VNIVERSIDAD NACIONAL
AVTNOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

RECURSIÓN EN ALGORITMOS DISTRIBUIDOS

TESIS
QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN CIENCIAS (COMPUTACIÓN)

PRESENTA:
JUAN CARLOS ONOFRE ROSAS

TUTORES PRINCIPALES:
DR. SERGIO RAJSBAUM GORODEZKY
INSTITUTO DE MATEMÁTICAS

MÉXICO, D. F. JUNIO 2014.

Agradecimientos

A mis padres Clara y Heriberto que con su apoyo contribuyeron a que concluyera mis estudios de maestría.

Al Dr. Sergio Rajsbaum por su apoyo y consejos para la realización de este trabajo.

Al los doctores Carlos Velarde, Francisco Hernández, Carlos Gershenson y David Flores por el tiempo dedicado a la revisión del trabajo y por las anotaciones realizadas que contribuyeron a mejorarlo.

A CONACYT por el apoyo económico brindado que contribuyó a realizar mis estudios de maestría.

Parte de este trabajo fue realizado en el marco del proyecto de investigación PAPIIT-IN104711.

Índice general

Abstract

En esta tesis se busca relacionar el cómputo distribuido, en particular algoritmos recursivos, con herramientas como la topología combinatoria. Tratar de dar un enfoque topológico a estos algoritmos es de gran utilidad, nos ayuda a analizarlos de una manera combinatoria y si los procesos que participan en los algoritmos son pocos, los podemos representar de manera gráfica fácilmente.

Primero veremos algunos conceptos básicos de cómputo distribuido así como de topología. Es de gran importancia debido a que utilizaremos palabras como tareas (*tasks*), entradas, identificadores, vistas (*view*), mapeo simplicial, recursión lineal, recursión ramificada. En la parte de topología utilizaremos palabras como espacio topológico, simplejo, complejo simplicial, vértices, etiquetas cromáticas, subdivisiones, unión (*join*) entre simplejos y eliminación de simplejos.

Después estudiaremos algoritmos distribuidos recursivos para resolver tareas como: foto inmediata (*immediate snapshot*), renombramiento (*renaming*) e intercambio (*swap*). Así mismo veremos si el árbol de recursión que se genera es simplemente una trayectoria o un árbol general. Además veremos las ventajas que tienen éstos algoritmos para demostrar correctez.

Daremos un algoritmo secuencial para construir de manera recursiva subdivisiones cromáticas de complejos simpliciales, en adelante referiremos únicamente como complejos.

En la parte de algoritmos distribuidos recursivos se observará que resolver el problema del *immediate snapshot* es prácticamente una implementación distribuida del algoritmo secuencial que construye subdivisiones cromáticas.

Finalmente mostraremos la utilidad de utilizar un enfoque topológico para describir un modelo y ver por qué ciertas tareas no se pueden resolver con determinado modelo.

Capítulo 1

Introducción

En este trabajo se estudian algoritmos recursivos, sus características en un ambiente distribuido y se les da un enfoque topológico para poder estudiarlos de manera sencilla. En este capítulo se muestra el interés por realizar dicha investigación y las bases necesarias para poder cumplir con el objetivo.

1.1. Motivación

En esta tesis se estudian las características de los algoritmos distribuidos recursivos, principalmente se abordan temas de recursión y se les da un enfoque topológico. Es interesante la representación topológica de los algoritmos distribuidos, porque ayuda a visualizar los diferentes escenarios que pueden darse durante la ejecución de un algoritmo distribuido.

Por otra parte el observar cómo es que los algoritmos recursivos distribuidos y la topología tienen una relación natural, es de gran interés. Por un lado podemos estudiar topología y dar algoritmos para construir complejos simpliciales, y por otro podemos ver cómo esas construcciones pueden derivar en algoritmos especializados en resolver una tarea en particular, como en el caso de la subdivisión cromática de un complejo simplicial y su mapeo de una manera transparente a un algoritmo que resuelve el *immediate snapshot*.

También la motivación de realizar este trabajo se debe, a que los algoritmos recursivos distribuidos han sido poco estudiados, existen artículos como [9] que mencionan las propiedades recursivas de los complejos simpliciales pero no profundizan mucho al respecto, por lo cual se cree que es importante aventurarse y ver las ventajas y desventajas de la recursividad en los algorit-

mos distribuidos, así como investigar las herramientas que puedan surgir de su estudio.

Este trabajo parte de un artículo publicado por Gafni y Rajsbaum quienes en [12], muestran las ventajas de plantear soluciones recursivas a diferentes tareas del cómputo distribuido y dan algunas definiciones sobre conceptos que surgen en los algoritmos distribuidos recursivos.

1.2. Objetivos

- Estudiar las propiedades recursivas de los complejos simpliciales.
- Estudiar tareas que podemos definir como cromáticas.
- Mostrar la relación entre los algoritmos que resuelven una tarea y los algoritmos que construyen complejos simpliciales.
- Dar herramientas para el estudio de las relaciones encontradas entre la topología y los algoritmos distribuidos recursivos.

1.3. Trabajos Relacionados

El área de los algoritmos distribuidos recursivos ha sido poco explorada, en particular la recursividad es poco estudiada, siendo que hay quienes como Stojmenovic I. en [28], recomiendan que debería formar parte de los conocimientos que se adquieren durante los primeros años de estudio en computación. En [28] se muestran las ventajas y desventajas de la recursividad secuencial, así como lo ineficiente que puede ser la recursividad secuencial cuando se ramifica (como al calcular los números de Fibonacci o los coeficientes binomiales). Cuando el árbol de recursión es lineal, sólo una trayectoria, posiblemente usar recursión resulte tan eficiente como hacerlo de manera iterativa (como en el cálculo del factorial).

En [12] se exploran algunos algoritmos, como el *immediate snapshot*, el *renaming* y finalmente el *swap*, todos éstos distribuidos y recursivos, y las ventajas que tienen en cuanto a complejidad en tiempo. En ese artículo se demuestra que resolver cada uno de dichos problemas de manera recursiva toma un tiempo de $O(n^2)$, incluso cuando la recursión es ramificada se logran buenos resultados, como lo veremos en el caso de la tarea de renombramiento (*renaming*).

En [7] se plantea un algoritmo recursivo síncrono, que resuelve la tarea del acuerdo enrejado (*lattice agreement*), para posteriormente dar un algoritmo asíncrono que resuelve la tarea. También los autores muestran que un algoritmo que resuelve la tarea del acuerdo enrejado puede ser transformado en una implementación para resolver la tarea *snapshot*. La tarea de *lattice agreement* consiste en que cada proceso inicia con un valor de entrada y durante la ejecución debe ir aprendiendo valores tal que el valor de salida pertenezca a la *lattice*. La meta es lograr que cada valor aprendido por el proceso sea mayor o igual que su valor de entrada, cada valor aprendido es la unión de un conjunto de valores de entrada.

Por otra parte, existen trabajos que muestran una manera de manipular los complejos simpliciales, así como las subdivisiones que de éstos podemos obtener, por ejemplo en [21] Kozlov muestra una forma de etiquetar todos los k-simplejos de un complejo simplicial cromático Δ^n, con $0 \leq k \leq n$, donde n es la dimensión de Δ. Etiquetar los simplejos y en particular los vértices, es fundamental pues ayuda a apegarse a una notación establecida. También, Kozlov hace una interpretación de estas etiquetas en cómputo distribuido, considerando que cada vértice es un proceso y que las etiquetas son de la forma (n, V_n), donde n representa el identificador del proceso y V_n representa la vista de un proceso en alguna ejecución.

Sobre el planteamiento de las tareas, existen artículos donde se definen formalmente, así como los algoritmos iterativos para resolverlas, como es el caso para: *snapshots* [1], *immediate snapshots* [8, 26], *renaming* [6], y *swap* [3, 29]. Estas tareas son importantes ya que se estudiarán en esta tesis, mostrando el análisis para resolverlas de manera recursiva y buscando su relación con la topología, por otro lado se verá que tomando la tarea del *immediate snapshot* y considerándola como parte de un modelo, se pueden resolver otras tareas.

