Tesis defendida por

Nelson Emmanuel Ordóñez Guillén

y aprobada por el siguiente comité

Dr. Israel Marck Martínez Pérez

Director del Comité

Dr. Carlos Alberto Brizuela Rodríguez

Miembro del Comité

Dr. Miguel Ángel Alonso Arévalo

Miembro del Comité

Dr. José Antonio García Macías

Coordinador del Programa de Posgrado en Ciencias de la Computación

---

Dr. Jesús Favela Vara

Director de Estudios de Posgrado

Febrero de 2014

CENTRO DE INVESTIGACIÓN CIENTÍFICA Y DE EDUCACIÓN SUPERIOR DE ENSENADA

Programa de Posgrado en Ciencias

en Ciencias de la Computación

Heurística para el problema MAX-CLIQUE basada en un modelo de cómputo con ADN utilizando GPU's

Tesis

para cubrir parcialmente los requisitos necesarios para obtener el grado de

Maestro en Ciencias

Presenta:

Nelson Emmanuel Ordóñez Guillén

Ensenada, Baja California, México

2014.

Resumen de la tesis de Nelson Emmanuel Ordóñez Guillén, presentada como requisito parcial para la obtención del grado de Maestro en Ciencias en Ciencias de la Computación.

Heurística para el problema MAX-CLIQUE basada en un modelo de cómputo con ADN utilizando GPU's

Resumen aprobado por:

Dr. Israel Marck Martínez Pérez

Director de Tesis

En sus inicios, los modelos de cómputo biomolecular fueron concebidos como un nuevo paradigma para enfrentar los problemas complejos aprovechando la gran densidad de información de las moléculas biológicas (principalmente ADN) y el paralelismo masivo de sus bio-operaciones. El modelo de etiquetas pertenece a estos modelos y es uno de los más pródigos en cuanto al desarrollo de algoritmos que abordan problemas de la clase NP-Completo. Uno de estos algoritmos es el llamado k-Cliques, que se utiliza para resolver el Problema del Clique Máximo (MCP, por sus siglas en inglés). Sin embargo, a pesar de sus ventajas, es ampliamente conocido que los algoritmos de cómputo biomolecular aplicados a tal clase de problemas, presentan algunos inconvenientes que evidencian su inviabilidad práctica. Estos inconvenientes son principalmente dos: las bio-operaciones son propensas a errores y la entrada (cantidad de ADN) aumenta de manera exponencial respecto al tamaño del problema. Una alternativa poco explorada que permite franquear el primero de estos obstáculos, consiste en la implementación *in silico* de los algoritmos de cómputo biomolecular en arquitecturas paralelas. El segundo de estos impedimentos se puede atacar combinando estos algoritmos con heurísticas que les proporcionen entradas de tamaño manejable. Tomando esto en consideración, la principal contribución del presente trabajo radica en el desarrollo de una heurística para el MCP, basada en la codificación de la entrada y el filtrado paralelo del algoritmo k-Cliques. En su parte central, esta heurística consiste en la generación y filtrado de tubos de prueba. Cada tubo de prueba contiene un subconjunto del espacio de soluciones completo y se genera tomando subconjuntos de cliques encontrados previamente. El paralelismo masivo de hebras en el filtrado del algoritmo k-Cliques se replica utilizando el paralelismo masivo de hilos de CUDA. Para verificar su desempeño, se realizaron pruebas sobre el conjunto de casos DIMACS. Los resultados obtenidos se comparan con un algoritmo *ramifica y acota* del estado-del-arte e indican competitividad en casos de prueba de tamaño moderado a grande. Además, una de las principales ventajas del enfoque propuesto, es que permite la obtención de más de una solución en algunos problemas, lo cual es una característica no encontrada en ningún otro trabajo previo de su tipo.

Palabras Clave: **Cómputo con ADN, Modelo de Etiquetas, Filtrado Paralelo, Problema del Clique Máximo, Heurísticas, CUDA.**

Abstract of the thesis presented by Nelson Emmanuel Ordóñez Guillén, in partial fulfillment of the requirements of the degree of Master in Sciences in Computer Science.

Heuristic for the MAX-CLIQUE problem based on a DNA computing model using GPU's

Abstract approved by:

Dr. Israel Marck Martínez Pérez

Thesis Director

Biomolecular computing models were initially conceived as a new paradigm to face hard problems by taking advantage from the high-information density existing in biological molecules (mainly in DNA) and massively parallel bio-operations. The sticker model is one of the most popular DNA-based models in the development of algorithms for NP-Complete problems. One of these algorithms, is the so-called k-Cliques sticker algorithm which is used for the Maximum Clique Problem (MCP). However, it has been widely known that the application of DNA computing algorithms for solving such class of problems exhibits two practical limitations: error-prone bio-operations and exponential growth of the input (amount of DNA) with respect to the size of the problem. A scarcely explored alternative which could be used to overcome the inaccuracy of bio-operations, consists of the *in silico* implementation of DNA-based algorithms using parallel architectures. Aditionally, the scaling-up issue may be tackled combining these algorithms with heuristics in order to provide them with inputs of a manageable size. Thus, the main contribution of this work resides precisely in the development of a heuristic for the MCP, based on the input coding and parallel filtering of the k-Cliques algorithm. In this heuristic, test tubes containing sets of candidate solutions are iteratively generated and filtered out to detect current solutions, where each set is in turn, constructed in an iterative fashion taking subsets from a previously found clique. The parallel filtering strategy of the k-Cliques algorithm is replicated by harnessing the massive threading and parallelism of CUDA. For comparison purposes, computational experiments were performed over DIMACS benchmark set and contrasted with those obtained in a state-of-the-art *branch-and-bound* algorithm. Experimental results show that the heuristic introduced here is competitive on instances ranging from moderate to large size. Moreover, a key feature of this approach resides in the generation of multiple solutions for some instances, a property not previously found in the literature.

Keywords: **DNA Computing, Sticker Model, Parallel Filtering, Maximum Clique Problem, Heuristics, CUDA.**

A mi madre, sigue iluminando mi camino...

Agradecimientos

A Dios, origen y destino, principio y fin, entrada y salida en este algoritmo de la vida.

A mi padre, de quien tomo como ejemplos la dedicación, tenacidad, deseos de superación y el salir adelante a pesar de las adversidades.

A mi familia, por estar siempre ahí para mí.

A mi director de tesis, Dr. Israel Marck Martínez Pérez, por la confianza depositada en mí y por la orientación proporcionada a lo largo de la tesis.

A los integrantes de mi comité de tesis, Dr. Carlos Alberto Brizuela Rodríguez y Dr. Miguel Ángel Alonso Arévalo, por todo el tiempo dedicado y por sus valiosos comentarios, consejos y observaciones que guiaron el desarrollo del presente trabajo.

A los investigadores del departamento de Ciencias de la Computación, por su gran calidad tanto académica como humana, y por todos los conocimientos que adquirí de ellos.

A mis compañeros, por todo lo que hemos compartido en estos años, tantos momentos y experiencias que quedarán indeleblemente impresos en mis recuerdos.

A mis amigos, quienes son parte fundamental en mi existencia.

Al CICESE por todas las atenciones brindadas durante este tiempo.

Al CONACyT por proporcionar los recursos económicos necesarios para la culminación del posgrado.

Contenido

Página

Página

Página

Lista de figuras

Figura Página

Figura Página

Lista de tablas

1.1. Antecedentes y motivación

A finales de los años cincuenta, Richard Feynman, famoso físico estadounidense, impartió la conferencia titulada "*There is Plenty of Room at the Bottom*", ante la Sociedad Americana de Física, en el Instituto Tecnológico de California (Feynman, 1960). En tal charla, Feynman introdujo la noción de utilizar estructuras biológicas, tales como células vivas y complejos moleculares para la construcción de computadoras submicroscópicas. Con esta revolucionaria visión, Feynman sentó las bases para el desarrollo del área del Cómputo Biomolecular. El Cómputo Biomolecular es un área de estudio relativamente nueva que implementa esta idea de Feynman: la información es codificada en moléculas biológicas y se aplican técnicas de laboratorio de Biología Molecular para modificar esta información y producir un resultado. Leonard Adleman es considerado el precursor del área de Cómputo Biomolecular, al demostrar la viabilidad experimental de este enfoque; resolviendo un pequeño caso específico del problema de rutas Hamiltonianas con moléculas de ADN (Adleman, 1994). Este logro de Adleman, provocó el interés del estudio dentro del área y surgieron nuevos modelos de cómputo basados en ADN. Algunos de estos modelos son: el modelo de Lipton (Lipton, 1995), el modelo de enzimas de restricción (Ouyang *et al.*, 1997), el modelo de etiquetas (*sticker*) (Roweis *et al.*, 1998), el modelo de cómputo basado en superficies (Smith *et al.*, 1998), el modelo de auto ensamblamiento (Winfree *et al.*, 1998) y el modelo de formación de horquillas (*hairpin*) (Sakamoto *et al.*, 2000). El modelo de etiquetas representa cadenas binarias utilizando hebras de ADN y define cinco operaciones que actúan concurrentemente sobre todas las hebras contenidas dentro de un tubo. Es un modelo de cómputo universal (Kari *et al.*, 1998). Dado que es un modelo no autónomo, masivamente paralelo que simula la operación de una máquina de registros vertical, resulta adecuado para su implementación *in silico*, particularmente en arquitecturas paralelas (Martínez-Pérez y Zimmermann, 2009). En la literatura existe una gran variedad de trabajos que utilizan el modelo de etiquetas principalmente enfocados en la solución de problemas NP-Completos (en el Capítulo 3 se presenta una revisión). Sin embargo, la mayoría de estos artículos presentan algoritmos que son meramente teóricos o son implementados en laboratorio para problemas de tamaños pequeños. Además, la imple-

mentación en arquitecturas paralelas de algoritmos basados en este modelo se ha explorado muy poco. Tomando esto en consideración, el presente trabajo de investigación toma como referencia principal lo propuesto en (Martínez-Pérez y Zimmermann, 2009) donde presentan algoritmos para problemas NP-Completos haciendo énfasis en su implementación *in silico*, y en (Martínez-Pérez *et al.*, 2008) donde implementan las operaciones del modelo de etiquetas en arquitecturas FPGA y combinan la exactitud del filtrado de un algoritmo molecular, con una heurística de generación del espacio de soluciones para resolver el problema de clique máximo. De esta forma, la presente tesis se enfoca en primer término, en el desarrollo de una plataforma para la implementación de algoritmos del modelo de etiquetas, utilizando CUDA para representar el paralelismo de las operaciones del modelo. En segundo término, desarrollar una heurística para la generación del espacio de soluciones basado en el modelo de etiquetas y combinarla con un algoritmo molecular para el filtrado paralelo de las soluciones para el problema de clique máximo.

1.2. Planteamiento del problema

En la mayoría de los modelos de cómputo biomolecular, las implementaciones experimentales *in vitro* de los algoritmos son propensas a errores; dado que los equipos, mecanismos y técnicas de laboratorio existentes en la actualidad, no permiten llevar a cabo con precisión algunas de las operaciones biológicas que se consideran en tales modelos, de forma que tales operaciones no producen el resultado esperado. Más aún, muchos de los materiales y equipos necesarios son caros o poco disponibles. Estas limitaciones dificultan el estudio de los modelos de cómputo biomolecular.

Las implementaciones *in silico* son simulaciones[1] en medios electrónicos de las operaciones biológicas de los algoritmos de cómputo biomolecular. Tales implementaciones constituyen una alternativa para los experimentos *in vitro*, costosos y propensos a errores. Algunos autores incluyen una implementación *in silico* de sus algoritmos, en lugar de probarlos experimentalmente. Sin embargo, existen pocas implementaciones *in silico* que consideren el paralelismo masivo de los modelos de cómputo biomolecular. Por lo tanto, es necesario el desarrollo de nuevas herramientas de simulación *in silico* que exploten las características de paralelismo

[1] A lo largo de la presente tesis, el término *simulación* es utilizado en un sentido más laxo: llevar a cabo la función lógica de los algoritmos, sin tomar en cuenta los factores físicos, químicos y biológicos involucrados en las operaciones biológicas.

de tales modelos, de manera que su operación sea lo más semejante posible a la forma en que fueron definidos biológicamente.

Por otro lado, los modelos de filtrado paralelo, tales como el modelo de etiquetas, consideran en sus algoritmos una entrada que consiste en el espacio de soluciones completo para un problema combinatorio. Particularmente, el algoritmo k-Cliques introducido en (Martínez-Pérez y Zimmermann, 2009), considera una biblioteca con un número combinatorio de cadenas binarias de longitud n. Esto trae como consecuencia que, aún para problemas de tamaño moderado, la representación del espacio de soluciones completo consuma recursos físicos (memoria, procesamiento) que no están disponibles ni en las más modernas supercomputadoras de la actualidad. Además, la cantidad de posibles soluciones es tan grande que, el utilizar un enfoque meramente aleatorio para la selección de soluciones candidatas, tendría probabilidades cercanas a cero en la búsqueda de soluciones. Por lo tanto, al realizar una implementación *in silico* del algoritmo *k-Cliques* es indispensable considerar estrategias heurísticas para la generación de un conjunto soluciones candidatas.

1.3. Objetivos

1.3.1. Objetivo general

Explorar el desempeño de una implementación *in silico* del paralelismo del modelo de etiquetas en la Arquitectura Unificada de Dispositivos de Cómputo (CUDA, por sus siglas en inglés). Utilizar el filtrado paralelo y la codificación del espacio de soluciones del algoritmo k-Cliques del modelo de etiquetas para diseñar e implementar una heurística para el Problema MAX-CLIQUE.

1.3.2. Objetivos específicos

1. Almacenar las hebras de forma que tomen un bit por región y diseñar funciones que operen a nivel de bit.

2. Implementar las operaciones del modelo de etiquetas de manera secuencial (CPU) y paralela (GPU).

3. Indagar las mejores opciones de filtrado paralelo en CUDA.

4. Obtener resultados experimentales de la aceleración (*speed-up*) del algoritmo k-Cliques en CUDA.

5. Diseñar una estrategia para aproximar los tamaños de clique óptimos.

6. Diseñar una estrategia heurística de generación del espacio de soluciones del algoritmo *k-Cliques* del modelo de etiquetas.

7. Obtener resultados experimentales, en tiempos y calidad de las soluciones, del desempeño de la heurística utilizando casos de prueba de DIMACS.

8. Comparar los resultados contra otras implementaciones.

1.3.3. Preguntas de investigación

De la presente investigación se desprenden las siguientes preguntas:

- ¿Qué tan eficiente es el paralelismo de hilos de CUDA para simular el paralelismo de las operaciones del modelo de etiquetas?
- ¿Cuál es la mejor estrategia para realizar el filtrado del algoritmo *k-Cliques* en CUDA?
- ¿En qué criterios nos podemos basar para la generación de un subespacio de soluciones para el Problema de Clique Máximo, basado en la codificación del algoritmo *k-Cliques*?

1.4. Metodología

Para la consecución de los objetivos planteados, el desarrollo del presente trabajo fue realizado en las siguientes etapas:

- **Revisión de literatura.** En la primera parte se efectuó una inmersión en el área de estudio. Se revisaron los artículos que implementan algoritmos del modelo de etiquetas, particularmente, los enfocados a implementaciones *in silico*. Se estudió a fondo la arquitectura de CUDA y el desarrollo de algunas implementaciones. Se hizo un repaso de las estructuras de datos y su manejo en C++.
- **Diseño y programación de plataforma.** Posteriormente, se hizo el diseño de la interfaz de la plataforma, se diseñaron las funciones necesarias para su funcionamiento

y se planteó la forma de una implementación paralela de las funciones secuenciales. Se implementaron las funciones diseñadas. Las funciones secuenciales fueron programadas utilizando el lenguaje de programación C y las funciones paralelas se implementaron utilizando las extensiones correspondientes del modelo de programación CUDA.

- **Pruebas de funcionamiento (secuencial).** El siguiente paso, una vez completa la plataforma, consistió en implementar el algoritmo k-Cliques y realizar pruebas de su correcto funcionamiento. Se propuso una mejora de tal algoritmo que consistió en la revisión de las adyacencias no existentes en el grafo. Se implementaron cuatro formas de algoritmo k-Cliques secuencial y se obtuvieron resultados experimentales de su desempeño utilizando casos de prueba de DIMACS. Estos resultados se incluyen en la sección de resultados preliminares del Capítulo 4.

- **Pruebas de funcionamiento (paralelo).** Se implementó el algoritmo k-Cliques utilizando las funciones paralelas del modelo de etiquetas. Adicionalmente, se diseñaron diferentes formas alternativas de filtrado en CUDA y se midió su desempeño en tiempo de ejecución, sobre diferentes casos de DIMACS. Con lo anterior, se obtuvieron resultados preliminares del desempeño de tres diferentes formas de filtrado paralelo y las aceleraciones sobre las diferentes implementaciones secuenciales.

- **Diseño y programación de estrategia de aproximación de cliques.** Se diseñó e implementó un algoritmo de búsqueda de cliques iniciales. Este algoritmo genera un conjunto de cliques iniciales de tamaño aproximado al máximo, que son procesados en una etapa posterior.

- **Diseño y programación de estrategia de generación de espacios de búsqueda.** Se diseñó una estrategia heurística para la generación de tubos de prueba que contienen espacios de búsqueda.

- **Pruebas exhaustivas.** En la etapa final, se realizaron pruebas de ejecución de la heurística para obtener resultados de desempeño en tiempos de ejecución y de calidad de soluciones. Se utilizaron 66 diferentes casos de prueba de DIMACS. Se realizaron 10 ejecuciones para cada caso de prueba registrando la mejor (la que obtiene los mejores resultados de acuerdo a ciertos criterios) y el promedio.

1.5. Organización de la tesis

La estructura de la presente tesis es como se describe a continuación: el Capítulo 1 consiste en una introducción al tema del cómputo biomolecular, motivación de la tesis y objetivos de la investigación. El Capítulo 2 presenta conceptos matemáticos y computacionales manejados a lo largo del documento, se define el problema de clique máximo y se presentan algunos trabajos previos en el tema. Además, se proporciona una breve introducción a la arquitectura de CUDA. El Capítulo 3 incluye los fundamentos biológicos para entender la operación de los modelos de cómputo biomolecular, se describen los trabajos seminales del área así como el modelo de etiquetas y trabajo previo relacionado. El Capítulo 4 aborda los aspectos de la implementación de la plataforma de simulación. Se presentan las consideraciones hechas al diseñar las funciones necesarias para la operación de dicha plataforma. En el Capítulo 5 se detalla la operación de la heurística para el problema de clique máximo propuesta en la presente tesis. En el Capítulo 6 se presentan los resultados experimentales preliminares y finales obtenidos. Por último, en el Capítulo 7 se discuten las conclusiones obtenidas y se presentan algunas perspectivas de trabajo futuro.

Capítulo 2. Fundamentos matemáticos y computacionales

En este capítulo se introducen los conceptos matemáticos y computacionales básicos que se manejan a lo largo de la presente tesis. Comprende definiciones de teoría de conjuntos, teoría de grafos, complejidad computacional, Problema de Clique Máximo y una introducción a CUDA. La bibliografía principal consultada para el desarrollo del presente capítulo incluye las siguientes refrencias: Cormen *et al.* (2001), Garey y Johnson (1990), Kreher y Stinson (1999), Bomze *et al.* (1999), Nickolls *et al.* (2008), Sanders y Kandrot (2010), NVIDIA Corporation (2013) y Kirk y Hwu (2010).

2.1. Conjuntos, permutaciones y combinaciones

Un **conjunto** es una colección de objetos distintos entre sí. A estos objetos se les llama elementos del conjunto. Para describir conjuntos, se utiliza la notación: $A = \{2, 3, 5, 8, 11\}$. Lo cual significa que el conjunto A consiste de los elementos 2, 3, 5, 8 y 11. No se considera el orden de los elementos en los conjuntos, de esta forma el conjunto $\{7, 2, 4, 1\}$ es equivalente al conjunto $\{4, 1, 7, 2\}$. La **cardinalidad** de un conjunto A denotada como $|A|$ es la cantidad de elementos en el conjunto. Se escribe $a \in A$ para indicar que a es un elemento del conjunto A, y $a \notin A$ para indicar lo contrario. Un **conjunto vacío** es el que tiene cardinalidad cero, y se denota con el símbolo $\emptyset$. Para un entero no negativo k, un *k-conjunto* es un conjunto de cardinalidad k.

Sean A y B dos conjuntos. Se dice que A es **subconjunto** de B, si cada elemento de A es también elemento de B, es decir, si se cumple la condición $a \in A \implies a \in B$. Cuando A es subconjunto de B se escribe $A \subseteq B$. Un *k-subconjunto* de A es un subconjunto que contiene k elementos de A. $A = B$ si $A \subseteq B$ y $B \subseteq A$, es decir, si se cumplen las condiciones $a \in A \implies a \in B$ y $b \in B \implies b \in A$. A es un *subconjunto propio* de B (se denota $A \subset B$), si A es subconjunto de B, pero $A \neq B$. El *producto cartesiano* (o *producto cruz*) de A y B se denota $A \times B$, es el conjunto de todos los pares ordenados (a, b) donde a es elemento de A y b es elemento de B. De manera formal $A \times B = \{(a, b) | a \in A \text{ y } b \in B\}$. El *conjunto potencia* de A se denota $\mathcal{P}(A)$ y es el conjunto de todos los k-subconjuntos de A, con $0 \leq k \leq |A|$. Si $|A| = n$, entonces $|\mathcal{P}(A)| = 2^n$. Una *relación* de A hacia B es un

conjunto que relaciona elementos de A con elementos de B. En otras palabras, una relación es un subconjunto $R \subseteq A \times B$. Una relación sobre A es un subconjunto $R \subseteq A \times A$.

Una **lista** es una colección ordenada de objetos llamados ítems de la lista. Para describir listas se utiliza la notación $X = [2, 6, 2, 7]$, lo que significa que la lista X contiene los ítems 2, 6, 2 y 7, en ese orden. Al ser una estructura ordenada, se tiene que $[2, 6, 2, 7] \neq [2, 2, 6, 7]$, por ejemplo. También cabe señalar que una lista puede contener ítems repetidos. La **longitud** de una lista X es el número de ítems que contiene. Una lista vacía es la que tiene longitud cero y se denota por $[\,]$. Para un entero no negativo n, una *n-tupla* es una lista de longitud n.

Si A es un conjunto de n elementos, entonces una *permutación* de A es una lista π de longitud n de manera que cada elemento de A aparece exactamente una vez en la lista π. Hay exactamente $n! = n \times (n-1) \times \cdots \times 1$ permutaciones de un n-conjunto. Para un entero positivo $k < n$, una *k-permutación* es una lista π de longitud k de forma que cada elemento de A aparece a lo más una vez en la lista π. Hay exactamente

$$\frac{n!}{(n-k)!} = n \times (n-1) \times \cdots \times (n-k+1) \tag{1}$$

k-permutaciones de un n-conjunto. De manera similar, para un entero positivo k, una *k-combinación* de A es un subconjunto $S \subseteq A$ de cardinalidad k. Hay exactamente

$$\frac{n!}{k!(n-k)!} \tag{2}$$

k-combinaciones de un n-conjunto.

2.2. Grafos

2.2.1. Definiciones básicas

Un grafo G es un par de conjuntos (V, E) donde V es un conjunto no vacío ($V \neq \emptyset$), y E es un conjunto (posiblemente vacío) que relaciona elementos de V, es decir, el conjunto E es una relación sobre V ($E \subseteq \{e_{ij} = (v_i, v_j) | v_i, v_j \in V, i \neq j\}$). Si se considera el orden en los pares, es decir, $(v_i, v_j) \neq (v_j, v_i)$ se trata de un grafo *dirigido*, si $(v_i, v_j) = (v_j, v_i)$, hablamos de un grafo *no dirigido*. Los elementos de V se llaman *vértices* y los elementos de

E se denominan *aristas*. Se escribe $V(G)$ y $E(G)$ para referenciar los vértices y las aristas del grafo G, respectivamente, o simplemente V y E, si el grafo G se conoce por contexto. Usualmente, se denota el número de vértices $|V|$ en un grafo con n y el número de aristas $|E|$ con m.

Las cantidades n y m corresponden al *orden* y el *tamaño* del grafo, respectivamente. Si $e_{ij} \in E$, entonces v_i, v_j son *adyacentes* y *terminales* de su arista *incidente* e_{ij}. Dos o más aristas son *adyacentes* cuando son incidentes a un mismo vértice. Las aristas que tienen los mismos vértices terminales son *paralelas*. Una arista incidente a un mismo vértice es un *bucle*. Un grafo es *simple* si no contiene aristas paralelas ni bucles. Un grafo que contiene aristas paralelas es llamado *multigrafo*. Un grafo que contiene bucles es un *pseudografo*. Un grafo sin aristas ($E = \emptyset$) es un *grafo vacío*. También se considera el concepto de un grafo sin vértices (y por lo tanto, sin aristas) el cual es llamado *grafo nulo*. Un grafo con un solo vértice es un *grafo trivial*. El *grado* de un vértice v_i, denotado por $d(v_i)$, es el número de aristas incidentes a v_i. El grado mínimo de los vértices en el grafo se denota $\delta(v_i)$ y el máximo $\Delta(v_i)$. Un vértice con grado igual a 1 es un vértice *colgante*. Una arista es colgante si tiene un vértice colgante. Un vértice con grado igual a cero es un vértice *aislado*. Si todos los vértices en el grafo tienen un mismo grado k, entonces se dice que el grafo es k-regular. Un grafo es *completo* si existe una arista para cada par de vértices del grafo. Dicho de otra forma, un grafo es completo si es $(n-1)$-regular. Un grafo puede ser finito o infinito, dependiendo si el conjunto de vértices es finito o infinito. En el presente texto consideramos solamente grafos finitos.

2.2.2. Representación de grafos

Un grafo $G = (V, E)$ está definido unívocamente por sus conjuntos V y E. Se puede representar en un diagrama dibujando los vértices como puntos en el espacio y las aristas como líneas que conectan estos puntos. En la Figura 1 se muestra un ejemplo de un grafo de 5 vértices. Para almacenar la información de un grafo, típicamente se utilizan dos representaciones: *listas de adyacencia* y *matriz de adyacencia*. En la primera, se mantiene una lista L_i para cada vértice v_i del grafo. En cada lista L_i se almacena el conjunto de vértices adyacentes a v_i. La segunda consiste en una matriz de $n \times n$, $A = [a_{ij}]$, donde $a_{ij} = 1$ si $(v_i, v_j) \in E$ y $a_{ij} = 0$, en caso contrario. La Figura 2 muestra la representación del grafo de la Figura 1 tanto en listas de adyacencia como matricial.

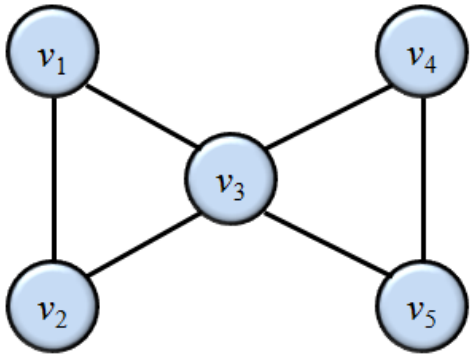


Figura 1: Un grafo G con $V = \{v_1, v_2, v_3, v_4, v_5\}$ y $E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_4), (v_3, v_5), (v_4, v_5)\}$.

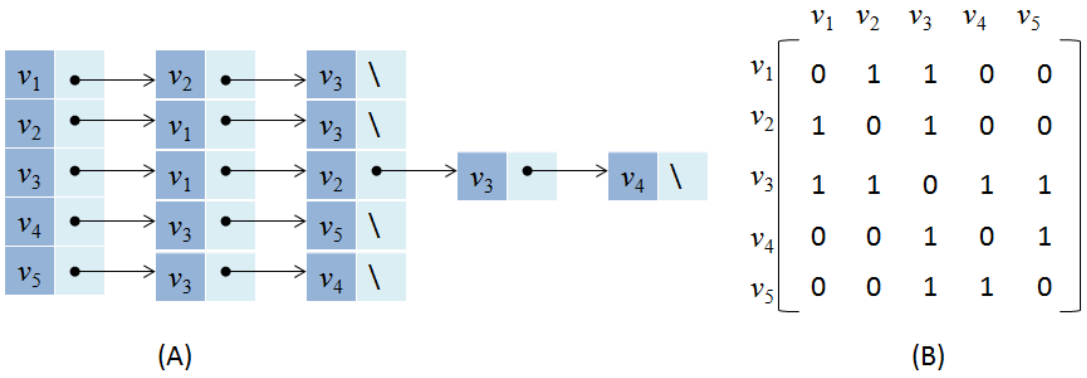


Figura 2: Representación del grafo de la Figura 1. (A) Listas de adyacencia. (B) Matriz de adyacencia.

2.2.3. Subgrafos

Dado un grafo no dirigido $G = (V, E)$, un subgrafo de G es un grafo $G' = (V', E')$ tal que cada vértice de G' es un vértice de G, y cada arista de G' es también una arista de G. En otras palabras, $V(G') \subseteq V(G)$ y $E(G') \subseteq E(G)$.

2.2.4. Grafo complemento

Dado un grafo no dirigido $G = (V, E)$, el grafo complemento de G es $\overline{G} = (V, \overline{E})$, donde $\overline{E} = \{(u, v) | u, v \in V, u \neq v, \text{ y } (u, v) \notin E\}$. En otras palabras, $\overline{G}$ es el grafo que contiene los mismos vértices que G, pero solamente las aristas que no están en G.

2.3. Complejidad computacional

2.3.1. Notación asintótica

La notación asintótica proporciona formas de expresar funciones en términos de cotas superiores e inferiores. Las notaciones más utilizadas son O, Ω y Θ. Sean dos funciones $f(n)$ y $g(n)$. Se dice que

- $f(n) = O(g(n))$ cuando para alguna elección de constantes c y n_0, $c \cdot g(n)$ es una cota superior de $f(n)$, es decir, $0 \leq f(n) \leq c \cdot g(n)$, $\forall n > n_0$ (Figura 3(B)).
- $f(n) = \Omega(g(n))$ cuando para alguna elección de constantes c y n_0, $c \cdot g(n)$ es una cota inferior de $f(n)$, es decir, $0 \leq c \cdot g(n) \leq f(n)$, $\forall n > n_0$ (Figura 3(C)).
- $f(n) = \Theta(g(n))$ cuando para alguna elección de constantes c_1, c_2 y n_0, $c_1 \cdot g(n)$ es una cota inferior y $c_2 \cdot g(n)$ es una cota superior de $f(n)$, es decir, $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$, $\forall n > n_0$ (Figura 3(A)).

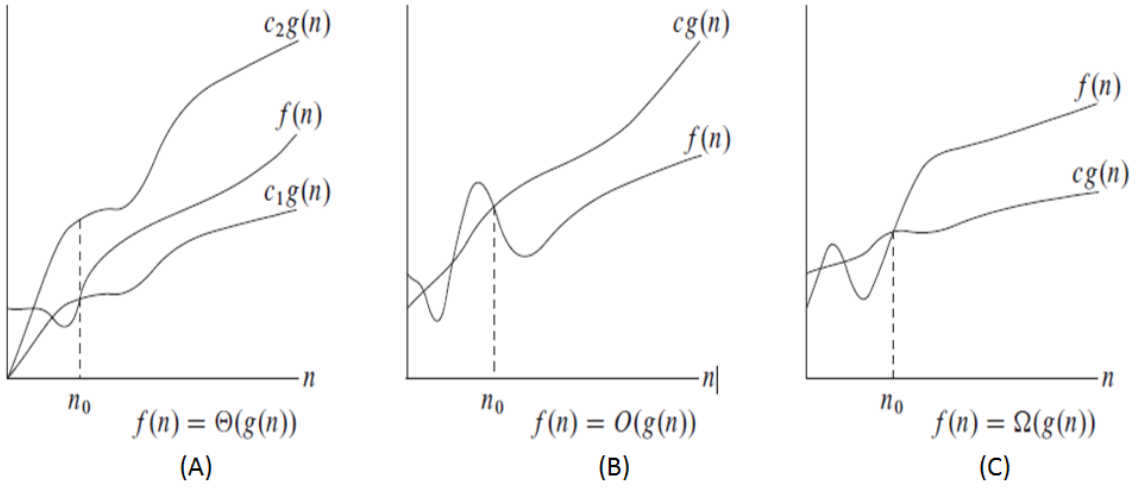


Figura 3: Crecimiento asintótico de las funciones: A) $f(n) = \Theta g(n)$, B) $f(n) = Og(n)$, C) $f(n) = \Omega g(n)$. Imagen extraída de Cormen *et al.* (2001, p. 45).

2.3.2. Tiempo polinomial

Un algoritmo se ejecuta en tiempo polinomial si su tiempo de ejecución T(n), en el peor de los casos, está acotado por arriba por un polinomio en el tamaño de la entrada n, en otras

palabras, si $T(n) = O(n^k)$, para alguna constante k. En tal caso, se dice que el algoritmo es polinomial.

2.3.3. Problemas, algoritmos y complejidad

Los problemas computacionales consisten en preguntas generales que se buscan responder, contienen usualmente parámetros o valores de entrada y definen una salida deseada, que corresponde a una configuración de los parámetros de entrada. Esta configuración que especifica valores particulares para cada parámetro de entrada determina un *caso específico* del problema. La definición formal de un problema computacional debe incluir: una descripción general de sus parámetros, y la relación explícita entre la entrada y la salida, esto es, las propiedades que una respuesta o solución debe satisfacer. El ejemplo clásico que ilustra el concepto anterior es el del problema de *ordenamiento*, el cual se define formalmente como sigue:

Entrada: una secuencia de n números $[a_1, a_2, ..., a_n]$.

Salida: una permutación (reordenamiento) $[a'_1, a'_2, ..., a'_n]$ que satisface la condición $a'_1 \leq a'_2 \leq ... \leq a'_n$.

Un *algoritmo* es un procedimiento computacional bien definido que resuelve un problema computacional bien definido. Un algoritmo construye un valor o conjunto de valores de salida a partir del parámetro o conjunto de parámetros de entrada. En base al ejemplo anterior, un caso específico del problema es, por ejemplo, ordenar la secuencia [3, 2, 5, 9, 1], en el que un algoritmo correcto debe producir la salida [1, 2, 3, 5, 9]. Un algoritmo es correcto cuando para cada caso específico de problema produce una salida correcta, de acuerdo a las especificaciones del problema, y se detiene.

El término complejidad computacional tiene dos connotaciones que están íntimamente relacionadas. En primer término, se refiere a la cantidad de recursos computacionales que un algoritmo requiere. Estos recursos pueden ser: tiempo de ejecución, memoria, ancho de banda, cantidad de procesadores, etc. Usualmente, el tiempo de ejecución es el recurso computacional que es sujeto de análisis en mayor medida. El tiempo de ejecución de un algoritmo se mide en función de las operaciones básicas que éste realiza. Una operación básica es una asignación, una comparación, una operación aritmética o lógica, un acceso a memoria. Las operaciones básicas no incluyen los ciclos, llamadas a funciones y bifurcaciones. En el análisis

de los recursos computacionales que consume un algoritmo, se busca obtener una cantidad de operaciones básicas en función del tamaño del problema. Por ejemplo, si el problema consiste en ordenar un vector de n números, se busca obtener una cantidad $f(n)$ de operaciones básicas para realizarlo.

Por otro lado, de forma más general, el estudio de la complejidad computacional es una rama de las Ciencias Computacionales. Se refiere a establecer clasificaciones para los problemas computacionales de acuerdo a su complejidad, a saber, la dificultad de resolverlos, tomando como métrica la cantidad de resursos computacionales utilizados. De acuerdo a su complejidad, los problemas son asignados a conjuntos llamados *clases de complejidad*. Es de particular interés la demostración que ciertos problemas son *intratables*, es decir, no existe hasta el momento forma alguna de encontrar un algoritmo que encuentre soluciones en tiempo razonable.

2.3.4. Problemas de optimización y problemas de decisión

La gran mayoría de los problemas de interés son problemas de optimización. En este tipo de problemas, las soluciones factibles tienen un valor asociado y se buscan aquellas soluciones que tengan el "mejor" valor, es decir, el valor máximo o mínimo de entre todos los demás. Por otro lado, los problemas de decisión son aquellos en los que se busca responder a una pregunta de forma binaria, es decir, se espera un "Sí" o un "No" como respuesta (o, equivalentemente 1 o 0). Aunque el estudio de la complejidad computacional se refiere únicamente a problemas de decisión, existe una relación estrecha entre ambos tipos de problemas. Así, cada problema de optimización tiene su correspondiente versión como problema de decisión.

2.3.5. Reducción de problemas

Sean A y B, dos problemas de decisión. Se dice que A se *reduce polinomialmente* a B, si existe un algoritmo que traduzca cada caso específico del problema A, a un caso específico del problema B. El algoritmo utilizado para la traducción debe ser polinomial y la respuesta de cada caso específico del problema B debe coincidir con la respuesta del caso específico correspondiente del problema A. Cuando tal reducción es posible, se denota por $A \leq_P B$.

2.3.6. Clases de complejidad

En este estudio, se consideran cuatro clases de complejidad para los problemas de decisión: *P*, *NP*, *NP*-Completo y *NP*-Difícil. La clase de complejidad *P* contiene los problemas que se pueden resolver en tiempo polinomial en el tamaño de la entrada del problema. La clase *NP* agrupa aquellos problemas que son "verificables" en tiempo polinomial. Esto significa que, dado un certificado de solución del problema, se puede diseñar un algoritmo polinomial que compruebe que este certificado cumple con los requerimientos para ser efectivamente una solución. Entonces, por definición se sabe que P es subconjunto de *NP*. La cuestión que queda por resolver y que actualmente constituye uno de los grandes problemas de las Ciencias Computacionales es si $P \subset NP$, o no.

La clase *NP*-Completa incluye los problemas que pertenecen a *NP* y que son al menos tan difíciles como cualquier otro problema en la clase *NP*. La demostración de esta cota en la dificultad de un problema A, se realiza comprobando que un problema B, el cual se sabe pertenece a *NP*-Completo, se reduce polinomialmente al problema A. Por último, la clase *NP*-Difícil contiene problemas para los cuales se ha demostrado la cota en dificultad con otros problemas de su misma clase, pero que no pertenecen a la clase *NP*.

2.4. Problema de Clique Máximo

Dado un grafo $G = (V, E)$, un *clique* es un subconjunto $C \subseteq V$ de manera que existe una arista en E para cualquier par de vértices de C. Dicho de otra forma, un clique es un subgrafo completo de G. Un clique es de tamaño *maximal* si no puede ser extendido agregando otro vértice o, de manera equivalente, no es subconjunto de ningún otro clique. Por ejemplo, el grafo de la Figura 4 tiene los cliques maximales $\{v_1, v_2, v_3, v_4\}, \{v_3, v_4, v_6\}, \{v_3, v_5\}$ y $\{v_4, v_7\}$. Un clique es de tamaño máximo si es el que tiene la mayor cantidad de vértices de los cliques maximales de un grafo, tal como $\{v_1, v_2, v_3, v_4\}$, en el ejemplo anterior. En general, un grafo puede tener más de un clique de tamaño máximo. El problema de clique máximo (MCP) consiste en encontrar los cliques de tamaño máximo en un grafo.

