dichas estructuras. Fue de absoluta importancia el uso de mutex para evitar la ocurrencia de condiciones de carrera.

Aún siendo que la complejidad de la implementación es para destacar, por ser sustancialmente distinta (y llevar muchas más horas de desarrollo) a la implementación secuencial, se ha logrado incorporar exitosamente dichas herramientas y técnicas, lo que nos abre un abanico de posibilidades como desarrolladores de software.

Los resultados experimentales evidencian que la implementación paralela obtenida muestra un rendimiento menor al de una implementacion secuencial. Se atribuye esto al costo inherente a realizar las uniones entre los árboles desarrollados por los threads, dado que la operación de fusión de arboles implica numerosas operaciones con las estructuras subyacentes y otras técnicas de sincronización como *busywaiting*.

Como conclusión general sobre la técnica de multithreading y algoritmia paralela, se considera que es una herramienta muy poderosa y representa una alternativa más que interesante a la programación habitual basada en la noción del uso de un solo thread. No obstante, es necesario invertir el tiempo necesario en optimización y diseño para lograr resultados superadores. Por otro lado, la experimentación resulta crucial para explorar estructuras que permitan aprovechar al máximo estos beneficios.

5.2. Experimentaciones futuras

Sería de gran interés ahondar en la optimización de la implementación lograda. Debría ser posible reducir el costo de la operación de unión entre árboles optimizando el uso de estructuras de datos (o proponiendo alternativas más adecuadas al problema en cuestión) y las operaciones necesarias sobre estas para realizar la fusión.

Otro tema que sería interesante indagar con mas detenimiento es el comportamiento del algoritmo para distintas densidades de ejes. Para esto se propone, como punto de partida, la familia de grafos "k-completos". Un grafo de esta G de esta familia tiene k nodos de grado $n-1$ y $n-k$ nodos de grado k. Dependiendo del valor elegido de k, la cantidad de ejes de G irá desde $n-1$ hasta $\frac{(n-1)*n}{2}$.