En [14] se muestra un algoritmo recursivo para resolver el acuerdo aproximado (*approximate agreement*). Primero plantean un modelo iterado de escritura y foto instantánea (*iterated write-snapshot*), donde los procesos se comunican por rondas a través de una memoria compartida. En particular la memoria utilizada es un arreglo, dicho arreglo provee la operación *snapshot*, tal que regresa el valor actual del arreglo completo. Un objeto *snapshot* es linearizable [19], es decir cada operación de escritura y snapshot parece realizarse de manera instantánea desde que se invoca hasta que regresa. Desde el planteamiento del modelo, se utiliza un algoritmo por rondas y en cada ronda se utiliza un nuevo arreglo compartido a partir del cual se obtienen

los *snapshot*, debido a que cada ronda utiliza un nuevo arreglo de memoria compartida y la utiliza como entrada para la siguiente ronda. Los autores llevan a un razonamiento recursivo del modelo **IWS**, *iterated write-snapshot*, y finalmente al planteamiento de un algoritmo recursivo para el acuerdo aproximado. También se dan definiciones sobre las tareas no cromáticas *(colorless tasks)*, donde no interesa que proceso escribió que valor, sólo nos interesan los valores de entrada escritos en la memoria para poder tomar decisiones. En [15] se da la descripción del modelo iterativo y del modelo recursivo del *snapshot* además, se estudia una técnica para la detección de fallas en estos modelos. Hay artículos como [11] que muestran la equivalencia entre modelos de cómputo, como lo son el modelo *snapshots* y el modelo *iterated snapshots*.

En [25] se describen de manera didáctica y con mayor profundidad las características del *snapshot*, *immediate snapshot*, *renaming* y *swap*, además se retoman y explican de manera alternativa los algoritmos presentados por Gafni y Rajsbaum en [12].

En [13] se estudia una amplia gama de algoritmos distribuidos, así como temas de imposibilidad, por ejemplo se explica detalladamente por qué el problema de los generales bizantinos es imposible resolverlo cuando se presenta una falla, la explicación se plantea a través de complejos simpliciales, como lo que se estudiará en esta tesis. También aborda temas como los mapeos simpliciales, para poder ver si una tarea se puede resolver con un modelo que genere un determinado protocolo, y si este protocolo respeta la especificación de la tarea y cubre la representación de ésta en el complejo de salida por medio de un mapeo simplicial, que en particular se llama mapeo de decisión, esto es interesante puesto que este tipo de mapeos los utilizaremos para lograr uno de nuestros resultados.

1.4. Resultados

Se ha encontrado en el estudio por separado de complejos simpliciales y algoritmos distribuidos recursivos, que tienen entre ellos una relación muy natural, un ejemplo claro es el algoritmo recursivo para el *immediate snapshot*, ya que a partir de él, se puede subdividir de manera cromática cualquier complejo Δ^n, y si se compara con un algoritmo secuencial especializado en construir la subdivisión cromática de Δ^n, se observa que básicamente es el mismo algoritmo. Además se da una definición de una función que dados dos complejos, tal que sus vértices están etiquetados de la forma (n, V_n), realiza

la unión cromática de éstos, permitiendo seguir la secuencia en la que los procesos se ejecutaron de acuerdo al algoritmo, es decir, se puede considerar el nuevo complejo como una representación de todos los posibles mundos que pueden surgir al ejecutar el algoritmo.

Por otra parte, considerando la subdivisión cromática inducida por el *immediate snapshot*, se puede analizar la forma de resolver otras tareas a partir de dicha subdivisión, con lo cual, se obtienen nuevas subdivisiones cromáticas, como la subdivisión inducida por el algoritmo que resuelve el problema del renombramiento. Lo interesante de este resultado es que a pesar de que el algoritmo del renombramiento utiliza llamadas recursivas etiquetadas, el análisis en la parte topológica suele ser sencillo y prácticamente el mismo que para algoritmos que no usan llamadas recursivas etiquetadas.

Finalmente se muestra el complejo del problema del intercambio, que se puede obtener a través de un razonamiento recursivo, en el cual ésta enfocando el trabajo. El resultado nos ayuda a mostrar que este problema no se puede resolver únicamente con registros de lectura y escritura, sino que requiere registros más poderosos para resolverlo. Además se observa como los registros 2-consenso rompen la simetría entre ejecuciones, y en particular para ejecuciones en donde participan dos procesos se forma un complejo con dos componentes que no están conectadas por trayectoria.

1.5. Organización del trabajo

En el capítulo 2 se tratan temas básicos sobre cómputo distribuido y topología combinatoria, estos sirven para familiarizarnos con algunos conceptos como: recursión, recursión lineal, recursión ramificada, unión de simplejos, eliminación de simplejos, simplejos y complejos simpliciales.

En el capítulo 3 se describen algunos algoritmos recursivos, básicamente tomados de [12], *immediate snapshot*, *renaming* y *swap*, para posteriormente en el capítulo 4 estudiar su relación con los complejos simpliciales, mediante la construcción de los complejos, sus estructuras y la forma en que nos pueden ayudar para estudiar la correctez y la complejidad de los algoritmos recursivos distribuidos.

Finalmente en el capítulo 5 se dan las conclusiones obtenidas de esta tesis, así como el trabajo a futuro que se puede realizar, considerando algunos de los resultados obtenidos.

Capítulo 2

Preliminares

En esta sección se hablará sobre conceptos básicos del cómputo distribuido y de topología combinatoria, principalmente los utilizados en el presente trabajo para cumplir con los objetivos. En la parte de cómputo distribuido se puede profundizar en los conceptos consultando los libros [16, 25] y los conceptos de topología matemática se pueden profundizar en [20, 24]. Se describen conceptos básicos como tareas (*tasks*), entradas y salidas distribuidas, identificadores, modelo libre de espera (*wait-free*), recursividad secuencial, recursividad distribuida.

En la parte de topología, se tiene definición de simplejo, complejo simplicial, unión entre simplejos, eliminación de simplejos, complejos cromáticos y definición de subdivisiones.

2.1. Cómputo distribuido

En cómputo secuencial se tiene la noción de función computable. Una función f es computable si existe una máquina de Turing tal que dada x como entrada, produzca una salida $f(x)$.

En cómputo distribuido la noción de computabilidad tiene que ver con las tareas que se pueden resolver. Estamos interesados en hacer el cómputo de *tareas*. Una tarea la se define como una tripleta $\langle \mathcal{I}, \mathcal{O}, \Delta \rangle$, donde $\mathcal{I}$ define las posibles entradas de la tarea, $\mathcal{O}$ define las posibles salidas de la tarea y Δ define una relación entre $\mathcal{I}$ y $\mathcal{O}$. Nos interesan las tareas donde la entrada $I \in \mathcal{I}$ y la salida $O \in \mathcal{O}$ son distribuidas, es decir, cada proceso que participa en la solución de la tarea obtiene parte de la entrada I, hace el cómputo

requerido y da una parte de la salida O. La entrada I está compuesta por pares (p,v), esto significa que la entrada del proceso p es v. Cuando p conoce parte de la entrada, v, no conoce la entrada de los otros procesos y esto se debe a que la entrada (p,v) puede ocurrir en otra entrada global I'. Si los procesos inician con entrada I y se comunican, entonces cada proceso produce la salida v_p, tal que el conjunto de pares (p, v_p) forma una salida O, tal que $O \in \Delta(I)$.

Las tareas que son consideradas más importantes son el *consenso* [10], donde cada proceso propone un valor de un conjunto dado, y deben de acordar todos un sólo valor de los propuestos, una tarea más general es el *k-set agreement*, donde pueden decidir a lo más k diferentes valores, en este mismo artículo [10] se demostró que es imposible resolver este problema en presencia de fallas.

Un problema fundamental en el cómputo distribuido es la caracterización de las tareas computables, lo cual es muy complejo debido a que una tarea puede ser computable en un modelo pero no en otro. Un modelo en cómputo distribuido está definido por el número de procesos que puede fallar, cómo ellos pueden fallar, si la velocidad de los procesos es homogénea o no, los retrasos que pueden haber en la comunicación y cómo es que se comunican. Esto resulta en estudiar las tareas que se quiere saber si son computables en un modelo básico y deducir los resultados en otro modelo por simulaciones o reducciones.

Los algoritmos que se tratarán en esta tesis, serán sobre un modelo de cómputo distribuido de memoria compartida con procesos tomados de un conjunto $\Pi = \{p_0, p_1, \ldots, p_n\}$, en una configuración libre de espera (*wait-free*) donde cualquier número de procesos puede fallar.

También se considerará que los procesos se comunican por medio de objetos compartidos que pueden ser invocados a lo más una vez por cada proceso. La especificación de la relación entrada/salida de un objeto está en términos de una tarea (*task*), definida por un conjunto de vectores de posibles entradas y un conjunto de posibles vectores de salida relacionados por medio de Δ, es decir, $\Delta(I) \in \mathcal{O}$.