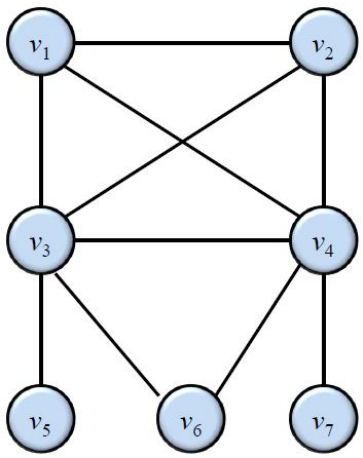


Figura 4: Grafo con diferentes cliques maximales ($\{v_1, v_2, v_3, v_4\}$, $\{v_3, v_4, v_6\}$, $\{v_3, v_5\}$, $\{v_4, v_7\}$).

2.4.1. Métodos de búsqueda de soluciones para el MCP

El problema de clique máximo es uno de los primeros problemas para los que fue demostrada su pertenencia a la clase *NP*-completo (Karp, 1972). Esto significa que a menos que *P*=*NP*, los algoritmos que lo resuelvan de manera exacta tomarán un tiempo que aumenta de manera exponencial en la cantidad de vértices del grafo. Incluso, en (Hastad, 1996) se demuestra que no es posible aproximar, en tiempo polinomial, el clique máximo de un grafo arbitrario dentro de un factor de $n^{1-\epsilon}$, para cualquier $\epsilon > 0$. Aún así, existe una gran variedad de artículos que presentan alternativas de solución para el problema de clique máximo. De manera general, estas alternativas se dividen en métodos exactos (para casos especiales o pequeños) y heurísticas. Las heurísticas son métodos que no garantizan una calidad específica de solución. Dado que a menudo proporcionan soluciones aproximadas al óptimo, en la literatura se refieren como métodos aproximados (aunque esto puede ser inapropiado, dado que existen los algoritmos de aproximación los cuales sí garantizan cierta calidad aproximada de solución). Un estudio extenso sobre las diferentes estrategias de solución, además de las aplicaciones del problema de clique máximo se puede encontrar en (Bomze *et al.*, 1999). Entre las diferentes estrategias heurísticas para abordar el problema de clique máximo se incluyen algoritmos basados en redes neuronales (Grossman y Jagota, 1993) y (Jagota *et al.*, 1996), recocido simulado (Aarts y Korst, 1989), (Jerrum, 1992) y (Homer y Peinado, 1996), algoritmos genéticos (Hifi, 1997) y (Marchiori, 1998), búsqueda tabú (Soriano y Gendreau, 1996). La estrategia de ramificar y acotar (*branch-and-bound*) ha sido una de las que más se

ha trabajado y que ha demostrado ser de las más veloces. Uno de los primeros algoritmos ramifica y acota es (Carraghan y Pardalos, 1990). Su idea principal es cortar ramificaciones utilizando como cota el tamaño máximo de clique en un conjunto de vértices candidatos. Además, considera un ordenamiento especial de los vértices para reducir la extensión del árbol de búsqueda. Otro algoritmo es el propuesto en (Fahle, 2002) donde se aplica una estrategia de cota más eficiente. Aquí se utiliza el número cromático como una cota superior del tamaño de clique máximo. Para esto, se utiliza una heurística para el coloreado de un grafo de vértices candidatos. Entonces, se toma el número de colores como una cota superior del tamaño de clique máximo. También utiliza técnicas de filtrado basadas en costos y de acuerdo a ciertas observaciones en el conjunto de vértices candidatos. Los algoritmos MCQ, MCR y MCS encontrados en (Tomita y Seki, 2003), (Tomita y Kameda, 2007) y (Tomita *et al.*, 2010), respectivamente, son versiones muy parecidas entre sí que utilizan la idea de coloreo de grafos no solo con propósitos de acotar sino también como criterios de ramificación. MCR y MCS son versiones mejoradas de MCQ. La única diferencia entre MCQ y MCR consiste en el ordenamiento inicial de los vértices, MCQ en orden no creciente de los grados mientras que MCR utiliza el ordenamiento especial de (Carraghan y Pardalos, 1990). MCS agrega además una nueva rutina que colorea un vértice de color más grande con otro más pequeño. El trabajo más reciente encontrado en la literatura es (Batsyn *et al.*, 2013). Aquí se mejora el algoritmo MCS de (Tomita *et al.*, 2010) utilizando una heurística de búsqueda local iterada (ILS) introducida en (Andrade *et al.*, 2012) para la búsqueda de una solución inicial de alta calidad para acotar ramas. Esto produce una reducción en el tamaño del árbol de búsqueda especialmente en grafos grandes y densos.

2.5. Arquitecturas de cómputo paralelo

El paralelismo es la característica que presentan algunos problemas de poder dividirse en una cantidad de tareas o subproblemas independientes que pueden ser llevados a cabo de manera simultánea. En el sentido más simple, el cómputo paralelo es el uso de diferentes recursos computacionales para la solución de un problema computacional que exhibe paralelismo. El problema es dividido en partes discretas que pueden ser resueltas concurrentemente. Para cada parte, se diseña una serie de instrucciones que se ejecutan simultáneamente en diferentes elementos de procesamiento.

En general, las arquitecturas de cómputo paralelo se dividen por la forma en la que accesan los datos en memoria, existiendo tres grupos: sistemas de memoria compartida, sistemas de memoria distribuída y sistemas híbridos. En el primer grupo, la memoria se comparte a través de un espacio de direcciones global, accesible para todos los procesadores. Los procesadores operan de manera independiente pero comparten los mismos recursos de memoria. Los cambios en alguna localidad son visibles para todos los procesadores. En el segundo grupo, cada procesador tiene su espacio de memoria y este espacio solo es compartido a través de un ducto de comunicaciones. La comunicación de datos entre procesadores debe ser definida explícitamente por los programadores, así como la sincronización de las tareas. Finalmente, los sistemas híbridos son los que emplean tanto arquitecturas de memoria compartida como distribuída. Este grupo comprende las computadoras más veloces actualmente, al integrar las capacidades de ambas arquitecturas.

El paralelismo puede ser soportado en diferentes niveles en el hardware. Bajo esta noción, las computadoras paralelas pueden ser clasificadas en diferentes grupos: cómputo multinúcleo, multiprocesadores simétricos (SMP's), cómputo distribuído, clústers, procesadores masivamente paralelos (MPP's), grids, computadoras paralelas especializadas, cómputo reconfigurable con arreglos de compuertas programables en campo (FPGA's), cómputo de propósito general con unidades de procesamiento gráfico (GPGPU), circuitos integrados de aplicación específica y procesadores vectoriales. Estos grupos no son excluyentes entre sí, de manera que, comúnmente, las computadoras paralelas se clasifican en más de uno de ellos.

2.6. Fundamentos de CUDA

2.6.1. Introducción

Debido a la creciente demanda de aplicaciones que requieren cómputo intensivo, particularmente aplicaciones que requieren procesamiento gráfico tales como los videojuegos, la capacidad de los procesadores gráficos (GPU's) se ha ido incrementando considerablemente en los últimos años. El procesamiento gráfico es una tarea inherentemente paralela, en la que grandes conjuntos de pixeles y vértices son renderizados independientemente unos de los otros. Esta característica ha provocado que las compañías fabricantes de tarjetas gráficas hayan invertido una gran cantidad de recursos para el desarrollo de nuevas y mejores plataformas de hardware, que soporten las cada vez más exigentes necesidades de procesamiento de las nuevas aplicaciones. Como resultado, los nuevos GPU's, son dispositivos especializados en cómputo paralelo e intensivo que han logrado establecer una gran brecha en desempeño en tales aplicaciones, con respecto a los procesadores tradicionales (CPU's) (Figuras 5 y 6).

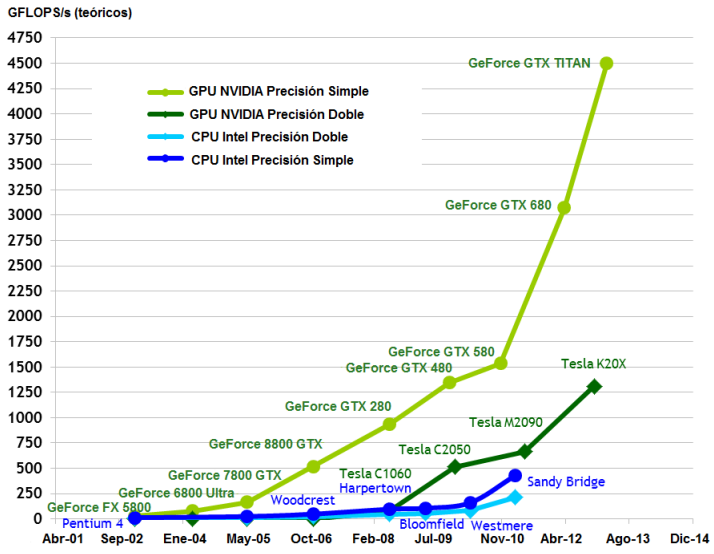


Figura 5: CPU's vs. GPU's. Operaciones de punto flotante por segundo. Imagen adaptada de NVIDIA Corporation (2013, p. 2).

La Arquitectura de Dispositivo de Cómputo Unificado (CUDA, por sus siglas en inglés), lanzada en 2007 por NVidia, es al mismo tiempo una arquitectura física (*hardware*) y un modelo de programación. Como arquitectura física, CUDA consiste en una disposición de los

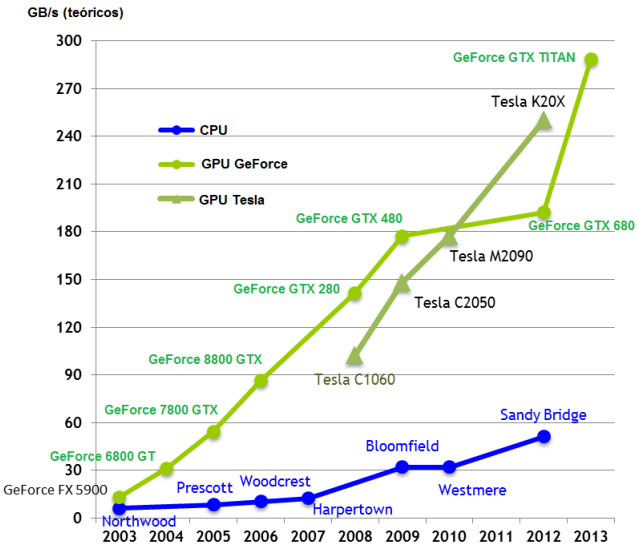


Figura 6: CPU's vs. GPU's. Ancho de banda de memoria. Imagen adaptada de NVIDIA Corporation (2013, p. 3).

componentes de hardware en los GPU's. Los GPU's compatibles con CUDA están conformados por un conjunto de multiprocesadores de flujo (*Streaming Mutiprocessors*-SM) SIMT (*Single Instruction Multiple Thread*), cada uno integrado por varios procesadores escalares (*Scalar Processors*-SP). La Figura 7 muestra la configuración de un GPU que contiene 14 SMs. Cada SM consiste de ocho núcleos escalares SP, dos Unidades de Funciones Especiales (SFU - *Special Function Units*) para funciones trascendentales, una Unidad de Instrucción Multihilo (MT IU - *Multithreaded Instruction Unit*), y memoria *on-chip* compartida.

Como modelo de programación, CUDA es un paradigma que pone las capacidades altamente paralelas de los GPU's a disposición de los programadores. CUDA hace posible la escritura de programas no enfocados exclusivamente en problemas gráficos, sino aplicaciones de cómputo de propósito general (*GPGPU Computing*). A través de algunas extensiones del lenguaje C/C++, los programadores escriben funciones que generan millones de hilos, con miles de ellos ejecutándose de manera concurrente.

2.6.2. El paradigma de CUDA

La idea principal detrás de la filosofía de CUDA es lo que se denomina cómputo heterogéneo. Esto consiste en permitir que el GPU funcione como un co-procesador que se encarga

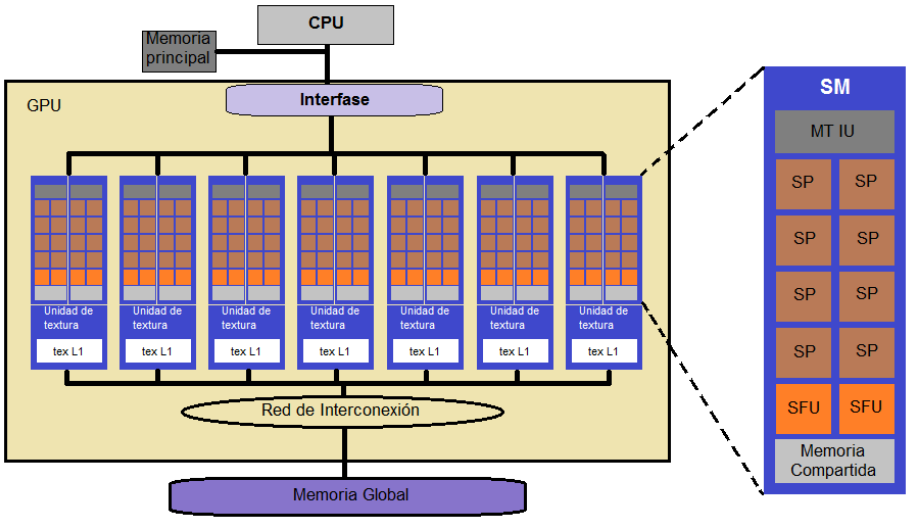


Figura 7: GPU dividido en 14 SMs con 8 SPs cada uno. Imagen basada en Nickolls *et al.* (2008).

de las tareas que exhiben paralelismo, mientras el CPU funge como el procesador maestro a cargo de la ejecución secuencial. Un programa en CUDA se organiza en dos partes: el código que se ejecuta en el CPU, llamado código del anfitrión (*host*); y el código que se ejecuta en el GPU, llamado código del dispositivo (*device*). El código del dispositivo a menudo es una función llamada *kernel* en la terminología de CUDA. Cuando se invoca un *kernel*, se produce un conjunto estructurado de hilos que ejecutan las instrucciones del *kernel* de manera independiente sobre diferentes datos. El flujo típico de un programa en CUDA es mostrado en la Figura 8. El CPU empieza a ejecutar las porciones iniciales de código secuencial. Cuando se va a ejecutar una tarea paralela, se "lanza" un *kernel* que crea un *grid* de hilos independientes. Cuando todos los hilos en el grid terminan su ejecución, el flujo regresa al CPU hasta que una nueva tarea paralela requiera ser llevada a cabo.

2.6.3. Organización de los hilos

Cuando un *kernel* se invoca, se crea un grid de hilos. El grid está formado por bloques, donde cada bloque es un conjunto de hilos que reside físicamente en un procesador. Los hilos dentro de un mismo bloque forman un conjunto fuertemente acoplado que pueden colaborar a través de un espacio de memoria compartido y de sincronización por medio de barreras. Por otro lado, el modelo de programación de CUDA requiere que los hilos de diferentes bloques dentro del grid sean independientes, de forma que puedan ser ejecutados en cualquier

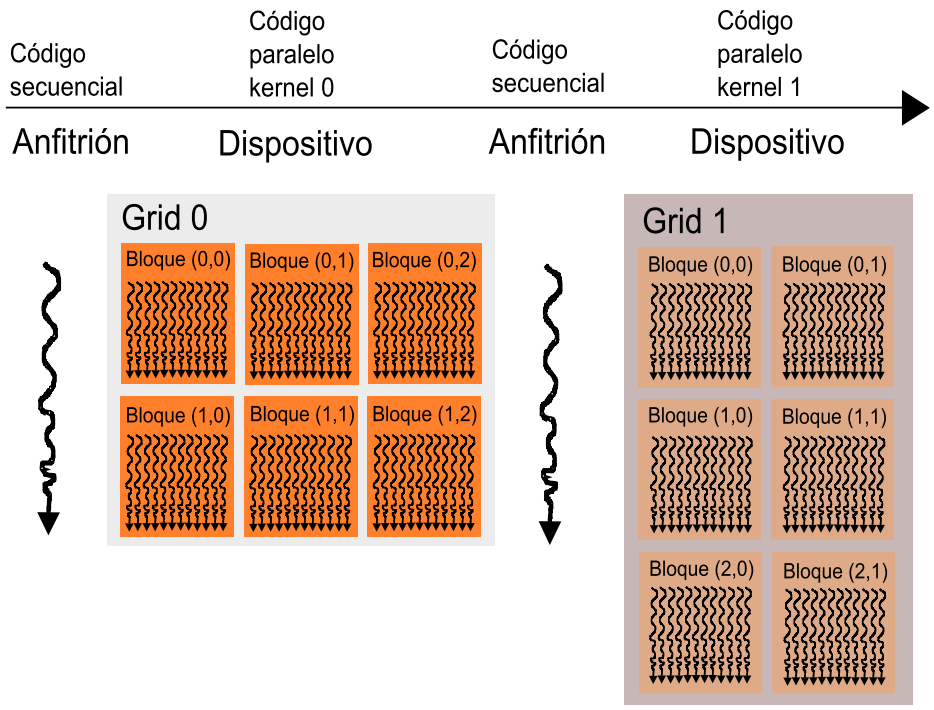


Figura 8: Flujo de un programa en CUDA: La parte secuencial se ejecuta en el CPU mientras que las tareas paralelas se envían al GPU. Imagen basada en NVIDIA Corporation (2013).

orden sobre cualquier cantidad de núcleos, para permitir la escalabilidad transparente del modelo en una cantidad arbitraria de núcleos de procesamiento. Debido a lo anterior, los hilos de diferentes bloques son débilmente acoplados, de forma que sus capacidades de trabajo colaborativo son más limitadas.

Un grid puede ser una estructura unidimensional, bidimensional o tridimensional de bloques. Asimismo, los bloques también pueden contener hilos estructurados en una, dos o tres dimensiones. Todos los bloques dentro del grid tienen las mismas dimensiones. CUDA proporciona variables integradas que permiten la identificación unívoca de cada hilo. Estas variables almacenan las dimensiones del grid, las dimensiones de los bloques, las coordenadas de cada bloque dentro del grid y las coordenadas de cada hilo dentro del bloque. Con estos valores, es posible calcular un índice único que identifique a cada hilo dentro del grid. La Figura 9 describe un grid de 2×3 bloques de 3×5 hilos. Cada hilo y cada bloque tiene sus coordenadas dentro del bloque y dentro del grid, respectivamente.

Las dimensiones del grid y los bloques son definidas por el programador al momento de invocar un *kernel*. Una vez que se ejecuta el *kernel*, el hardware de CUDA se encarga de las tareas de administración de los hilos (creación, calendarización y destrucción). Debido a esto,

se considera que los hilos de CUDA son *ligeros*. Además, el hardware también divide cada bloque en grupos de 32 hilos (normalmente) llamados *warps* que son calendarizados para ejecutarse simultáneamente dentro del procesador. Aunque esto último es transparente para el programador, es importante a la hora de seleccionar las dimensiones del grid y los bloques para maximizar el desempeño del *kernel*.

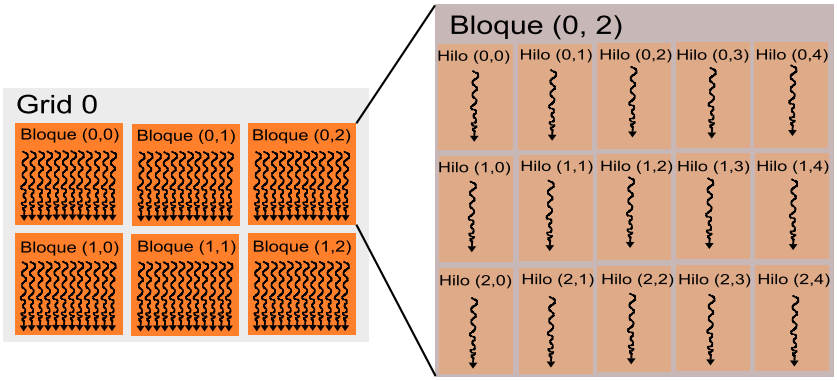


Figura 9: Organización de los hilos en un grid. Imagen basada en NVIDIA Corporation (2013).

2.6.4. Asignación de hilos a datos

Dado que se cuenta con un identificador único para cada hilo, se puede asignar un hilo para que se encargue de un dato particular identificado por su posición en memoria (o su índice dentro de un arreglo). Por ejemplo, suponiendo que la invocación de un grid genera N hilos en total, y que es necesario procesar un conjunto de $M \geq N$ datos independientes, se puede hacer un "mapeo" de hilos a datos de forma que el i-ésimo hilo se encargue de las posiciones $\langle i, i+N, i+2N, ... \rangle$ de datos.

2.6.5. Jerarquía de memorias

Dentro del dispositivo, existen diferentes espacios de memoria que pueden ser accesados por los hilos durante su ejecución. Estos espacios son: memoria global, memoria de constantes, memoria de texturas, memoria compartida, memoria local y registros. La Figura 10 describe los diferentes espacios de almacenamiento. Los registros y la memoria compartida son memorias *on-chip*. Los datos almacenados en estos espacios pueden ser accesados a grandes velocidades y de manera altamente paralela. Los registros son de acceso exclusivo de

cada hilo y su duración es la misma que la vida del hilo, mientras que los datos en memoria compartida están disponibles para todos los hilos de un bloque durante la ejecución del bloque. La memoria local es espacio reservado en memoria global para acceso exclusivo de cada hilo. Se utiliza para vaciado de registros cuando el espacio en registros es insuficiente. La memoria global es la memoria principal del dispositivo a la que todos los hilos tienen acceso. Finalmente, las memorias de constantes y de texturas son memorias caché nivel 2 de sólo lectura. La memoria de constantes es útil cuando muchos hilos requieren leer los mismos datos. La memoria de texturas es un tipo de memoria especializada para aplicaciones gráficas. Los datos que residen en memoria global, de constantes, y de texturas, duran hasta que la aplicación termine o que los espacios sean reasignados a nuevos datos.

Los espacios de memoria en el dispositivo están físicamente separados del anfitrión. Por esto, en el caso de las memorias global, de constantes y de texturas, se debe asignar los espacios y enviar los datos a procesar (cuando se requiera), del anfitrión al dispositivo antes de la ejecución de un *kernel*. Más aún, una vez terminada la ejecución del *kernel*, los datos deben ser regresados del dispositivo al anfitrión (en el caso de la memoria global). Entonces, la comunicación anfitrión-dispositivo-anfitrión, agrega una latencia al tiempo global de procesamiento del *kernel*.

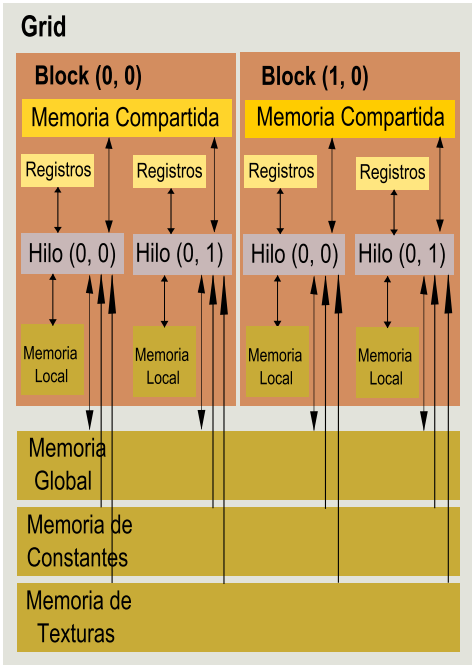


Figura 10: Distintos tipos de memoria en CUDA. Imagen basada en NVIDIA Corporation (2013).

El uso correcto de los distintos tipos de memoria, además de las técnicas de coalescencia de memoria (almacenar en localidades consecutivas los datos que accesan hilos consecutivos), constituyen un factor determinante para alcanzar el máximo desempeño de las capacidades de CUDA. Sin embargo, a excepción de la memoria global, todas las memorias son de capacidad limitada. Por lo tanto, lo contrario también se cumple: el uso indiscriminado de registros y memoria compartida, puede provocar que muy pocos hilos puedan ser clendarizados para ejecutarse concurrentemente dentro del multiprocesador, obteniendo un pobre desempeño como consecuencia.

2.6.6. Limitaciones

A pesar de las capacidades de cómputo y paralelismo del modelo de programación de CUDA, existen algunas restricciones:

- El compilador de CUDA soporta solamente lenguaje C/C++ simple. El uso de las características de programación orientada a objetos de C++ no es soportado en el código de dispositivo.
- Debido a la interacción CPU-GPU de la arquitectura de cómputo heterogéneo, los datos deben ser copiados de la memoria del anfitrión a la memoria del dispositivo. También deben ser copiados de regreso del dispositivo al anfitrión, una vez terminada la operación del *kernel*.
- Un grid solamente puede ser creado en la invocación a un *kernel*, desde el CPU. No es posible invocar a otro *kernel* cuando se está ejecutando código en el GPU, por lo que no se pueden crear más hilos después de la invocación inicial.
- Las funciones *kernel* no regresan ningún valor (directamente).
- La recursión no se permite, debido a la gran cantidad de requerimientos de memoria para los miles de hilos.
- La asignación y liberación de memoria sólo se puede realizar en el anfitrión, antes y después, respectivamente, de la ejecución del *kernel*.
- La memoria compartida sólo es visible para los hilos de un mismo bloque.

- La comunicación y sincronización de hilos sólo es posible para los hilos dentro de un bloque.
- Restricciones de los tipos de variables en memoria global, compartida y de constantes. Estas restricciones consisten en que no se puede utilizar todos los tipos de variables de C/C++ en variables que se alojan en el GPU.

Si bien, estas limitaciones constituyen un obstáculo para el desarrollo de ciertas aplicaciones, el modelo de programación de CUDA continúa siendo uno de los modelos de cómputo paralelo más robustos actualmente.

Capítulo 3. Fundamentos de cómputo biomolecular

En el presente capítulo se presentan los fundamentos biológicos del Cómputo Biomolecular, se revisan los trabajos considerados seminales dentro del área. Se define el modelo de etiquetas y se hace una revisión de la literatura relacionada a tal modelo. La bibliografía más relevante consultada para el desarrollo del capítulo consiste en: Ignatova *et al.* (2008), Amos (2005), Kari *et al.* (2012), Martínez-Pérez (2007), Adleman (1994), Lipton (1995), Ouyang *et al.* (1997) y Roweis *et al.* (1998).

3.1. Conceptos de biología molecular

3.1.1. Estructura del ADN

El modelo de la estructura del ADN se debe a James Watson y Francis Crick, quienes en 1953 descubrieron como está conformada esta macromolécula (Watson y Crick, 1953). Una hebra simple de ADN consiste en una secuencia de cuatro diferentes tipos de unidades llamadas *bases* o *nucleótidos* (abreviado *nt*) encadenadas juntas en una columna vertebral orientada, como si fueran cuentas unidas a un cable. Estas bases están conformadas por un grupo fosfato, un azúcar (desoxirribosa), y una base nitrogenada. Existen cuatro tipos de bases nitrogenadas que se clasifican en dos grupos: en el grupo purinas están la Adenina y la Guanina, mientras que en el grupo pirimidinas se tiene a la Citosina y a la Timina. Cada base se abrevia por su inicial (A, G, C, T). El grupo fosfato y el azúcar de las bases forman una columna vertebral que las une en una secuencia de enlaces covalentes, del grupo fosfato de una base al grupo hidroxilo de la siguiente. Estos enlaces se conocen como enlaces *fosfodiéster* y son los que le dan la orientación a la columna vertebral que encadena las bases en las hebras. Por convención, se dice que las hebras inician en un grupo $5'$-fosfato y terminan en un grupo $3'$-hidroxilo. Comúnmente, en la literatura se hace referencia a una secuencia corta de nucleótidos (generalmente menor de 30) como *oligonucleótido* o simplemente *oligo*. Matemáticamente, una hebra de ADN se define como una palabra sobre el alfabeto {A, G, C, T}.

La estructura química de las bases permite la formación eficiente de enlaces de hidrógeno entre dos pares de bases. De esta forma, A se puede unir mediante un enlace doble de hidró-

geno a T, y viceversa, mientras que G se puede unir mediante un enlace triple de hidrógeno a C, y viceversa. Esta propiedad de unión entre pares de bases se conoce como *complementariedad de Watson-Crick*. Dos hebras simples de ADN (ssADN) con orientación opuesta y con bases complementarias en cada una de sus posiciones (llamadas hebras antiparalelas) se pueden unir para formar una hebra doble (dsADN) en un proceso conocido como *emparejamiento de bases* (ver Figura 11). Cuando se trata de hebras dsADN se utiliza la abreviatura *pb* (pares de bases) para denotar la longitud de las secuencias. A la unión de una subsecuencia o fragmento de hebra con su respectiva subsecuencia o fragmento complementario se le conoce como *hibridación* (*annealing* o *hybridization*). Al proceso opuesto de separación de dos fragmentos de hebras unidos se le denomina *desnaturalización* (*denaturation*). Ambos procesos son fundamentales en los modelos de cómputo biomolecular y serán definidos con mayor detalle más adelante.

3.1.2. ARN

La molécula de ARN (ácido ribonucleico) es otro tipo de molécula que puede ser utilizada para realizar cómputo. El ARN es muy similar al ADN, de hecho sólo difiere en tres aspectos principales: los nucleótidos en el ARN contienen el azúcar ribosa, mientras que el ADN contiene el azúcar desoxirribosa; el ARN usualmente consiste de hebras simples a diferencia del ADN que generalmente se presenta en hebras dobles; en el ARN la base Uracilo (U) sustituye a la base Timina del ADN. La regla de complementariedad de Watson-Crick también aplica en el ARN, con la diferencia de que es la base Uracilo la que se une a la base Adenina.

3.1.3. Enzimas

Otros componentes que juegan un papel muy importante en los procesos biológicos y que particularmente contribuyen en la actividad molecular del ADN son las enzimas. Las enzimas son los agentes naturales que fungen como catalizadores en las reacciones químicas y como decodificadores de la información contenida en el ADN. Las enzimas de restricción reconocen ciertas regiones en el ADN y cortan la secuencia de nucleótidos para formar subconjuntos de secuencias. Las *endonucleasas* son enzimas de restricción que reconocen una subsecuencia corta específica dentro de la hebra, llamada *sitio de restricción*, y corta los enlacen covalentes entre nucleótidos adyacentes. Las *exonucleasas* hidrolizan los enlaces fosfodiéster de los

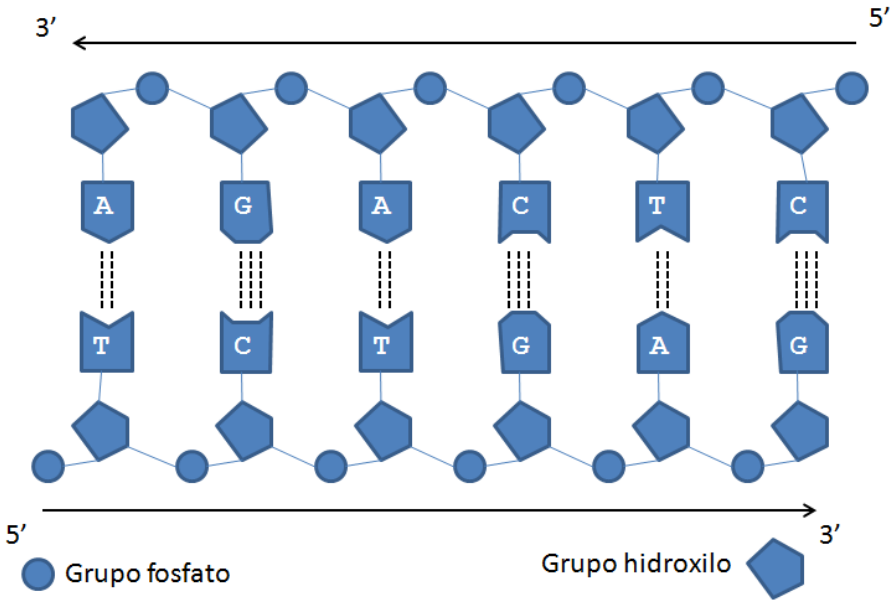


Figura 11: Diagrama esquemático del ADN. Dos hebras simples antiparalelas se unen para formar una hebra doble. Formación de enlaces dobles y triples de hidrógeno entre (A, T) y (G, C), respectivamente. Imagen basada en Amos (2005).

extremos 5′ o 3′ tanto de hebras simples como dobles y remueven los residuos uno a la vez. La enzima *ligasa* crea enlaces covalentes entre los extremos 3′-hidroxilo de un nucleótido y 5′-fosfato de otro, uniendo fragmentos de hebras de esta forma.

Aunque el uso de enzimas se incluye en varios modelos de cómputo basado en ADN, éste constituye una característica no deseable en tales modelos. Esto es debido a que las enzimas son materiales relativamente costosos y de corta vida útil, por lo que su utilización altera en buena medida el siempre importante factor económico de las implementaciones prácticas de cómputo biomolecular.

3.1.4. Bio-operaciones

Las operaciones definidas en los modelos de cómputo basado en ADN consisten en procedimientos biológicos (llamados comúnmente bio-operaciones) bien definidos, que son aplicables a conjuntos de hebras de ADN. Esta bio-operaciones son realizadas de manera cotidiana por biólogos moleculares en laboratorios equipados adecuadamente.

Síntesis

La síntesis de ADN de estado sólido está basada en un método en el cual se une un nucleótido inicial a un soporte sólido y, paso a paso, se van añadiendo nucleótidos sucesivos en la dirección $3'$ a $5'$ en una solución reactiva (Figura 12). Hebras de ADN de longitudes de hasta 100 bases pueden ser fácilmente sintetizadas de forma completamente automatizada. Como resultado, se obtiene un pequeño tubo de prueba con una pequeña cantidad de masa blanca amorfa que contiene una población homogénea de ADN de hasta 10^{18} hebras.

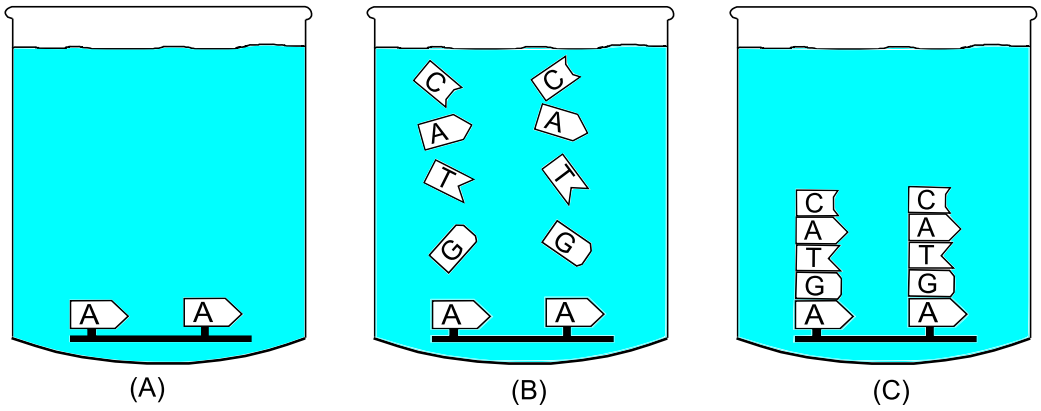


Figura 12: Síntesis de ADN. (A) Nucleótidos iniciales se adhieren a un soporte sólido. (B) Se agregan nucleótidos paso a paso en la dirección $3' \rightarrow 5'$ en una solución reactiva. (C) La solución reacciona y se van formando las hebras.

Hibridación y desnaturalización

Estos procesos son también llamados recocido (*annealing*) y fusión (*melting*), respectivamente. La unión con los puentes de hidrógeno entre dos hebras antiparalelas es más débil que los enlaces covalentes que unen nucleótidos en la misma secuencia. Estas operaciones son llevadas a cabo alterando condiciones fisico-químicas simples tales como la temperatura y el pH de las soluciones que contienen al ADN. El incremento en la temperatura rompe los puentes de hidrógeno separando las hebras y la disminución forma de nuevo estas uniones. Cada hebra doble tiene una temperatura específica, llamada temperatura de fusión, en la cual se separa en sus hebras constituyentes (Figura 13).

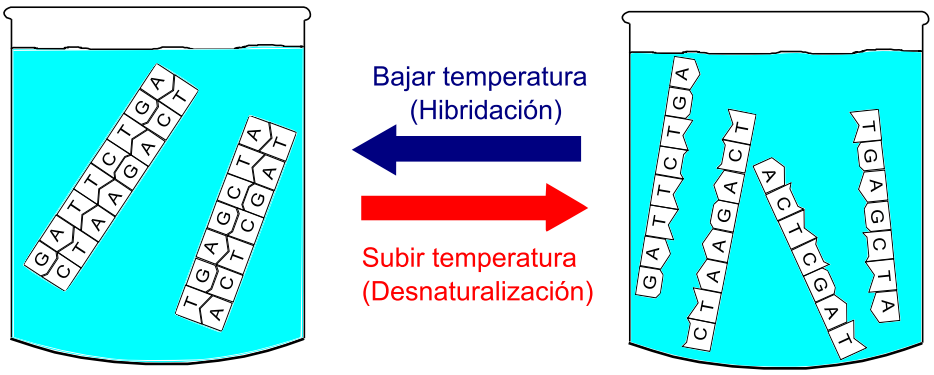


Figura 13: Procesos de hibridación y desnaturalización.

Cortado

También conocido como digestión, en el que se utiliza enzimas de restricción (*endonucleasas*). Cada enzima de restricción reconoce una secuencia corta específica de ADN (*sitio de restricción*). Cualquier hebra doble de ADN que contenga este sitio de restricción en su secuencia, es cortada de acuerdo a un patrón específico. Según el tipo de enzima utilizado, la hebra doble puede quedar dividida en dos hebras con los extremos lisos o con fragmentos de hebra simple llamados *extremos pegajosos* (*sticky ends*) (Figura 14). Existen cientos de enzimas de restricción conocidos y muchas de ellas están disponibles comercialmente.

Ligado

Proceso opuesto al anterior. Realizado por otra enzima llamada ADN *ligasa* que crea los enlaces covalentes en hebras con extremos complementarios restituyendo las partes cortadas en la columna vertebral por las enzimas de restricción (Figura 14).