2.1.1. Recursividad Secuencial

La recursividad se puede definir como la capacidad que una máquina de Turing T tiene para obtener una descripción de ella misma $< T >$ y realizar el cómputo de ésta con entrada $< w >$. Con esta misma idea, los algoritmos

recursivos aprovechan su propia definición para realizar el cómputo de una función. En cómputo secuencial, las características de la recursividad son bien conocidas, así como su implementación.

En algunas técnicas para el diseño de algoritmos, como lo es la de programación dinámica, es muy recurrente utilizar recursividad, pues esta técnica trata de resolver un problema a partir de subproblemas del mismo más sencillos. Es decir, si resolvemos primero un subproblema, podemos utilizar el resultado para resolver el problema más grande.

Por otra parte en cómputo secuencial se puede tener una recursividad que se desenvuelva de manera lineal, la podríamos representar mediante una trayectoria, por ejemplo, encontrar el factorial de un número. También se puede tener una recursividad ramificada, como lo es la construcción de la serie de Fibonacci.

Como se menciona en [28], a pesar de que las características y las técnicas recursivas secuenciales son conocidas y utilizadas, no siempre se profundiza en lo ineficiente que puede llegar a ser, por ejemplo: en el cálculo de la serie de Fibonacci, siempre se hacen cálculos repetidos y ésto es más costoso cuando calculamos números muy grandes de la serie, a diferencia de resolverlo con un algoritmo iterativo, utilizando una matriz, que con sólo calcular una vez el n-ésimo término, cuando sea requerido, sólo se direcciona en la memoria y se obtiene el valor, sin necesidad de volverlo a calcular.

También en cómputo secuencial se sabe que cualquier algoritmo recursivo puede ser transformado a un algoritmo iterativo y viceversa utilizando algunas estructuras de datos como lo son las pilas, esto no se sabe en cómputo distribuido, y es una de las preguntas abiertas en el área.

2.1.2. Recursividad Distribuida

En [12] se discuten diferentes algoritmos recursivos para resolver tareas importantes como: *snapshots* [1], *immediate snapshots* [8, 26], *renaming* [6], y *swap* [3, 29]. Cuando se habla de recursividad en cómputo distribuido, surgen algunos temas interesantes que originalmente no se encuentran en la recursividad secuencial, descritos en [12], como son:

a) *Nombrado de llamadas recursivas*: Las llamadas recursivas deben nombrarse para que los procesos identifiquen la llamada que están ejecutando, por ejemplo, en una búsqueda binaria secuencial, la llamada que se realiza es la misma ya sea con la parte izquierda o con la parte derecha del

arreglo, siendo ejecutada una sola, no ambas; por otra parte en cómputo distribuido se tienen ejecuciones concurrentes, por lo cual se pueden ejecutar ambas llamadas.

b) *Ramificación concurrente*: En cómputo secuencial, las funciones recursivas pueden dividirse en lineales, es decir, que desarrollan una sola rama la cual es una trayectoria, por ejemplo al calcular el factorial de un número, o pueden ramificarse, por ejemplo calcular el n-ésimo término de la serie de Fibonacci. En cómputo distribuido pueden presentarse de la misma forma ambos tipos de ramificaciones, ya que como se mencionó anteriormente, es posible ejecutar llamadas recursivas de manera concurrente.

c) *Memoria iterada*: En las llamadas recursivas secuenciales, cada llamada tiene su propio espacio de memoria. En algoritmos distribuidos, cada proceso tiene memoria local y puede tener acceso a memoria compartida, ambos tipos de memoria deben ser locales a la llamada recursiva, por lo que cada llamada recursiva tiene su propio espacio de memoria compartida. No hay efectos laterales en el sentido de que una invocación no puede tener acceso a la memoria compartida de otra invocación. Es por esta razón que los algoritmos distribuidos recursivos, corren sobre un modelo de cómputo iterado, donde la memoria iterada está dividida en secciones.

d) *Objetos compartidos*: El caso más simple de memoria compartida que es accedida en cada iteración es un arreglo compartido Un-Escritor/Múltiples-Lectores *(Single-Writer/Multi-Reader)*. Es decir, el algoritmo distribuido recursivo escribe en un sólo lugar del arreglo, lee cada sección del arreglo y después realiza algún cómputo local para producir una salida o volver a invocar recursivamente el algoritmo. En general los procesos se comunican por medio de un objeto compartido, más poderoso que un arreglo compartido SW/MR. Cada objeto es invocado a lo más una vez por cada proceso. El comportamiento de cada objeto lo especificamos como una tarea (*task*), esencialmente su relación de entrada y salida.

e) *Razonamiento inductivo*: Como no hay efectos laterales entre llamadas recursivas, nosotros podemos imaginar que los procesos van al mismo paso de un objeto compartido al siguiente, sólo variando el orden en el que los procesos invocan a los objetos. Este orden induce un conjunto estructurado de ejecuciones que facilitan un razonamiento inductivo y simplifica el entendimiento de algoritmos distribuidos.

Cada uno de estos pequeños detalles se observará en el análisis de los algoritmos que se aborden en esta tesis.

Tratar de dar un enfoque topológico a estos algoritmos recursivos, creemos que puede llevar a saber si cualquier algoritmo distribuido recursivo puede transformarse en un algoritmo distribuido iterativo, debido a que ambos tipos de algoritmos inducen complejos simpliciales.

2.1.3. Propiedades de exactitud y progreso

En [18] se puede encontrar un tema sobre las propiedades de exactitud y progreso en un sistema distribuido.

Para las propiedades de exactitud, se referirá a $p_i \to p_j$ como p_i precede a p_j, para poder establecer un orden de alguna ejecución. En los diagramas las llamadas a métodos sobre la misma línea representan que fueron ejecutadas por un mismo proceso.

Por el lado de las propiedades de exactitud tenemos la propiedad de **Consistencia de quietud** (*Quiescent consitency*), introducida en [5] y detallada en [27]. Esta propiedad la definen dos principios.

Principio 1 (Principio 3.3.1 de [18]) *Las llamadas a métodos deben parecer que ocurren una a la vez en un orden secuencial.*

Principio 2 (Principio 3.3.2 de [18]) *Las llamadas a métodos separadas por un periodo de quietud deben parecer que toman efecto en su orden de tiempo real.*

Por ejemplo en la Figura 2.1 se muestra un objeto que representa una cola. El periodo de quietud se representa por la línea vertical. Es decir, del lado izquierdo de la línea vertical se tiene q.enq(x) y q.enq(y) estas llamadas son concurrentes por lo que es válido el orden $q.enq(x) \to q.enq(y)$ ó $q.enq(y) \to q.enq(x)$. De lado derecho de la línea también existen dos posibles ordenamientos en el que $q.deq(x) \to q.deq(y)$ ó $q.deq(y) \to q.deq(x)$. Por el orden de tiempo real sabemos que no es posible que $q.deq(y) \to q.enq(x)$, y un posible orden respetando la definición de la cola es: $q.enq(x) \to q.enq(y) \to q.deq(x) \to q.deq(y)$.

La siguiente propiedad se conoce como **consistencia secuencial** (*sequential consistency*) introducida en [22]. Esta propiedad ayuda a verificar la correcta ejecución de un algoritmo distribuido, por lo que se basa en el siguiente principio:

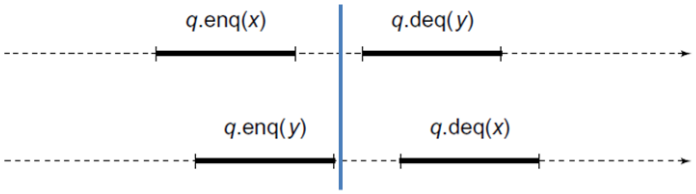


Figura 2.1: Se muestra como después de un periodo de quietud se pueden tener dos posibles ordenamientos en las llamadas, $q.enq(x) \rightarrow q.enq(y) \rightarrow q.deq(x) \rightarrow q.deq(y)$ ó $q.enq(y) \rightarrow q.enq(x) \rightarrow q.deq(y) \rightarrow q.enq(x)$, por lo que el objeto q cumple con la propiedad de quietud.

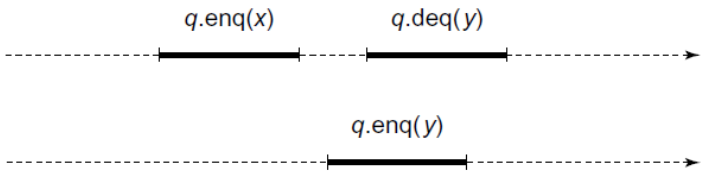


Figura 2.2: En esta ejecución aparentemente se viola la especificación de un objeto FIFO, pero la realidad es que se puede reordenar la llamada $q.enq(y)$ para que se cumpla con la especificación, ya que ésta es ejecutada por un hilo diferente por lo que no se afecta el orden de programa.

Principio 3 (Principio 3.4.1 de [18]) *Las llamadas a métodos deben parecer que toman efecto en orden de programa.*

El orden en el cual un único proceso realiza las llamadas a métodos se le llama orden de programa. Entonces para que un algoritmo sea consistente secuencialmente, las llamadas a métodos realizadas por cada proceso deben parecer que toman efecto respetando el orden de programa. Existe un orden de programa por cada proceso. Por ejemplo, la Figura 2.2 muestra una ejecución de un objeto FIFO, el cual es consistente secuencialmente.

Para la siguiente propiedad de exactitud daremos la definición de una llamada a un método tomada de [18].

Definición 1 *Una llamada a un método en una historia H es un par de eventos que consisten en una invocación que esta relacionada con una respuesta en la historia H.*

Un sistema concurrente puede ser modelado a través de una historia de su ejecución. Básicamente si cada objeto del sistema cumple con su especificación secuencial es linearizable, y si cada objeto es linearizable entonces el sistema es linearizable.

Entre los eventos de invocación y respuesta existe un tiempo que es el que consideraremos de ejecución de la llamada, en un sistema concurrente las llamadas se pueden traslapar por lo que nos apoyaremos de estos eventos para poder determinar un orden en la ejecución de la llamada del método.

El concepto de **linearización** (*linearizability*) introducido en [19], está basado en el siguiente principio.

Principio 4 (Principio 3.5.1 de [18]) *Cada llamada a un método debe parecer que toma efecto instantáneamente en algún momento entre su invocación y su respuesta.*

El principio nos dice que el orden de tiempo real debe preservarse. La manera de demostrar que un objeto concurrente cumple la propiedad, es identificar para cada método un punto de linearización, es decir, justo donde la llamada toma efecto. Por ejemplo, para métodos basados en bloqueos cada sección crítica puede funcionar como un punto de linearización.

Informalmente podemos decir que una ejecución concurrente cumple la propiedad de linearización si es equivalente a una ejecución secuencial válida. En general las tareas pueden tener una especificación secuencial, es decir, definen un comportamiento definido en caso de que las llamadas a los métodos del objeto no se traslapen.

Así, cada sistema que tiene una historia H puede tener una historia H' linearizable, si para cada objeto x de H, la subhistoria $H|x$ es linearizable.

Teorema 1 (Teorema 1 de [19]) *H es linearizable si y sólo si, para cada objeto x, $H|x$ es linearizable*

En las propiedades de progreso tenemos las de **bloqueo** y las de **no bloqueo**. En esta tesis se tratará la propiedad de no bloqueo libre de espera definida de la siguiente manera.

Definición 2 *Un método es libre de espera si garantiza que cada llamada termina su ejecución en un número finito de pasos.*

Básicamente esta propiedad nos garantiza que cualquier proceso termina su ejecución pese a un inesperado retraso de algún otro proceso, incluso si algún proceso falla.

2.2. Complejos simpliciales

Existen varias formas de representar los espacios topológicos, una de ellas es mediante la unión de simplejos de una forma estructurada. En esta tesis se usarán los complejos simpliciales como representación de espacios topológicos y posteriormente se relacionarán con representaciones de ejecuciones de algoritmos distribuidos recursivos. Se pueden representar dichos complejos de dos formas, una será combinatoria y la otra gráfica, con el fin de visualizar lo que ocurre en las ejecuciones de los algoritmos distribuidos recursivos. Gran parte de la teoría aquí contenida, es tomada de [24, 20] donde se puede encontrar de manera más detallada los conceptos que aquí se presentan.

Definición 3 *Un $k-$simplejo, σ^k, es la envolvente convexa* (convex hull) *de $k+1$ puntos independientes en sentido afín, $\sigma^k = \text{conv}\{u_0, u_1, u_2, ..., u_k\}$.*

Por ejemplo: en la Figura 2.3 se muestra del lado izquierdo, un conjunto de puntos $\{a_0, a_1, a_2, a_3\}$ y la envolvente convexa se muestra del lado derecho, decimos que la envolvente es convexa porque si unimos con una línea recta cualesquiera 2 puntos, esta línea se encuentra dentro de la envolvente.

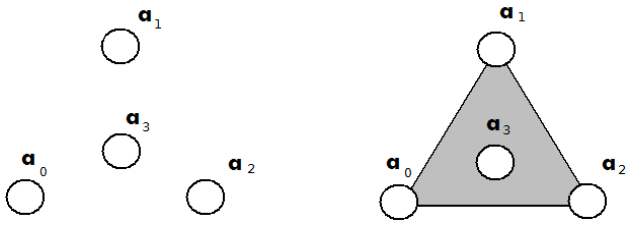


Figura 2.3: De lado izquierdo se muestra un conjunto de puntos y del lado derecho se muestra su envolvente convexa.

Ahora, ¿qué significa que los puntos sean independientes en sentido afín?, el conjunto de puntos mostrados en la Figura 2.3 no son independientes en sentido afín, pues a_3 se puede representar como una combinación afín de los otros 3 puntos.

Por ejemplo, supongamos que el conjunto de puntos independientes en sentido afín es $\{a_0, a_1, a_2\}$ y que están en $\mathbb{R}^2$, si se representa cada punto de acuerdo a su posición en la Figura 2.3 como: $a_0 = (1, 1)$, $a_1 = (2, 3)$, $a_2 = (3, 1)$, $a_3 = (2, 2)$, cualquier punto en la envolvente convexa está dado

por:

$$x = \sum_{i=0}^{n} t_i a_i \ , \quad 0 \leq t_i \leq 1 \tag{2.1}$$

Y por ser una combinación afín $\sum t_i = 1$. Ahora, si se asignan los siguientes valores para cada una de las t_i se tendrá a_3.

$$t_0 = 0.25, t_1 = 0.5, t_2 = 0.25, \tag{2.2}$$

Entonces de acuerdo con la ecuación 2.1 tenemos que:

$$\begin{aligned} x &= (0.25)(1,1) + (0.5)(2,3) + (0.25)(3,1) \\ &= (0.25, 0.25) + (1, 1.5) + (0.75, 0.25) = (2,2) \end{aligned} \tag{2.3}$$

Ahora se puede ver que $x = a_3$, por lo que a_0, a_1 *y* a_2 son los elementos del conjunto de puntos independientes en sentido afín y por lo tanto son los puntos que definen un $2-$*simplejo* como se muestra en la Figura 2.3.

Por simplicidad comúnmente se nombra a los simplejos de dimensión chica de la siguiente manera: *vértice* para el $0-$*simplejo*, *arista* para el $1-$*simplejo*, triángulo para el $2-$*simplejo* y *tetraedro* para el $3-$*simplejo*. Cualquier subconjunto de un conjunto de puntos independientes en sentido afín es independiente en sentido afín por lo tanto también define un simplejo.

Una *cara* de un simplejo σ es la envolvente convexa de un subconjunto de puntos y es propia si no es el conjunto entero. Se denota $\tau \leq \sigma$ si τ es una cara de sigma y $\tau < \sigma$ si τ es una cara propia de σ.

Un k simplejo tiene 2^{k+1} caras, como se muestra en la Figura 2.4 para un 2-simplejo, en esta tesis se considera la cara vacía, la dimensión del simplejo vacío es -1, siguiendo las definiciones encontradas en [21] y utilizadas en esta investigación, ya que posteriormente se hará uso de este elemento para construir los complejos simpliciales de nuestro interés.

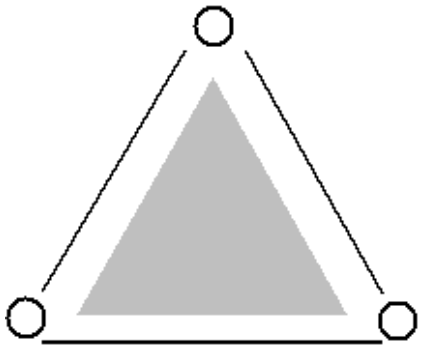


Figura 2.4: Un 2-simplejo tiene $2^{k+1} = 2^{2+1} = 2^3$ caras, en total 8, como se observa se tienen 3 que son vértices, 3 que son aristas, 1 que es el triángulo interior y la cara vacía.