Separación

Clasificación de hebras de ADN por tamaño utilizando una técnica llamada *electroforesis en gel*. La electroforesis consiste en el movimiento de moléculas cargadas a través de un campo eléctrico. Las moléculas de ADN tienen la misma carga negativa por unidad de longitud, de manera que tienden a moverse uniformemente a través de un medio acuoso. Sin embargo, cuando el medio consiste en un gel (de agarosa, poliacrilamida o combinación de las dos), la tasa de migración de las moléculas se ve afectado por su tamaño. El procedimiento consiste

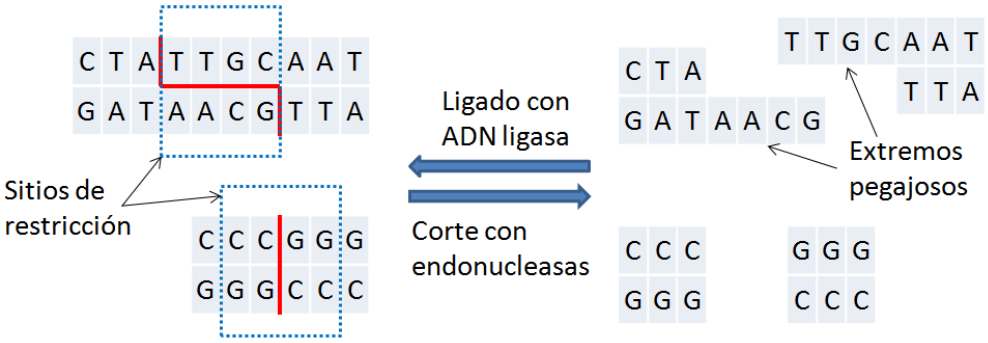


Figura 14: Acción de las enzimas sobre las dsADN. Dos enzimas de restricción hipotéticas reconocen diferentes sitios de restricción y cortan la dsADN dejando extremos escalonados (pegajosos) o lisos. La ADN ligasa realiza la operación opuesta uniendo los extremos cortados.

en colocar las moléculas de ADN en una ranura hecha sobre el gel y aplicar una carga eléctrica. Las moléculas se distribuyen en bandas a lo largo del gel quedando más cerca de la carga positiva las de menor tamaño. Por último, se tiñe el ADN con tinta y se visualizan los resultados utilizando luz ultravioleta (Figura 15).

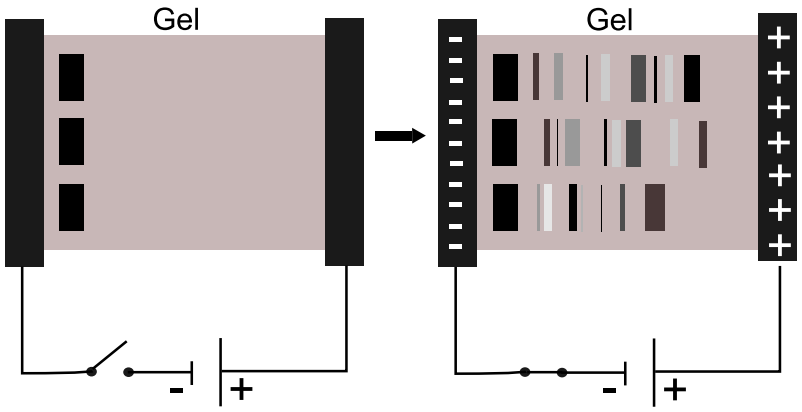


Figura 15: Electroforesis en gel. Se aplica un campo eléctrico que mueve las moléculas de ADN a través del gel, quedando clasificadas por tamaño.

Extracción

Se extraen las hebras que contienen una subsecuencia específica en una mezcla heterogénea de ssADN. El proceso mediante el cual se lleva a cabo esto se conoce como *purificación por afinidad*. Se utilizan *sondas* las cuales son hebras simples que son antiparalelas a la región o subsecuencia que se quiere detectar. Se fijan estas sondas a un soporte sólido (tal como

cuentas magnéticas). Se hace pasar la solución heterogénea de ssADN a través del medio donde se encuentran las sondas. Las hebras que contienen la subsecuencia deseada se adhieren (hibridan) a las sondas y son retenidas. Las demás hebras pasan por el medio libremente. Al final, las cuentas magnéticas pueden ser removidas de la mezcla y las hebras deseadas son extraídas (Figura 16).

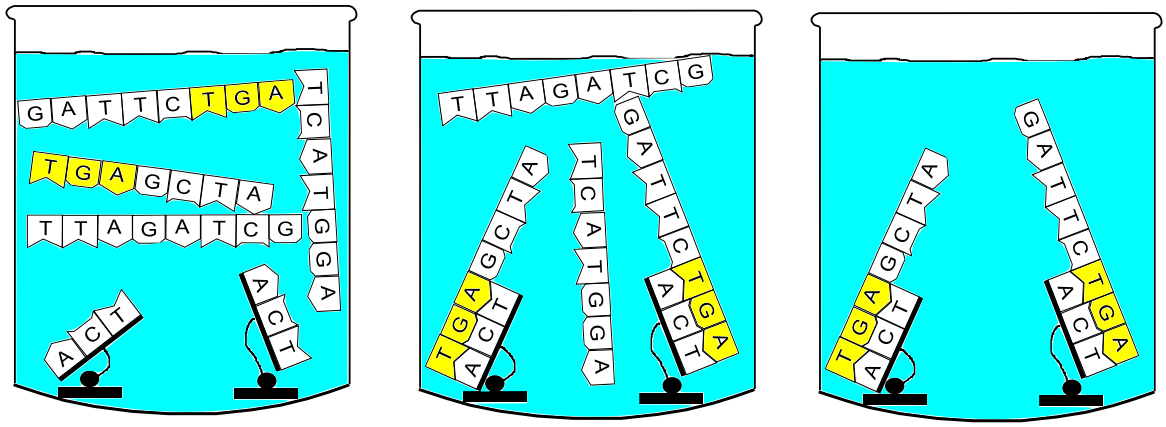


Figura 16: Proceso de purificación por afinidad. Para extraer (separar) las hebras que contienen la subsecuencia $5'-\text{TGA}-3'$ se agregan las hebras a una solución que contiene sondas adheridas a un soporte sólido. Estas sondas corresponden a la subsecuencia $3'-\text{ACT}-5'$ complementaria. Las hebras que contienen la subsecuencia deseada hibridan con las sondas. Se lava el contenido removiendo de la mezcla aquellas hebras que no contienen la subsecuencia deseada. En la práctica, se requieren subsecuencias más largas para que el proceso funcione.

Replicación

Esta operación también es conocida como *amplificación*. Copias idénticas de ADN son creadas en un proceso iterativo llamado *Reacción en Cadena de la Polimerasa* (PCR por su siglas en inglés) que se basa en el uso de la enzima ADN Polimerasa. Dada una dsADN que sirve como *plantilla*, oligos cortos *iniciadores* (*primers*) que contienen las primeras bases de los extremos $5'$ de cada ssADN que compone la plantilla, y la presencia de suficientes nucleótidos, la ADN polimerasa extiende los primers cuando están hibridados en la plantilla. Un ciclo de PCR consiste de los siguientes pasos: calentamiento para desnaturalizar las plantillas, enfriamiento para la hibridación de los primers en los extremos $5'$ de cada ssADN de la plantilla, la ADN polimerasa extiende los primers añadiendo sucesivamente nucleótidos en sus extremos $3'$ (siempre en la dirección $5' \rightarrow 3'$) (Figura 17). Al final de n iteraciones se

producen 2^n copias de una plantilla.

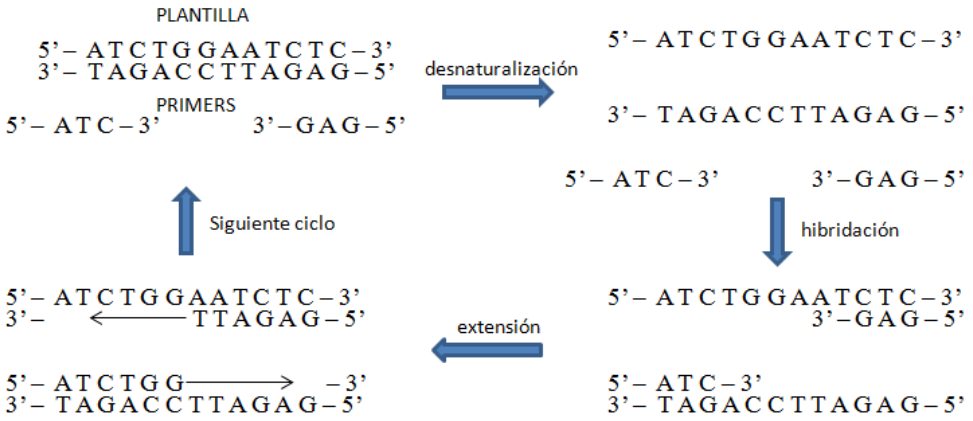


Figura 17: Un ciclo de PCR. Se aplica calor para separar la plantilla en ssADN. Se enfría la solución para que los primers hibriden en los extremos $5'$ de las ssADN. La ADN polimerasa extiende los primers en la dirección $5' \to 3'$ hasta que cada ssADN forme una dsADN. El proceso se repite duplicando la cantidad de hebras en cada iteración.

3.2. Cómputo biomolecular

3.2.1. Antecedentes

El cómputo biomolecular es un campo de estudio relativamente nuevo donde los componentes fundamentales de cómputo consisten en moléculas biológicas. Es un área de convergencia de conocimientos y experiencia de diversas áreas de estudio, tales como la química, ciencias computacionales, biología molecular, física y matemáticas. Históricamente, la idea de usar moléculas biológicas como sustrato físico para almacenar información, se debe al físico estadounidense Richard Feynman, quien a finales de los años cincuentas impartió la conferencia *There is plenty of room at the bottom*, donde propuso este nuevo paradigma de entender cómo los complejos moleculares almacenan una gran cantidad de información y además, que estos complejos no son estáticos, sino que existen mecanismos naturales que los hacen muy activos. Con esta idea de poder manipular esta capacidad de almacenamiento y dinamismo de los complejos moleculares, Feynman sentó las bases para el desarrollo del cómputo biomolecular. El cómputo con ADN, que forma parte del cómputo biomolecular, ofrece un nuevo paradigma de computación. Este paradigma consiste fundamentalmente en codificar la información en hebras de ADN y aplicar operaciones basadas en técnicas de la-

boratorio de biología molecular. Estas operaciones son llamadas bio-operaciones, y a través de ellas se simulan operaciones aritméticas y lógicas sobre la información codificada en las hebras y así, se obtiene un resultado. Una de las características principales de los modelos de cómputo con ADN es la vasta capacidad de información que es posible almacenar en espacios muy pequeños. Además, el cómputo con ADN ofrece un paralelismo masivo de datos, dado que las bio-operaciones actúan de manera concurrente sobre la información codificada en una gran cantidad de hebras.

3.2.2. Trabajos principales

Experimento de Adleman

El interés por el estudio en los modelos de cómputo basados en ADN surgió a partir del trabajo de Leonard Adleman, quien fue el primero en lograr resultados experimentales resolviendo un pequeño caso particular del problema de búsqueda de trayectorias de Hamilton (HPP, por sus sigles en inglés) utilizando moléculas de ADN y técnicas de laboratorio de biología molecular para realizar cómputo con ellas (Adleman, 1994). Dado un grafo dirigido $G = (V, E)$ con $V = \{v_1, v_2, \ldots, v_n\}$ y un par de vértices v_{i_1} y v_{i_n} designados del grafo, una trayectoria de Hamilton consiste en una secuencia $\langle v_{i_1}, v_{i_2}, \ldots, v_{i_{n-1}}, v_{i_n} \rangle$ de los n vértices de manera que $(v_{i_r}, v_{i_{r+1}}) \in E$, $\forall (1 \leq r < n)$. Adleman resolvió el caso particular de grafo de la Figura 18 a través del Algoritmo 1.

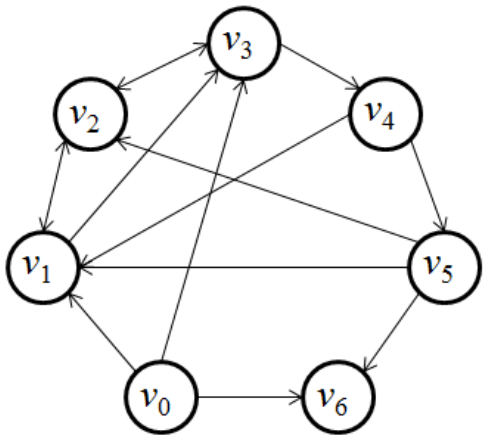


Figura 18: El caso particular de grafo de Adleman. La secuencia de vértices $\langle v_0, v_1, v_2, v_3, v_4, v_5, v_6 \rangle$ corresponde a una ruta de Hamilton.

Algoritmo 1: Rutas de Hamilton − Adleman

Entrada : Grafo $G = (V, E)$ con $V = \{v_1, v_2, \ldots, v_n\}$, vértices inicial y final v_{i_1}, v_{i_n}.

Salida : Trayectoria(s) de Hamilton en G.

1 Generar trayectorias aleatorias en G;

2 Mantener sólo aquellas trayectorias que empiecen en v_{i_1} y finalicen en v_{i_n};

3 Mantener sólo aquellas trayectorias que contengan exactamente n vértices;

4 Mantener sólo aquellas trayectorias que consideren cada vértice al menos una vez;

5 Leer las trayectorias restantes (si existen);

Cada paso en el algoritmo de Adleman se implementa mediante un procedimiento de laboratorio biomolecular. Para la codificación del caso particular (paso 1) se diseñaron y sintetizaron secuencias de 20nt para los vértices y las aristas dirigidas. La secuencia de cada arista dirigida se divide en dos partes, cada una correspondiente a una mitad del complemento de la secuencia de un vértice origen v_o y un vértice destino v_d. Así, las secuencias que codifican vértices se unen a secuencias que codifican aristas para después ligarse entre sí (Figura 19). Para el paso 2, se amplifican las hebras obtenidas en el paso 1, utilizando como primers las secuencias correspondientes a v_{i_1} y el complemento de v_{i_n}. Sólo las hebras que inician en v_{i_1} y finalizan en v_{i_n} son amplificadas. El paso 3 se realiza separando las hebras por tamaño a través de electroforesis en gel. Las hebras que incluyen exactamente los 7 vértices del caso particular tienen una longitud de $7 \times 20 = 140$nt. Para el paso 4, primero se producen ssADN de cada dsADN obtenida del paso anterior (desnaturalización). Posteriormente, para cada vértice se aplica el procedimiento de purificación por afinidad de forma iterativa. En cada iteración $1 \leq i \leq n$ se mantienen las hebras que incluyen al i-ésimo vértice. Finalmente, para el paso 5 se amplifican las hebras (en caso de existir) resultantes del paso anterior para su detección.

El trabajo de Adleman se considera seminal en el área del cómputo biomolecular por demostrar la viabilidad práctica de resolver problemas utilizando el paradigma de cómputo con moléculas biológicas. Esto provocó el interés por desarrollar nuevas investigaciones en el campo del cómputo biomolecular.

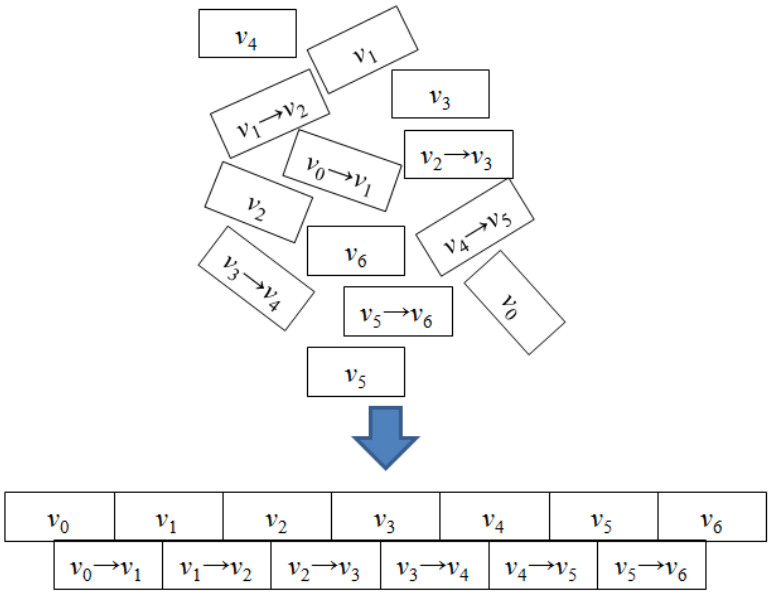


Figura 19: Codificación del caso particular del problema. Las aristas están formadas por mitades antiparalelas de las secuencias de los vértices. Una ruta posible se produce por la hibridación de los fragmentos que codifican vértices con los que codifican aristas, y el posterior ligado entre los vértices.

El problema de satisfactibilidad

En (Lipton, 1995) se describe una metodología para el resover el problema de *satisfactibilidad* (*satisfiability*– SAT). Este problema se define como sigue: dado un conjunto finito de variables lógicas $V = \{v_1, v_2, \ldots, v_n\}$, se define una *literal* como una variable v_i o su complemento $\overline{v_i}$. Un conjunto de literales $\{v_1^j, v_2^j, \ldots, v_l^j\}$ forma una *cláusula* C_j. Una caso particular I del problema SAT consiste en un conjunto de cláusulas y una solución se compone de una asignación de valores booleanos para cada una de las variables, de manera que cada cada cláusula sea verdadera. Entonces se dice que el caso particular I se *satisface*. El problema de satisfacibilidad pertenece a la clase NP-Completo (Cook, 1971).

Lipton enfrentó este problema basándose en el mismo enfoque general de Adleman: generar todas las posibles formas de solución para mantener aquellas que se puedan considerar "legales" (que cumplan los requerimientos de una solución). La codificación del problema también está basada en la utilizada por Adleman. Para esto, la idea clave fue codificar una cadena binaria arbitraria como una ruta a través de un grafo específico G_n. La Figura 20 muestra el grafo de Lipton para una codificación de dos bits. Al generar trayectorias a través del grafo G_n se producen cadenas correspondientes.

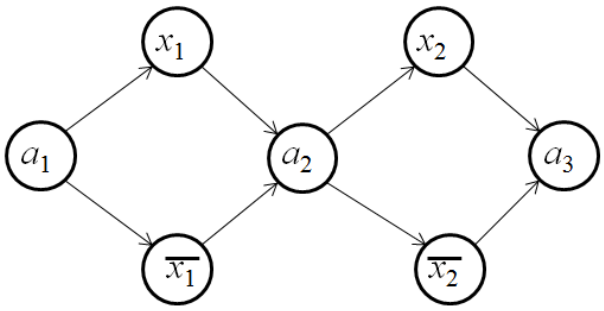


Figura 20: Grafo G_n que produce la codificación de 2 bits. Las trayectorias $\langle a_1, x_1, a_2, x_2, a_3\rangle$, $\langle a_1, \overline{x_1}, a_2, x_2, a_3\rangle$, $\langle a_1, x_1, a_2, \overline{x_2}, a_3\rangle$, $\langle a_1, \overline{x_1}, a_2, \overline{x_2}, a_3\rangle$, codifican las cadenas 11, 01, 10 y 00, respectivamente.

Aunque en (Lipton, 1995) no se definen explícitamente las operaciones necesarias, la propuesta de solución puede ser planteada con el Algoritmo 2 en términos de las siguientes operaciones definidas en (Adleman, 1994).

- *Separar*(T, S). Dado un conjunto T de cadenas y una subcadena S, obtener un conjunto $+(T, S)$ con todas las cadenas que contienen S y un conjunto $-(T, S)$ con todas las cadenas que no contienen S.
- *Unir*$(T_1, T_2, \ldots, T_n)$. Dados los conjuntos $T_1, T_2, \ldots, T_n$ obtener $T_1 \cup T_2 \cup \cdots \cup T_n$.
- *Detectar*(T). Dado el conjunto T, regresar VERDADERO si $T \neq \emptyset$. Regresar FALSO, en caso contrario.

Algoritmo 2: Satisfacibilidad − Lipton

Entrada : Conjunto de cláusulas $I = \{C_1, C_2, \ldots, C_m\}$.

Salida : VERDADERO, si la fórmula se satisface; FALSO, en caso contrario.

```
1 Generar T;
2 for a ← 1 to |I| do
3 |   for b ← 1 to |C_a| do
4 |   |   if v_b^a = x_j then T_b ← +(T, v_b^a = 1);
5 |   |   else T_b ← +(T, v_b^a = 0);
6 |   end
7 |   T ← Unir(T_1, T_2, ..., T_b);
8 end
9 Regresar(Detectar(T));
```

El primer paso (línea 1) consiste en generar todas las posibles rutas a través de G_n (equivalentemente, todas las diferentes cadenas de n bits). Cada iteración del ciclo de las líneas $2-8$ procesa una cláusula $C_a = \{v_1^a, v_2^a, \ldots, v_l^a\}$ del conjunto I. Para cada cláusula C_a, el ciclo de las líneas $3-4$ recorre las literales que la forman, manteniendo aquellas cadenas que contienen una asignación de la literal que hace verdadera la cláusula. Por ejemplo, sea $I = (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_3) \wedge \overline{x_3}$. Inicialmente, T contiene las hebras que codifican las cadenas $\{000, 001, 010, 011, 100, 101, 110, 111\}$. Las tres iteraciones del ciclo de las líneas $2-8$ mantienen los conjuntos $\{000, 001, 100, 101, 110, 111\}$, $\{000, 001, 101, 111\}$ y $\{000\}$, respectivamente. Al final, el algoritmo regresa VERDADERO (línea 9) dado que T contiene la cadena 000 que satisface el caso particular I del problema.

Cómputo de cliques máximos

El problema de clique máximo (MCP) fue inicialmente abordado en (Ouyang *et al.*, 1997), donde se presentan resultados experimentales de la solución de un caso particular de grafo de seis vértices (Figura 21). El primer paso en la estrategia utilizada consiste en codificar cada posible clique en el grafo con una cadena de n bits $b_{n-1}, b_{n-2}, \ldots, b_1, b_0$. Por ejemplo, el clique de tamaño máximo de la Figura 21(A) es $\{v_5, v_4, v_3, v_2\}$ que se representa con la cadena 111100. Después, para cada arista $(v_i, v_j) \in \overline{E}$ se remueven las cadenas que tienen las posiciones correspondientes a v_i y v_j en estado "encendido" (1). Por definición, estas cadenas no pueden representar un clique en G dado que consideran un par de vértices v_i, v_j para los cuales la arista $(v_i, v_j) \notin E$. Tomando de nuevo el ejemplo de la Figura 21(B), las aristas (v_0, v_2), (v_0, v_5), (v_1, v_3), $(v_1, v_5) \in \overline{E}$, por lo que las cadenas XXX1X1, 1XXXX1, XX1X1X y 1XXX1X deben ser removidas (las X representan cualquier estado de bit). Las cadenas restantes representan cliques de G. Estas cadenas son ordenadas posteriormente para distinguir los cliques de tamaño máximo, esto es, las cadenas que contengan la mayor cantidad de 1's.

Para construir un conjunto de hebras de ADN que representen todos los posibles subgrafos se consideran hebras dsADN. Cada bit i en un número binario se representa por dos secciones: una que codifica el valor del bit (V_i) y la otra su posición (P_i). Para un número de seis bits hay seis secciones de valor $V_0 - V_5$ intercaladas con siete secciones de posición $P_0 - P_6$ (la sección extra P_6 se agrega con propósitos de amplificación por PCR). Cada sección P_i se construye

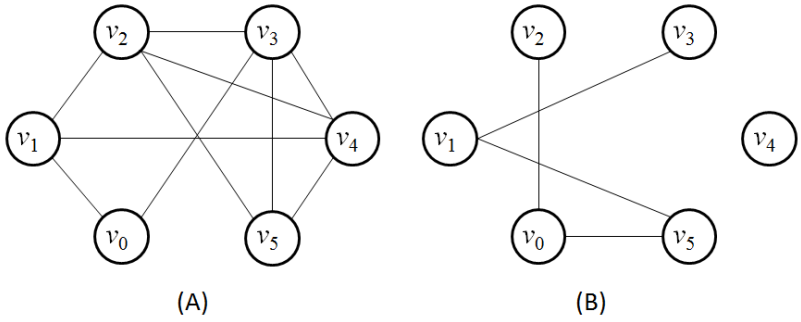


Figura 21: Grafo utilizado en (Ouyang *et al.*, 1997). (A) $G = (V, E)$. (B) $\overline{G} = (V, \overline{E})$.

utilizando secuencias de 20pb. Para la longitud de las secciones de valor se consideran dos casos: si $V_i = 1$ se destinan 0pb, si $V_i = 0$ se utilizan secuencias de "relleno" de 10pb. De esta forma, la secuencia de mayor longitud es de 200pb (correspondiente al número 000000), mientras que la menor es de 140pb (que codifica el número 111111). La remoción de cadenas que no pueden codificar un clique se realiza utilizando enzimas de restricción. Cada enzima reconoce y digiere una sección $V_i = 1$. Para cada arista $(v_i, v_j) \in \overline{E}$ se separan las hebras en dos tubos T_0 y T_1. En T_0 quedan las cadenas con sección $V_i = 1$, en donde se agrega la enzima correspondiente que reconoce y corta la sección. De la misma forma, en T_1 se mezclan las cadenas con sección $V_j = 1$ y las enzimas correspondientes. Los contenidos de los tubos T_0 y T_1 son unidos en una operación posterior. Así, las hebras con secciones cortadas por el paso anterior no son amplificadas exponencialmente por medio de PCR utilizando primers P_0 y $\overline{P_6}$, siendo eliminadas del cómputo. Finalmente, la lectura de los resultados obtenidos se implementa con electroforesis en gel. Debido a la codificación utilizada, las hebras de menor tamaño son las que codifican cadenas con mayor cantidad de 1's. En la banda más corta se encontraron hebras de 160pb correspondientes a las cadenas 111100 que codifican el clique máximo en el grafo.

3.2.3. Modelo de etiquetas

El modelo de etiquetas (*sticker model*) introducido en (Roweis *et al.*, 1998) es un modelo de cómputo biomolecular que utiliza hebras de ADN como el sustrato físico donde se almacena la información. El mecanismo central de cómputo está basado en la separación por hibridación. Introduce una ventaja sobre los modelos previamente definidos que consiste en el manejo de una memoria de acceso aleatorio manteniendo una longitud fija de las hebras.

Además, las bio-operaciones definidas en el modelo no requieren la utilización de enzimas que modifiquen las hebras, haciendo que éstas sean teóricamente reutilizables en diferentes cómputos.

Representación de la información

El modelo de etiquetas representa cadenas de bits en moléculas de ADN. Para representar una cadena de bits se consideran dos tipos de moléculas ssADN: las llamadas *hebras de memoria* y las *hebras etiquetas* (*sticker strands*) o simplemente *etiquetas*. Una hebra de memoria se compone de N bases en total y se divide en K regiones no traslapadas de M bases. De esta forma $N \geq KM$. Para cada una de las K regiones, se considera una hebra antiparalela de M bases. Estas hebras son las llamadas etiquetas. Cada etiqueta solamente puede hibridar en su región correspondiente y en ninguna otra región de la hebra de memoria. Cuando una región tiene "pegada" su etiqueta correspondiente, tal región codifica un 1. En caso contrario, codifica un 0. A una hebra de memoria con sus etiquetas asociadas se le denomina *complejo de memoria* (*memory complex*). Así, cada complejo de memoria codifica un número binario de K bits (Figura 22).

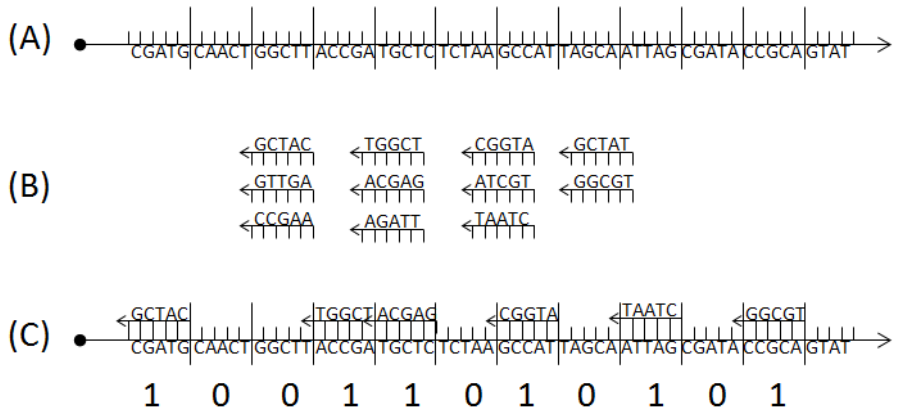


Figura 22: Representación de información en el modelo de etiquetas. (A) Una hebra de memoria de $N = 59$ bases dividida en $K = 11$ regiones de $M = 5$ bases. (B) Etiquetas para cada una de las regiones. (C) Un complejo de memoria formado por una hebra de memoria y etiquetas asociadas a sus regiones.

A una colección de complejos de memoria se le denomina *tubo*. Esta representación genera una gran cantidad de números binarios utilizando una misma molécula ssADN como hebra de memoria y diferentes asignaciones de etiquetas a sus respectivas regiones. Esto constituye

otra ventaja respecto a los modelos anteriores (Lipton, 1995) y (Adleman, 1994) donde cada bit se representa por la presencia o ausencia de una subsecuencia diferente en la hebra, por lo que cada número es una molécula diferente.

Siguiendo el paradigma de los modelos de filtrado en la búsqueda de soluciones de problemas *NP*-completos, el cómputo en el modelo empieza con un tubo inicial. Sobre este tubo se realiza una secuencia de operaciones que de manera paralela filtra (elimina) los complejos de memoria que no corresponden a soluciones de algún problema particular. En lo que resta del presente documento, los términos bit/región, complejo de memoria/hebra/cadena, tubo/conjunto de cadenas, serán equivalentes a menos que se indique lo contrario.

Operaciones sobre colecciones de cadenas

En el modelo se definen cinco operaciones las cuales son bastante flexibles para la implementación de algoritmos de propósito general. Estas operaciones consisten en:

- **Merge(T, T_0, T_1).** *Combinar* (*Unir*) dos conjuntos de cadenas en uno. Esta es la operación más básica que consiste en la generación de un nuevo conjunto que contiene la unión de todas las cadenas de los conjuntos de entrada. En ADN esto significa producir un nuevo tubo (T) que contiene todos los complejos de memoria de ambos tubos de entrada (T_0, T_1).

- **Separate(T, T_0, T_1, i).** Separar un conjunto de cadenas en dos nuevos conjuntos, uno que contiene todas las cadenas que tienen un bit particular i encendido y el otro todas las cadenas con el bit apagado. Esto consiste en aislar en el tubo T aquellos complejos que presentan la etiqueta en alguna región i de las que no la presentan (sin alterar las etiquetas en los complejos).

- **Set(T, i).** Encender un bit particular i en cada cadena de un conjunto. La etiqueta correspondiente a la región i se "adhiere" (si no está presente) en todos los complejos contenidos en un tubo T.

- **Clear(T, i).** Apagar un bit particular i en cada cadena de un conjunto. La etiqueta correspondiente a la región i se "despega" (si está presente) en todos los complejos contenidos en un tubo T.

- **Discard(T).** Eliminar (descartar) un conjunto de cadenas. Desechar el contenido de un tubo T.

Conjunto inicial (K, L)

Un conjunto inicial de cadenas (llamado *conjunto biblioteca* (K, L)) contiene cadenas de longitud K generadas tomando todas las posibles cadenas de longitud $L \leq K$ seguidas de $K - L$ ceros. Así, hay 2^L cadenas en el conjunto. Por ejemplo, el conjunto biblioteca $(6, 3)$ es {000000, 001000, 010000, 011000, 100000, 101000, 110000, 111000}. Los primeros L bits son utilizados para codificar el problema y son la parte aleatoria de la biblioteca, mientras que los $K - L$ bits restantes se utilizan para almacenamiento intermedio y codificación de respuesta (en caso de ser necesario). Con este conjunto inicial es posible codificar una gran variedad de problemas combinatorios.

Trabajo previo

El modelo de etiquetas es un modelo de cómputo universal (Kari *et al.*, 1998). Pertenece a la clase de modelos de filtrado, en el sentido que toma una entrada, codificada en hebras de ADN, que consiste en todas las posibles soluciones para un caso particular de problema, y aplica una secuencia de operaciones (algoritmo), para eliminar (filtrar) aquellas hebras que no correspondan al conjunto de soluciones. En la literatura, existe una gran variedad de algoritmos que se enfocan en la solución de problemas de la clase NP-Completo. En (Zimmermann, 2002) se proporcionan algoritmos que calculan k-cliques, k-conjuntos independientes, rutas Hamiltonianas y árboles de Steiner. Estos algoritmos reportan todas las soluciones de cada problema en caso de que existan. Los problemas de la mochila (*Knapsack*) y de suma de subconjuntos (SubSet Sum - S^3) se abordan en (Pérez-Jiménez y Sancho-Caparrini, 2002). Estos algoritmos se basan en un enfoque que permite ordenar por cardinalidad una familia finita de conjuntos finitos. La técnica usada para lograr esto es diferente de la aplicada por (Roweis *et al.*, 1998) para resolver el problema de Cubrimiento Mínimo de Conjuntos. Xu *et al.* (2004a) presentan un estudio que resume los trabajos de otros autores. Además, introduce los modelos *k-bit sticker* y *full-message sticker* los cuales son una extensión del modelo original que redefinen la estructura de los complejos de memoria, así como las operaciones que se aplican en el modelo. Adicionalmente, presentan un artículo complementario (Xu *et al.*, 2004b)

donde se enfocan en las aplicaciones del modelo, introducen una representación matricial del modelo, y realizan una recopilación de los algoritmos en la literatura, por último, presentan un algoritmo para el problema de isomorfismo de grafos. En (Martínez-Pérez y Zimmermann, 2009) se proponen algoritmos para diversos problemas NP-Completos. Hacen énfasis sobre la implementación *in silico* para resolver problemas de tamaños pequeños a moderados de manera exacta y de forma heurística los de tamaño considerable. Los algoritmos dados son probados en un entorno de simulación en computadora. En (Ahrabian *et al.*, 2008) se presenta una algoritmo para el problema de optimización combinatoria de las N Reinas. Este algoritmo puede ser extendido para resolver problemas como el de Asignación Cuadrática, Calendarización de Mantenimiento, Asignación de Recursos, y de Emparejamiento de Grafos. El problema de *Bin-Packing* es tratado en (Alonso Sanchez y Soma, 2009), mientras que en (Darehmiraki, 2009) se resuelve el problema de Cliques Maximales. En (Liu *et al.*, 2010) se utiliza el modelo híbrido Adleman-Lipton-sticker que utiliza las ventajas de ambos. Se incluyen las operaciones definidas en el modelo de Adleman-Lipton utilizando la codificación de información del modelo de etiquetas. En este artículo se resuelve el problema de Bisección Mínima de un Grafo. En (Razzazi y Roayaei, 2011) presentan tres nuevos algoritmos para sus correspondientes problemas de grafos: el problema de *Domatic-Partition*, *Induced Path* y *Kernel*. También se incluye la simulación de los algoritmos propuestos para demostrar su efectividad.

Capítulo 4. Diseño de plataforma de simulación

En este capítulo se analizan los aspectos del diseño de la plataforma de simulación de algoritmos del modelo de etiquetas. Se definen los algoritmos de las funciones necesarias para la operación de tal plataforma.

4.1. Paralelismo

Dentro del modelo de etiquetas, el paralelismo se puede analizar en tres niveles: a nivel de regiones (bits), a nivel de hebras (cadenas), y a nivel de tubos. A nivel de regiones, el modelo planteado actúa de manera secuencial, es decir, las operaciones no consideran concurrencia en el manejo de las regiones. Por ejemplo, la operación *SET* no considera el encendido concurrente de dos o más regiones en la misma hebra. En tal caso, se requiere una secuencia de operaciones *SET*, una para el encendido de cada región. Es a nivel de hebras donde ocurre el paralelismo masivo del modelo de etiquetas. Las operaciones del modelo actúan concurrentemente sobre todas las hebras contenidas dentro de un tubo. Finalmente, a nivel de tubos, se puede considerar la ejecución concurrente de dos o más operaciones, con la condición de que no haya dependencia entre los tubos involucrados. Por ejemplo, las operaciones $SEPARATE(T_0, T_{on}, T_{off}, b)$ y $CLEAR(T_{on}, b)$ no se pueden efectuar en paralelo debido a que el tubo T_{on}, que es el tubo de entrada de la operación *CLEAR*, no está listo hasta que se termine la operación *SEPARATE* (Figura 23(A)). En contraste, las operaciones $SEPARATE(T_0, T_{on}, T_{off}, b)$ y $SEPARATE(T_1, T_{on}, T_{off}, b)$ sí pueden realizarse al mismo tiempo, pues aunque involucra los mismos tubos de salida, los tubos de entrada no dependen de la finalización de una operación anterior (Figura 23(B)).

4.2. Paralelismo de hebras vs. paralelismo de hilos

El modelo de etiquetas y el modelo de programación paralela de CUDA comparten ciertas analogías esenciales. El paralelismo de hebras del modelo de etiquetas corresponde al paralelismo de hilos de CUDA. Esta correspondencia es, en el sentido que las hebras son como estructuras que no tienen dependencia entre sí, y por lo tanto, las operaciones definidas en el modelo actúan concurrentemente sobre ellas. Los hilos, por su parte, son instancias

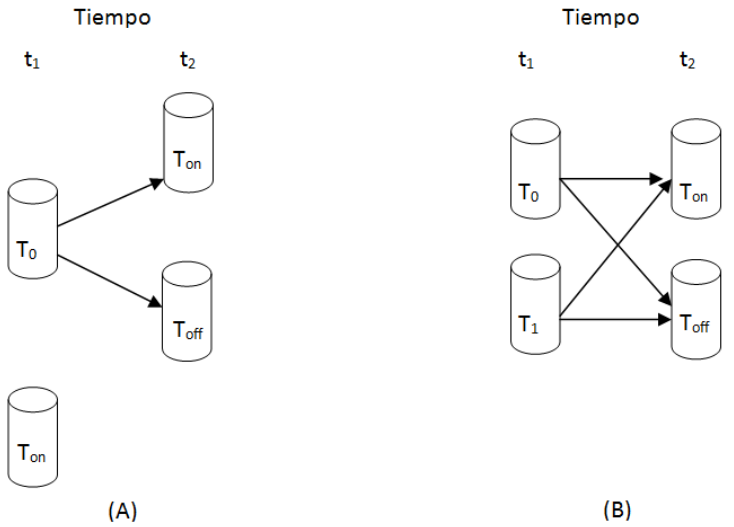


Figura 23: Paralelismo de tubos. A) Operaciones no paralelizables: $SEPARATE(T_0, T_{on}, T_{off}, b)$ y $CLEAR(T_{on}, b)$. B) Operaciones paralelizables: $SEPARATE(T_0, T_{on}, T_{off}, b)$ y $SEPARATE(T_1, T_{on}, T_{off}, b)$.

independientes de ejecución de un mismo conjunto de instrucciones sobre diferentes datos, generalmente independientes. De esta forma, el "mapeo" de hebras a hilos es natural: un hilo por cada hebra y cada hilo se encarga de ejecutar las operaciones sobre la hebra correspondiente, de manera concurrente. No obstante, existe un factor limitante para esta similitud. Esta limitante consiste en que la cantidad de hebras que se considera en el modelo, es mucho más grande (por varios órdenes de magnitud) que la cantidad de hilos que se pueden ejecutar concurrentemente en una implementación en CUDA, y que la memoria física del dispositivo disponible para almacenar estas hebras.