El límite de σ, bd σ, es la unión de todas las caras propias de σ, y el interior es int $\sigma = \sigma -$ bd σ. La dimensión de un simplejo es dim $\sigma = |\sigma| - 1$

2.2.1. Complejo Simplicial

Definición 4 *Un complejo simplicial abstracto es una colección finita de conjuntos A tales que $\alpha \in A$ y $\beta \subseteq \alpha \Rightarrow \beta \in A$.*

Esta forma de ver los complejos es puramente combinatoria. Los conjuntos en A son simplejos.

Los complejos también tienen dimensión, la dimensión del complejo simplicial K es la máxima dimensión de cualquiera de sus simplejos.

Definición 5 *Un subcomplejo L de un complejo simplicial K es un subconjunto $L \subseteq K$, tal que en si mismo L es un complejo simplicial.*

El subcomplejo llamado $j-esqueleto$ consiste en todos los simplejos de dimensión j o menor: $Skel^j(K) = \{\sigma \in K | \dim \sigma \leq j\}$. El $0-esqueleto$ es el conjunto de vértices de K, es decir, $V(K) = Skel^0(K)$.

Un complejo simplicial abstracto también es conocido como sistema independiente (*independence system*), sistema de conjuntos hereditarios (*hereditary set system*) y sistema de conjuntos cerrado hacia abajo (*downward-closed set system*). Fueron propuestos como modelos de espacios topológicos por Pavel Alexandrov [4].

Para poder dar un ejemplo de un complejo simplicial abstracto primero se dará la definición de un espacio topológico, tomada de [23].

Definición 6 *Un espacio topológico es una pareja (X, O), donde X es un conjunto, y $O \subseteq 2^X$ es un sistema de conjuntos (set system), cuyos miembros son llamados conjuntos abiertos (open sets), tal que $\emptyset \in O$, $X \in O$, y la intersección finita de varios conjuntos abiertos es un conjunto abierto y la unión arbitraria de conjuntos abiertos es conjunto abierto.*

Como se mencionó anteriormente, los simplejos son una representación de espacio topológico, por lo que un complejo simplicial abstracto lo podemos ejemplificar de la siguiente manera:

Sea $X = \{a_0, a_1, a_2\}$ un conjunto de puntos independientes en sentido afín, y sea $O \subseteq 2^X$, entonces, como se ejemplificó anteriormente estos puntos definen un simplejo, a partir del cual se puede definir un complejo simplicial abstracto O y lo escribimos de la forma siguiente:

$$O = \{\{\}, \{a_0\}, \{a_1\}, \{a_2\}, \{a_0, a_1\}, \{a_0, a_2\}, \{a_1, a_2\}, \{a_1, a_2, a_3\}\}$$

Todo complejo simplicial abstracto K tiene una realización geométrica salvo homomorfismos. La realización geométrica es el espacio topológico obtenido a partir de la unión de los simplejos σ en $\mathbb{R}^{|V(K)|}$, para todo $\sigma \in K$ y es denotado por $|K|$. Por ejemplo, la realización geométrica del complejo simplicial O se muestra en la Figura 2.5.

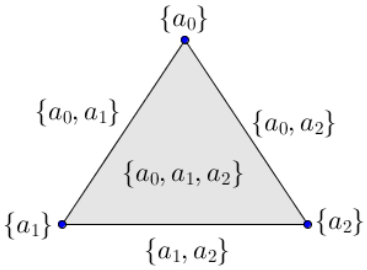


Figura 2.5: $|O| = \bigcup_{\sigma \in O} \sigma$

2.2.2. Operaciones entre Simplejos

En esta sección se hablará de complejos simpliciales abstractos y se hará referencia a ellos simplemente como complejos.

Se tomarán en cuenta algunas operaciones entre simplejos debidas a Kozlov en [20] que se describirán a continuación:

Eliminación: Sea K un complejo, y sea τ un simplejo en K. La eliminación de τ es un subcomplejo de K denotado por $\mathrm{dl}_K(\tau)$, definida por:

$$\mathrm{dl}_K(\tau) := \{\sigma \in K | \sigma \not\supseteq \tau\} \tag{2.4}$$

En otras palabras consiste en eliminar de K los simplejos que contienen a τ. Por ejemplo

Sea $K = \{\{\}, \{0\}, \{1\}, \{2\}, \{0,1\}, \{0,2\}, \{1,2\}, \{0,1,2\}\}$

y sea $\tau = \{0,1\}$

Si aplicamos $\mathrm{dl}_K(\tau)$ obtendremos como resultado el subcomplejo

$$K' = \{\{\}, \{0\}, \{1\}, \{2\}, \{0,2\}, \{1,2\}\}.$$

Además, si S es un conjunto de simplejos en K entonces se extiende la función de eliminación de la siguiente manera:

$$\mathrm{dl}_K(S) := \{\sigma \in K | \sigma \not\supseteq \tau \, para \; todo \; \tau \in S\} = \bigcap_{\tau \in S} \mathrm{dl}_K(\tau) \tag{2.5}$$

Otra operación que se le puede aplicar a un simplejo es *Link*. Sea K un complejo, y sea τ un simplejo de K. El *link* de τ es el subcomplejo simplicial abstracto de K, denotado por $\mathrm{lk}_K(\tau)$ definido por:

$$\mathrm{lk}_K(\tau) := \{\sigma \in K | \sigma \cap \tau = \emptyset, y \; \sigma \cup \tau \in K\} \tag{2.6}$$

Es decir, son los simplejos σ que son disjuntos con τ pero que al unir σ y τ forman un simplejo válido en K. Por ejemplo:

Sea $K = \{\{\}, \{0\}, \{1\}, \{2\}, \{3\}, \{0,1\}, \{0,2\}, \{0,3\}\{1,2\}, \{1,3\}, \{2,3\}$
$\{0,1,3\}, \{0,2,3\}, \{1,2,3\}\}$
y sea $\tau = \{3\}$.

Entonces el resultado de aplicar $\mathrm{lk}_K(\tau)$ es:

$$K' = \{\{\}, \{0\}, \{1\}, \{2\}, \{0,1\}, \{0,2\}, \{1,2\}\}.$$

Por otra parte, si tenemos un conjunto de simplejos S, entonces la función de *link* se puede extender de la siguiente manera:

$$\mathrm{lk}_K(S) := \{\sigma \in K | \sigma \cap \tau = \emptyset, and\ \sigma \cup \tau \in K, para\ todo\ \tau \in S\} = \bigcap_{\tau \in S} \mathrm{lk}_K(\tau) \tag{2.7}$$

La operación *Star*. Sea K un complejo simplicial abstracto, y sea τ un simplejo de K. La estrella cerrada $\mathrm{star}_K(\tau)$ es un subcomplejo de K definido por:

$$\mathrm{star}_K(\tau) := \{\sigma \in K | \sigma \cup \tau \in K\} \tag{2.8}$$

Se puede ver como los simplejos que contienen a τ o que unidos con τ forma un simplejo válido en K, por ejemplo:

Sea $K = \{\{\}, \{0\}, \{1\}, \{2\}, \{3\}, \{0,1\}, \{0,2\}, \{0,3\}, \{1,2\}, \{1,3\}, \{2,3\}$
$\{0,1,3\}, \{0,2,3\}, \{1,2,3\}\}$

y sea $\tau = \{0\}$

Entonces el resultado de aplicar $\mathrm{star}_K(\tau)$ es:

$$K' =$$
$$\{\{\}, \{0\}, \{1\}, \{2\}, \{3\}, \{0,1\}, \{0,2\}, \{0,3\}, \{1,3\}, \{2,3\}, \{0,1,3\}, \{0,2,3\}\}$$

También se define la estrella abierta, que en general no nos da como resultado un subcomplejo, y es la unión de los interiores de los simplejos que contienen a τ. $\mathrm{ostar}_K(\tau)$ se define de la siguiente manera:

$$\mathrm{ostar}_K(\tau) := \bigcup_{\sigma \supseteq \tau} \mathrm{int}\ \sigma. \tag{2.9}$$

En este caso se debe considerar que el resultado no es un subcomplejo si no un espacio topológico. La Figura 2.6 muestra ejemplos gráficos de cada una de las operaciones descritas anteriormente.

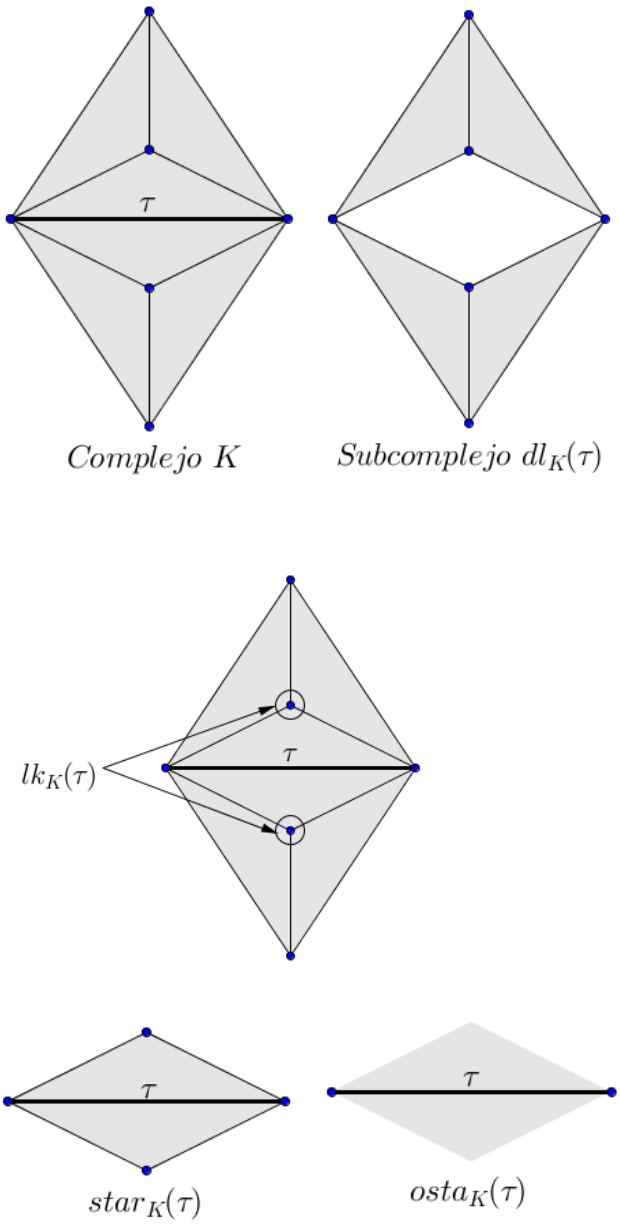


Figura 2.6: Ejemplo de las operaciones entre complejos.

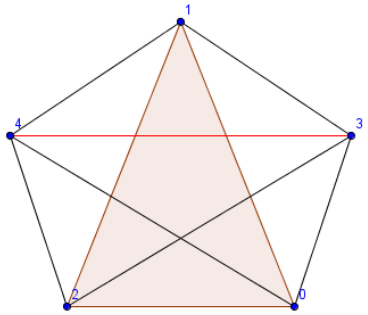


Figura 2.7: Representación gráfica de la unión de dos complejos $K_1 * K_2$, donde K_1 es el triángulo $\{0, 1, 2\}$ y K_2 es la arista $\{3, 4\}$.

2.2.3. Unión entre complejos

Sean K_1 y K_2 dos complejos cuyos vértices están indexados por conjuntos disjuntos. El *join* de K_1 y K_2 es el complejo simplicial $K_1 * K_2$ definido como sigue: el conjunto de vértices de $K_1 * K_2$ es $V(K_1) \cup V(K_2)$, y el conjunto de simplejos está dado por:

$$K_1 * K_2 = \{\sigma \subseteq V(K_1) \cup V(K_2) | \sigma \cap V(K_1) \in K_1 \; y \; \sigma \cap V(K_2) \in K_2\}. \quad (2.10)$$

Consideremos el complejo:
$K_1 = \{\{\}, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\}\}$
Y el complejo:
$K_2 = \{\{\}, \{3\}, \{4\}, \{3, 4\}\}$
Entonces el resultado de $K_1 * K_2$ es:
$K_1 * K_2 = \{\{\}, \{3\}, \{4\}, \{3, 4\}, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\}, \{0, 3\}, \{0, 4\}, \{0, 3, 4\}, \{1, 3\}, \{1, 4\}, \{1, 3, 4\}, \{2, 3\}, \{2, 4\}, \{2, 3, 4\}, \{0, 1, 3\}, \{0, 1, 4\}, \{0, 1, 3, 4\}, \{0, 2, 3\}, \{0, 2, 4\}, \{0, 2, 3, 4\}, \{1, 2, 3\}, \{1, 2, 4\}, \{1, 2, 3, 4\}, \{0, 1, 2, 3\}, \{0, 1, 2, 4\}, \{0, 1, 2, 3, 4\}\}$.

La representación gráfica de este complejo se muestra en la Figura 2.7. Cabe mencionar que, si bien se ilustra de forma plana, en realidad es un complejo de dimensión cuatro, que contiene caras como tetraedros, triángulos, aristas y vértices.

A partir de las operaciones definidas en las secciones anteriores se definirán nuevas operaciones para construir complejos simpliciales de manera

recursiva.

2.2.4. Subdivisiones

La subdivisión de un complejo es de gran interés, pues se estará hablando de subdivisiones de un complejo simplicial o de un simplejo durante la tesis, entonces consideremos K como un complejo simplicial geométrico en un espacio E. Un complejo K' se llama la subdivisión de K si cumple con las siguientes dos propiedades:

- Cada simplejo de K' está contenido en un simplejo de K.
- Cada simplejo de K es la unión finita de simplejos de K'.

Estas condiciones implican que los espacios topológicos correspondientes a los complejos simpliciales son los mismos, es decir $|K| = |K'|$. La Figura 2.8 muestra una subdivisión de un complejo.

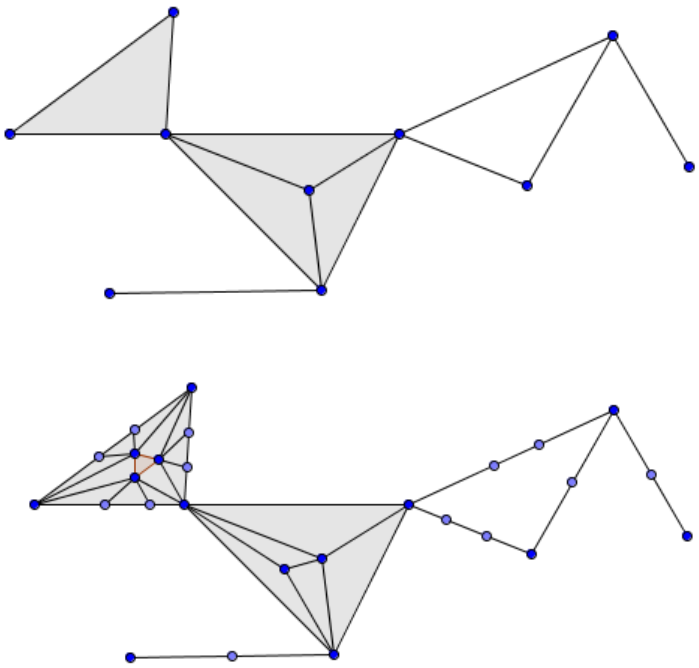


Figura 2.8: Complejo simplicial y una subdivisión de él.

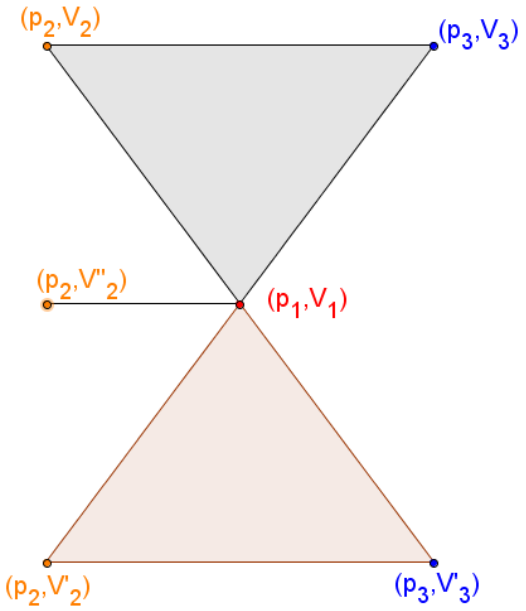


Figura 2.9: Complejo cromático.