4.3. Modelo no autónomo vs. cómputo heterogéneo

Otra característica común entre el modelo de etiquetas y CUDA, es la forma en la que se realiza el cómputo. El modelo de etiquetas es un modelo no autónomo, es decir, cada operación requiere de algún mecanismo que la lleve a cabo. Este mecanismo puede ser una persona que implemente los procedimientos de laboratorio correspondientes a cada operación, o bien, un sistema de control robótico que automatice estos procedimientos. En cualquier caso, la definición de algoritmos del modelo de etiquetas, considera que las operaciones paralelas del modelo son llevadas a cabo de manera secuencial. Por otro lado, CUDA es una arquitectura de cómputo heterogéneo. Esto significa que ciertas tareas, aquellas intrínsecamente secuenciales,

son ejecutadas en el CPU, mientras que las tareas que exhiben paralelismo se destinan al GPU. De esta forma, las operaciones en el GPU son paralelas, pero requieren del CPU para su ejecución secuencial.

4.4. Tipos de operaciones

Las cinco operaciones definidas en el modelo de etiquetas se pueden clasificar en dos categorías:

1. **Operaciones de modificación de hebras.** Son las que realizan alguna alteración de la información codificada en las hebras. Dentro de esta categoría se incluyen *SET* y *CLEAR*, que son las únicas que escriben/borran información a través del encendido/apagado, respectivamente, de regiones en las hebras.

2. **Operaciones de transferencia de hebras.** Son las que realizan el movimiento de hebras entre tubos. Aquí se considera a las operaciones *SEPARATE*, *MERGE* y *DISCARD*.

4.5. Representación de estructuras

4.5.1. Hebras

En el modelo de etiquetas, las hebras de ADN codifican y almacenan la información de la instancia del problema. Dada la densidad de almacenamiento en las hebras de ADN, el modelo de etiquetas considera una cantidad masiva de éstas para la representación del espacio de soluciones de un problema. Debido a lo anterior, una implementación *in silico* de una plataforma para la simulación de algoritmos del modelo de etiquetas está limitada por la cantidad de memoria física disponible. Uno de los objetivos de la presente tesis, es encontrar una representación eficiente de las estructuras utilizadas en el modelo de etiquetas. Dicha representación debe maximizar el aprovechamiento de la memoria, es decir, utilizar la memoria bit por bit, sin consumir mucho tiempo de ejecución en el acceso lectura-escritura de cada uno de ellos.

De esta forma, en el presente trabajo, cada hebra se representa como un arreglo de datos de tipo entero. Cada entero consiste de $SIZE\_INT$ bits. Esta cantidad de bits por

entero es dependiente de la plataforma de ejecución del sistema, aunque en la mayoría de las plataformas $SIZE\_INT = 32$ bits $= 8$ bytes. Para representar cadenas binarias de K bits de longitud, se necesitan al menos $K/SIZE\_INT$ enteros cuando la división es exacta y $(K/SIZE\_INT) + 1$ enteros cuando la división tiene módulo. En la Figura 24 se muestra la representación de una cadena de longitud $K = 30$ bits utilizando 8 enteros, y su correspondiente arreglo de enteros. Por razones de simplicidad se utiliza un valor $SIZE\_INT = 4$ (cuatro bits por entero). Las X representan bits no utilizados en el entero (los bits no utilizados se manejan como ceros).

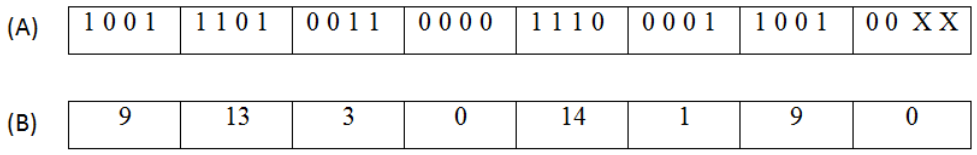


Figura 24: Representación de cadenas binarias con arreglos de enteros. (A) Ejemplo de cadena con $K = 30$ bits tomando 4 bits por entero ($SIZE\_INT = 4$. Las X representan bits no utilizados en el entero. (B) El correspondiente arreglo de enteros.

Con esta representación, se utiliza la memoria física bit por bit. Aunque la complejidad asintótica del espacio de almacenamiento sigue siendo $O(K)$, en la práctica, la reducción en el uso de memoria es considerable. Esta reducción es por un factor de $SIZE\_INT$ enteros por cadena. Para ilustrar lo anterior, supongamos que se desea almacenar cadenas de longitud K. La Tabla 1 muestra el uso de memoria para diferentes valores de K, utilizando una representación trivial que utiliza un arreglo de K enteros y la representación que utiliza $SIZE\_INT = 32$ bits por entero.

Tabla 1: Reducción en uso de memoria por cadena para diferentes valores K (en bytes).

K	Representación trivial	Representación bit por bit
64	256	8
200	800	28
800	3200	100
1024	4096	128
3000	12000	376

4.5.2. Tubos

Dentro del modelo de etiquetas, las hebras de ADN son contenidas en tubos. Cada tubo es un recipiente que mantiene temporalmente conjuntos de hebras. Más aún, las operaciones definidas en el modelo de etiquetas actúan sobre tubos de entrada, produciendo tubos como salida. Así, el enfoque natural de implementación de los tubos en una plataforma de simulación *in silico*, sería como una estructura formada por un campo identificador del tubo y un campo que consiste en un arreglo o lista enlazada de hebras. De esta forma, se tendrían diferentes estructuras tipo tubo, cada una almacenando un diferente conjunto de hebras, y la implementación estaría más apegada a la definición del modelo de etiquetas.

Sin embargo, al realizar un análisis más profundo de los aspectos técnicos de implementación, notamos que este enfoque natural presenta dos inconvenientes en cuanto al desempeño:

1. **Gestión de memoria.** En la operación *SEPARATE*, no se conoce *a priori* cuántas de las hebras de un tubo de entrada T_0 tienen el bit índice i encendido y cuántas lo tienen apagado. Esto representa un problema al momento de la gestión de memoria pues no se puede asignar el espacio exacto para almacenar las hebras correspondientes en los tubos de salida T_{on} y T_{off}. Así, se hace necesaria una función de conteo de hebras previo a la ejecución de *SEPARATE* (con su correspondiente contribución al tiempo de ejecución), o manejar un espacio fijo de memoria para los tubos (provocando un gran desperdicio de memoria).

2. **Transferencia de hebras.** En las operaciones de transferencia, se requiere el movimiento de hebras entre diferentes tubos. Cuando los tubos se implementan en diferentes estructuras cada uno, es necesario pasar hebras de una estructura tubo a otra, implicando movimientos de información en diferentes localidades físicas de memoria. Una vez más, debido a que en la implementación se considera manejar una cantidad grande de hebras (lo mayor posible), estas transferencias consumirían una buena cantidad de tiempo de ejecución, lo que afectaría negativamente el desempeño de la plataforma.

Por estos motivos, se decidió no implementar el almacenamiento de los tubos en diferentes estructuras como sería el enfoque natural. En lugar de esto, la alternativa que resultó más conveniente fue almacenar todas las hebras en una sola estructura *TubeRack* (haciendo alusión a un “estante"de tubos) y almacenar la información de los tubos en las mismas hebras, es

decir, cada hebra registra el tubo al cual pertenece. *TubeRack* almacena R cadenas S_i, donde cada S_i está asignada a un tubo T_j. La estructura *TubeRack* puede definirse formalmente como sigue:

$$TubeRack = \{S_i \in T_j : 0 \leq i \leq R-1; 0 \leq j \leq MAX\_TUBES - 1\}, \tag{3}$$

donde MAX_TUBES es una constante que define el número máximo de tubos a utilizar. Esto se realizó agregando un campo a la estructura hebra para guardar el tubo correspondiente. El primer inconveniente que surge de este enfoque es que el tamaño de las hebras es mayor, aunque su manejo se simplifica. La Figura 25 muestra la diferencia entre los enfoques de implementación de los tubos. Es importante señalar que aunque esta modificación no corresponde a la definición del modelo de etiquetas, es equivalente a ella y en esencia su operación es la misma.

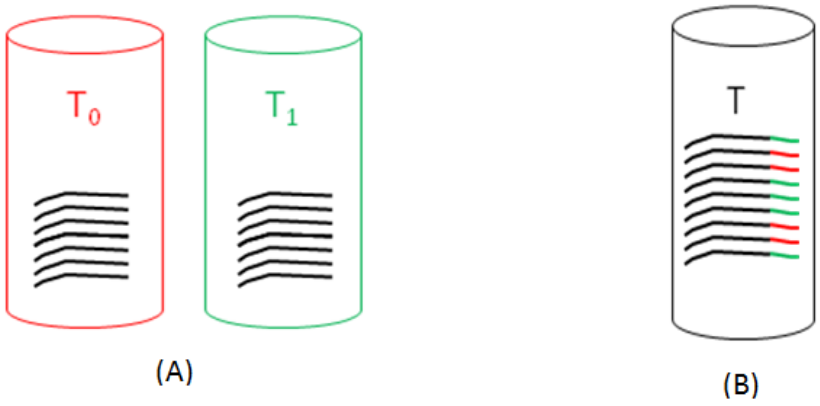


Figura 25: Diferentes enfoques para el manejo de los tubos. (A) Diferentes tubos. (B) Un solo tubo, las hebras registran el tubo al que pertenecen.

4.6. Funciones de acceso a bits

Una vez definida la forma de almacenar las hebras, es necesario determinar los mecanismos de acceso a nivel de bit en los arreglos de enteros. Esto es, dado un arreglo de enteros de $SIZE\_INT$ bits cada uno, diseñar funciones de encendido, apagado y consulta del estado de una posición *index* de bit dentro del arreglo. La Figura 26 ilustra el ejemplo de un arreglo de cuatro enteros que almacena una cadena de $K = 30$ bits, con $SIZE\_INT = 8$ bits y una posición de bit $index = 19$.

Aunque en la codificación de las hebras no importa la significancia de los bits, puesto que no se toma en cuenta el valor posicional del bit sino simplemente su estado, la convención que seguimos para su indexación es de izquierda a derecha. Así, el bit cero es el que se encuentra en la posición más a la izquierda del entero más a la izquierda. De manera similar, los índices de los enteros también se toman de izquierda a derecha.

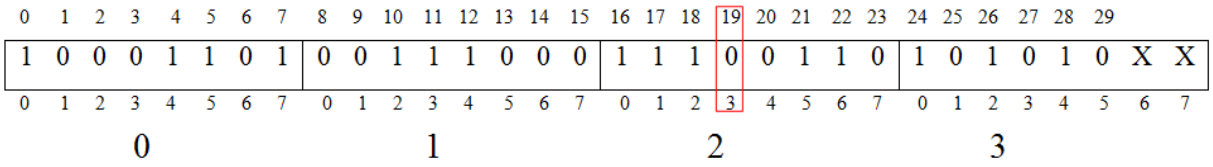


Figura 26: Posiciones de bits dentro de un arreglo de 4 enteros tomando 8 bits por entero. Los números en la parte de arriba de la cadena indican las posiciones de bit en la cadena y los números por debajo de la cadena indican las posiciones dentro de cada entero. En la parte inferior se indican las posiciones de los enteros.

Nótese que la posición de bit $index = 19$ corresponde a la posición de bit 3 dentro del entero 2, en el arreglo de enteros. Sean $intPOS$ y $bitPOS$ el índice del entero que contiene la posición de bit $index$ y la posición de bit que corresponde a $index$ dentro de tal entero, respectivamente. Es sencillo ver que

$$intPOS = \frac{index}{SIZE\_INT} \tag{4}$$

y que

$$bitPOS = (index)MOD(SIZE\_INT) \tag{5}$$

para todo $0 \leq index \leq K - 1$.

Una vez que se ha calculado la posición del entero y el índice de bit dentro del entero, sólo resta determinar la forma de encender, apagar y consultar el estado del bit. Sean x, y y z tres enteros, de manera que:

- x es el entero en la posición $intPOS$ en el arreglo.
- y es el entero que tiene todas sus posiciones de bit apagadas a excepción de la posición $bitPOS$.
- z es el complemento a uno de y (operación NOT bit a bit).

Para encender un bit en x, se lleva a cabo una operación lógica OR bit a bit, entre x y y, escribiendo el resultado en x (Figura 27(A)). Para apagar un bit en x, se realiza una operación lógica AND bit a bit, entre x y z, escribiendo el resultado en x (Figura 27(B)). Para consultar el estado de un bit en x, se hace una operación lógica AND bit a bit, entre x y y, escribiendo el resultado en otro entero w. El bit se encuentra apagado si $w = 0$ y encendido en caso contrario (Figura 27(C)).

Tomando en cuenta lo anterior, se implementaron las funciones *setStrandIndex* (Algoritmo 3), *clearStrandIndex* (Algoritmo 4) y *getStrandIndex* (Algoritmo 5), que reciben como entrada una hebra S y un índice *index* y encienden, apagan y consultan el estado del bit *index* en la hebra S, respectivamente. El tiempo de ejecución de las tres funciones es $\Theta(1)$.

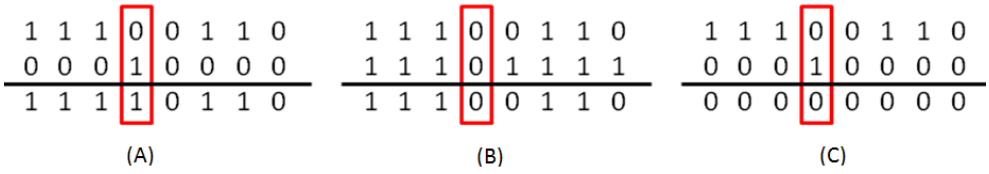


Figura 27: Encendido, apagado y consulta de bit (según ejemplo en Figura 26). (A) $x = x \vee y$. (B) $x = x \wedge z$. (C) $w = x \wedge y$.

Algoritmo 3: *setStrandIndex*(S, *index*)

Entrada : Hebra S, índice de bit *index*.

Salida : Hebra S con posición de bit *index* encendida.

1 $intPOS \leftarrow \lfloor \frac{index}{SIZE\_INT} \rfloor$;

2 $bitPOS \leftarrow MOD(index, SIZE\_INT)$;

3 $x \leftarrow S[intPOS]$;

4 $y \leftarrow 2^{SIZE\_INT - bitPOS}$;

5 $x \leftarrow x \vee y$;

6 Regresar S;

Algoritmo 4: *clearStrandIndex*(S, *index*)

Entrada : Hebra S, índice de bit *index*.

Salida : Hebra S con posición de bit *index* apagada.

1 $intPOS \leftarrow \lfloor \frac{index}{SIZE\_INT} \rfloor$;
2 $bitPOS \leftarrow MOD(index, SIZE\_INT)$;
3 $x \leftarrow S[intPOS]$;
4 $y \leftarrow \neg(2^{SIZE\_INT - bitPOS})$;
5 $x \leftarrow x \wedge y$;
6 Regresar S;

Algoritmo 5: *getStrandIndex*(S, *index*)

Entrada : Hebra S, índice de bit *index*.

Salida : TRUE si posición de bit *index* está encendida en S, FALSE, en caso contrario.

1 $intPOS \leftarrow \lfloor \frac{index}{SIZE\_INT} \rfloor$;
2 $bitPOS \leftarrow MOD(index, SIZE\_INT)$;
3 $x \leftarrow S[intPOS]$;
4 $y \leftarrow 2^{SIZE\_INT - bitPOS}$;
5 $w \leftarrow x \wedge y$;
6 **if** $w = 0$ **then**
7 | Regresar FALSE;
8 **else**
9 | Regresar TRUE;
10 **end**

4.7. Ordenamiento de hebras por tubo

Las operaciones de transferencia modifican el tubo al que pertenece cada hebra que se encuentra registrado dentro de la misma. Dado que todas las hebras se almacenan en una sola estructura *TubeRack*, si no se mantiene un orden en base a los tubos, las hebras que pertenecen a un mismo tubo se encontrarán dispersas dentro de la estructura. Como conse-

cuencia, se tienen que considerar todas las hebras dentro de la estructura cuando se quiera realizar alguna operación sobre un tubo particular T_j, en lugar de realizar la operación solamente sobre las hebras de ese tubo T_j. Por ejemplo, suponga que después de realizar una operación *SEPARATE* las hebras dentro de la estructura quedan como en la Figura 28(A) y que posteriormente, se tiene una operación sobre el tubo 2. Debido a que no se sabe en qué posiciones quedaron las hebras que pertenecen al tubo 2, se tiene que considerar a todas las hebras, y verificar, una por una, si pertenecen a tal tubo. En contraste, si se conoce el intervalo de posiciones en las cuales se encuentran las hebras del tubo 2, solamente se considera ese intervalo (Figura 28(B)).

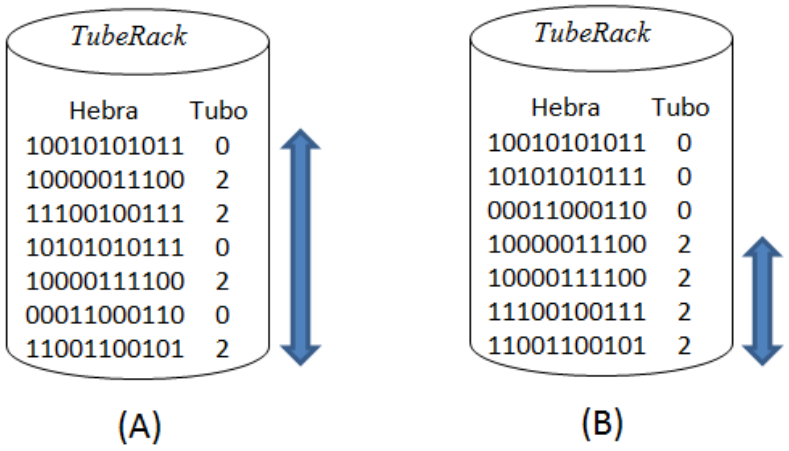


Figura 28: Ordenamiento de las hebras en base a los tubos a los que pertenecen. (A) Sin orden: se debe recorrer toda la estructura. (B) Con orden: se recorre solamente el intervalo.

Nótese que esta situación solamente se presenta después de una operación de transferencia de hebras, pues son las que cambian las hebras de tubo. Mantener el orden de las cadenas por tubo en la estructura $TubeRack$, significa mantener todas las cadenas $S_i \in T_j$ en un intervalo $[l_j, ..., u_j]$ de posiciones adyacentes dentro de la estructura $TubeRack$. Para mantener este orden se implementó lo siguiente:

- Dos arreglos L y U que mantienen los intervalos de las hebras que pertenecen a cada tubo T_j. $L[j]$ y $U[j]$ mantienen la posición inicial y final, respectivamente, de las hebras que pertenecen al tubo T_j dentro de la estructura $TubeRack$. Por convención, si $L[j] = U[j] = -1$, entonces $T_j = \emptyset$ (Figura 29).

- Una función $SortTubes$ que ordena las hebras por tubo dentro de la estructura $TubeRack$ y mantiene los intervalos en los arreglos L y U (Algoritmo 6).

- Después de la operación *DISCARD* se realiza un ordenamiento con *SortTubes*.
- Si la operación *MERGE* se realiza sobre dos tubos adyacentes dentro de *TubeRack*, no se requiere ordenamiento posterior, únicamente actualizar los arreglos L y U. Si los tubos no son adyacentes, se ordena con *SortTubes*.
- La operación *SEPARATE* hace la separación de manera "ordenada", esto es, al escribir el tubo correspondiente de cada hebra, dependiendo del estado del bit i, mantiene las hebras de un mismo tubo destino, en el mismo intervalo (adyacentes). Al final, actualiza los intervalos correspondientes en los arreglos L y U.

TubeRack

Pos	0 ... 21	22 ... 50	51 ... 70	71 ... 71	72 ... 100
Tubo	3	1	2	4	5

Arreglos de intervalos L y U

Pos	0	1	2	3	4	5
L	-1	22	51	0	71	72
U	-1	50	70	21	71	100

Figura 29: Arreglos L y U con los intervalos de posiciones de cada tubo.

Además, para el correcto funcionamiento de lo anterior, se requiere imponer las siguientes restricciones:

- Se debe conocer el número de tubos a utilizar o no sobrepasar un máximo MAX_TUBES preestablecido de tubos.
- La operación *SEPARATE* debe tener ambos tubos destino vacíos.

No obstante, ninguna de estas restricciones introduce alguna limitante en la implementación de algoritmos del modelo de etiquetas, como se explica a continuación:

- La primera restricción es solamente para hacer explícita una condición ya existente en el modelo de etiquetas, pues en todos los algoritmos existentes en la literatura, la cantidad de tubos a utilizar es constante y conocida de antemano.

- Si se necesitara utilizar un tubo T_x no vacío como tubo destino en la operación *SEPARATE*, se podría realizar haciendo la operación *SEPARATE* sobre un tubo temporal T_y vacío, y posteriormente unir los tubos T_x y T_y con una operación *MERGE*.

Algoritmo 6: *SortTubes(TubeRack*

Entrada : Estructura *TubeRack* con la lista de R cadenas $\langle S_0, S_1, ..., S_{R-1} \rangle$. Cada $S_i \in T_j, 0 \leq j \leq MAX\_TUBES - 1$.

Salida : Estructura *TubeRack* con la lista de cadenas en orden no decreciente por tubo. Arreglos L y U con las posiciones inicial y final de las cadenas que pertenecen a cada tubo T_j.

```
1  p ← 0;
2  for j ← 0 to MAX_TUBES − 1 do
3      if T_j ≠ ∅ then
4          L[j] ← p;
5          for k ← p to R − 1 do
6              if S_k ∈ T_j then
7                  Intercambiar hebras S_i ↔ S_p;
8                  p ← p + 1;
9              end
10         end
11         U[j] ← p − 1;
12     end
13 end
```

El algoritmo *SortTubes* recorre la estructura *TubeRack* desde la posición p hasta $R-1$, para cada tubo T_j no vacío. En cada iteración del ciclo de las líneas $(2-13)$, coloca las cadenas $S_k \in T_j$ en el intervalo de posiciones adyacentes $[p...p+|T_j|-1]$ en *TubeRack*. Dado que el número de tubos T_j es constante, el algoritmo *SortTubes* ordena las cadenas por tubo en $\Theta(R)$ pasos, siendo R el número total de cadenas en *TubeRack*.

4.8. Implementación de las operaciones

4.8.1. Implementación secuencial

Para la implementación secuencial de las cinco operaciones del modelo de etiquetas se consideraron dos opciones: con ordenamiento de las cadenas por tubo y sin ordenamiento. El ordenamiento con el algoritmo *SortTubes* agrega una complejidad de $\Theta(R)$ pasos en la operación *DISCARD*. Lo mismo sucede en la operación *MERGE* cuando los tubos a unir no son adyacentes dentro de la estructura $TubeRack$. Dado que, en general, la cantidad R de cadenas en la estructura $TubeRack$ es muy grande, este ordenamiento puede consumir mucho tiempo de ejecución. Sin embargo, el tiempo invertido en ordenamiento se compensa en las operaciones, la cuales se realizan solamente sobre las hebras de un tubo particular, en lugar de la estructura completa. Así, el ordenamiento es un factor de intercambio entre invertir tiempo en llevarlo a cabo y disminuir el tiempo necesario para las operaciones del modelo de etiquetas.

SET

La operación *SET* toma un tubo de entrada T_j y una posición de bit *idx* y enciende la posición *idx* en cada una de las cadenas $S_i \in T_j$. Sin ordenamiento, esta operación consiste en un ciclo que recorre las R cadenas en la estructura $TubeRack$ y en cada iteración k, enciende la posición de bit *idx* de cada cadena $S_k \in T_j$. Esta operación se realiza con el Algoritmo 7 en $\Theta(R)$ pasos.

Algoritmo 7: $SET1(T_j,\ idx)$

Entrada : Tubo de entrada T_j, índice *idx*

Salida : Cadenas $S_i \in T_j$ con posición *idx* encendida.

```
1 for k ← 0 to R − 1 do
2 |   if S_k ∈ T_j then
3 |   |   setStrandIndex(S_k, idx);          ▷ Encender bit idx en k-ésima cadena
4 |   end
5 end
```

Con ordenamiento, el recorrido del ciclo se reduce al intervalo de las cadenas que pertenecen al tubo T_j. Este intervalo es $[L[j]...U[j]]$. Con esto, el número de pasos de la operación

SET se reduce a $\Theta(|T_j|)$. El Algoritmo 8 muestra este cambio.

Algoritmo 8: $SET2(T_j,\ idx)$

Entrada : Tubo de entrada T_j, índice *idx*, arreglos L y U.

Salida : Cadenas $S_i \in T_j$ con posición *idx* encendida.

```
1 for k ← L[j] to U[j] do
2 |   setStrandIndex(S_k, idx);          ▷ Encender bit idx en k-ésima cadena
3 end
```

CLEAR

La operación *CLEAR* es casi idéntica a *SET*, con la única diferencia del apagado del bit *idx*, en lugar de encenderlo. Los Algoritmos 9 y 10 realizan la operación *CLEAR* sin ordenamiento y con ordenamiento, respectivamente.

Algoritmo 9: $CLEAR1(T_j,\ idx)$

Entrada : Tubo de entrada T_j, índice *idx*

```
1 . Salida : Cadenas S_i ∈ T_j con posición idx encendida.
2 for k ← 0 to R − 1 do
3 |   if S_k ∈ T_j then
4 |   |   clearStrandIndex(S_k, idx);      ▷ Apagar bit idx en k-ésima cadena
5 |   end
6 end
```

Algoritmo 10: $CLEAR2(T_j,\ idx)$

Entrada : Tubo de entrada T_j, índice *idx*, arreglos L y U.

Salida : Cadenas $S_i \in T_j$ con posición *idx* apagada.

```
1 for k ← L[j] to U[j] do
2 |   clearStrandIndex(S_k, idx);        ▷ Apagar bit idx en k-ésima cadena
3 end
```

SEPARATE

La operación *SEPARATE* toma un tubo de entrada T_j y un índice de bit *idx*, y transfiere las cadenas con la posición de bit *idx* encendida a un tubo T_r, y las cadenas con la posición de bit *idx* apagada a un tubo T_s. El tubo de entrada T_j queda vacío al final. La complejidad de la operación es $\Theta(R)$ pasos (Algoritmo 11).

Algoritmo 11: $SEPARATE1(T_j,\ T_r,\ T_s,\ idx)$

Entrada : Tubo de entrada T_j, índice *idx*.

Salida : Tubos $T_r = \{S_i \in T_j : idx = 1\}$ y $T_s = \{S_i \in T_j : idx = 0\}$.

```
1 for k ← 0 to R − 1 do
2     if S_k ∈ T_j then
3         if getStrandIndex(S_k, idx) then
4             T_r ← T_r ∪ {S_k};
5         else
6             T_s ← T_s ∪ {S_k};
7         end
8     end
9 end
```

Considerando el orden en $TubeRack$, la oparación *SEPARATE* debe mantener las cadenas de los tubos de salida T_r y T_s, en posiciones adyacentes en $TubeRack$. Para esto, se escriben las cadenas de T_r al inicio del intervalo del tubo de entrada T_j, en el intervalo de posiciones $[L[j], L[j] + 1, ..., U[r]]$. Así, las cadenas de T_s terminan al final del intervalo del tubo de entrada T_j, en las posiciones $[U[r]+1, U[r]+2, ..., U[j]]$. Este manejo de posiciones no agrega complejidad adicional a la operación *SEPARATE* que en este caso, al recorrer solamente las cadenas de T_j es $\Theta(|T_j|)$.

Algoritmo 12: *SEPARATE2*(T_j, T_r, T_s, *idx*)

Entrada : Tubo de entrada T_j, índice *idx*, arreglos L y U.

Salida : Tubos $T_r = \{S_i \in T_j : idx = 1\}$ y $T_s = \{S_i \in T_j : idx = 0\}$, arreglos L y U con intervalos actualizados.

```
1  p ← L[j];
2  for k ← L[j] to U[j] do
3      if getStrandIndex(S_k, idx) then
4          T_r ← T_r ∪ {S_k};
5          Intercambiar hebras S_k ↔ S_p;
6          p ← p + 1;
7      else
8          T_s ← T_s ∪ {S_k};
9      end
10 end
11 L[r] ← L[j];
12 U[r] ← p − 1;
13 L[s] ← p;
14 U[s] ← U[j];
15 L[j] ← −1;
16 U[j] ← −1;
```

El Algoritmo 12 realiza la operación de separación manteniendo el orden descrito. En la línea 1, se inicializa el apuntador p que indica la siguiente posición disponible para escribir una cadena en T_r. El ciclo de las líneas $(2-10)$ recorre las cadenas de T_j, verificando el estado del bit *idx* en cada iteración, para enviar las cadenas a los tubos T_r o T_s cuando *idx*=1 o *idx*=0, respectivamente. Las líneas $(11-14)$ actualizan los apuntadores en los arreglos L y U para los tubos de salida T_r y T_s. Al final, las líneas $(15-16)$ actualizan los apuntadores en L y U para indicar que el tubo de entrada T_j termina vacío.

MERGE

La operacón *MERGE* une el contenido de dos tubos T_j y T_r, esto es, pasa las cadenas de T_r a T_j. Sin ordenamiento, esta operación se realiza en $\Theta(R)$ pasos (Algoritmo 13).

Algoritmo 13: $MERGE1(T_j, T_r)$

Entrada : Tubos de entrada T_j y T_r.

Salida : Tubo $T_j = T_j \cup T_r$.

```
1 for k ← 0 to R − 1 do
2 |   if S_k ∈ T_r then
3 |   |   T_j ← T_j ∪ {S_k};
4 |   |   T_r ← T_r \ {S_k};
5 |   end
6 end
```

Con ordenamiento, el número de pasos se reduce a $\Theta(|T_r|)$. Sin embargo, cuando los tubos T_j y T_r no son adyacentes, se debe ordenar con *SortTubes* (línea 8), lo cual agrega $\Theta(R)$ pasos adicionales. La adyacencia de dos tubos con la función *adyacent* (línea 5) y la actualización de los intervalos con *updateIntervals* (línea 6) se pueden comprobar en tiempo $\Theta(1)$, pues sólo requieren verificar y modificar, respectivamente, los límites en los arreglos L y U, para los tubos T_j y T_r. El Algoritmo 14 realiza la operación *MERGE* con ordenamiento.

Algoritmo 14: $MERGE2(T_j, T_r)$

Entrada : Tubos de entrada T_j y T_r, arreglos L y U.

Salida : Tubo $T_j = T_j \cup T_r$, arreglos L y U con intervalos actualizados.

```
1 for k ← L[r] to U[r] do
2 |   T_j ← T_j ∪ {S_k};
3 |   T_r ← T_r \ {S_k};
4 end
5 if adyacent(T_j, T_r) then
6 |   updateIntervals(T_j, T_r, L, U);
7 else
8 |   SortTubes;
9 end
```

DISCARD

La operación *DISCARD* vacía el contenido de un tubo T_j. El Algoritmo 15 realiza esta operación sin considerar ordenamiento. Recorre las R cadenas en $TubeRack$ (ciclo de las líneas $(1-5)$) y elimina aquellas que pertenecen a T_j (línea 3). La complejidad del algoritmo es también $\Theta(R)$ pasos.

Algoritmo 15: $DISCARD1(T_j)$

Entrada : Tubo de entrada T_j.

Salida : Tubo $T_j = \emptyset$.

```
1 for k ← 0 to R − 1 do
2 |   if S_k ∈ T_j then
3 |   |   T_j ← T_j \ {S_k};
4 |   end
5 end
```

Con ordenamiento, se recorren solamente las cadenas de T_j (ciclo de las líneas $(1-3)$) en $\Theta(|T_j|)$ pasos. Sin embargo, se requiere llamar a *SortTubes* (línea 4) para ordenar la estructura $TubeRack$, agregando $\Theta(R)$ pasos. El Algoritmo 16 muestra los pasos para realizar la operación *DISCARD* con ordenamiento.

Algoritmo 16: $DISCARD2(T_j)$

Entrada : Tubo de entrada T_j, arreglos L y U.

Salida : Tubo $T_j = \emptyset$, arreglos L y U con intervalos actualizados.

```
1 for k ← L[j] to U[j] do
2 |   T_j ← T_j \ {S_k};
3 end
4 SortTubes;
```

4.8.2. Implementación paralela

La implementación paralela de las operaciones del modelo de etiquetas en CUDA se realizó siguiendo el enfoque simple de paralelismo de datos. Cada hebra se puede accesar de manera independiente con respecto a las demás dentro de la estructura $TubeRack$. En la implementación secuencial, se destina una iteración k que se encarga de realizar alguna

operación sobre la k-ésima hebra en el tubo, ya sea encender o apagar un bit dentro de la hebra, transferirla a otro tubo, o desecharla. El enfoque simple de paralelización en CUDA consiste en cambiar iteraciones por hilos. Así, se destina un hilo para que se encargue de realizar las operaciones correspondientes para cada hebra, de manera concurrente.

Los cambios en los algoritmos secuenciales presentados (sin considerar ordenamiento) que paralelizan su funcionamiento son mínimos. Básicamente, cada ciclo

$$\textbf{for } k \leftarrow 0 \textbf{ to } R-1 \textbf{ do}$$

se convierte en

$$\textbf{for each } \textit{thread } k \textbf{ do}$$

que transforma las instrucciones ejecutadas en cada iteración k, a un hilo k que realiza exactamente lo mismo pero sobre una hebra diferente en $TubeRack$ y de manera concurrente a los demás hilos.

En este capítulo se introduce la heurística propuesta para el Problema de Clique Máximo. Se define la codificación utilizada para representar el espacio de soluciones. Se introduce la mejora implementada para el algoritmo de filtrado. Se realiza el planteamiento del problema y se describen las estructuras utilizadas. Por último, se explica el esquema general de la heurística y los módulos que la componen.

5.1. Representación de cliques

La representación de cliques como cadenas de bits en el presente trabajo está basada en la definición de Biblioteca Pascal (Zimmermann, 2002). Esta definición se extiende introduciendo los conceptos presentados en esta sección.

Una cadena $S_{(n,k)}$ es una cadena de n bits de longitud de los cuales k bits se encuentran encendidos. Por ejemplo, la cadena 10010100 es una cadena $S_{(8,3)}$. Una cadena $\overline{S}_{(n,k)}$ denota a la cadena que se produce al realizar la operación complemento a uno (operación NOT, bit a bit) a una cadena $S_{(n,k)}$ dada. En el ejemplo anterior, la cadena $\overline{S}_{(8,3)}$ correspondiente es 01101011. Nótese que una cadena $\overline{S}_{(n,k)}$ es una cadena $S_{(n,n-k)}$. Una biblioteca $B_{[n,k]}$ es el conjunto de todas las diferentes cadenas $S_{(n,k)}$ posibles. La cantidad de cadenas en $B_{[n,k]}$ es

$$|B_{[n,k)]}| = \binom{n}{k}, \tag{6}$$

que considera todas las formas posibles de tener k bits encendidos en un cadena de n bits y cada forma es una cadena $S_{(n,k)}$ distinta. La biblioteca $\overline{B}_{[n,k]}$ representa la biblioteca complementaria de $B_{[n,k])}$, en otras palabras, es el conjunto de todas las cadenas $\overline{S}_{(n,k)}$, o de forma equivalente, de todas las cadenas $S_{(n,n-k)}$.

Dada una lista $P = \langle p_1, p_2, ..., p_n \rangle$ de n elementos que consiste en una permutación de los elementos de un n-conjunto, una sublista $P' = \langle p'_1, p'_2, ..., p'_k \rangle$ de k elementos de P, con $0 \leq k \leq n$, se puede representar con una cadena $S_{(n,k)}$ de la siguiente forma:

- Haciendo corresponder el j-ésimo bit b_j en la cadena, con el j-ésimo elemento $p_j \in P$, y

- el estado cada bit b_j indica si $p_j \in P'$.

Por ejemplo, para la lista $P = \langle 4, 5, 7, 1, 3, 2, 8, 6 \rangle$ y la sublista $P' = \langle 7, 3, 5, 8 \rangle$, la cadena $S_{(8,4)}$ que representa a P' es 01101010. Esto es, los bits encendidos b_2, b_3, b_5 y b_7, indican que los elementos $p_2 = 5, p_3 = 7, p_5 = 3$ y $p_7 = 8$, están en la sublista P'.

Es importante notar que en esta representación depende el orden de los elementos de la lista P, de forma que, una permutación diferente de tales elementos puede producir una cadena distinta para la misma sublista. Por ejemplo, si $P = \langle 3, 8, 7, 5, 4, 2, 6, 1 \rangle$, para la misma sublista del ejemplo anterior $P' = \langle 7, 3, 5, 8 \rangle$, la cadena $S_{(8,4)}$ que representa a P' es 11110000.

Por definición un clique en un grafo es un subconjunto del conjunto de vértices V. Si $V = \{v_1, v_2, ..., v_n\}$ es el conjunto de vértices, entonces un clique $C = \{v_{i_1}, v_{i_2}, ..., v_{i_k}\}$ de tamaño k, es un k-subconjunto de V. Si consideramos una lista P que sea una permutación de los n vértices, entonces un clique de tamaño k corresponde a una sublista P' de k vértices de P. De esta forma, un clique de tamaño k en un grafo de n vértices, se puede representar con una cadena $S_{(n,k)}$.

De manera similar, el conjunto de todos los posibles cliques de tamaño k en un grafo de n vértices, se representa con una biblioteca $B_{[n,k]}$. La Figura 30 muestra un ejemplo de un grafo y los posibles cliques de tamaño 3.

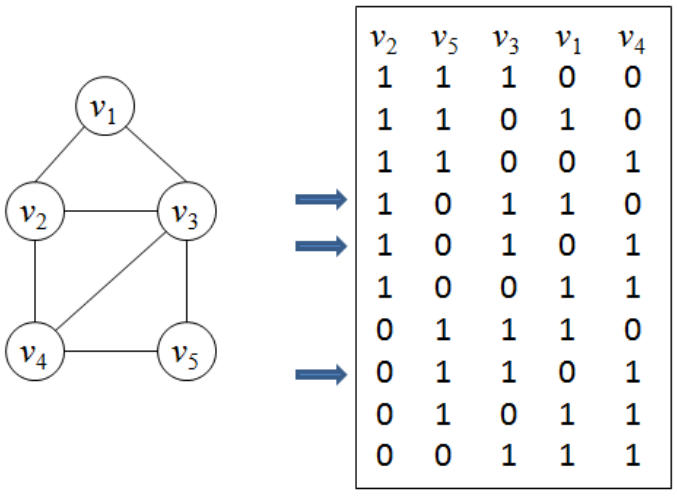


Figura 30: Un grafo de 5 vértices con su correspondiente biblioteca $B_{[5,3]}$ que representa todos los posibles cliques de tamaño 3. Las cadenas $S_{(5,3)}$ señaladas con flecha corresponden a cliques existentes en el grafo para la lista $P = \langle v_2, v_5, v_3, v_1, v_4 \rangle$.

5.2. Algoritmo k-Cliques

El algoritmo k-Cliques introducido en (Martínez-Pérez y Zimmermann, 2009) es un algoritmo de cómputo basado en ADN del modelo de etiquetas. Dado un grafo $G = (V, E)$ y un tubo de entrada T que contiene una biblioteca $B_{[n,k]}$ de hebras de ADN, el algoritmo proporciona todos los cliques de tamaño k en el grafo, eliminando aquellas hebras que codifican un clique no existente en el grafo. Básicamente, el algoritmo k-Cliques funciona como un algoritmo de verificación de certificados de solución para el problema de búsqueda de cliques de tamaño k. Se le proporciona el espacio de soluciones y el algoritmo verifica todas las soluciones en paralelo. El seudocódigo del algoritmo k-Cliques se presenta en el Algoritmo 17.

Algoritmo 17: k-Cliques

Entrada : Grafo $G = (V, E)$, tubo T con biblioteca $B_{[n,k]}$.

Salida : Tubo T con cliques de tamaño k.

```
1  for i ← 1 to n − 1 do
2      for j ← i + 1 to n do
3          SEPARATE(T, T1, T0, i);
4          SEPARATE(T1, T11, T10, j);
5          MERGE(T0, T10, T);
6          if adyacentes(vi, vj) then
7              MERGE(T11, T);
8          end
9          DISCARD(T11);
10     end
11 end
12 if T ≠ ∅ then
13     Regresar T;
14 else
15     reportar "no se encontraron cliques de tamaño k";
16 end
```

En las líneas 3 y 4, se separan en el tubo T_{11} las cadenas que consideran a los vértices v_i y v_j, es decir, las cadenas que tienen las posiciones de bit i y j encendidas. La línea 5 regresa

las cadenas restantes al tubo T para que sean procesadas en iteraciones posteriores. En la línea 6 se prueba la adyacencia entre los vértices v_i y v_j, de manera que si son adyacentes, las cadenas en T_{11} se regresan al tubo T para que sean procesadas en iteraciones posteriores. Por definición, si no existe la adyacencia (v_i, v_j), las cadenas en T_{11} no pueden formar un clique, y son desechadas en la línea 9. Cada iteración de los ciclos anidados de las líneas 1 y 2 produce una combinación diferente de dos vértices distintos (v_i, v_j), asegurando con esto la prueba de adyacencia para cada par de vértices en el grafo. Al final, las líneas 12-16 reportan los cliques en el tubo T, en caso de existir alguno.

5.3. Algoritmo k-Cliques mejorado

El algoritmo k-Cliques realiza exactamente $\frac{n(n-1)}{2}$ iteraciones, por lo que el número de operaciones que realiza es $O(n^2)$. Así, el algoritmo k-Cliques revisa todas las adyacencias posibles en el grafo. Sin embargo, para obtener los cliques de tamaño k, lo que interesa es eliminar aquellas cadenas que consideran a los vértices v_i y v_j, cuando no existe la adyacencia (v_i, v_j). Entonces, una mejora consiste en verificar estas dos condiciones en orden inverso, es decir, primero comprobar que no existe la adyacencia y, entonces, revisar si las cadenas consideran a los vértices v_i y v_j. Esto se realiza haciendo algunos cambios simples en el algoritmo k-Cliques. En el Algoritmo 18 se presentan estos cambios.

Algoritmo 18: k-Cliques2

Entrada : Grafo $G = (V, E)$, tubo T con biblioteca $B_{[n,k]}$.

Salida : Tubo T con cliques de tamaño k.

```
1  for i ← 1 to n − 1 do
2      for j ← i + 1 to n do
3          if ¬adyacentes(v_i, v_j) then
4              SEPARATE(T, T_1, T_0, i);
5              SEPARATE(T_1, T_11, T_10, j);
6              MERGE(T_0, T_10, T);
7              DISCARD(T_11);
8          end
9      end
10 end
11 if T ≠ ∅ then
12     Regresar T;
13 else
14     reportar "no se encontraron cliques de tamaño k";
15 end
```

Aunque la cantidad de iteraciones se mantiene en $\mathrm{O}(n^2)$, la condición en la línea 3 reduce la ejecución del cuerpo del ciclo de forma que solamente se ejecuta cuando los vértices no son adyacentes, en lugar de ejecutarse para cada par de vértices. La cantidad de operaciones disminuye a $\mathrm{O}(\overline{E})$. Esta mejora es particularmente notoria cuando los grafos son densos.

5.4. Planteamiento del problema

El presente trabajo se basa en la codificación del espacio de búsqueda y el filtrado paralelo del Algoritmo 17 para la búsqueda de cliques en un grafo. Como ya se ha visto, el algoritmo k-Cliques proporciona los cliques de tamaño k de un grafo. Sin embargo, para la búsqueda de cliques de tamaño máximo k_{max}, se requiere un ciclo externo que vaya incrementando sucesivamente el tamaño k del clique y se detenga cuando el tubo T termine vacío, reportando los cliques en la iteración anterior, o bien, se puede considerar un ciclo con valores decrecientes

de k, que se detenga cuando el tubo T no termine vacío y reporte los cliques encontrados en esa iteración.

Ahora, aunque la cantidad de cadenas en la biblioteca $B_{[n,k]}$ crece polinomialmente para valores sucesivos de $k = 0, 1, ..., \lfloor \frac{n}{2} \rfloor$, y está acotada por un polinomio Q de grado k ($Q = \mathrm{O}(n^k)$), al aproximarse a valores cercanos a $\lfloor \frac{n}{2} \rfloor$, esta cantidad es exponencial. Más aún, el total de cadenas de todas las bibliotecas $B_{[n,k]}$ con $0 \leq k \leq n$ es

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n, \tag{7}$$

que corresponde a la cardinalidad del conjunto potencia del conjunto V de vértices. La Figura 31 muestra la gráfica de $|B_{[n,k]}| = f(k)$, con $n = 64$ y $0 \leq k \leq n$. Observe que esta función alcanza un máximo para los valores

- $k = \frac{n}{2}$, cuando n es par y,
- $k = \frac{n-1}{2}$ y $k = \frac{n+1}{2}$, cuando n es impar.

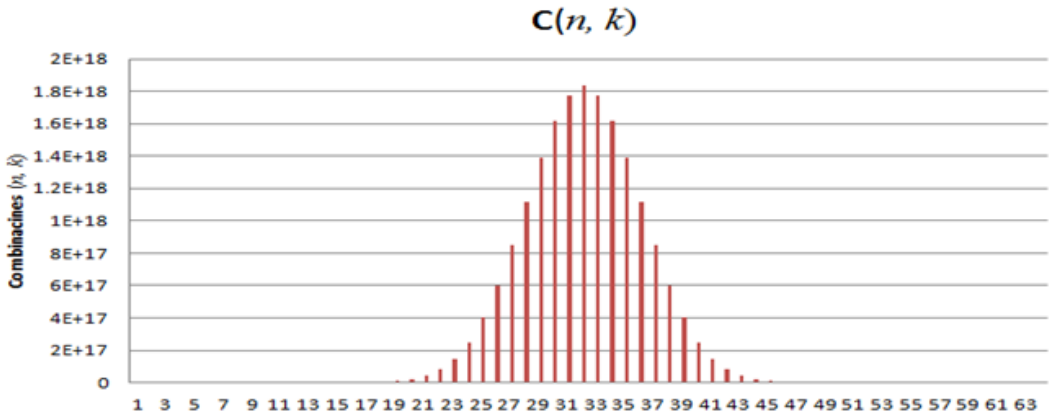


Figura 31: Crecimiento de la función $|B_{[n,k]}| = f(k)$, con $n = 64$ y para valores $0 \leq k \leq n$. La función alcanza un máximo para el valor $k = \frac{n}{2} = 32$.

Además, nótese también que al aproximarse a un valor de $k = \frac{n}{2} = 32$, la cantidad de cadenas es exponencial, haciendo que la generación, almacenamiento y procesado de tal biblioteca $B_{[n,k]}$ sea inviable.

Para tener una mejor idea de los números involucrados, suponga que se desea procesar la biblioteca $B_{[64,32]}$ (que contiene más de 1.8 e+18 cadenas) y que cada cadena se almacena en

64 bits de memoria (8 bytes). Además, para generar, almacenar y procesar tal biblioteca, se divide en bloques de 1 GB de memoria y se cuenta con una función que genera y almacena cada bloque en 10 ms y otra que procesa (filtra) cada bloque en 10 ms. Con estos parámetros, el tiempo que tomaría procesar la biblioteca completa es de aproximadamente 8.66 años. Cabe señalar que en el ejemplo anterior las suposiciones hechas son muy optimistas en cuanto al tiempo de ejecución, el tamaño del problema (considerado modesto con $n = 64$) y que sólo se está tomando en cuenta un valor de k, por lo que es posible tener que repetir el proceso para valores sucesivos de k.

Debido a lo anteriormente expuesto, es indispensable determinar criterios para la selección de un subconjunto de cadenas de una biblioteca $B_{[n,k]}$ de manera que sea posible su generación, almacenamiento y procesado, en un tiempo razonable. Adicionalmente, es necesario diseñar una forma de aproximar el valor k_{max}, es decir, encontrar un valor inicial k_0, de manera que la búsqueda empiece con el valor $k_0 + 1$.

5.5. Estructuras de datos utilizadas

5.5.1. Vector de adyacencias

Con el propósito de mejorar el desempeño de la consulta de adyacencias, utilizando la memoria de constantes de CUDA, se implementó una forma reducida de guardar la información del grafo de entrada. La memoria de constantes es una memoria de acceso rápido, sin embargo, su capacidad es muy limitada, por lo que esta reducción resulta de particular importancia. Como se revisó en el Capítulo 2, existen dos formas comúnmente utilizadas para la representación de grafos: matriz y listas de adyacencias. La primera ocupa $\Theta(|V|^2)$ espacios de memoria y el tiempo de acceso para la consulta de una adyacencia es $O(1)$. Esta representación es la más apropiada para grafos densos (aquellos donde $|E| = O(|V|^2)$). La segunda requiere $\Theta(|V| + |E|)$ espacios en memoria y el tiempo de acceso de consulta de adyacencias es $O(|V|)$. El uso de esta representación resulta adecuado cuando los grafos son dispersos (aquellos donde $|E| = O(|V|)$).

Cuando los grafos son simples, no dirigidos, y las aristas no tienen peso, la matriz de adyacencia es simétrica con respecto a la diagonal principal, de manera que registra las adyacencias por duplicado para cada par de vértices. Entonces, es suficiente almacenar la información que se encuentra en alguno de los dos triángulos, superior o inferior, respecto a

la diagonal principal. La Figura 32 muestra esta condición.

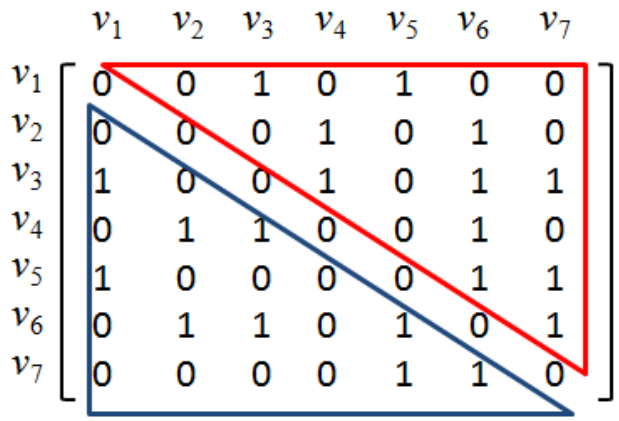


Figura 32: Duplicidad de la información en la matriz de adyacencias. La matriz es simétrica respecto a la diagonal principal, por lo que los triángulos superior e inferior contienen la misma información.

De esta forma, se define la matriz de adyacencia reducida como $A = [a_{ij}](j > i)$, para hacer referencia a los elementos por arriba de la diagonal principal, únicamente. Aunque la complejidad asintótica del espacio de almacenamiento se mantiene en $O(|V|^2)$, en la práctica, la cantidad de espacios de memoria se reduce a menos de la mitad. Por ejemplo, en la Figura 32 la cantidad de espacios de memoria se reduce de 49 a 21.

Además, dado que las aristas no tienen peso, la información que se desea almacenar consiste en datos binarios, por lo que para obtener un máximo aprovechamiento de la memoria se puede emplear un arreglo ADJ de enteros y utilizarlos bit por bit, tal como se describió en el Capítulo 4. En este caso, la cantidad de bits necesarios para almacenar las adyacencias del grafo está dada por

$$Bits = \frac{n(n-1)}{2}. \tag{8}$$

Si $SIZE\_INT$ es la cantidad de bits por entero, entonces el número de enteros necesarios es

$$\frac{Bits}{SIZE\_INT}, \tag{9}$$

si la división es exacta. Si la división tiene módulo se requiere un entero más.

El arreglo ADJ almacena la matriz $A = [a_{ij}](j > i)$ de forma "vectorizada". Esto es, primero los $n-1$ bits correspondientes a las adyacencias de v_1, luego los $n-2$ bits de v_2, y así sucesivamente. Para cada vértice v_i se guardan las adyacencias (v_i, v_j) para $i < j \leq n$.

La Figura 33 describe lo anterior para el ejemplo de la Figura 32, y muestra el arreglo ADJ de enteros, suponiendo un valor $SIZE\_INT = 6$.

v_i	v_0						v_1					v_2			v_3			v_4		v_5			
v_j	v_1	v_2	v_3	v_4	v_5	v_6	v_2	v_3	v_4	v_5	v_6	v_3	v_4	v_5	v_6	v_4	v_5	v_6	v_5	v_6	v_6			
(v_i, v_j)	0	1	0	1	0	0	0	1	0	1	0	1	0	1	1	0	1	0	1	1	1			
Adj	0	1	0	1	0	0	0	1	0	1	0	1	0	1	1	0	1	0	1	1	1	x	x	x

Figura 33: Forma de almacenamiento del vector de adyacencias ADJ tomando el ejemplo de la Figura 32. Se supone un valor $SIZE\_INT = 6$. Las líneas punteadas indican las adyacencias (v_i, v_j) con $i < j < n$, para cada vértice v_i con $0 \leq i < n-1$. Las líneas continuas indican los diferentes enteros en el arreglo ADJ. Las x representan bits no utilizados en el arreglo ADJ.

Por conveniencia, la indexación de los vértices cambia de $\{1, 2, ..., n\}$ a $\{0, 1, ..., n-1\}$. De esta forma, los valores que toman los índices i, j son: $0 \leq i < n-1$ y $i < j < n$. Así, solo resta determinar la posición de bit *index* dentro del vector de adyacencias ADJ que corresponda a la adyacencia (v_i, v_j) dados los índices i y j. Para esto, primero se calcula el desplazamiento de las adyacencias para un valor de i. Esto es, la posición $offset_1$ donde inician las adyacencias del i-ésimo vértice. Dado que el i-ésimo vértice tiene $n - (i + 1)$ adyacencias, entonces para $i > 0$ la posición $offset_1$ del i-ésimo vértice es la suma de las anteriores:

$$offset_1 = (n-1) + (n-2) + \ldots + (n-i). \tag{10}$$

Entonces, al ser i términos la suma es

$$offset_1 = i \times n - \sum_{r=1}^{i} r. \tag{11}$$

Note que con esta expresión, se incluye la posición para $i = 0$. La Figura 34 muestra los desplazamientos $offset_1$ utilizando el mismo ejemplo de las Figuras 32 y 33.

La posición calculada con $offset_1$ accede a la adyacencia (v_i, v_j) con $j = i + 1$, para cada vértice v_i. Entonces, es necesario agregar un desplazamiento $offset_2$ en función de todos los valores de j en el intervalo $[i + 1, n)$. Esto es, para los valores sucesivos de $j = i+1, i+2, i+3, ..., n-1$ corresponden los valores sucesivos de $offset_2 = 0, 1, 2, ..., n-i-2$. De manera

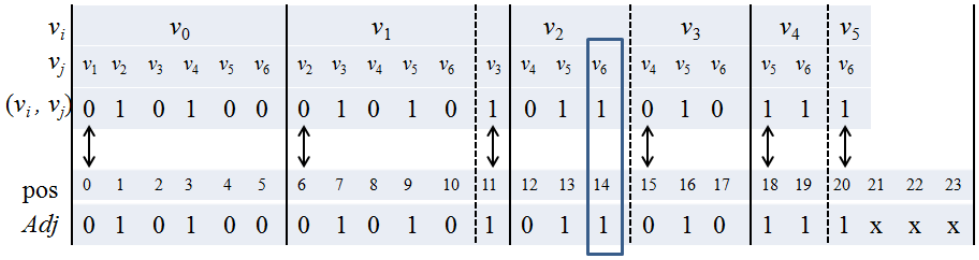


Figura 34: Cálculo de posición *index* en vector de adyacencia *ADJ*. Para cada adyacencia (v_i, v_j) se muestra la posición correspondiente en el arreglo *ADJ*. Las flechas indican los desplazamientos $offset_1$ calculados con la Fórmula 11 para cada vértice v_i, con $0 \leq i < n-1$. El rectángulo indica un ejemplo para la adyacencia (v_2, v_6) a la que corresponde la posición 14 en el arreglo *ADJ*. Esta posición se calcula con la Fórmula 13.

general

$$offset_2 = j - (i+1). \tag{12}$$

Así, la posición de bit *index* dentro del arreglo *ADJ* está dada por

$$index = offset_1 + offset_2 = i \times n - \sum_{r=0}^{i} r + j - (i+1), \tag{13}$$

la cual se puede calcular en tiempo O(1) sustituyendo la suma $\sum_{r=0}^{i} r$ por la expresión equivalente $\frac{i(i+1)}{2}$.

Por ejemplo, en la Figura 34, el rectángulo indica que la adyacencia (v_2, v_6) se guarda en la posición $index = 14$. Utilizando la Fórmula 13, se puede calcular esta posición a partir de los valores $i = 2$ y $j = 6$, con $n = 7$:

$$index = (2)(7) - \frac{2(2+1)}{2} + 6 - (2+1) = 14,$$

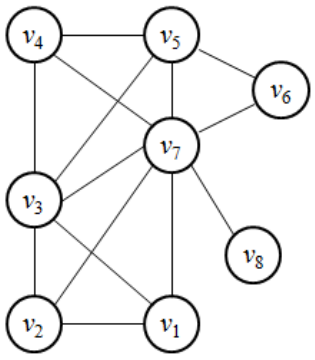
y así para cualquier par de valores (i, j) tales que $0 \leq i < n-1$, $i < j < n$.

Una vez calculada la posición de bit *index* en función de los valores i, j de la adyacencia que se desea acceder, se utiliza un procedimiento muy similar a las funciones *setStrandIndex* y *getStrandIndex* vistas en el Capítulo 4, para registrar y consultar una adyacencia, respectivamente. Es importante señalar que ambos procedimientos, de registro y consulta de adyacencia, llevan a cabo su función en una cantidad O(1) de operaciones.

5.5.2. Vector de grados

En la presente implementación se consideran distintos ordenamientos del conjunto V de vértices. Particularmente, se desea mantener ordenados los vértices de acuerdo a su grado, de mayor a menor. También, se utiliza la información de los grados para seleccionar un vértice con una probabilidad proporcional a su grado. Los vértices con mayor grado son seleccionados con mayor probabilidad. Todas estas consideraciones se verán a detalle en secciones posteriores.

Para mantener la información de los vértices y sus grados, de manera que se puedan considerar distintos ordenamientos, se utilizó un vector de grados. El vector de grados DEG es un arreglo de n posiciones que almacena una permutación del conjunto de vértices del grafo (índices) y el grado correspondiente de cada vértice. La Figura 35 muestra, a manera de ejemplo, un vector de grados para un grafo de 8 vértices.



(A)

v_i	v_7	v_3	v_5	v_4	v_2	v_1	v_6	v_8
$d(v_i)$	7	5	4	3	3	3	2	1

(B)

Figura 35: (A)Un grafo $G = (V, E)$ con $V = \{v_1, v_2, ..., v_8\}$ y (B) su correspondiente vector de grados ordenado de mayor a menor.

5.5.3. Vector de cliques

El vector de cliques CV es una estructura que mantiene la información de los cliques encontrados hasta algún momento. La estructura está formada por los siguientes campos:

- Un arreglo CLQS que almacena un vector de grados en cada posición. Cada vector de grados corresponde a un clique encontrado.

- Un valor *Csize* que guarda el tamaño de los cliques encontrados. Todos los cliques en el arreglo CLQS tienen el mismo tamaño *Csize*.
- Un valor *size* que indica el tamaño del vector de cliques, es decir, la cantidad de cliques almacenados en el arreglo CLQS. Este valor también es un apuntador que señala la siguiente posición disponible para guardar un nuevo clique en el arreglo CLQS.
- Un valor *next* que indica el siguiente clique a ser procesado, en otras palabras, la siguiente posición en el arreglo CLQS a ser procesada.

En la Figura 36 se muestra cómo se mantiene la información de los cliques en la estructura CV. Cada posición en el arreglo CLQS es un ordenamiento de los vértices, es decir, un vector de grados. En la figura se omiten los grados de los vértices, pero también están almacenados en cada vector de grados. El campo $Csize$ indica cuáles vértices pertenecen al clique y cuáles no, para cada vector de grados. Los primeros $Csize$ vértices de cada vector de grados forman un clique. Los $n-Csize$ vértices restantes, son *candidatos* para extender los cliques almacenados en cada posición. Los campos *size* y *next* son apuntadores a posiciones dentro del arreglo CLQS, como se decribió anteriormente.

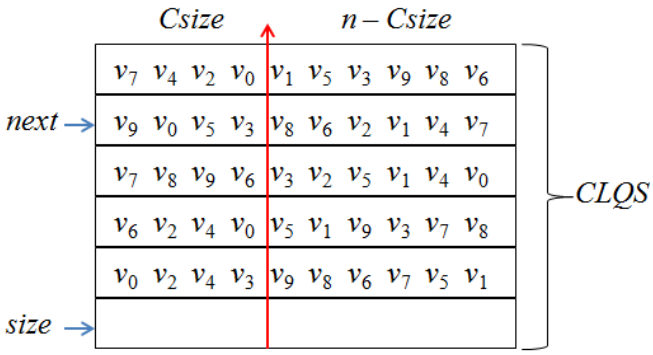


Figura 36: Ejemplo de vector de cliques. Cada posición es un vector de grados (permutación de los vértices). El valor $Csize$ indica el tamaño de los cliques y los valores *next* y *size* indican la siguiente posición a ser procesada y la siguiente posición disponible para un nuevo clique, respectivamente.

5.5.4. Vector de hebras

Para la codificación del espacio de búsqueda se utilizó una estructura tipo arreglo que almacena una hebra y el tubo al que pertenece en cada posición. Cada hebra codifica un posible

clique en el grafo. El arreglo *TubeRack* almacena este vector de hebras. La implementación del vector de hebras se realizó como fue descrito en el Capítulo 4.

Es importante tener en consideración que las hebras no guardan información sobre los índices de los vértices, sino de la posición en el conjunto de vértices. Es decir, cada hebra representa un subconjunto de vértices, donde el i-ésimo bit representa al i-ésimo vértice, esto es, al vértice en la i-ésima posición en la permutación, pero no al vértice que tiene el índice i. El estado del i-ésimo bit indica la inclusión ("*on*") o no inclusión ("*off*") del i-ésimo vértice, en el subconjunto representado por la hebra. Es por esta razón que es necesario guardar la información sobre diferentes ordenamientos de los vértices, como se revisó en la Sección 5.5.2.

5.6. Funciones de filtrado paralelo

En esta sección se describe la implementación de tres diferentes enfoques de filtrado paralelo en CUDA. Se presentan tres algoritmos que llevan a cabo la misma función: dado el vector de adyacencia del grafo y un vector de hebras que almacena un subconjunto $B'_{[n,k]} \subseteq B_{[n,k]}$, determinar cuáles de las hebras codifican un clique de tamaño k existente en el grafo. Cada algoritmo presenta diferencias técnicas en cuanto a implementación, que produce diferente desempeño en los tiempos de ejecución.

La función de filtrado paralelo consiste en "marcar" aquellas hebras que codifican un clique no existente en el grafo. Esta operación de marcado se implementa utilizando los tubos a los que pertenece cada hebra. La Figura 37 describe esta operación. Al inicio, todas las hebras en el vector están en un tubo cero. La función de filtrado determina, en paralelo, cuáles hebras codifican un clique no existente y asigna tales hebras a un tubo M. Así, al final se obtiene el mismo vector con las mismas hebras, pero asignadas a un tubo diferente (las que no corresponden a cliques).

Los detalles técnicos a considerar en cada uno de los algoritmos son:

- La cantidad de llamadas a *kernel*.
- La cantidad de operaciones que realiza cada *kernel*.
- La memoria donde se almacena la información de adyacencias del grafo. Esta memoria puede ser de dos tipos: memoria del *host* (CPU) y memoria del dispositivo (GPU).

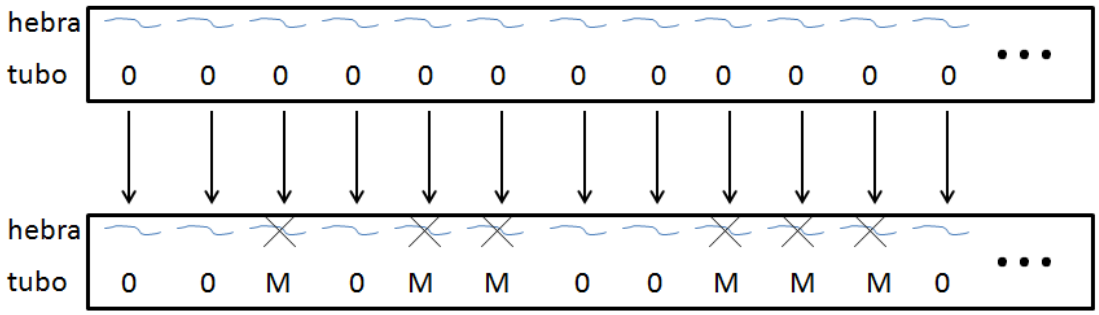


Figura 37: Operación de filtrado paralelo. Al inicio todas las hebras en el vector pertenecen al tubo cero. Se detectan cuáles hebras no codifican un clique existente en el grafo y se cambian a un tubo M. Las hebras que codifican un clique siguen perteneciendo al tubo cero.

5.6.1. Algoritmo A

Este algoritmo es la implementación del algoritmo k-Cliques (Algoritmo 17) utilizando la implementación de las operaciones del modelo de etiquetas de forma paralela (descrita en el Capítulo 4). Cada operación del modelo de etiquetas es un *kernel*, por lo que la cantidad de llamadas a *kernel* corresponde a la cantidad de iteraciones, que es $O(n^2)$. Cada *kernel* realiza $O(1)$ operaciones. La consulta de las adyacencias se realiza en el CPU. Cabe mencionar que en esta implementación, no se consideró el ordenamiento de las hebras por tubo, como se describió en el Capítulo 4.

5.6.2. Algoritmo B

El funcionamiento de este algoritmo es equivalente a una implementación paralela del algoritmo k-Cliques mejorado pero con ligeras diferencias técnicas. El seudocódigo correspondiente se presenta en el Algoritmo 19.

Algoritmo 19: Filtrado paralelo B

Entrada : Grafo $G = (V, E)$, tubo T con biblioteca $B'_{[n,k]}$.

Salida : Tubo T con cliques de tamaño k.

```
1  for i ← 1 to n − 1 do
2      for j ← i + 1 to n do
3          if ¬adyacentes(v_i, v_j) then
4              for each S_r ∈ T in parallel do
5                  if bit i = bit j = 1 en S_r then
6                      Marcar S_r como "no clique";
7                  end
8              end
9          end
10     end
11 end
```

De la misma forma que en el algoritmo k-Cliques mejorado, los ciclos de las líneas 1 y 2 producen todas las combinaciones de dos vértices v_i, v_j. También, la consulta de adyacencia en la línea 3 asegura que únicamente se consideren los casos cuando v_i y v_j no son adyacentes. Las líneas $4-8$ corresponden a la implementación de un *kernel* que se encarga de verificar en paralelo para cada hebra, si tiene las posiciones de bit i y j encendidas, y entonces, marcar esa hebra como "no clique".

En este enfoque, el número de llamadas a *kernel* también es $O(\overline{E})$, tal como sería una implementación paralela del algoritmo k-Cliques mejorado. La diferencia radica en que en este caso se realiza una sola llamada a *kernel* por cada no adyacencia del grafo, mientras que en el otro caso serían cuatro, una para cada operación del modelo de etiquetas. La cantidad de operaciones que realiza el *kernel* es $O(1)$ y la adyacencia debe ser consultada en la memoria del CPU, al igual que el Algoritmo A.

5.6.3. Algoritmo C

En esta función, se trasladan las iteraciones de los ciclos que generan las combinaciones de pares de vértices a un solo *kernel*. Cada hilo se encarga de realizar todas las operaciones

necesarias para verificar si una hebra codifica un clique o no. Esto requiere que la información de adyacencias del grafo se encuentre almacenada en el GPU. Dado que la información de adyacencias no se modifica en ningún caso, se puede utilizar la memoria de constantes del GPU. Con esto, se agiliza la consulta de adyacencias mejorando el tiempo de ejecución del *kernel*. Cabe mencionar que, dado que la capacidad de la memoria de constantes es muy limitada, sólo se puede considerar para los casos en los que el tamaño del vector de adyacencias, el cual está en función del tamaño del problema $|V|$, no sobrepase esta capacidad. El seudocódigo que realiza lo anteriormente descrito se muestra en el Algoritmo 20.

Algoritmo 20: Filtrado paralelo C

Entrada : Grafo $G = (V, E)$, tubo T con biblioteca $B'_{[n,k]}$.

Salida : Tubo T con cliques de tamaño k.

```
1  for each S_r ∈ T in parallel do
2      for i ← 1 to n − 1 do
3          if bit i = 1 in S_r then
4              for j ← i + 1 to n do
5                  if bit j = 1 in S_r then
6                      if ¬adyacentes(v_i, v_j) then
7                          Marcar S_r como "no clique";
8                      end
9                  end
10             end
11         end
12     end
13 end
```

Las comparaciones en las líneas 3 y 5, permiten que solo se consideren los casos donde las hebras tengan ambos bits i y j encendidos. Dado que cada hebra tiene exactamente k bits encendidos, la cantidad de operaciones realizadas por cada hilo se reduce de $\mathrm{O}(n^2)$ a $\mathrm{O}(k^2)$. Esta reducción es significativa cuando k es mucho menor que n.

5.7. Esquema general de la heurística

En esta sección, se decribe de manera general, el funcionamiento de los diferentes módulos de los que consta la implementación, para la búsqueda heurística de cliques de tamaño máximo. La Figura 38 muestra la operación de la heurística de manera esquemática.

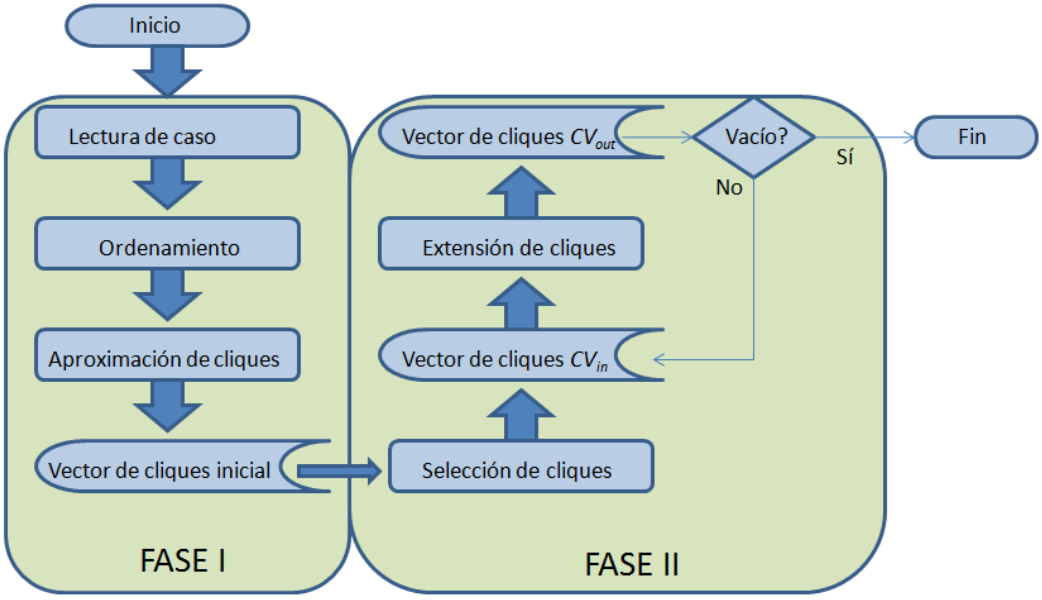


Figura 38: Diagrama esquemático del funcionamiento general de la heurística. Ésta se divide en dos fases: en la Fase I se produce un vector de cliques inicial con cliques de tamaño $k_0 \leq k_{max}$; en la segunda fase se extienden los cliques encontrados en la Fase I, en tamaños sucesivamente más grandes $(k_0 + 1, k_0 + 2, ...)$.

La heurística se divide en dos fases, cada una compuesta por diferentes etapas. En la primera fase, se incluyen las etapas de llenado de estructuras, ordenamiento del vector de grados y aproximación de los tamaños máximos de clique. La primera fase produce como resultado un vector de cliques inicial, que se toma como entrada en la segunda fase. La segunda fase se encarga de la búsqueda de cliques sucesivamente más grandes, de forma iterativa. En cada iteración $r = 0, 1, 2...$, se procesa un conjunto de cliques de tamaño $k_0 + r$, almacenado en un vector de cliques CV_{in}, y se buscan cliques extendidos de tamaño $k_0 + r + 1$. Estos cliques son almacenados en otro vector CV_{out}. La Fase II finaliza cuando el vector de cliques CV_{out} termina vacío al final de alguna iteración r, lo cual significa que no se encontraron nuevos cliques de tamaño extendido $k_0 + r + 1$.

5.7.1. Fase I

La Fase I de la heurística comprende las etapas descritas a continuación:

- **Lectura de caso de prueba.** En esta parte, se realiza el llenado del vector de adyacencias y del vector de grados, correspondientes al caso de prueba del grafo a procesar.
- **Ordenamiento.** Se ordenan los vértices de mayor a menor grado en el vector de grados. En otras palabras, se obtiene una permutación $\langle v_{i_1}, v_{i_2}, ..., v_{i_n} \rangle$, de forma que $d(v_{i_1}) \geq d(v_{i_2}) \geq \cdots \geq d(v_{i_n})$. Para este ordenamiento se utiliza el algoritmo *Insertion Sort*.
- **Aproximación de cliques.** En esta etapa se implementa una estrategia para la aproximación del tamaño máximo de clique k_{max}. Esta estrategia se describe en la Sección 5.8.
- **Vector de cliques inicial.** Como resultado de la etapa anterior, se obtiene un vector de cliques que puede contener hasta n cliques de algún tamaño $k_0 \leq k_{max}$. Los cliques encontrados forman la entrada de la Fase II.

5.7.2. Fase II

Está conformada por las siguientes etapas:

- **Selección de cliques.** Para el procesamiento de la Fase II se establece una cota en la cantidad de cliques que se pueden almacenar en los vectores de cliques. El parámetro de la heurística M define esta cota. Cuando la cantidad de cliques encontrados en la Fase I excede el valor M, se hace una selección aleatoria de los cliques almacenados en el vector de cliques inicial.
- **Vector de cliques CV_{in}.** Almacena los cliques a procesar al inicio de una iteración. Puede contener hasta M cliques de tamaño $k_0 + r$ en la r-ésima iteración.
- **Extensión de cliques.** En esta parte se implementa una estrategia que extiende progresivamente los cliques encontrados. Esta estrategia se revisará a detalle en la Sección 5.9.

En esta sección, se explicó de manera general las etapas que componen las fases I y II de la heurística propuesta. En las Secciones 5.8 y 5.9 se tratará a detalle el funcionamiento de

las etapas de aproximación de cliques y de extensión de cliques, respectivamente. Las etapas restantes que componen las fases I y II no requieren explicación adicional.

5.8. Aproximación de cliques

En esta sección se describe la estrategia empleada para la aproximación del tamaño de clique máximo k_{max} en un grafo. Además, con esta estrategia se encuentran cliques de un tamaño inicial k_0 que sirven como base para la generación de un vector de hebras que contiene un subconjunto de la biblioteca $B_{[n,k_0+1]}$. Así, estos cliques sirven como "materia prima" para el funcionamiento de la Fase II de la heurística.

5.8.1. Cómputo de clique

En principio, la estrategia se basa en la identificación de un clique C_D dada una permutación $D = \langle v_{i_1}, v_{i_2}, \ldots, v_{i_n} \rangle$ de los vértices. El Algoritmo 21 muestra esta función. En primer término, en la línea 3 se agrega el primer vértice de la permutación al clique C_D. El ciclo de las líneas $4 - 10$ considera el resto de los vértices uno por uno, verificando si cada vértice es adyacente a todos los vértices que han sido agregados al clique (línea 5), para agregarlo al clique C_D (línea 6). Cuando un vértice no es adyacente a alguno de los vértices en el clique C_D, por definición, no puede ser parte de tal clique, y es agregado a otro conjunto N_D (línea 8).

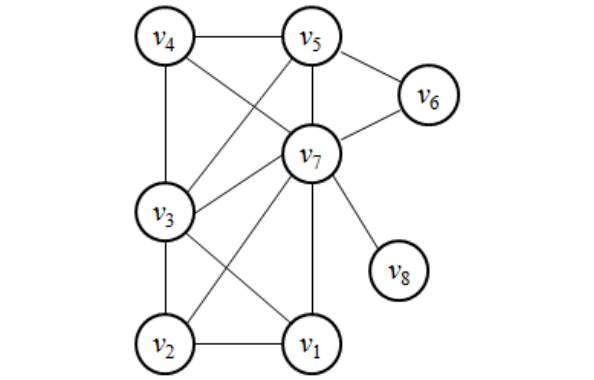
Algoritmo 21: Cómputo de clique

Entrada : Grafo $G = (V, E)$, permutación de vértices $D = \langle v_{i_1}, v_{i_2}, \ldots, v_{i_n} \rangle$.