2.2.5. Complejo Cromático

Hasta aquí se han visto simplejos y complejos como un conjunto de puntos. Ahora trabajaremos con una forma particular de nombrar sus vértices, y que posteriormente utilizaremos para hablar de ejecuciones de los algoritmos distribuidos.

Un complejo es cromático cuando los vértices de cada simplejo tienen nombres diferentes.

Los vértices son parejas (p, V), donde p es tomado de un conjunto de identificadores $\Pi = \{0, 1, 2, ..., n\}$ y V es un subconjunto de Π. V ayudará a relacionar simplejos y además tendrá su interpretación en cómputo como los procesos vistos por p.

El nombre de cada vértice será p y se obtendrá mediante la función $nombre(v)$, en general $nombre(\sigma)$ regresa el nombre de cada vértice en σ y además como hablamos de complejos cromáticos se utilizarán los nombres como los colores de los vértices. La Figura 2.9 muestra un complejo cromático, donde los vértices de cada simplejo tienen colores diferentes. Y además se observa que si dos vértices son del mismo color pero su V es diferente, se considera un vértice diferente.

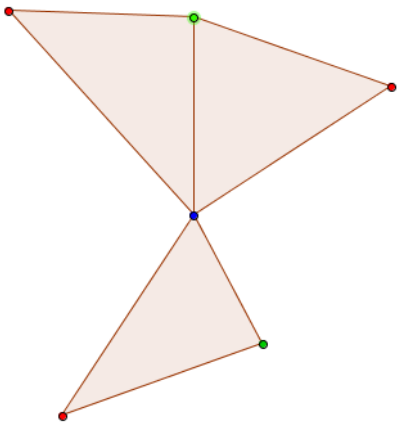


Figura 2.10: Complejo puro de dimensión 2

En está investigación se trabajará con subdivisiones cromáticas, es decir, dado un complejo cromático obtener una subdivisión cromática. La subdivisión debe cumplir con la definición dada anteriormente y además los vértices de un simplejo en el complejo subdividido deben tener nombres diferentes.

Un complejo es puro de dimensión n ó puro n-dimensional cuando todos sus simplejos estén contenidos en simplejos de dimensión n, por ejemplo en la Figura 2.10 se muestra un complejo puro de dimensión 2.

En este trabajo se considera sólo este tipo de complejos, en la parte de cómputo distribuido se verá que si algún proceso falla no significa que no se visualice en el complejo, debido a que no se tendrá forma de saber cuando un proceso falla y cuando no, pero esto no impide que se termine la tarea dada.

En la Figura 2.11 se muestra una subdivisión cromática del complejo de la Figura 2.10. Más adelante se verá un algoritmo distribuido que construye esta subdivisión y lo que representa cada uno de sus simplejos.

2.2.6. Operación $Skel^k_\tau$

En esta sección y en la siguiente se dará una descripción de una subdivisión cromática, construida con ayuda de funciones que han sido definidas en este trabajo.

Los complejos puros n-dimensionales K^n, se pueden subdividir de una manera recursiva, a partir de este momento cuando se hable de subdivisiones

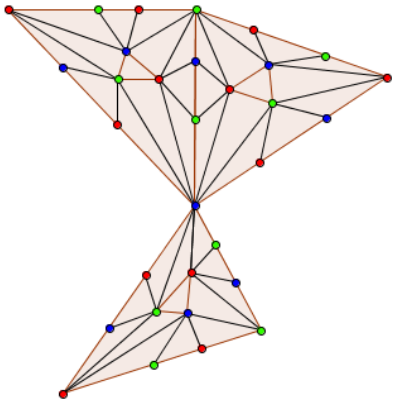


Figura 2.11: Subdivisión cromática del complejo puro de dimensión 2 de la Figura 2.10

serán subdivisiones cromáticas, subdividiendo primero los simplejos de menor dimensión hasta llegar a los simplejos de dimensión mayor.

La operación $Skel^j(C)$ se lee como el esqueleto de dimensión j del complejo C, y la subdivisión cromática de un complejo se denota por $\chi(C)$.

Parte del trabajo consiste en ir subdividiendo de manera recursiva los esqueletos del complejo, por ejemplo, para un complejo puro 3-dimensional, K^3, se requiere tener subdividido el 2-esqueleto, $\chi(Skel^2(K^3))$, que a su vez necesita de subdividir el 1-esqueleto, $\chi(Skel^1(K^3))$, el cual requiere la subdivisión del 0-esqueleto, $\chi(Skel^0(K^3))$.

El 0-esqueleto es el conjunto de vértices por lo que están cromáticamente subdivididos.

Recordemos que los vértices están etiquetados de la forma (p, V), ahora definamos el complejos U^n como el que contiene simplejos, τ^n, de la forma (p_i, V), con $0 \leq i \leq n$ y V es la misma para cada vértice de un simplejo. Es por esta razón que se necesita definir una función que relacione un simplejo τ^n con parte de algún esqueleto subdividido del complejo original, por lo que se da la definición de la siguiente función.

$$Skel^k_\tau(C) = \{\sigma | nombre(\sigma) \subset V_\tau, V_\sigma \subset V_\tau, \sigma \in C)\} \tag{2.11}$$

Está función la leemos de la siguiente manera: El k-esqueleto de τ en el complejo C, son los simplejos σ tal que los nombres de σ están en V de τ, V de σ está contenida en V de τ y σ está en C.

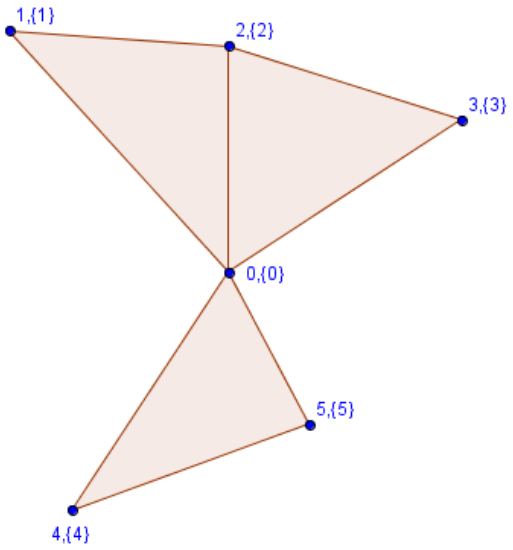


Figura 2.12: Complejo K^2

Para ilustrar esta función, en la Figura 2.12 se muestra el complejo K^2, con sus vértices etiquetados de la forma (i, V) por simplicidad en la notación. En la Figura 2.13 se muestra el 1-esqueleto subdividido de K^2, L^1, y un simplejo τ^2 el cual se busca relacionar con simplejos en L^1.

2.2.7. Operación $ChrJoin$

Realizar una unión cromática es parte importante de este trabajo. Se tratará de explicar de manera intuitiva esta operación. Se tiene que los vértices están etiquetados por pares (p, V), donde p es tomado de un conjunto de identificadores Π, y V es un subconjunto de vértices.

En la parte de cómputo se considera a p como el nombre del proceso y a V como su vista, *view*.

Entonces dados dos complejos con etiquetas en sus vértices de la forma (p, V), la unión cromática los une considerando lo siguiente:

- Para cualesquiera dos vértices adyacentes $p_i \neq p_j$.
- Para cualesquiera dos vértices adyacentes $V_i \subseteq V_j$ o $V_j \subseteq V_i$.

Las uniones cromáticas siempre se harán entre complejos de dimensión

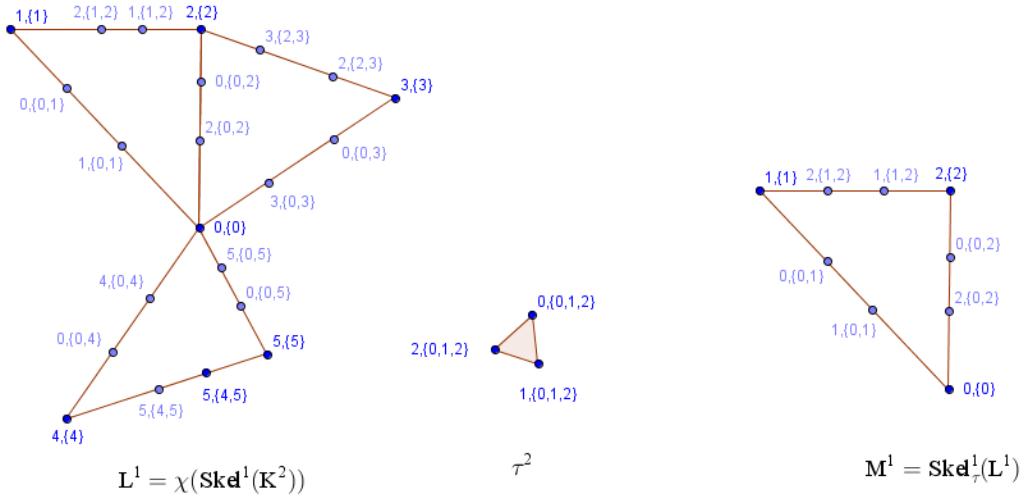


Figura 2.13: De izquierda a derecha se muestra el 1-esqueleto subdividido de un complejo $L^1 = \chi(Skel^1(K^2))$, un simplejo τ^2 con vértices que tienen la misma vista y finalmente los simplejos en L^1 con los que se puede relacionar τ^2.