Salida : Un clique C_D en G.

```
1  C_D ← ∅;
2  N_D ← ∅;
3  C_D ← C_D ∪ {v_{i_1}};
4  for k ← 2 to n do
5      if v_{i_k} es adyacente con cada vértice en C_D then
6          C_D ← C_D ∪ {v_{i_k}};
7      else
8          N_D ← N_D ∪ {v_{i_k}};
9      end
10 end
```

La cantidad de pasos que realiza el Algoritmo 21 es $O(n \times |C_D|)$. Cabe señalar que cuando el valor de $|C_D|$ es cercano a n, el algoritmo realiza $O(n^2)$ pasos. Sin embargo, aunque estos casos producen la mayor cantidad de pasos, los tamaños de clique obtenidos son cercanos al máximo posible en un grafo de n vértices, por lo que el tamaño de clique obtenido debe ser muy cercano al máximo existente en el grafo. La Figura 39 muestra un ejemplo del funcionamiento del Algoritmo 21 para un grafo y una permutación de vértices dados.



C_D	N_D
$\{v_7\}$	$\emptyset$
$\{v_7, v_3\}$	$\emptyset$
$\{v_7, v_3, v_5\}$	$\emptyset$
$\{v_7, v_3, v_5, v_4\}$	$\emptyset$
$\{v_7, v_3, v_5, v_4\}$	$\{v_2\}$
$\{v_7, v_3, v_5, v_4\}$	$\{v_2, v_1\}$
$\{v_7, v_3, v_5, v_4\}$	$\{v_2, v_1, v_6\}$
$\{v_7, v_3, v_5, v_4\}$	$\{v_2, v_1, v_6, v_8\}$

(B)

Figura 39: Identificación de un clique C_D con el Algoritmo 21. (A) Un grafo y una permutación de vértices dados. (B) Conformación de los conjuntos C_D y N_D con el Algoritmo 21. Al final, se obtiene un clique de tamaño 4 que en este caso corresponde al tamaño máximo de clique en el grafo.

Observe que el resultado del Algoritmo 21 depende del orden de los vértices en la permutación. En total, existen $n!$ permutaciones diferentes para un conjunto de n vértices. Es evidente que no es posible considerar el total de diferentes permutaciones por lo que, el enfoque que se sigue, es tomar un subconjunto $O(n)$ de tales permutaciones.

Se consideran tres formas de generar las permutaciones de los vértices, para cada forma se producen n permutaciones:

- Determinista.
- Probabilística (basada en probabilidades, similar a una ruleta).
- Aleatoria uniforme.

5.8.2. Permutación determinista

De forma determinista, se generan n permutaciones conocidas de antemano. El Algoritmo 22 muestra cómo se toman estas permutaciones.

Algoritmo 22: Permutación determinista

Entrada : Permutación de vértices $D = \langle v_{i_1}, v_{i_2}, \ldots, v_{i_n} \rangle$, posición r de vértice.

Salida : Permutación D' de vértices con v_{i_r} al inicio y los demás vértices en el mismo orden relativo.

```
1 v_t ← v_{i_r};
2 for k ← r downto 2 do
3 |   v_{i_k} ← v_{i_{k-1}};
4 end
5 v_{i_1} ← v_t;
```

Dada una permutación de entrada $D = \langle v_{i_1}, v_{i_2}, \ldots, v_{i_n} \rangle$ de vértices ordenados de forma que $d(v_{i_1}) \geq d(v_{i_2}) \geq \cdots \geq d(v_{i_n})$, y una posición r de vértice, el algoritmo coloca v_{i_r} al inicio (línea 5), y los demás vértices en el mismo orden relativo de la permutación de entrada D. La Figura 40 muestra las permutaciones D y D', para un valor $r = 5$.

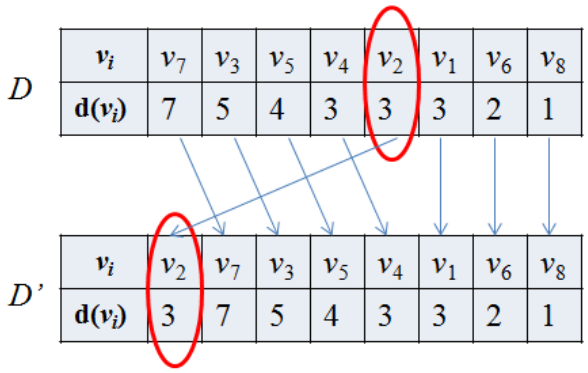


Figura 40: Ejemplo de funcionamiento del Algoritmo 22 para una permutación de entrada D. La permutación de salida D' tiene al inicio el vértice en la posición $r = 5$ y se mantiene el mismo orden relativo de los demás vértices.

Cada permutación determinista se obtiene en $O(n)$ pasos. En total se consideran las n permutaciones que se obtienen para los valores de $r = 1, 2, \ldots, n$.

5.8.3. Permutación probabilística

La permutación probabilística se basa en el grado de los vértices. Un vértice con mayor grado tiene una mayor probabilidad de ser seleccionado antes que los vértices con grado menor. El algoritmo implementado para realizar lo anterior funciona como una "ruleta" o "rueda de la fortuna". Cada vértice no seleccionado tiene una "sección" o "área" en la ruleta, que es proporcional a su grado. En cada "giro", los vértices con mayor área tienen mayores probabilidades de ser seleccionados.

Lo anterior se define formalmente de la siguiente forma: sea S el conjunto de los vértices seleccionados y $\overline{S} = V \setminus S$ el conjunto de los vértices no seleccionados, en un momento dado. Cada vértice $v_i \in \overline{S}$ se representa como una sección en la ruleta. El área $A(v_i)$ de la sección correspondiente a cada vértice $v_i \in \overline{S}$ es proporcional a la razón del grado del vértice sobre la suma de los grados de los vértices no seleccionados:

$$A(v_i) \propto \frac{d(v_i)}{\sum_{v_j \in \overline{S}} d(v_j)}. \tag{14}$$

De esta forma, para seleccionar un vértice del conjunto $\overline{S}$, se "gira la ruleta", se agrega el vértice v_s que corresponde a la sección donde "cayó" la ruleta al conjunto S, y se recalculan las áreas de los vértices restantes en $\overline{S}$ para generar una nueva ruleta. Este procedimiento se repite hasta que se hayan seleccionado los p vértices deseados (en este caso, deseamos una permutación de los n vértices, por lo que $p = n$). La Figura 41 proporciona un ejemplo de la selección probabilística usando este mecanismo de ruleta.

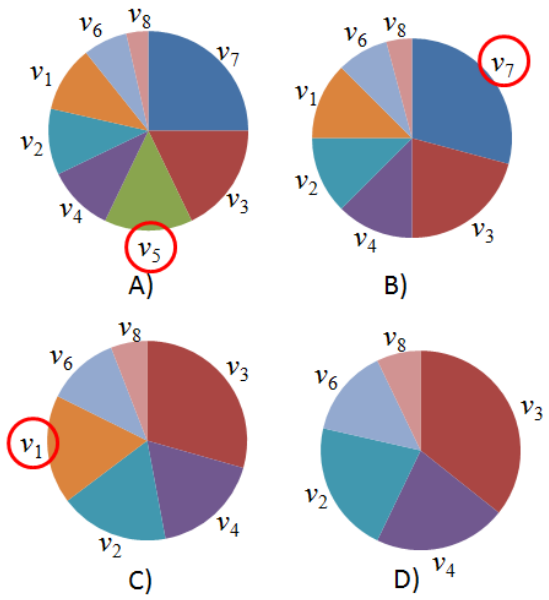


Figura 41: Ejemplo de funcionamiento del mecanismo de ruleta. Cada inciso muestra la configuración de la ruleta incluyendo los vértices indicados. Los vértices encerrados en círculos son los seleccionados en cada configuración. Nótese el cambio de áreas en cada configuración debido a que las proporciones cambian al no considerar vértices previamente seleccionados.

Algoritmo 23: Permutación probabilística

Entrada : Permutación de vértices $D = \langle v_{i_1}, v_{i_2}, \ldots, v_{i_n} \rangle$, cantidad p de vértices a seleccionar.

Salida : Permutación D' de p vértices de D.

```
1  S ← ∅;
2  S̄ ← D;
3  for k ← 1 to p do
4  |  Calcular A(v_i) para todo v_i ∈ S̄;
5  |  Girar ruleta;
6  |  Verificar sección s donde cayó;          ▷ v_s es el vértice seleccionado
7  |  S ← S ∪ {v_s};
8  |  S̄ ← S̄ \ {v_s};
9  end
10 D' ← S;
```

El Algoritmo 23 muestra el seudocódigo de la selección probabilística de una permutación

de vértices. La ruleta se implementa como un arreglo A de $|\overline{S}|$ enteros. La i-ésima posición en el arreglo A guarda la suma de los grados hasta el i-ésimo vértice que es elemento de $\overline{S}$, esto es,

$$A_i = \sum_{k=1}^{i} d(v_{i_k}). \tag{15}$$

Con esto, el área en la ruleta para el vértice en la i-ésima posición de $\overline{S}$ corresponde al intervalo $[A_{i-1}+1, A_i]$ con $1 \leq i \leq |\overline{S}|$ y $A_0 = 0$. Así, al girar la ruleta se genera un número entero aleatorio en el intervalo $[1, A_{|\overline{S}|}]$. Para verificar dónde cayó la ruleta se consulta el intervalo en A, dentro del cual está el número aleatorio generado. Si $[A_{s-1}+1, A_s]$ es el intervalo que contiene el número aleatorio generado, entonces el vértice v_s en la posición s dentro de $\overline{S}$ es el seleccionado. El vértice v_s se agrega al conjunto S de vértices seleccionados, y se elimina del conjunto $\overline{S}$ de vértices no seleccionados.

Por ejemplo, sea $\overline{S} = \{v_7, v_2, v_5, v_8, v_1, v_3, v_4\}$ el conjunto de vértices no seleccionados con grados $\langle 3, 5, 2, 7, 5, 6, 1\rangle$, respectivamente. Entonces, el arreglo A tendrá los valores $\langle 3, 8, 10, 17, 22, 28, 29\rangle$. Estos valores almacenan los intervalos $[1, 3]$, $[4, 8]$, $[9, 10]$, $[11, 17]$, $[18, 22]$, $[23, 28]$ y $[29, 29]$ que corresponden, respectivamente, a la posición de cada vértice en $\overline{S}$. Al girar la ruleta generamos un número entero aleatorio en el intervalo $[1, 29]$. Suponga que el número aleatorio generado es 25. Este número está contenido en el intervalo $[23, 28]$ que corresponde a la posición 6 en $\overline{S}$. Por lo tanto, el vértice seleccionado es v_3.

Dado que el conjunto $\overline{S}$ contiene inicialmente todos los n vértices de V, el cómputo del arreglo A y la búsqueda del intervalo donde está contenido el número aleatorio generado son ambos $\mathrm{O}(n)$. Entonces, la cantidad total de pasos del Algoritmo 23 es $\mathrm{O}(p \times n)$. Cuando se quiere calcular una permutación completa de los n vértices, como es el caso en esta sección, la cantidad de pasos es $\mathrm{O}(n^2)$.

5.8.4. Permutación aleatoria uniforme

Una permutación aleatoria consiste en una selección de vértices en la cual, todos los vértices tienen la misma probabilidad de ser seleccionado. Esto es, si $\overline{S}$ es el conjunto de vértices no seleccionados en un momento dado, todo vértice $v_i \in \overline{S}$ tendrá una probabilidad $\frac{1}{|\overline{S}|}$ de ser seleccionado. El Algoritmo 24 selecciona p vértices de un conjunto de n de manera aleatoria.

Algoritmo 24: Permutación aleatoria

Entrada : Permutación de vértices $D = \langle v_{i_1}, v_{i_2}, \ldots, v_{i_n} \rangle$, cantidad p de vértices a seleccionar.

Salida : Permutación D' de p vértices de D.

```
1 S ← ∅;
2 S̄ ← D;
3 for k ← 1 to p do
4 |   s ← generar número entero aleatorio en el intervalo [1, |S̄|];
5 |   v_s ← vértice en la posición s;
6 |   S ← S ∪ {v_s};
7 |   S̄ ← S̄ \ {v_s};
8 end
9 D' ← S;
```

Para cada posición $1 \leq k \leq p$ se genera un número entero aleatorio s en el intervalo $[1, |\overline{S}|]$. Este número determina la posición del vértice a ser seleccionado. Por ejemplo, suponga que en algún momento $\overline{S} = \{v_2, v_4, v_1, v_8, v_9, v_5, v_7\}$ y $s = 5$. Entonces, v_9 es el vértice seleccionado, dado que le corresponde la posición 5 en el conjunto $\overline{S}$. La cantidad de pasos que realiza el Algoritmo 24 es $\mathrm{O}(p)$.

5.8.5. Funcionamiento general

Una vez que se ha descrito la forma de generar un clique a partir de distintos tipos de permutaciones, sólo resta definir de manera general, el funcionamiento de la etapa de aproximación de cliques. El Algoritmo 25 describe este funcionamiento.

Algoritmo 25: Etapa de aproximación de cliques

Entrada : Grafo $G = (V, E)$, permutación inicial $D = \{v_{i_1}, v_{i_2}, \ldots, v_{i_n} | d(v_{i_1}) \geq d(v_{i_1}) \geq \cdots \geq d(v_{i_1})\}$, cantidad p de vértices a seleccionar.

Salida : Conjunto CV_{in} con $|CV_{in}| \leq n$ cliques de tamaño $k_0 \leq k_{max}$.

```
1  CV_in ← ∅;
2  k_0 ← 1;
3  for each tipo de permutación perm = {det, prob, aleat} do
4      for k ← 1 to n do
5          Generar permutación D de tipo perm ;        ▷ Algoritmos 22, 23 y 24.
6          Calcular clique C_D de tamaño k_D ;                ▷ Algoritmo 21.
7          if k_D > k_0 then
8              CV_in ← ∅;
9              CV_in ← CV_in ∪ {C_D};
10             k_0 ← k_D;
11         else
12             if k_D = k_0 y C_D ∉ CV_in then
13                 CV_in ← CV_in ∪ {C_D};
14             end
15         end
16     end
17 end
```

En primer término, se incializan el vector de cliques CV_{in} para almacenar hasta n cliques y el tamaño de clique k_0 (líneas $1-2$). El ciclo de las líneas $3-17$ considera los tres tipos de permutación revisados (determinista, probabilística y aleatoria) de manera que para cada tipo, el ciclo interno de las líneas $4-16$ genera n permutaciones, usando los Algoritmos 22, 23 y 24 (línea 5). Por cada permutación D, se calcula el clique C_D de tamaño k_D correspondiente, usando el Algoritmo 21 (línea 6). Si el tamaño del clique encontrado es mayor que el mayor actual, se agrega al vector de cliques CV_{in}, borrando todo clique contenido anteriormente y se actualiza el tamaño de clique mayor actual (líneas $7-10$). Si el tamaño del clique encontrado

es igual al mayor actual, se verifica que el clique no se encuentre en el vector CV_{in} para agregarlo (líneas $12-13$). De esta forma, el vector de cliques de salida CV_{in} termina con los cliques de mayor tamaño, calculados a partir de las $3n$ permutaciones totales.

Se puede dar el caso de que los cliques encontrados en cada iteración se repitan, ya sea exactamente o con diferente orden de los vértices. La verificación de la línea 12, que se encarga de eliminar estos casos, requiere que el clique actual C_D sea comparado con todos los cliques almacenados en el vector de cliques CV_{in}. Existen dos posibilidades para implementar esta comparación: mantener los vértices ordenados (por índice) dentro de los cliques (lo cual requiere un algoritmo de ordenamiento en cada iteración, con $\mathrm{O}(k_D \log_2 k_D)$ pasos) o sin mantener orden. Si no se considera un orden de los vértices en los cliques, la verificación de igualdad de cada par de cliques requiere que se compare cada par de vértices en los cliques, resultado en $\mathrm{O}({k_D}^2)$ comparaciones de vértices. Dado que la cantidad de cliques en el vector CV_{in} es $\mathrm{O}(n)$ y el tamaño del clique encontrado es también $\mathrm{O}(n)$, la cantidad de pasos total en la verificación de la línea 12 es $\mathrm{O}(n^3)$. Sin embargo, esta complejidad se puede reducir utilizando el Algoritmo 26 para comparar dos cliques.

Algoritmo 26: Comparación de cliques

Entrada : Cliques C_1 y C_2 a comparar.

Salida : TRUE si $C_1 = C_2$, FALSE en caso contrario.

```
1  Inicializar dos hebras S_1 y S_2 en ceros;
2  for each v_i ∈ C_1 do
3  |  setStrandIndex(S_1, i) ;          ▷ Encender posición i en hebra S_1
4  end
5  for each v_j ∈ C_2 do
6  |  setStrandIndex(S_2, j) ;          ▷ Encender posición j en hebra S_2
7  end
8  if S_1 = S_2 then
9  |  Regresar TRUE;
10 else
11 |  Regresar FALSE;
12 end
```

El Algoritmo 26 toma dos cliques C_1 y C_2, y genera dos hebras correspondientes S_1 y S_2

(ciclos de las líneas 2−4 y 5−7, respectivamente). Si las hebras generadas son iguales, entonces los cliques de entrada son iguales y el algoritmo regresa TRUE, en caso contrario regresa FALSE (líneas $8-12$). La operación de convertir un clique en una hebra correspondiente, donde la posición de bit en la hebra codifica el índice de cada vértice, equivale a ordenar los vértices en el clique. De esta forma, la verificación de igualdad se puede hacer comparando el estado de cada bit en la hebra (Figura 42).

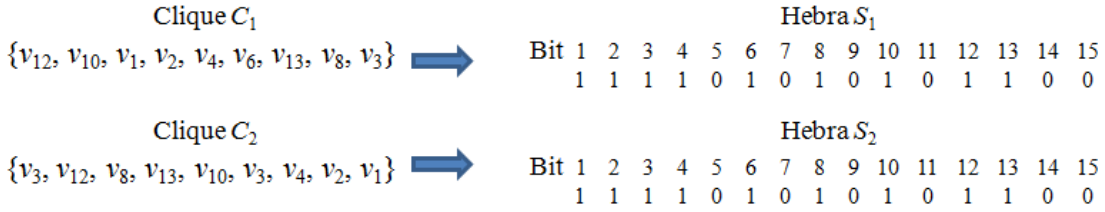


Figura 42: Conversión de cliques a hebras. Dos cliques iguales con vértices en diferente orden se traducen en la misma hebra. Esta conversión equivale a ordenar los vértices por índice en los cliques para comparar, posición por posición, si son iguales.

Dado que cada clique puede tener a lo más n vértices, los ciclos de las líneas $2-4$ y $5-7$ son ambos $O(n)$. La verificación de igualdad de las dos hebras en la línea 8 es también $O(n)$. Por lo tanto, la cantidad total de pasos del Algoritmo 26 es $O(n)$. Utilizando esta estrategia, la cantidad de pasos en la verificación de la línea 12 del Algoritmo 25 se reduce a $O(n^2)$. De esta forma, la complejidad total del Algoritmo 25 es $O(n^3)$.

5.9. Extensión de cliques

En esta sección se describe la estrategia empleada para la búsqueda de cliques de tamaños progresivamente más grandes $[k_0+1, k_0+2, \ldots]$, dado un conjunto inicial de cliques de tamaño k_0. Primeramente, esta estrategia se describe de forma general, donde se considera la iteración sobre distintos conjuntos de cliques y posteriormente, de forma particular, se considera la formación de un tubo de prueba que se genera a partir de un clique del conjunto.

5.9.1. Descripción general

El Algoritmo 27 describe la operación de extensión de cliques de la heurística. Toma como entrada un vector de cliques CV_{in} obtenido de la etapa de aproximación de cliques (Sección 5.8). Inicialmente, este vector contiene cliques de tamaño k_0. La Figura 43 describe

de forma general el funcionamiento de la etapa de extensión de cliques, para una iteración r. Con el propósito de simplificar la notación definimos $k_r = k_0 + r$, que utilizaremos en lo que resta del presente trabajo.

Algoritmo 27: Extensión de cliques

Entrada : Vector de cliques CV_{in} con cliques de tamaño k_0.

Salida : Vector de cliques CV_{out} con cliques de tamaño $k_r \geq k_0$.

```
1  if |CV_in| > M then
2  |  Seleccionar M cliques de CV_in de manera aleatoria;
3  end
4  clq ← TRUE;
5  r ← 0;
6  while clq = TRUE y r ≤ n − k_0 do
7  |  while |CV_in| > 0 y |CV_out| < M do
8  |  |  C ← siguiente clique en CV_in;
9  |  |  Generar tubo de prueba T;
10 |  |  Filtrar tubo de prueba T;
11 |  |  for each hebra S_i ∈ T do
12 |  |  |  if S_i es un clique then
13 |  |  |  |  CV_out ← CV_out ∪ {S_i};
14 |  |  |  end
15 |  |  end
16 |  end
17 |  if CV_out = ∅ then
18 |  |  clq ← FALSE;
19 |  else
20 |  |  CV_in ↔ CV_out;
21 |  |  r ← r + 1;
22 |  end
23 end
```

La cantidad de cliques en los vectores está limitada por un valor predefinido en un parámetro de la heurística M. Cuando la cantidad de cliques en CV_{in} excede este valor, se hace

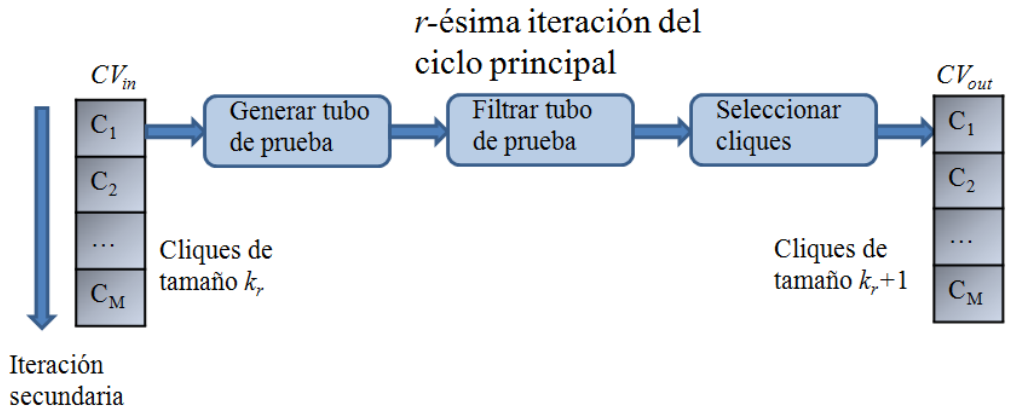


Figura 43: Descripción general de la etapa de extensión de cliques. Consiste en dos ciclos anidados. En la r-ésima iteración del ciclo principal, se itera sobre los cliques de tamaño k_r contenidos en CV_{in}. Para cada clique se genera un tubo de prueba que contiene posibles cliques de tamaño $k_r + 1$. El tubo de prueba se filtra para marcar los cliques existentes en el grafo. Si la cantidad de cliques encontrados es mayor que el parámetro M, se seleccionan aleatoriamente M cliques y se escriben en el vector CV_{out}. Si $|CV_{out}| = \text{M}$ (se llena), o se terminaron de procesar todos los cliques de CV_{in} y $CV_{out} \neq \emptyset$, los nuevos cliques de tamaño $k_r + 1$ son procesados en la iteración $r + 1$. Cuando $CV_{out} = \emptyset$ al final de una iteración r, la heurística termina.

una selección aleatoria de M cliques en CV_{in} (líneas $1-3$). En cada iteración del ciclo externo de las líneas $6 - 23$, se procesan los cliques almacenados en el vector CV_{in}. En la r-ésima iteración, con $0 \leq r \leq n - k_0$, los cliques en CV_{in} son de tamaño k_r, y se buscan cliques de tamaño k_r+1. El ciclo externo de las líneas $6-23$ termina cuando en alguna iteración r, no se encuentra ningún clique de tamaño $k_r + 1$, o equivalentemente, el vector CV_{out} termina vacío (líneas $17-18$). En caso contrario, si fueron encontrados nuevos cliques, estos son procesados en una iteración posterior, intercambiando los vectores CV_{in} y CV_{out} (líneas $19 - 21$).

El ciclo interno de las líneas $7 - 16$ se ejecuta mientras haya cliques por procesar en CV_{in} y el vector CV_{out} tenga espacio para almacenar nuevos cliques. En cada iteración del ciclo interno de las líneas $7 - 16$ se procesa un clique C de tamaño k_r de CV_{in}. Se genera un tubo de prueba T con posibles cliques de tamaño $k_r + 1$, en base al clique actual C. El tubo de prueba T se filtra utilizando alguno de las funciones de filtrado descritas en la Sección 5.6. El ciclo de las líneas $11 - 14$ recorre las hebras del tubo de prueba T para detectar cuáles fueron marcadas como cliques y copiarlas al vector CV_{out}.

5.9.2. Estrategia de generación de tubo de prueba

La operación que se realiza en la línea 9 del Algoritmo 27 es fundamental en la heurística. Consiste en la generación de un tubo de prueba T que contiene un subconjunto $B' \subseteq B_{[n,k_r+1]}$, para un valor k_0 y un número r de iteración dados. Este subconjunto se genera heurísticamente, tomando subconjuntos de vértices que pertenecen a un clique C de tamaño k_r conocido, y uniéndolos con subconjuntos de vértices que no pertenecen al clique, de manera que se obtengan subconjuntos de $k_r + 1$ vértices (formando posibles cliques de tamaño $k_r + 1$).

El clique C que se procesa en cada iteración, corresponde a una posición dentro del vector CV_{in} y, por lo tanto, es una permutación de los n vértices separados en dos grupos: los primeros k_r vértices pertenecen al clique, y los restantes $n - k_r$ vértices no pertenecen al clique. La Figura 44 muestra gráficamente un ejemplo de lo anterior.

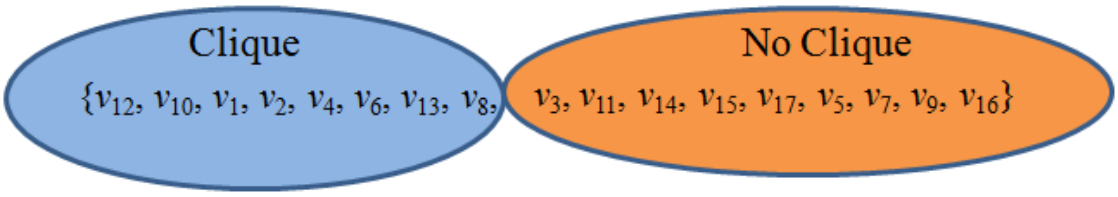


Figura 44: Permutación de vértices separados en dos grupos: los primeros k_r vértices pertenecen al clique conocido y los últimos $n - k_r$ vértices no pertenecen.

La estrategia de generación heurística del tubo T se basa en tomar, iterativamente, subconjuntos de ambos grupos de vértices. Lo que se pretende es tomar en cada iteración, una cantidad progresivamente menor de vértices que forman parte del clique y, correspondientemente, una cantidad progresivamente mayor de vértices que no pertenecen al clique.

De manera formal, la estrategia se define como sigue: en la iteración i, con $0 \leq i < k_r$, se genera un conjunto P_i de los subconjuntos de cardinalidad $k_r - i$, tomados de los k_r vértices que pertenecen al clique C, y se genera un conjunto S_i de los subconjuntos de cardinalidad $i + 1$, tomados de los $n - k_r$ vértices que no pertenecen al clique C. Los conjuntos P_i y S_i se denominan, respectivamente, conjunto de *prefijos* y conjunto de *sufijos*. Nótese que en la

iteración i, la cardinalidad de P_i está dada por

$$|P_i| = \binom{k_r}{k_r - i}, \tag{16}$$

mientras que la cardinalidad de S_i está dada por

$$|S_i| = \binom{n - k_r}{i + 1}. \tag{17}$$

Una vez que se tienen los conjuntos P_i y S_i, se une cada elemento de P_i con cada elemento de S_i. Esto es, se forma el conjunto

$$T_i = \{X \cup Y \mid X \in P_i \ \wedge \ Y \in S_i\}. \tag{18}$$

Observe que esta operación es equivalente al producto cartesiano $P_i \times S_i$. La única diferencia es que en lugar de obtener el conjunto de todos los pares, se obtiene el conjunto de la unión de todos los pares. Así, el conjunto T_i tiene una cardinalidad $|T_i| = |P_i| \times |S_i|$ y sus elementos son subconjuntos de $k_r + 1$ vértices. La Figura 45 muestra los conjuntos P_i y S_i para los dos grupos de vértices dados, con $i = 1$. La Figura 46 muestra el conjunto T_i que se forma a partir de estos conjuntos P_i y S_i.

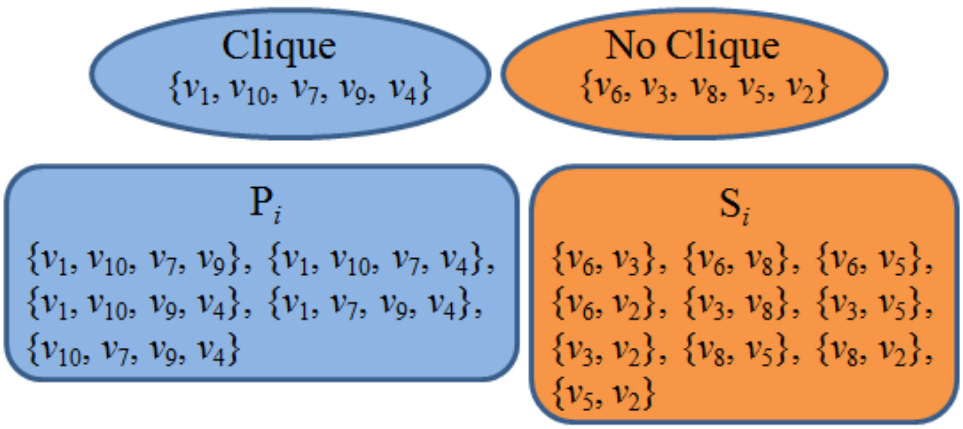


Figura 45: Ejemplos de conjuntos P_i y S_i que se forman a partir de los dos grupos dados, con $i = 1$, $k_r = 5$ y $n - k_r = 5$. P_i es el conjunto de todos los subconjuntos de $k_r - i = 4$ vértices, tomados de los $k_r = 5$ vértices que forman un clique. S_i es el conjunto de todos los subconjuntos de $i + 1 = 2$ vértices, tomados de los $n - k_r = 5$ vértices que no forman parte del clique. Observe que $|P_i| = \binom{5}{4} = 5$ y $|S_i| = \binom{5}{2} = 10$.

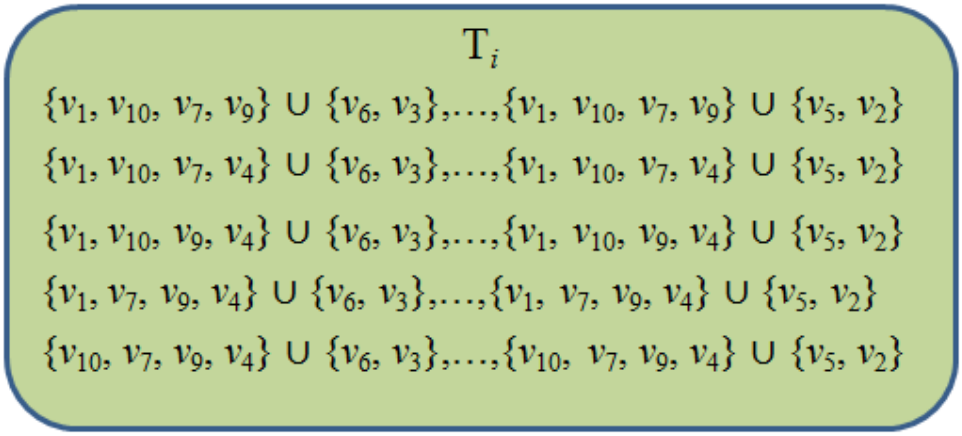


Figura 46: Conjunto T_i formado a partir de los conjuntos S_i y P_i de la Figura 45. T_i es el conjunto que se forma al unir cada subconjunto que es elemento de P_i con cada subconjunto que es elemento de S_i. Cada renglón en la figura corresponde a la unión de un elemento de P_i con cada elemento de S_i. En total, $|T_i| = |P_i| \times |S_i| = 50$.

El conjunto T_i se forma en cada iteración i, con $0 \leq i < k_r$. Entonces, se define el conjunto T, que contiene la unión de todos los conjuntos T_i como

$$T = \bigcup_{0 \leq i < k_r} T_i, \tag{19}$$

y la cantidad de subconjuntos en T está dada por

$$|T| = \sum_{0 \leq i < k_r} |T_i| = \sum_{0 \leq i < k_r} |P_i| \times |S_i|. \tag{20}$$

De esta forma, el conjunto T contiene subconjuntos de vértices de cardinalidad $k_r + 1$, donde cada subconjunto representa un posible clique de tamaño $k_r + 1$.

Como se recordará de la Sección 5.1, los conjuntos P_i y S_i se pueden representar con las bibliotecas $B_{[k_r, k_r - i]}$ y $B_{[n-k_r, i+1]}$, respectivamente. Sin embargo, en la presente implementación se maneja una ligera diferencia respecto a lo descrito en tal sección. Esta diferencia consiste en hacer que todas las cadenas de las bibliotecas sean de una misma longitud n, pero se utilizan solamente los primeros k_r bits en el conjunto de prefijos, y los últimos $n - k_r$ bits en el conjunto de sufijos. Los bits restantes en cada caso son puestos en cero. Es importante señalar que esta diferencia no cambia las cantidades $|P_i|$ y $|S_i|$. De esta forma, se simplifica la operación de unión de elementos de P_i con elementos de S_i, resultando en una simple

operación OR bit a bit, entre las cadenas que codifican cada elemento. La Figura 47 muestra las bibliotecas correspondientes a los conjuntos P_i y S_i de la Figura 45.

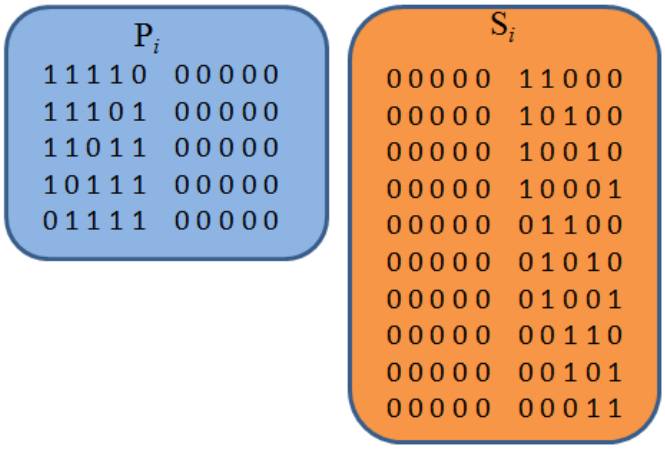


Figura 47: Bibliotecas correspondientes a los conjuntos P_i y S_i de la Figura 45. En P_i se utilizan solamente los primeros $k_r = 5$ bits y en S_i se utilizan los últimos $n - k_r = 5$ bits. Así, cada elemento de T_i corresponde a la operación OR bit a bit de una cadena de P_i con una cadena de S_i.

Las bibliotecas correspondientes a los conjuntos P_i y S_i llegan a tener una cantidad exponencial de cadenas, a medida que i crece. Más aún, la cantidad de cadenas en el conjunto T_i es el producto $|P_i| \times |S_i|$, por lo que es aún más grande. Por ejemplo, sean $n = 64$ y $k_r = 32$. Para $i = 16$, se tiene que $|P_i| = \binom{32}{16} = 601,080,390$, $|S_i| = \binom{32}{17} = 565,722,720$, y $|T_i| > 340 \times 10^{15}$.

El ejemplo anterior proporciona una idea de la cantidad de cadenas involucrada, incluso para tamaños modestos de problema ($n = 64$), y considerando solamente una iteración $i = 16$. Habría que considerar aún, la suma de las cadenas en todas las iteraciones para calcular la cantidad de cadenas en T.

Con esto, es evidente la inviabilidad computacional de la generación total de un conjunto T. Por este motivo, es indispensable la introducción de criterios que acoten esta generación. Un primer criterio básico, consiste en reducir el número de iteraciones, esto es, hacer $i < \lfloor \frac{k_r}{2} \rfloor$. Aunque puede parecer arbitraria, esta reducción tiene bastante sentido, dado que al generar nuevos subconjuntos de vértices, es deseable tomar la mayor cantidad de vértices del clique, y esta cantidad se reduce a medida que i crece.

5.9.3. Parámetros de la heurística

Para acotar las cantidades $|P_i|$, $|S_i|$, $|T_i|$ y $|T|$, se definen los siguientes parámetros de la heurística:

- MAX_{P_i}. Cota superior para la cantidad de prefijos en la iteración i ($|P_i| \leq MAX_{P_i}$).
- MAX_{S_i}. Cota superior para la cantidad de sufijos en la iteración i ($|S_i| \leq MAX_{S_i}$).
- MAX_{T_i}. Cota superior para cantidad de cadenas totales en la iteración i ($|T_i| \leq MAX_{T_i}$). Se calcula como $MAX_{T_i} = MAX_{P_i} \times MAX_{S_i}$.
- MAX_T. Cota superior para la cantidad de cadenas a generar en el tubo de prueba T ($|T| \leq MAX_T$).

Además, se considera la probabilidad de repetición de cadenas cuando no se generan de forma determinista (descrito en la Sección 5.9.4). Esta probabilidad es inversamente proporcional a la razón $\frac{\text{cadenas totales}}{\text{cadenas a generar}}$. Esto se debe a que la generación no determinista de cadenas es, en cierta forma, equivalente al experimento de extracción de elementos de un conjunto con repetición: mientras mayor sea la proporción $\frac{\text{cantidad de elementos del conjunto}}{\text{número de extracciones}}$, menor es la probabilidad de obtener elementos repetidos, y viceversa. Utilizando esta noción, se definen los siguientes parámetros para reducir la probabilidad de repetición de cadenas:

- MIN_P. Cota inferior para la razón $\frac{\text{prefijos totales}}{\text{prefijos a generar}}$ para utilizar generación probabilística en el conjunto P_i.
- MIN_S. Cota inferior para la razón $\frac{\text{sufijos totales}}{\text{sufijos a generar}}$ para utilizar generación aleatoria en el conjunto S_i.

5.9.4. Formas de generación de conjuntos P_i y S_i

Los conjuntos P_i y S_i se generan en tres de cuatro diferentes formas. Tanto P_i como S_i pueden obtenerse de forma determinista total y determinista parcial. P_i también se puede obtener de forma probabilística, mientras que S_i se puede generar de forma aleatoria. A continuación se describen estas cuatro formas:

- **Generación determinista total.** Es la generación del espacio combinatorio completo, es decir, todas las combinaciones de subconjuntos (equivalentemente, todas las cadenas) que se pueden formar en cada conjunto.

- **Generación determinista parcial.** Consiste en tomar un subconjunto que contiene alguna cantidad de las primeras cadenas del espacio combinatorio completo.
- **Generación probabilística**. Cada cadena se obtiene tomando en cuenta los grados de los vértices, utilizando el Algoritmo 23.
- **Generación aleatoria.** Cada cadena se genera utilizando el Algoritmo 24.

Las primeras dos formas son deterministas en el sentido que se tiene la seguridad de que no se generan cadenas repetidas en los conjuntos. En contraste, las generaciones probabilística y aleatoria dependen del azar, por lo que no se puede garantizar que no se produzcan cadenas repetidas. En la siguiente sección se describen las condiciones que definen el uso de cada forma, en cada conjunto.

5.9.5. Generación de los conjuntos P_i y S_i

Dado que el conjunto T se forma de la unión de los conjuntos T_i, y cada conjunto T_i se obtiene a partir de los conjuntos P_i y S_i, el aspecto fundamental de esta sección de la heurística consiste en la formación de estos conjuntos. Como ya se revisó, no es posible generar los conjuntos de forma total en todas la iteraciones, sino que se deben determinar cantidades que acoten el número de cadenas.

Las ecuaciones 16 y 17 definen las cantidades totales de sufijos y prefijos en cierta iteración i. Estas cantidades representan el espacio total (todas las cadenas *posibles*) de los conjuntos P_i y S_i, y son constantes para una iteración i dada. Para diferenciar la cantidad de cadenas *posibles*, de la cantidad de cadenas *actualmente almacenadas* en los conjuntos P_i y S_i, se redefinen $|P_i|$ y $|S_i|$ como las cantidades de cadenas *actualmente almacenadas* en los conjuntos P_i y S_i, respectivamente. Además, las cantidades totales se definen como:

- C_{P_i}. El total de prefijos posibles en la iteración i dado por $\binom{k_r}{k_r-i}$.
- C_{S_i}. El total de sufijos posibles en la iteración i dado por $\binom{n-k_r}{i+1}$.

Adicionalmente, se definen las cantidades:

- A_{P_i}. La cantidad "aceptable" de cadenas para el conjunto P_i.
- A_{S_i}. La cantidad "aceptable" de cadenas para el conjunto S_i.

Estas cantidades se dicen "aceptables" en el sentido que no exceden la cota MAX_{T_i} de cadenas máximas por iteración. Se calculan de forma que maximicen la cantidad de cadenas en T_i, tomando en cuenta las cotas anteriormente descritas. Esto se describirá más adelante en esta sección.

El conjunto de prefijos P_i contiene cadenas que codifican subconjuntos de vértices que forman parte de un clique, por lo que cualquier cadena en P_i es un clique. Por otra parte, en el caso del conjunto de sufijos S_i, no se conoce cuáles cadenas codifican un clique. Si se consideran cadenas que no forman un clique en los sufijos, al unirlas con las cadenas en el conjunto de prefijos, las cadenas resultantes no codificarán un clique. Esta situación produce un gran desperdicio de recursos computacionales (memoria, tiempo de procesamiento) en el manejo de cadenas que no pueden formar un clique. Por ejemplo, suponga que se tienen $2,000$ cadenas en P_i, y $2,000$ cadenas en S_i de las cuales $1,500$ cadenas no codifican un clique. Al realizar la unión de cada cadena de P_i con cada cadena de S_i se obtendrán $2,000 \times 1,500 = 3,000,000$ cadenas que no forman un clique de $4,000,000$ totales.

Por este motivo, antes de generar el conjunto P_i, se genera el conjunto S_i de manera que sólo contenga cadenas que forman un clique. Para esto se realiza lo siguiente:

- **Generar el máximo posible de cadenas en S_i.** El máximo posible de cadenas está acotado por el parámetro MAX_T. El Algoritmo 28 se encarga de generar el conjunto inicial de sufijos en alguna de las formas revisadas en la Sección 5.9.4. Si $C_{S_i} \leq MAX_T$ (línea 2), se generan todas las C_{S_i} cadenas en S_i de forma determinista (línea 9). En caso contrario (líneas $3-7$), se genera el máximo MAX_T de cadenas en S_i.

- **Filtrar las cadenas en S_i.** Utilizando el Algoritmo 19 o el Algoritmo 20 para marcar las cadenas que no forman un clique. La selección del algoritmo a utilizar depende de la densidad del grafo y del tamaño k de clique a filtrar.

- **Eliminar las cadenas que no forman un clique.** Recorrer las cadenas en S_i para eliminar aquellas marcadas como "no clique" por el algoritmo de filtrado.

Algoritmo 28: Generación inicial del conjunto de sufijos S_i.

Entrada : Parámetros MAX_T y MIN_S, iteración i.

Salida : Conjunto S_i de sufijos iniciales.

```
1  C_{S_i} ← (n−k_r choose i+1);
2  if C_{S_i} > MAX_T then
3  |  if C_{S_i} / MAX_T ≥ MIN_S then
4  |  |  Generar MAX_T cadenas de forma aleatoria;
5  |  else
6  |  |  Generar las primeras MAX_T cadenas de forma determinista;
7  |  end
8  else
9  |  Generar todas las C_{S_i} cadenas de forma determinista;
10 end
```

Con esto, se obtiene el conjunto S_i con alguna cantidad $|S_i|$ de cadenas que forman un clique. El siguiente paso, consiste en determinar la cantidad A_{P_i} de prefijos a generar, y la cantidad A_{S_i} de sufijos a considerar (de los $|S_i|$ que contiene actualmente). Estas cantidades se calculan en base en las cantidades $|S_i|$ de sufijos, C_{P_i} de prefijos posibles, y utilizando las cotas MAX_{P_i}, MAX_{S_i} y MAX_{T_i}. El Algoritmo 29 realiza este cálculo considerando cuatro casos. Consiste en una "jerarquía" de criterios (condiciones) que produce una asignación diferente a las cantidades A_{P_i} y A_{S_i}. El criterio principal es que el producto $|S_i| \times C_{P_i}$ no exceda la cota $MAX_{T_i} = MAX_{P_i} \times MAX_{S_i}$ de cadenas por iteración. Esto se verifica en la línea 1. Cuando no se excede esta cota, se pueden tomar las cantidades $|S_i|$ y C_{P_i} sin modificación alguna (Caso 4, líneas $15-16$). El siguiente criterio es que la cantidad $|S_i|$ no rebase la cota MAX_{S_i} (línea 2). Cuando se cumple, se toman todos los $|S_i|$ sufijos y una fracción de $\frac{1}{|S_i|}$ del máximo MAX_{T_i} de cadenas por iteración (Caso 3, líneas $11-12$). El último criterio es que la cantidad C_{P_i} no rebase la cota MAX_{P_i} (línea 3). Si esto sucede, se toman todos los C_{P_i} prefijos y una fracción de $\frac{1}{C_{P_i}}$ sufijos del máximo MAX_{T_i} de cadenas por iteración (Caso 2, líneas $7-8$). Si no se cumple este último criterio, significa que ambas cantidades $|S_i|$ y C_{P_i} excedieron sus respectivas cotas, por lo que se deben tomar los máximos MAX_{P_i} y MAX_{S_i} de sufijos y prefijos por iteración, respectivamente (Caso 1, líneas $4-5$). Nótese que en los

casos 1, 2 y 3, donde no se cumple el criterio principal, las cantidades asignadas a A_{P_i} y A_{S_i}, maximizan la cantidad $|T_i|$ de cadenas por iteración, haciendo $A_{P_i} \times A_{S_i} \approx MAX_{T_i}$.

Algoritmo 29: Cálculo de A_{P_i} y A_{S_i}

Entrada : Cantidades $|S_i|$, C_{P_i} y cotas MAX_{P_i}, MAX_{S_i} y MAX_{T_i}.

Salida : Cantidades A_{P_i} y A_{S_i}.

```
1  if |S_i| × C_{P_i} > MAX_{T_i} then
2      if |S_i| > MAX_{S_i} then
3          if C_{P_i} > MAX_{P_i} then
4              A_{P_i} ← MAX_{P_i} ;                          ▷ Caso 1
5              A_{S_i} ← MAX_{S_i};
6          else
7              A_{P_i} ← C_{P_i} ;                            ▷ Caso 2
8              A_{S_i} ← ⌊MAX_{T_i} / C_{P_i}⌋;
9          end
10     else
11         A_{P_i} ← ⌊MAX_{T_i} / |S_i|⌋ ;                    ▷ Caso 3
12         A_{S_i} ← |S_i|;
13     end
14 else
15     A_{P_i} ← C_{P_i} ;                                    ▷ Caso 4
16     A_{S_i} ← |S_i|;
17 end
```

Una vez que se han calculado las cantidades definitivas a tomar para los prefijos y los sufijos, el siguiente paso es tomar los A_{S_i} de los $|S_i|$ almacenados y generar los A_{P_i} prefijos. Si es posible, se toman todas las cadenas de S_i, esto es, si $A_{S_i} \geq |S_i|$, se toman todas las $|S_i|$ cadenas. En caso contrario, se seleccionan A_{S_i} cadenas de las $|S_i|$ totales, de forma aleatoria. La generación de los A_{P_i} prefijos se hace en una de las tres formas anteriormente descritas: determinista total, determinista parcial y probabilística. El Algoritmo 30 se encarga de la generación del conjunto P_i de prefijos. Se generan todos los C_{P_i} prefijos de forma determinista cuando esta cantidad no excede la cantidad A_{P_i} calculada con el Algoritmo 29.

En caso contrario, se generan A_{P_i} prefijos de forma probabilística cuando las probabilidades de producir cadenas repetidas son pocas (de acuerdo al parámetro MIN_P), o de forma determinista (parcialmente).

Algoritmo 30: Generación del conjunto de prefijos P_i.

Entrada : Cantidad A_{P_i}, parámetro MIN_P, iteración i.

Salida : Conjunto P_i de prefijos.

```
1  C_{P_i} ← (k_r choose k_r − i);
2  if C_{P_i} > A_{P_i} then
3      if C_{P_i} / A_{P_i} ≥ MIN_P then
4          Generar A_{P_i} cadenas de forma probabilística;
5      else
6          Generar las primeras A_{P_i} cadenas de forma determinista;
7      end
8  else
9      Generar todas las C_{P_i} cadenas de forma determinista;
10 end
```

5.9.6. Generación de T_i

Una vez que se tienen los conjuntos P_i y S_i, cada elemento del primero se une con cada elemento del segundo. Esta operación es equivalente a obtener el producto cartesiano $P_i \times S_i$, con la excepción de que en lugar de obtener el conjunto de todos los pares de elementos de P_i y S_i, se obtiene el conjunto de la unión de cada par. De esta forma, T_i se genera a partir de los conjuntos P_i y S_i como se describe en la Ecuación 18. Finalmente, se agregan todas las cadenas en T_i al conjunto final T, y se repite el proceso para la siguiente iteración $i+1$ mientras T no alcance el límite predeterminado de capacidad MAX_T.

5.9.7. Generación de tubo de prueba T

El Algoritmo 31 muestra el seudocódigo para generar el tubo de prueba T con un subconjunto $B' \subseteq B_{[n,k_r+1]}$ generado de manera heurística de la forma anteriormente descrita.

Algoritmo 31: Generación heurística del tubo de prueba T.

Entrada : Permutación de vértices $C = \{v_{i_1}, v_{i_2}, \ldots, v_{i_n}\}$ donde los primeros k_r vértices forman un clique.

Salida : Tubo T con biblioteca $B' \subseteq B_{[n,k_r+1]}$.

```
1  i ← 0;
2  while i < ⌊k_r/2⌋ y |T| < MAX_T do
3      P_i, S_i, T_i ← ∅;
4      Generar S_i;
5      Filtrar S_i;
6      for each X ∈ S_i | X no codifica un clique do
7          S_i ← S_i \ {X};
8      end
9      Calcular A_{P_i} y A_{S_i};
10     if |S_i| > A_{S_i} then
11         Seleccionar A_{S_i} cadenas de S_i de manera aleatoria;
12     end
13     Generar P_i;
14     for each X ∈ P_i do
15         for each Y ∈ S_i do
16             T_i ← T_i ∪ {X ∪ Y};
17         end
18     end
19     T ← T ∪ T_i;
20     i ← i + 1;
21 end
```

El ciclo de las líneas $2-21$ realiza las iteraciones $0 \leq i < \lfloor \frac{k_r}{2} \rfloor$ mientras el conjunto T no alcance su capacidad máxima predeterminada en el parámetro MAX_T. En cada iteración, se genera un conjunto T_i formado a partir de las uniones de los elementos del conjunto P_i de prefijos y S_i de sufijos. Para esto, se inicializan primero los conjuntos P_i, S_i y T_i (línea 3). Se genera el conjunto S_i con la mayor cantidad de cadenas posible utilizando el Algoritmo 28

(línea 4). Este conjunto de sufijos se filtra con algún algoritmo de la Sección 5.6 para marcar las cadenas que no codifican un clique (línea 5). El ciclo de las líneas $6-8$ recorre las cadenas en S_i para eliminar las cadenas marcadas por el algoritmo de filtrado. Se calculan las cantidades A_{P_i} y A_{S_i} usando el Algoritmo 29 (línea 9). Se seleccionan las A_{S_i} cadenas de forma aleatoria, si esta cantidad es menor que las $|S_i|$ que actualmente están en el conjunto S_i (líneas $10-12$). El conjunto P_i de prefijos se genera usando el Algoritmo 30 (línea 13). Los ciclos anidados en las líneas $14-18$ se encargan de calcular la unión de cada cadena del conjunto de prefijos con cada cadena del conjunto de sufijos y agregar la cadena resultante al conjunto T_i. Finalmente, la línea 19 agrega las cadenas por iteración al tubo de prueba T y la línea 20 aumenta la iteración i.

El ciclo principal de las líneas $2-21$ tiene dos condiciones de rompimiento (la cantidad i de iteraciones y que el tubo T alcance su capacidad MAX_T). En cada iteración se agregan $\mathrm{O}(MAX_{T_i})$ cadenas a T, por lo que el número de iteraciones que se requieren para llenar T es aproximadamente $\frac{MAX_T}{MAX_{T_i}}$. Así, el número de iteraciones que produce el ciclo principal es $\mathrm{O}(\text{mín}(\lfloor \frac{k_r}{2} \rfloor, \frac{MAX_T}{MAX_{T_i}}))$.

Al final del ciclo principal de las líneas $2-21$, el tubo de prueba T contiene a lo más MAX_T cadenas. Cada cadena representa un subconjunto de k_r+1 vértices de los n totales. Cada subconjunto fue formado a partir de la unión de dos conjuntos disjuntos que forman un clique cada uno. De esta forma, cada subconjunto elemento de T es un buen candidato a formar un clique, debido a que una buena cantidad de los vértices que lo forman son adyacentes por pares.

En este capítulo se presentan los resultados obtenidos de las ejecuciones de los algoritmos presentados en los Capítulos 4 y 5. De acuerdo a los objetivos de la presente investigación, estos resultados se dividen en dos secciones. En la primera sección se registran tiempos de ejecución para diferentes implementaciones del algoritmo k-Cliques de manera secuencial y paralela, así como las relaciones entre velocidades paralelas y secuenciales (*aceleraciones*) de tales implementaciones. La segunda sección contiene resultados del funcionamiento de la heurística para el problema de clique máximo propuesta en el Capítulo 5.

6.1. Plataforma computacional y casos de prueba

6.1.1. Equipo utilizado

Las implementaciones aquí presentadas fueron ejecutadas en una computadora de escritorio equipada con las siguientes características de hardware y software:

- **Procesador:** Intel Core i7-3820 @3.60 GHz.
- **Memoria RAM:** 8 GB DDR3 RAM.
- **Dispositivos CUDA:** NVidia GeForce GTX 680 (para procesamiento), NVidia GeForce GT 620 (para gráficos independientes).
- **Sistema operativo:** Windows 7 Home Premium SP1 64 bits.
- **Compiladores:** Microsoft Visual Studio 2010 Version 10.0.30319.1 RTMRel, NVidia Nsight Visual Studio Edition Version 2.2 Build number 2.2.0.12313.

6.1.2. Casos de prueba DIMACS

Las pruebas de funcionamiento fueron ejecutadas utilizando los casos de prueba de grafos del segundo desafío de implementación DIMACS (*Center for Discrete Mathematics and Theoretical Computer Science*). Estos casos de prueba son utilizados en una variedad de trabajos, con propósitos de comparación y cuando un análisis formal de la complejidad de los algoritmos no es posible.

Se utilizaron un total de 66 casos de prueba de grafos con tamaños variados desde 28 hasta 3361 vértices. Para su manejo, estos casos de prueba fueron separados en seis clasificaciones de acuerdo al tamaño del problema. Las tablas 2 a 7 muestran los datos de los distintos casos utilizados para cada clasificación, respectivamente. Con el propósito de reducir el tamaño de las tablas, a cada caso se le asigna un número, el cual será utilizado en referencias posteriores en el presente documento. La información de cada caso consiste en el número asignado, nombre, cantidad de vértices del grafo, densidad y el tamaño de clique máximo (cuando incluye el signo mayor o igual es el máximo encontrado al momento).

Tabla 2: Casos de prueba DIMACS con $n \leq 256$ (Clasificación 1).

No. caso	Nombre	n	Densidad	C*
1	johnson8-2-4	28	0.5556	4
2	MANN_a9	45	0.9273	16
3	hamming6-2	64	0.9048	32
4	hamming6-4	64	0.3492	4
5	johnson8-4-4	70	0.7681	14
6	johnson16-2-4	120	0.7647	8
7	keller4	171	0.6491	11
8	brock200_1	200	0.7454	21
9	brock200_2	200	0.4963	12
10	brock200_3	200	0.6054	15
11	brock200_4	200	0.6577	17
12	c-fat200-1	200	0.0771	12
13	c-fat200-2	200	0.1626	24
14	c-fat200-5	200	0.4258	58
15	san200_0.7_1	200	0.7	30
16	san200_0.7_2	200	0.7	18
17	san200_0.9_1	200	0.9	70
18	san200_0.9_2	200	0.9	60
19	san200_0.9_3	200	0.9	44
20	sanr200_0.7	200	0.6969	18
21	sanr200_0.9	200	0.8976	42
22	hamming8-2	256	0.9686	128
23	hamming8-4	256	0.6392	16

6.2. Resultados preliminares

En esta sección, se presentan resultados en tiempos de ejecución y aceleración para distintas implementaciones del algoritmo k-Cliques de forma secuencial y paralela. Estos resultados

Tabla 3: Casos de prueba DIMACS con $300 \leq n \leq 400$ (Clasificación 2).

No. caso	Nombre	n	Densidad	C*
24	p_hat300-1	300	0.2438	8
25	p_hat300-2	300	0.4889	25
26	p_hat300-3	300	0.7445	36
27	MANN_a27	378	0.9901	126
28	brock400_1	400	0.7484	27
29	brock400_2	400	0.7492	29
30	brock400_3	400	0.7479	31
31	brock400_4	400	0.7489	33
32	san400_0.5_1	400	0.5	13
33	san400_0.7_1	400	0.7	40
34	san400_0.7_2	400	0.7	30
35	san400_0.7_3	400	0.7	22
36	san400_0.9_1	400	0.9	100
37	sanr400_0.5	400	0.5011	13
38	sanr400_0.7	400	0.7001	21

se obtuvieron utilizando los primeros cinco casos de grafos DIMACS de la Tabla 2. Por ser los de menor tamaño, estos casos permiten la generación de bibliotecas $B_{[n,k]}$ completas (para ciertos valores de k). De esta forma, es posible probar el filtrado de los algoritmos de manera exacta, obteniendo todos los cliques de un tamaño k dado, existentes en cada caso. Cada medición de tiempo se obtuvo ejecutando 10 veces cada caso y obteniendo el promedio de estos tiempos.

6.2.1. Funciones secuenciales de filtrado

En este apartado, se presentan los tiempos de ejecución para el algoritmo k-Cliques introducido en el Capítulo 5. Este algoritmo se implementó en cuatro variantes utilizando la implementación de las funciones del modelo de etiquetas de manera secuencial, tal como se describió en el Capítulo 4. Estas variantes son las siguientes:

- **kCliques1:** Algoritmo k-Cliques sin considerar ordenamiento de las hebras por tubo.
- **kCliques2:** Algoritmo k-Cliques mejorado sin considerar ordenamiento de las hebras por tubo.
- **kCliques3:** Algoritmo k-Cliques considerando ordenamiento de las hebras por tubo.

Tabla 4: Casos de prueba DIMACS con $496 \leq n \leq 700$ (Clasificación 3).

No. caso	Nombre	n	Densidad	C*
39	johnson32-2-4	496	0.8788	16
40	c-fat500-1	500	0.0357	14
41	c-fat500-10	500	0.3738	126
42	c-fat500-2	500	0.0733	26
43	c-fat500-5	500	0.1859	64
44	p_hat500-1	500	0.2531	9
45	p_hat500-2	500	0.5046	36
46	p_hat500-3	500	0.7519	50
47	p_hat700-1	700	0.2493	11
48	p_hat700-2	700	0.4976	44
49	p_hat700-3	700	0.748	62

Tabla 5: Casos de prueba DIMACS con $776 \leq n \leq 1035$ (Clasificación 4).

No. caso	Nombre	n	Densidad	C*
50	keller5	776	0.7515	27
51	brock800_1	800	0.6493	23
52	brock800_2	800	0.6513	24
53	brock800_3	800	0.6487	25
54	brock800_4	800	0.6497	26
55	p_hat1000-1	1000	0.2448	10
56	p_hat1000-2	1000	0.4901	46
57	p_hat1000-3	1000	0.7442	68
58	san1000	1000	0.5015	15
59	hamming10-2	1024	0.9902	512
60	hamming10-4	1024	0.8289	40
61	MANN_a45	1035	0.9963	345

- **kCliques4:** Algoritmo k-Cliques mejorado considerando ordenamiento de las hebras por tubo.

Las tablas 8 a 12 muestran los resultados para cada uno de los cinco casos seleccionados. Cada tabla incluye distintos valores de k (los posibles para cada caso). Para cada valor k se muestra la cantidad de hebras de la biblioteca $B_{[n,k]}$ completa (dada por $\binom{n}{k}$), y los tiempos de ejecución de cada variante. Todos los tiempos están dados en segundos. Los tiempos menores se resaltan en negritas.

En las tablas 8 a 12 se puede apreciar el comportamiento exponencial en el tiempo de ejecución de cada una de las variantes, para valores sucesivamente más grandes de k. A excepción del caso 2 para el valor $k = 2$, en todos los casos y para todos los valores de k

Tabla 6: Casos de prueba DIMACS con $1500 \leq n \leq 1500$ (Clasificación 5).

No. caso	Nombre	n	Densidad	C*
62	p_hat1500-1	1500	0.2534	12
63	p_hat1500-2	1500	0.5061	65
64	p_hat1500-3	1500	0.7536	≥ 94

Tabla 7: Casos de prueba DIMACS con $3321 \leq n \leq 3361$ (Clasificación 6).

No. caso	Nombre	n	Densidad	C*
65	MANN_a81	3321	0.9988	≥ 1100
66	keller6	3361	0.8182	≥ 59

considerados, la variante kCliques4 es la más veloz. Otra característica importante de resaltar, es la notoria disminución de los tiempos de ejecución entre los algoritmos que consideran ordenamiento respecto de los que no lo consideran. Esta característica se puede apreciar comparando la columna kCliques3 con la columna kCliques1 y la columna kCliques4 con la columna kCliques2. En todos los casos y para todos los valores de k, los resultados indican que es computacionalmente menos costoso realizar el ordenamiento de las hebras por tubo cuando es requerido, que no realizarlo, e iterar sobre todas las hebras en cada operación.

Por otro lado, las tablas también precisan la ventaja del algoritmo k-Cliques mejorado sobre el algoritmo k-Cliques. Esto se visualiza comparando la columna kCliques2 con la columna kCliques1 y la columna kCliques4 con la columna kCliques3. Esta mejora es más marcada en los casos de grafos densos (tablas 9 y 10), que en los de menor densidad (tablas 8, 11 y 12). Como se recordará del Capítulo 4, esta mejora consiste en la reducción de iteraciones que realiza el algoritmo: en lugar de iterar sobre todas las adyacencias posibles, se itera solamente sobre las no adyacencias del grafo. Así, la reducción en tiempo de ejecución es proporcional a la densidad del grafo.

6.2.2. Funciones paralelas de filtrado

En esta subsección se mide el desempeño de las implementaciones de los algoritmos paralelos de filtrado introducidos en el Capítulo 5. Como se recordará del capítulo mencionado, se diseñaron e implementaron tres algoritmos diferentes, nombrados algoritmos A, B y C. Las tablas 13 a 17 describen los tiempos de ejecución de cada uno de estos algoritmos, utilizando los mismos cinco casos seleccionados en la subsección anterior y los mismos valores de k. En

Tabla 8: Tiempos de ejecución (en segundos) para las variantes del algoritmo k-Cliques secuencial sobre el caso 1 ($n = 28$) y los valores $2 \leq k \leq 14$.

k	Número de hebras	kCliques1	kCliques2	kCliques3	kCliques4
2	378	0.0026	0.0017	0.0022	**0.0010**
3	3,276	0.028	0.0102	0.011	**0.005**
4	20,475	0.1075	0.0507	0.0425	**0.0229**
5	98,280	0.478	0.2148	0.1449	**0.0714**
6	376,740	1.744	0.7724	0.4371	**0.2123**
7	1,184,040	5.43	2.4	1.14	**0.5585**
8	3,108,105	14.30	6.38	2.54	**1.25**
9	6,906,900	31.60	14.01	4.81	**2.46**
10	13,123,110	59.10	26.41	7.97	**4.08**
11	21,474,180	96.3	42.91	11.4	**5.89**
12	30,421,755	135.19	60.29	14.18	**7.37**
13	37,442,160	165.27	74.14	15.17	**8.11**
14	40,116,600	176.06	79.11	14.28	**7.65**

Tabla 9: Tiempos de ejecución (en segundos) para las variantes del algoritmo k-Cliques secuencial sobre el caso 2 ($n = 45$) y los valores $2 \leq k \leq 7$.

k	Número de hebras	kCliques1	kCliques2	kCliques3	kCliques4
2	990	0.0172	**0.0012**	0.013	0.0014
3	14,190	0.2551	0.019	0.1594	**0.0146**
4	148,995	2.58	0.1881	1.51	**0.1409**
5	1,221,759	20.93	1.52	11.21	**1.04**
6	8,145,060	132.15	9.71	65.69	**6.23**
7	45,379,620	649.94	51.57	287.23	**30.14**

estas tablas se omite la cantidad de hebras por cada valor de k pues son exactamente las mismas que las indicadas en las tablas 8 a 12. Los tiempos están dados en milisegundos. Los tiempos menores están resaltados en negritas.

Tabla 10: Tiempos de ejecución (en segundos) para las variantes del algoritmo k-Cliques secuencial sobre el caso 3 ($n = 64$) y los valores $2 \leq k \leq 6$.

k	Número de hebras	kCliques1	kCliques2	kCliques3	kCliques4
2	2,016	0.0713	0.0075	0.0486	**0.0065**
3	41,664	1.47	0.1440	0.9219	**0.1057**
4	635,376	21.03	2.11	12.81	**1.43**
5	7,624,512	257.98	25.48	143.23	**15.78**
6	74,974,368	2349.45	232.15	1220.79	**131.59**

Tabla 11: Tiempos de ejecución (en segundos) para las variantes del algoritmo k-Cliques secuencial sobre el caso 4 ($n = 64$) y los valores $2 \leq k \leq 6$.

k	Número de hebras	kCliques1	kCliques2	kCliques3	kCliques4
2	2,016	0.0648	0.0425	0.0409	**0.0267**
3	41,664	1.22	0.7908	0.5311	**0.3597**
4	635,376	16.89	10.97	6.13	**3.80**
5	7,624,512	206.49	134.21	62.56	**42.66**
6	74,974,368	1968.58	1290.28	531.83	**358.79**

Tabla 12: Tiempos de ejecución (en segundos) para las variantes del algoritmo k-Cliques secuencial sobre el caso 5 ($n = 70$) y los valores $2 \leq k \leq 5$.

k	Número de hebras	kCliques1	kCliques2	kCliques3	kCliques4
2	2,415	0.1009	0.0247	0.0674	0.0182
3	54,740	2.27	0.5358	1.25	0.3386
4	918,895	37.44	8.72	17.50	4.48
5	12,103,014	449.11	104.10	191.03	47.97

Tabla 13: Tiempos de ejecución (en milisegundos) para las tres implementaciones de algoritmos paralelos de filtrado sobre el caso 1 ($n = 28$)y los valores $2 \leq k \leq 14$.

k	A	B	C
2	124.33	11.36	**1.07**
3	124.63	11.37	**1.18**
4	128.86	12.04	**1.97**
5	146.33	15.35	**5.23**
6	210.04	26.31	**15.29**
7	360.75	57.82	**38.67**
8	685.15	128.24	**75.95**
9	1289.25	239.7	**129.59**
10	2225.69	436.88	**193.18**
11	3429.03	713.02	**262.33**
12	4682.88	1018.06	**312.35**
13	5626.16	1241.12	**317.64**
14	5926.18	1329.46	**308.74**

Tabla 14: Tiempos de ejecución (en milisegundos) para las tres implementaciones de algoritmos paralelos de filtrado sobre el caso 2 ($n = 45$) y los valores $2 \leq k \leq 7$.

k	A	B	C
2	304	38.77	**2.22**
3	323.98	31.56	**2.51**
4	420.59	28.29	**22.23**
5	1121.14	**23.14**	165.49
6	5560.03	**116.48**	843.29
7	29902.82	**610.17**	4085.2

Tabla 15: Tiempos de ejecución (en milisegundos) para las tres implementaciones de algoritmos paralelos de filtrado sobre el caso 3 ($n = 64$) y los valores $2 \leq k \leq 6$.

k	A	B	C
2	622.52	33.27	**3.41**
3	677.21	35.88	**10.94**
4	1494.91	**40.76**	138.93
5	10688.37	**286.4**	1333.32
6	100969.84	**2396.91**	11260.36

Tabla 16: Tiempos de ejecución (en milisegundos) para las tres implementaciones de algoritmos paralelos de filtrado sobre el caso 4 ($n = 64$) y los valores $2 \leq k \leq 6$.

k	A	B	C
2	610.63	88.8	**3.31**
3	655.26	100.64	**8.39**
4	1400.39	277.54	**82.52**
5	8801.17	1958.43	**570.01**
6	78720.52	16938.90	**3752.08**

Tabla 17: Tiempos de ejecución (en milisegundos) para las tres implementaciones de algoritmos paralelos de filtrado sobre el caso 5 y los valores $2 \leq k \leq 5$.

k	A	B	C
2	743.51	53.17	**3.74**
3	845.64	60.73	**14.27**
4	2320.55	**143.61**	194.41
5	21526.9	**1262.13**	1832.00

Del análisis de las tablas 13 a 17 se desprende que, en general, el Algoritmo C es el más veloz. Sin embargo, en los casos de grafos densos y para valores de k mayores que un k_0 inicial, el Algoritmo B registra mejores tiempos (tablas 14, 15 y 17). Esto se debe a las variaciones en los detalles técnicos de cada algoritmo (revisadas en el Capítulo 5). En todos los casos, el Algoritmo A registra los mayores tiempos y es más lento con respecto de los algoritmos B y C por varios órdenes de magnitud. La principal causa del pobre desempeño de este algoritmo es la cantidad de llamadas a kernel que realiza. Una vez más, los tiempos para todos los algoritmos crecen de forma exponencial para valores de k sucesivamente más grandes.

6.2.3. *Aceleraciones*

Utilizando los tiempos de los apartados anteriores, en esta subsección se presentan las aceleraciones de los tiempos de ejecución de las tres implementaciones paralelas con respecto de cada una de las cuatro implementaciones secuenciales. Las figuras 48 a 62 muestran estos datos para cada algoritmo paralelo y por cada una de los cinco casos de prueba seleccionados.

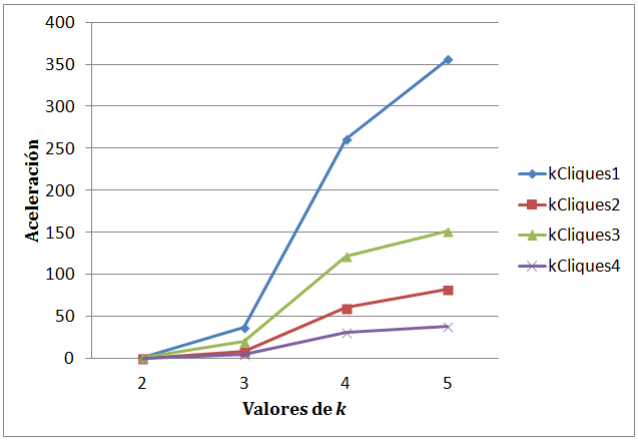


Figura 48: Aceleración del algoritmo de filtrado paralelo A sobre las diferentes variaciones secuenciales para el caso 1.

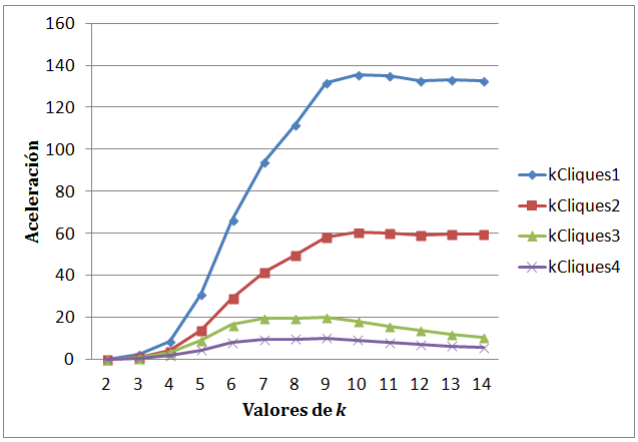


Figura 49: Aceleración del algoritmo de filtrado paralelo B sobre las diferentes variaciones secuenciales para el caso 1.

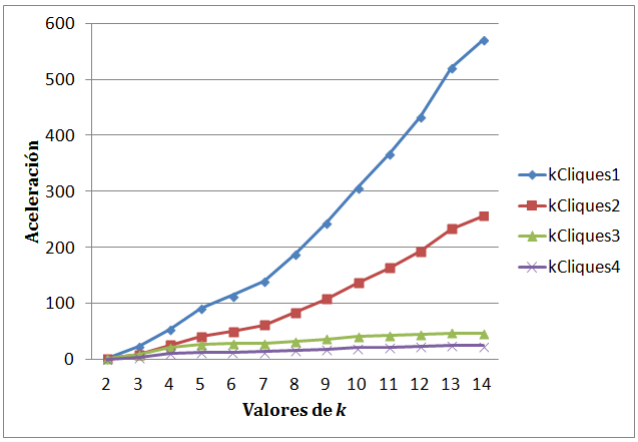


Figura 50: Aceleración del algoritmo de filtrado paralelo C sobre las diferentes variaciones secuenciales para el caso 1.

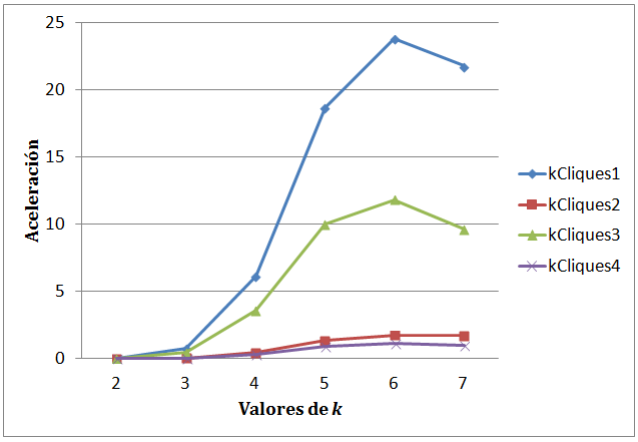


Figura 51: Aceleración del algoritmo de filtrado paralelo A sobre las diferentes variaciones secuenciales para el caso 2.

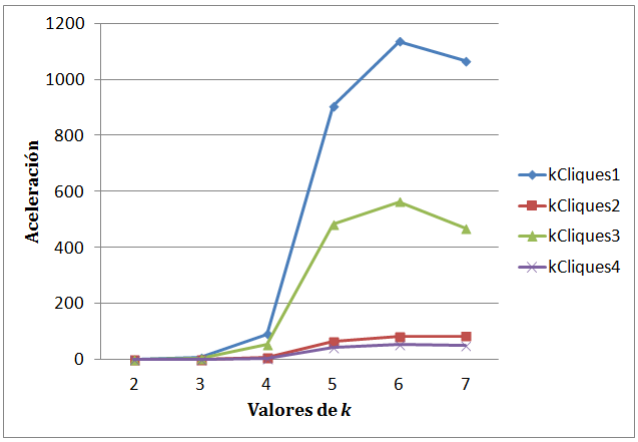


Figura 52: Aceleración del algoritmo de filtrado paralelo B sobre las diferentes variaciones secuenciales para el caso 2.

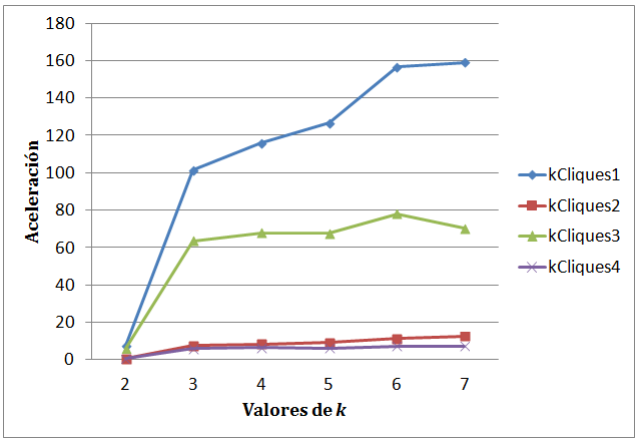


Figura 53: Aceleración del algoritmo de filtrado paralelo C sobre las diferentes variaciones secuenciales para el caso 2.

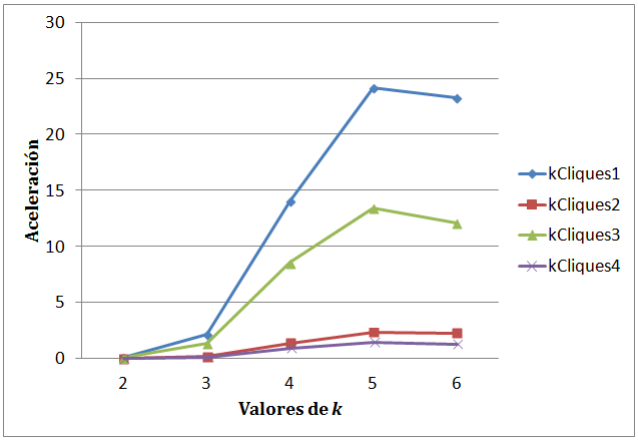


Figura 54: Aceleración del algoritmo de filtrado paralelo A sobre las diferentes variaciones secuenciales para el caso 3.

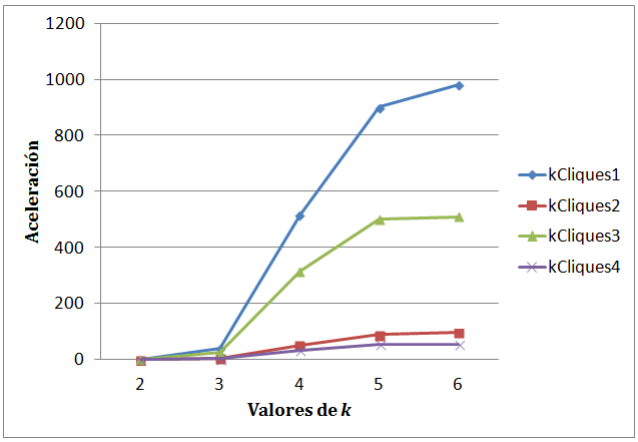


Figura 55: Aceleración del algoritmo de filtrado paralelo B sobre las diferentes variaciones secuenciales para el caso 3.

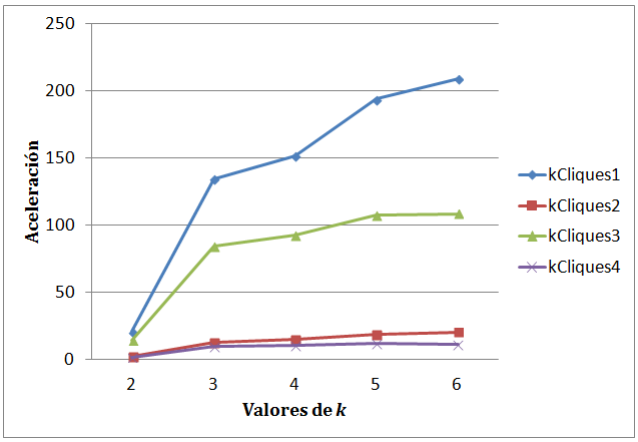


Figura 56: Aceleración del algoritmo de filtrado paralelo C sobre las diferentes variaciones secuenciales para el caso 3.

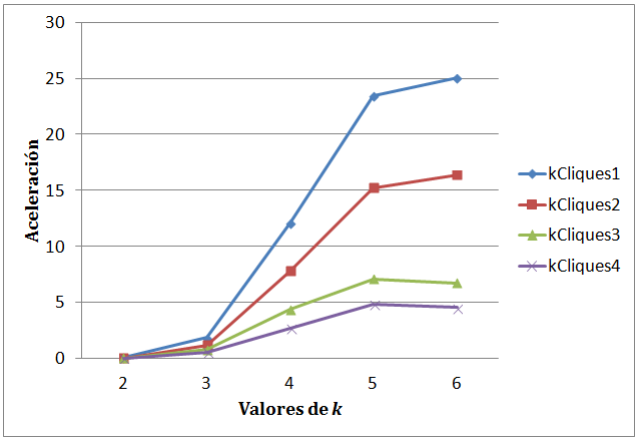


Figura 57: Aceleración del algoritmo de filtrado paralelo A sobre las diferentes variaciones secuenciales para el caso 4.

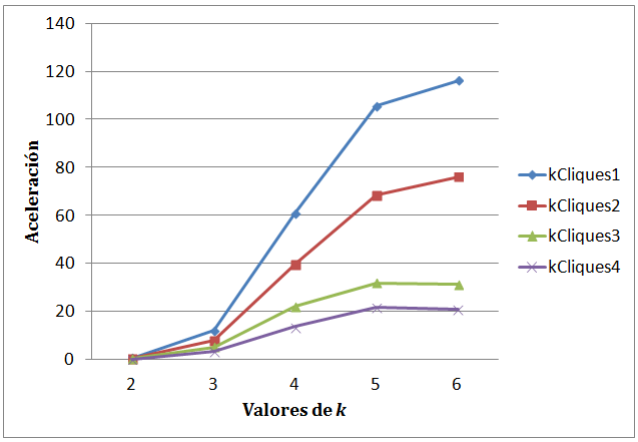


Figura 58: Aceleración del algoritmo de filtrado paralelo B sobre las diferentes variaciones secuenciales para el caso 4.

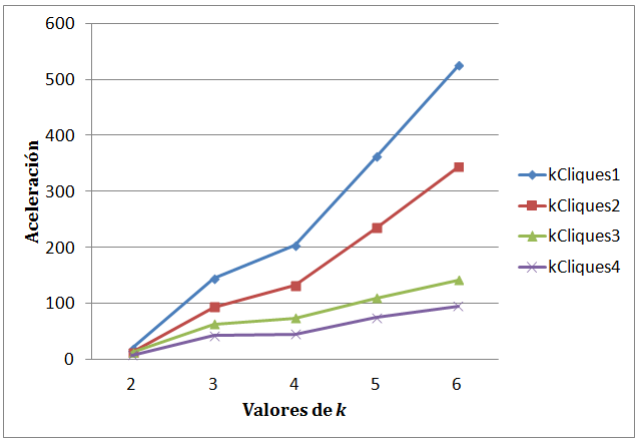


Figura 59: Aceleración del algoritmo de filtrado paralelo C sobre las diferentes variaciones secuenciales para el caso 4.

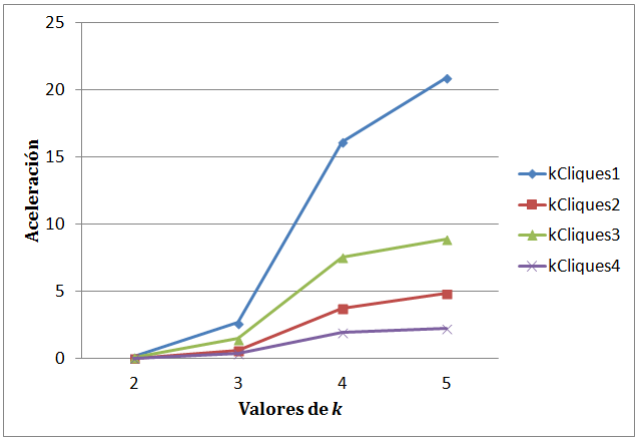


Figura 60: Aceleración del algoritmo de filtrado paralelo A sobre las diferentes variaciones secuenciales para el caso 5.

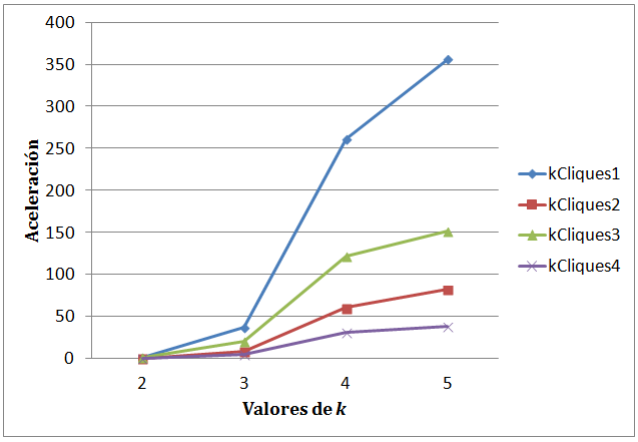


Figura 61: Aceleración del algoritmo de filtrado paralelo B sobre las diferentes variaciones secuenciales para el caso 5.

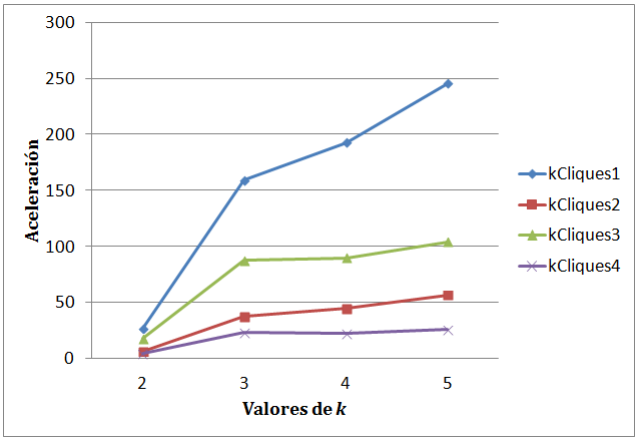


Figura 62: Aceleración del algoritmo de filtrado paralelo C sobre las diferentes variaciones secuenciales para el caso 5.

De las figuras 48 a 62 se destaca la característica de los aumentos en las aceleraciones de todos los algoritmos a medida que k crece (produciendo una mayor cantidad de hebras). Esta condición es producto de la característica de CUDA de obtener un mejor desempeño en el paralelismo sobre una cantidad masiva de datos.

6.3. Resultados de la heurística

6.3.1. Configuración de parámetros

Como se recordará del Capítulo 5, el funcionamiento de la heurística propuesta depende de los valores de algunos parámetros. Particularmente, el uso de memoria juega un papel esencial en la calidad de los resultados obtenidos. Los siguientes valores son los que se utilizaron en la ejecución de todos las corridas de la implementación de la heurística:

- **M (Cantidad máxima de cliques en el vector de cliques):** 20.
- MAX_{P_i} **(cantidad máxima de prefijos por iteración):** 1024.
- MAX_{S_i} **(cantidad máxima de sufijos por iteración):** 8192.
- MAX_{T_i} **(cantidad máxima de hebras por iteración):** $MAX_{P_i} \times MAX_{S_i}$.
- MIN_P **(cantidad mínima de prefijos para generación probabilística):** 200.
- MIN_S **(cantidad mínima de sufijos para generación aleatoria):** 200.

La cantidad total de hebras por tubo acotada por el parámetro MAX_T se configuró para cada clasificación de los casos. La Tabla 18 muestra los distintos valores del parámetro MAX_T dependiendo de la clasificación del caso. Para cada clasificación, se incluyen también los valores (n) mínimos y máximos, el tamaño máximo de cada hebra (Bytes) y el máximo de memoria a utilizar (MBytes). Las cantidades de hebras fueron definidas de manera que la memoria máxima total a utilizar no rebase los 512 MBytes de memoria en cada clasificación.

6.3.2. Resultados

Las tablas 19 a 24 presentan los resultados obtenidos de la heurística. Se incluyen los tamaños de clique y tiempos promedio de las 10 ejecuciones de cada caso. Además, se agrega el mejor resultado de estas ejecuciones. El mejor resultado consiste en la ejecución con el mayor

Tabla 18: Cantidad de hebras y uso de memoria para las distintas clasificaciones de casos.

Clasificación	MAX_T	n min	n max	Tamaño*	Memoria*
1	16,000,000	28	256	33	503.5
2	9,570,000	300	400	56	511.1
3	5,830,000	496	700	92	511.5
4	3,945,000	776	1,035	136	511.7
5	2,795,000	1,500	1,500	192	511.8
6	1,254,000	3,321	3,361	428	511.8

tamaño de clique, en caso de empate se considera la mayor cantidad de cliques encontrados y por último, en caso de coincidir en los primeros criterios, la de menor tiempo de ejecución. Con propósitos de comparación, se agregan los resultados de la heurística ILS propuesta en (Batsyn *et al.*, 2013) donde a su vez, comparan sus resultados con el algoritmo MCS de (Tomita *et al.*, 2010) que también son agregados en las tablas. Cabe mencionar que tanto la heurística ILS como el algoritmo MCS obtienen los mismos tamaños de clique máximos y solamente varía su tiempo de ejecución. Todos los tiempos están dados en segundos.

De los resultados presentados anteriormente en las tablas 19 a 24, una de las características más notables es que los tiempos de ejecución no dependen directamente de los tamaños de problema. Por ejemplo, el caso número 63 con un tamaño $n = 1500$ toma en promedio 401 segundos en ejecutarse, mientras que el caso número 6 con un tamaño $n = 120$ significativamente menor toma en promedio 456 segundos. Lo mismo ocurre con los tamaños de clique máximo de los casos, los cuales no influyen en los tiempos de ejecución. Esto se debe a que no es posible establecer un análisis de complejidad del tiempo de ejecución de la heurística, en función del tamaño del caso particular de problema. Lo anterior es consecuencia de la naturaleza de la heurística. Como se recordará del Capítulo 5, la Fase II es "alimentada" inicialmente por cliques encontrados en la Fase I. Posteriormente, la Fase II procesa iterativamente conjuntos de cliques y busca cliques progresivamente más grandes. Así, el tiempo de ejecución de la Fase II depende de factores tales como: la cantidad de cliques iniciales a procesar, la calidad de los cliques iniciales (si contienen una buena cantidad de vértices presentes en el clique máximo del grafo), y el tamaño inicial de clique k_0 (la medida en que se aproxima al tamaño del clique máximo del grafo). Estos factores derivan del desempeño de la etapa de aproximación de cliques. Tanto la etapa de aproximación como la de extensión de cliques incluyen componentes no determinísticos en su operación.

Tabla 19: Comparación de resultados para clasificación de casos 1. Los tamaños de clique más grandes se resaltan en negritas. En los casos de empate en los tamaños de clique, se subrayan los mejores tiempos. Todos los tiempos están en segundos.

Caso		**Promedio**		**Mejor caso**				**ILS**	**MCS**
No.	|C*|	*tam.*	*tiempo*	*cant.*	*tam.*	*tiempo*	*tam.*	*tiempo*	*tiempo*
1	4	**4**	2.52	20	**4**	2.46	**4**	0	0
2	16	**16**	65.67	20	**16**	60.49	**16**	0	0
3	32	**32**	4.25	2	**32**	4.22	**32**	0	0
4	4	**4**	5.06	20	**4**	5	**4**	0	0
5	14	**14**	77.45	15	**14**	71.77	**14**	1	0
6	8	**8**	456.53	20	**8**	146.5	**8**	2	0
7	11	**11**	155.57	11	**11**	256.45	**11**	7	0
8	21	20	655.52	1	**21**	883.08	**21**	7	1
9	12	10.5	36.23	1	**12**	11.31	**12**	12	0
10	15	13.2	179.68	1	**15**	11.86	**15**	10	0
11	17	15	77.13	4	15	153.7	**17**	9	0
12	12	**12**	883.45	14	**12**	813.46	**12**	11	0
13	24	**24**	13.04	1	**24**	12.93	**24**	10	0
14	58	**58**	124.07	3	**58**	123.68	**58**	7	0
15	30	**30**	430.94	1	**30**	17.2	**30**	6	0
16	18	15.6	214.13	1	**18**	70.06	**18**	7	0
17	70	**70**	980.06	1	**70**	683.72	**70**	2	0
18	60	58	1164.08	1	**60**	853.22	**60**	2	0
19	44	36.1	188.22	3	38	742.29	**44**	2	0
20	18	**18**	140.92	1	**18**	103.46	**18**	7	1
21	42	39.8	886.96	4	40	886.51	**42**	28	42
22	128	**128**	40.58	2	**128**	40.41	**128**	1	0
23	16	**16**	218.75	20	**16**	214.58	**16**	14	0

Tabla 20: Comparación de resultados para clasificación de casos 2. Los tamaños de clique más grandes se resaltan en negritas. En los casos de empate en los tamaños de clique, se subrayan los mejores tiempos. Todos los tiempos están en segundos.

Caso		**Promedio**		**Mejor caso**				**ILS**	**MCS**
No.	|C*|	*tam.*	*tiempo*	*cant.*	*tam.*	*tiempo*	*tam.*	*tiempo*	*tiempo*
24	8	**8**	144.31	5	**8**	244.8	**8**	27	0
25	25	**25**	122.09	4	**25**	153.24	**25**	17	0
26	36	**35**	1451.06	1	**35**	1111.19	**35**	11	3
27	126	125	189.57	20	125	188.93	**126**	4	0
28	27	**23**	519.61	2	**23**	680.35	**23**	483	460
29	29	22.1	120.84	1	23	291.16	**24**	212	199
30	31	22	112.28	2	22	202.24	**24**	346	320
31	33	24.1	483.26	1	**33**	2414.23	26	135	161
32	13	9	338.78	1	**13**	603.73	**13**	65	0
33	40	27.5	420.38	1	**40**	20.2	**40**	26	1
34	30	19.2	192.78	1	**30**	967.92	**30**	33	0
35	22	16.5	276.71	4	18	356.74	**22**	35	2
36	100	91	2960.74	1	**100**	2548.5	**100**	6	1
37	13	**12**	97.29	6	**12**	115.35	**12**	50	1
38	21	19.1	60.62	1	**20**	103.19	**20**	180	171

Tabla 21: Comparación de resultados para clasificación de casos 3. Los tamaños de clique más grandes se resaltan en negritas. En los casos de empate en los tamaños de clique, se subrayan los mejores tiempos. Todos los tiempos están en segundos.

Caso		**Promedio**		**Mejor caso**				**ILS**	**MCS**
No.	|C*|	*tam.*	*tiempo*	*cant.*	*tam.*	*tiempo*	*tam.*	*tiempo*	*tiempo*
39	16	**16**	321.43	20	**16**	315.17	NA	NA	NA
40	14	**14**	1013.1	19	**14**	939.88	**14**	62	0
41	126	**126**	181.38	3	**126**	180.95	**126**	42	0
42	26	**26**	92.67	19	**26**	91.78	**26**	42	0
43	64	**64**	109.33	3	**64**	108.7	**64**	56	0
44	9	**9**	28.66	5	**9**	38.164	**9**	75	0
45	36	**35**	55.45	1	**35**	55.04	34	42	1
46	50	**49**	1727.18	17	**49**	1724.74	48	171	164
47	11	9	73.85	8	9	86.84	**11**	147	0
48	44	**43**	180.2	1	**43**	179.93	42	81	6
49	62	**60**	2411.94	3	**60**	2410.06	**60**	1303	1529

Tabla 22: Comparación de resultados para clasificación de casos 4. Los tamaños de clique más grandes se resaltan en negritas. En los casos de empate en los tamaños de clique, se subrayan los mejores tiempos. Todos los tiempos están en segundos.

Caso		**Promedio**		**Mejor caso**			**ILS**	**MCS**
No.	|C*|	*tam.*	*tiempo*	*cant.*	*tam.*	*tiempo*	*tam.*	*tiempo*	*tiempo*
50	27	26.2	67.93	2	**27**	65.77	**27**	78957	78875
51	23	**19**	34.11	2	**19**	50.28	**19**	6482	6337
52	24	19	50.59	2	19	50.28	**20**	5886	5737
53	25	**20**	260.68	2	**20**	245.04	19	3994	3830
54	26	**19**	33.06	1	**19**	32.98	**19**	3009	2849
55	10	**10**	38.57	3	**10**	48.76	**10**	302	0
56	46	43	69.86	1	43	69.33	**44**	360	272
57	68	63	1296.22	9	63	1225.45	**67**	214611	> 1000000
58	15	9.9	156.39	5	10	142.92	**15**	464	3
59	512	**512**	817.82	2	**512**	814.95	473	0	0
60	40	37.1	223.86	2	**38**	224.77	NA	NA	NA
61	345	342	8715.46	20	342	8644.69	**344**	130	126

Tabla 23: Comparación de resultados para clasificación de casos 5. Los tamaños de clique más grandes se resaltan en negritas. En los casos de empate en los tamaños de clique, se subrayan los mejores tiempos. Todos los tiempos están en segundos.

Caso		**Promedio**		**Mejor caso**			**ILS**	**MCS**
No.	|C*|	*tam.*	*tiempo*	*cant.*	*tam.*	*tiempo*	*tam.*	*tiempo*	*tiempo*
62	12	**11**	65.49	2	**11**	71.84	**11**	696	3
63	65	**61**	401.38	1	**61**	343.49	**61**	10231	11859
64	>= 94	**84**	1141.84	1	**84**	820.45	NA	NA	NA

Tabla 24: Comparación de resultados para clasificación de casos 6. Los tamaños de clique más grandes se resaltan en negritas. En los casos de empate en los tamaños de clique, se subrayan los mejores tiempos. Todos los tiempos están en segundos.

Caso		**Promedio**		**Mejor caso**			**ILS**	**MCS**
No.	|C*|	*tam.*	*tiempo*	*cant.*	*tam.*	*tiempo*	*tam.*	*tiempo*	*tiempo*
65	>= 1100	**1096**	105272.05	20	**1096**	104772	NA	NA	NA
66	>= 59	45.7	17452.82	1	**47**	17236.24	NA	NA	NA

Por otro lado, la heurística propuesta produjo resultados competitivos al encontrar el tamaño de clique máximo en 25 casos en el promedio de la ejecuciones y en 6 casos en la mejor ejecución, del total de 66 casos de DIMACS. Con respecto a los resultados presentados por Batsyn y colaboradores (2013), la heurística encontró cliques de tamaños más grandes en los casos 45, 46, 48, 53 y 59 en el promedio de las ejecuciones y en el caso 31 con la mejor ejecucón. En los casos 51, 53, 54 y 63 se encuentra el mismo tamaño de clique que ambos algoritmos ILS y MCS, pero con menores tiempos de ejecución. En el caso 62, el tiempo es significativamente menor que la heurística ILS. En los casos 50, 52 y 57, no obstante que se encuentran cliques de menor tamaño, los tiempos son una fracción de los algoritmos comparados. En (Batsyn *et al.*, 2013) no incluyen resultados de los casos 39, 60, 64, 65 y 66.

Por último, cabe mencionar que una de las principales características de la heurística introducida en el presente trabajo radica en la cantidad de cliques encontrados. La columna *cantidad* que se incluye en la sección de mejor caso de ejecución de las tablas, exhibe esta característica. Otros enfoques revisados en la literatura, se basan principalmente en estrategias de ramificación y cota. Dado que esta estrategia construye cliques más grandes al añadir vértices sucesivamente al mayor clique encontrado, producen una solución única para cada caso particular de problema. Así, esta característica cuantitativa constituye una de las principales ventajas del enfoque presentado en este trabajo.

Capítulo 7. Conclusiones y trabajo futuro

7.1. Conclusiones

En el presente trabajo de tesis, en primer término, se indagó sobre el desempeño de una plataforma para la simulación del paralelismo de los algoritmos del llamado modelo de etiquetas. Para esto, se utilizó la arquitectura de paralelismo masivo de hilos de CUDA, estableciendo una correspondencia con el paralelismo masivo de hebras del modelo de etiquetas. Se probó experimentalmente el desempeño utilizando el algoritmo k-Cliques definido en (Martínez-Pérez y Zimmermann, 2009) como caso de prueba. Este paralelismo de hebras del modelo de etiquetas radica esencialmente en sus operaciones, pues éstas actúan concurrentemente sobre todas las hebras contenidas en un tubo, ya sea para modificar la información o transferirlas. En el caso de CUDA, el paralelismo de hilos se realiza a través de funciones llamadas *kernels* que crean un grid masivo de hilos. De esta forma, la definición de cada operación del modelo de etiquetas en la arquitectura CUDA es natural (y unívoca): cada operación es un *kernel*. Sin embargo, a pesar de que la implementación en CUDA produjo aceleraciones notables, lo anterior trae como consecuencia que el tiempo de ejecución de los algoritmos del modelo de etiquetas no sea el mejor. Esto se debe principalmente a que los algoritmos tienen una cantidad polinomial de operaciones, lo que se traduce en una cantidad polinomial de llamadas a *kernel* en CUDA, con su correspondiente retraso.

Por otro lado, para el logro del segundo objetivo general de la presente investigación, se diseñó una heurística para el problema de clique máximo y se obtuvieron resultados experimentales de su desempeño sobre los casos de prueba de DIMACS. Esta heurística está basada en la codificación del espacio de soluciones y el filtrado del algoritmo k-Cliques del modelo de etiquetas. En un principio, se buscan cliques iniciales de un tamaño que se toman como base para la generación de espacios de búsqueda posteriores. Una vez obtenido un conjunto inicial de cliques se procesa iterativamente uno por uno, generando un espacio de búsqueda con cada clique. Este espacio de búsqueda consiste en un tubo de prueba que contiene hebras que codifican posibles cliques en el grafo. La generación del espacio de búsqueda está basada en la propiedad de que se puede tomar cualquier subconjunto de vértices de un clique, e ir agregando subconjuntos de vértices que formen un clique entre sí. De esta forma, cada hebra

obtenida contiene dos cliques separados y es más probable que codifique un clique existente en el grafo. Este espacio de búsqueda es filtrado posteriormente para detectar las hebras que codifican un clique. Para lo anterior, se diseñaron dos funciones de fitrado paralelo que realizan la misma operación que el algoritmo k-Cliques, pero optimizadas para un mejor desempeño en CUDA. La operación de filtrado "marca" las hebras que no codifican un clique las cuales son eliminadas en una operación posterior. Las hebras que codifican un clique son procesadas en iteraciones posteriores y este proceso se repite mientras se sigan encontrando nuevos cliques.

Los resultados obtenidos en la ejecución de los 66 casos de prueba de DIMACS se compararon con aquellos publicados en (Batsyn *et al.*, 2013), los cuales constituyen el estado del arte en algoritmos ramifica y acota para el problema de clique máximo. Estos resultados parecen indicar que la heurística introducida, se desempeña mejor en los casos de prueba de tamaños moderados a grandes. Además, cabe mencionar que una de las características más sobresalientes de la heurística desarrollada en el presente trabajo, consiste en el enfoque de construcción de un espacio de búsqueda y posterior filtrado para la detección de soluciones. Este enfoque permite la obtención de más de una solución en muchos de los casos, a diferencia de trabajos anteriores donde se va construyendo una solución única. Hasta el momento, y en lo que al conocimiento del autor de la presente tesis respecta, este el el primer trabajo donde se reportan múltiples soluciones para un mismo caso de prueba en una heurística para el problema de clique máximo.

7.2. Trabajo futuro

El presente trabajo puede servir como punto de partida para futuras investigaciones que involucren la implementación de simuladores de modelos de cómputo biomolecular en arquitecturas paralelas y en heurísticas que se basen en la codificación y filtrado paralelo del modelo de etiquetas. Los siguientes aspectos se dejan como oportunidades de mejora a la presente implementación:

- **Robustecimiento de plataforma.** Implementar otros algoritmos del modelo de etiquetas existentes en la literatura. Obtener resultados del desempeño.

- **Experimentación con grafos aleatorios.** Obtener resultados del desempeño de la presente implementación sobre grafos generados aleatoriamente.

- **Paralelización de otros módulos.** A excepción del filtrado, todo el funcionamiento de la heurística es secuencial. Los módulos de aproximación inicial de cliques, generación de tubo de prueba a partir de un clique y selección de cliques son paralelizables. Con esto, se espera que los tiempos de ejecución se reduzcan significativamente.

- **Escalamiento.** Aprovechando las características de escalabilidad que ofrece CUDA y habiendo paralelizado los módulos secuenciales, es posible probar el desempeño de la presente heurística en una plataforma GPGPU extendida. Esta plataforma puede consistir en una computadora con múltiples GPU's o un clúster de computadoras.

- **Operación asíncrona.** Los módulos de generación de tubo de prueba y de filtrado son independientes. Con esto es posible traslapar su operación de manera que se maximice la utilización de los recursos computacionales, reduciendo los tiempos de ejecución.

- **Generación de tubo "plantilla".** En un vector de cliques, los tamaños de clique son los mismos, por lo que lo único que varía es el ordenamiento de los vértices en cada clique. Tomando esta consideración en cuenta, es posible generar un único tubo de prueba para todos los cliques en el vector, en lugar de generar un tubo de prueba para cada uno. Lo anterior podría no afectar significativamente la calidad de los resultados obtenidos, pero sí reducir considerablemente los tiempos de ejecución.

Referencias bibliográficas

Aarts, E. y Korst, J. (1989). *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Wiley. ISBN 0-471-92146-7.

Adleman, L. (1996). On constructing a molecular computer. En: *1st DIMACS workshop on DNA based computers*, Vol. 27, pp. 1–21. American Mathematical Society.

Adleman, L. M. (1994). Molecular Computation of Solutions to Combinatorial Problems. *Science*, **266**(11): 1021–1024.

Ahrabian, H., Mirzaei, A., y Nowzari-Dalini, A. (2008). A DNA sticker algorithm for solving N-Queen problem. *International Journal of Computer Science and Applications*, **5**(2): 12–22.

Alonso Sanchez, C. y Soma, N. Y. (2009). A Polynomial-time DNA Computing Solution for the Bin-Packing Problem. *Applied Mathematics and Computation*, **215**(6): 2055–2062.

Amos, M. (2005). *Theoretical and experimental DNA computation*. Natural Computing Series. Springer. ISBN 978-3-540-28131-3.

Andrade, D. V., Resende, M. G., y Werneck, R. F. (2012). Fast local search for the maximum independent set problem. *Journal of Heuristics*, **18**(4): 525–547.

Arnold, M. G. (2011). An improved DNA-sticker addition algorithm and its application to logarithmic arithmetic. En: *DNA Computing and Molecular Programming*, Vol. 6937, pp. 34–48. Springer.

Barua, R. y Misra, J. (2003). Binary arithmetic for dna computers. En: M. Hagiya y A. Ohuchi (eds.), *DNA Computing*, Vol. 2568 de *Lecture Notes in Computer Science*, pp. 124–132. Springer. ISBN 978-3-540-00531-5.

Batsyn, M., Goldengorin, B., Maslov, E., y Pardalos, P. M. (2013). Improvements to MCS algorithm for the maximum clique problem. *Journal of Combinatorial Optimization*, **27**(2): 397–416.

Bomze, I. M., Budinich, M., Pardalos, P. M., y Pelillo, M. (1999). The maximum clique problem. En: *Handbook of Combinatorial Optimization*, Vol. 4, pp. 1–74. Kluwer Academic.

Braich, R. S., Chelyapov, N., Johnson, C., Rothemund, P. W. K., y Adleman, L. (2002). Solution of a 20-variable 3-SAT problem on a DNA computer. *Science*, **296**(5567): 499–502.

Carraghan, R. y Pardalos, P. M. (1990). An exact algorithm for the maximum clique problem. *Operations Research Letters*, **9**(6): 375–382.

Cook, S. A. (1971). The complexity of theorem-proving procedures. En: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pp. 151–158. ACM Press.

Cormen, T. H., Stein, C., Rivest, R. L., y Leiserson, C. E. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education, segunda edición. ISBN 0070131511.

Darehmiraki, M. (2009). A new solution for maximal clique problem based sticker model. *Biosystems*, **95**(2): 145–149.

Fahle, T. (2002). Simple and fast: Improving a branch-and-bound algorithm for maximum clique. En: R. Mohring y R. Raman (eds.), *Algorithms – ESA 2002*, Vol. 2461 de *Lecture Notes in Computer Science*, pp. 485–498. Springer.

Faulhammer, D., Cukras, A. R., Lipton, R. J., y Landweber, L. F. (2000). Molecular computation: RNA solutions to chess problems. *Proceedings of the National Academy of Sciences of the USA*, **97**(4): 1385–1389.

Feynman, R. P. (1960). There's plenty of room at the bottom. *Engineering and Science*, **23**(5): 22–36.

Garey, M. R. y Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman. ISBN 0716710455.

Grossman, T. y Jagota, A. (1993). On the equivalence of two Hopfield-type networks. En: *IEEE International Conference on Neural Networks*, pp. 1063–1068. IEEE.

Guo, M., Ho, M. S.-H., y Chang, W.-L. (2004). Fast parallel molecular solution to the dominating-set problem on massively parallel bio-computing. *Parallel Computing*, **30**(9-10): 1109–1125.

Guo, P. y Zhang, H. (2009). DNA implementation of arithmetic operations. En: *2009 Fifth International Conference on Natural Computation*, Vol. 6, pp. 153–159. IEEE.

Hastad, J. (1996). Clique is hard to approximate within $n^{1-\epsilon}$. En: *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pp. 627–636. IEEE.

Head, T. (1987). Formal language theory and DNA: an analysis of the generative capacity of specific recombinant behaviors. *Bulletin of Mathematical Biology*, **49**(6): 737–759.

Hifi, M. (1997). A genetic algorithm-based heuristic for solving the weighted maximum independent set and some equivalent problems. *Journal of the Operational Research Society*, **48**(6): 612–622.

Homer, S. y Peinado, M. (1996). Experiments with polynomial-time CLIQUE approximation algorithms on very large graphs. En: *Proc. Cliques, coloring and satisfiability: 2nd DIMACS Implementation Challenge*, pp. 147–167. American Mathematical Society.

Hsieh, S.-Y. y Chen, M.-Y. (2008). A DNA-based solution to the graph isomorphism problem using Adleman-Lipton model with stickers. *Applied Mathematics and Computation*, **197**(2): 672–686.

Ignatova, Z., Martínez-Pérez, I. M., y Zimmermann, K.-H. (2008). *DNA Computing Models*. Springer. ISBN 0387736352, 9780387736358.

Jagota, A., Sanchis, L., y Ganesan, R. (1996). Approximately solving maximum clique using neural network and related heuristics. En: *Proc. 1st DIMACS Implementation Challenge*, pp. 169–204. American Mathematical Society.

Jerrum, M. (1992). Large cliques elude the metropolis process. *Random Struct. Algorithms*, **3**(4): 347–360.

Johnson, D. J. y Trick, M. A., (eds.) (1996). *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. American Mathematical Society. ISBN 0821866095.

Kari, L., Paun, G., Rozenberg, G., Salomaa, A., y Yu, S. (1998). DNA computing, sticker systems, and universality. *Acta Informatica*, **35**(5): 401–420.

Kari, L., Seki, S., y Sosik, P. (2012). *DNA Computing: Foundations and Implications*, Vol. 3 de *Handbook for Natural Computing*, pp. 1073–1127. Springer.

Karp, R. (1972). Reducibility among combinatorial problems. En: R. Miller y J. Thatcher (eds.), *Complexity of Computer Computations*, pp. 85–103. Plenum Press.

Kirk, D. B. y Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan-Kaufmann. ISBN 0123814723, 9780123814722.

Kreher, D. L. y Stinson, D. R. (1999). *Combinatorial Algorithms: generation, enumeration, and search*. CRC Press series on discrete mathematics and its applications. CRC Press. ISBN 084933988.

Lipton, R. (1995). DNA Solution of Hard Computational Problems. *Science*, **268**(5210): 542–545.

Liu, X., Yang, X., Li, S., y Ding, Y. (2010). Solving the minimum bisection problem using a biologically inspired computational model. *Theoretical Computer Science*, **411**(6): 888–896.

Marchiori, E. (1998). A simple heuristic based genetic algorithm for the maximum clique problem. En: *Proc. of the ACM Symposium on Applied Computing*, pp. 366–373. ACM Press.

Martínez-Pérez, I. M. (2007). *Biomolecular computing models for graph problems and finite state automata*. Tesis de doctorado, Hamburg Univ. Tech.

Martínez-Pérez, I. M. y Zimmermann, K.-H. (2009). Parallel bioinspired algorithms for NP complete graph problems. *Journal of Parallel and Distributed Computing*, **69**(3): 221–229.

Martínez-Pérez, I. M., Brandt, W., Wild, M., y Zimmermann, K.-H. (2008). Bioinspired Parallel Algorithms for Maximum Clique Problem on FPGA Architectures. *Journal of Signal Processing Systems*, **58**(2): 117–124.

Nickolls, J., Buck, I., Garland, M., y Skadron, K. (2008). Scalable Parallel Programming with CUDA. *ACM Queue*, **6**(2): 40–53.

NVIDIA Corporation (2013). *NVIDIA CUDA C Programming Guide version 5.5*. Recuperado de: http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

Ouyang, Q., Kaplan, P. D., Liu, S., y Libchaber, A. (1997). DNA Solution of the Maximal Clique Problem. *Science*, **278**(5337): 446–449.

Paun, G. y Rozenberg, G. (1998). Sticker systems. *Theoretical Computer Science*, **204**(1-2): 183–203.

Pérez-Jiménez, M. J. y Sancho-Caparrini, F. (2002). Solving knapsack problems in a sticker based model. En: *Revised Papers from the 7th International Workshop on DNA-Based Computers: DNA Computing*, pp. 161–171. Springer.

Razzazi, M. y Roayaei, M. (2011). Using sticker model of DNA computing to solve domatic partition, kernel and induced path problems. *Information Sciences*, **181**(17): 3581–3600.

Roweis, S., Winfree, E., Burgoyne, R., Chelyapov, N. V., Goodman, M. F., Rothemund, P. W. K., y Adleman, L. M. (1998). A sticker based model for DNA computation. *Journal of Computational Biology*, **5**(4): 615–629.

Sakamoto, K., Gouzu, H., Komiya, K., Kiga, D., Yokoyama, S., Yokomori, T., y Hagiya, M. (2000). Molecular Computation by DNA Hairpin Formation. *Science*, **288**(5469): 1223–1226.

Sanders, J. y Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley. ISBN 0131387685, 9780131387683.

Smith, L., Corn, R., Condon, A., Lagally, M., Frutos, A., Liu, Q., y Thiel, A. (1998). A surface-based approach to DNA computation. *Journal of Computational Biology*, **5**(2): 255–267.

Soriano, P. y Gendreau, M. (1996). Tabu search algorithms for the maximum clique problem. En: *Proc. Cliques, coloring and satisfiability: 2nd DIMACS Implementation Challenge.*, pp. 221–242. American Mathematical Society.

Tomita, E. y Kameda, T. (2007). An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global Optimization*, **37**(1): 95–111.

Tomita, E. y Seki, T. (2003). An efficient branch-and-bound algorithm for finding a maximum clique. En: C. Calude, M. Dinneen, y V. Vajnovszki (eds.), *Discrete Mathematics and Theoretical Computer Science*, Vol. 2731 de *Lecture Notes in Computer Science*, pp. 278–289. Springer.

Tomita, E., Sutani, Y., Higashi, T., Takahashi, S., y Wakatsuki, M. (2010). A simple and faster branch-and-bound algorithm for finding a maximum clique. En: M. Rahman y S. Fujita (eds.), *WALCOM: Algorithms and Computation*, Vol. 5942 de *Lecture Notes in Computer Science*, pp. 191–203. Springer. ISBN 978-3-642-11439-7.

Watson, J. D. y Crick, F. H. (1953). Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, **171**(4356): 737–738.

Winfree, E., Liu, F., Wenzler, L. A., y Seeman, N. A. (1998). Design and self-assembly of two-dimensional DNA crystals. *Nature*, **394**(6693): 539–544.

Xu, J., Dong, Y., y Wei, X. (2004a). Sticker DNA computer model-part I: Theory. *Chinese Science Bulletin*, **49**(8): 772–780.

Xu, J., Li, S., Dong, Y., y Wei, X. (2004b). Sticker DNA computer model-Part II: Application. *Chinese Science Bulletin*, **49**(9): 863–871.

Yang, C.-N. y Yang, C.-B. (2005). A DNA solution of SAT problem by a modified sticker model. *Bio Systems*, **81**(1): 1–9.

Yang, Y.-x., Wang, A.-m., y Ma, J.-l. (2009). A DNA computing algorithm of addition arithmetic. En: *2009 First International Workshop on Education Technology and Computer Science*, Vol. 1, pp. 1056–1059. IEEE.

Zimmermann, K.-H. (2002). Efficient DNA sticker algorithms for NP-complete graph problems. *Computer Physics Communications*, **144**(3): 297–309.