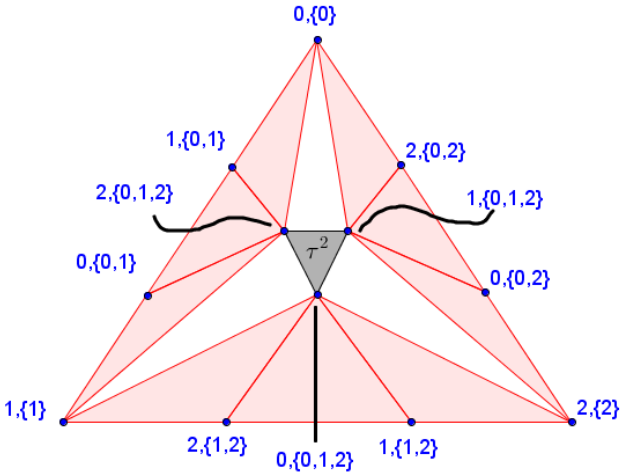
k y dimensión m, y como resultado siempre obtendremos un complejo de dimensión n.

Por ejemplo, los complejos de la Figura 2.13, podemos considerar un complejo $U^2 = \{\tau^2\}$ que sólo consta del simplejo τ^2 y el complejo M^1, entonces si aplicamos $ChrJoin(U^2 * M^1)$ intuitivamente la función hace lo siguiente:

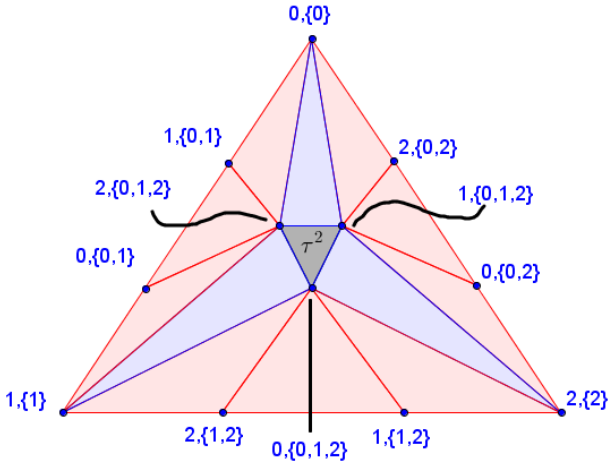
- La función *ChrJoin* tomará un simplejo τ de dimensión n en U^n, en este caso $n = 2$.
- La función *ChrJoin* tomará cada cara del simplejo τ^n empezando por las caras de dimensión 0 hasta las de dimensión $n - 1$.
- Para cada cara de dimensión k en τ^n se tomará un simplejo σ de dimensión m en M^{n-1} tal que $m + k + 1 = n$.

En la Firgura 2.14 se muestra como opera la función *ChrJoin*, tomando el complejo M^1 y el simplejo τ^2 de la Figura 2.13, primero la función toma cada cara de τ^2 de dimensión 0, es decir, sólo los vértices y cada uno de ellos se une con los simplejos de dimensión 1 en M^1, dejando en este primer paso el conjunto de simplejos rojos mostrados . Después se toma las caras de

dimensión 1 en τ^2 y se une con los vértices en M^1, formando los simplejos de color azul. Finalmente se toma la cara de dimensión 2 y se une con el simpejos vacío cuya dimensión es -1.



(a)



(b)

Figura 2.14: Representación gráfica de la función ChrJoin, donde primero en la figura (a) se toma cada simplejo de dimensión 0 en τ^2, $(0,\{0,1,2\}),(1,\{0,1,2\}),(2,\{0,1,2\})$, y se unen con simplejos de dimensión 1 en M^1, por ejemplo: $(0,\{0,1,2\})$ se une con $[(2,\{2\}),(1,\{1,2\})],[(1,\{1,2\}),(2,\{1,2\})]$ y con $[(2,\{1,2\}),(1,\{1\})]$, así se construyen los simplejos de color rojo; en la figura (b) ahora se toman los simplejos de dimensión 1 en τ^2, $[(0,\{0,1,2\}),(1,\{0,1,2\})],[(1,\{0,1,2\}),(2,\{0,1,2\})],[(0,\{0,1,2\}),(2,\{0,1,2\})]$ y se unen con simplejos de dimensión 0 en M^1 formando los simplejos azules, por ejemplo: $[(0,\{0,1,2\}),(1,\{0,1,2\})]$ se une con $(2,\{2\})$.

Capítulo 3

Algoritmos Distribuidos Recursivos

En este capítulo se abordarán los algoritmos estudiados en [12]. Se explican y relacionan con topología combinatoria, para posteriormente profundizar más a detalle en el capítulo 4. Cómo se había mencionado anteriormente, se pueden generar dos posibles estructuras, ya sea un árbol que consiste en una trayectoria, o un árbol general, por lo cual en este capítulo se estudian los dos casos. Por otra parte se describen las etiquetas de las llamadas recursivas, algo que no ocurre en el cómputo secuencial.

3.1. La tarea de Immediate Snapshot

La tarea de *immediate snapshot* tiene por objetivo hacer que la escritura y la lectura, a registros de lectura y escritura, parezca que ocurre de manera instantánea. Para resolver esta tarea, en [12] se presenta un algoritmo que consiste en un arreglo compartido $sm[]$ por $n' = n+1$ procesos. Los procesos que participan se identifican con una pareja (p_i, v), donde i es el identificador del proceso y v su valor de entrada, por conveniencia $v = i$. Cada proceso escribe en $sm[i]$ su valor de entrada v. El arreglo $sm[]$ se inicializa con $[\perp, \ldots, \perp]$. Cuando un proceso invoca un objeto *immediate snapshot*, obtiene una vista (*view*) la cual contiene los identificadores de los procesos que escribieron en el arreglo $sm[]$, si $sm[k] = \perp$ el valor de $sm[k]$ no es agregado a *view*. Cada proceso tiene su propia *view*, por lo cual se hará referencia por medio de $view_i$. Se usa la notación $n' = n + 1$, más adelante se toma n co-

mo la dimensión de algún simplejo que represente una ejecución y n' será la cantidad de procesos que participan en una ejecución.

Siguiendo con la notación se hace referencia a las llamadas recursivas como: la llamada recursiva n' para denotar que es la llamada recursiva donde se espera que participen a lo más n' procesos. Por ejemplo, si decimos la *llamada recursiva* 2 significa que es la llamada donde se espera que participen a lo más 2 procesos. Cuando decimos que se espera que participen a lo más n' es por que los algoritmos son libres de espera. Puede haber procesos más rápidos que otros. En una llamada recursiva n', puede ser el caso que sólo 1 proceso participe.

La vista de cada proceso debe cumplir con las siguientes propiedades:

- **Autoinclusión:** $\forall i : i \in sm_i$
- **Contención:** $\forall i, j : sm_i \subseteq sm_j \vee sm_j \subseteq sm_i$
- **Inmediatez:** $\forall i, j : i \in sm_j \Rightarrow sm_i \subseteq sm_j$

Recursión lineal En el Algoritmo 1 en la línea 1 los procesos se comunican a través de un arreglo de memoria compartida SW/MR, por sus siglas en inglés Single-Writer/Multi-Reader, invocado por medio de la operación WSCAN. El proceso p_i escribe su id i y almacena en *view* el conjunto de ids que lee del arreglo, es decir, las entradas diferentes de $\perp$. En la línea 2 los procesos revisan si su *view* contiene los n' ids, donde n' es el número de procesos que están participando en la llamada recursiva n'. El último proceso en escribir en el arreglo verá que $|view| = n'$ por lo que regresará su *view* y terminará el algoritmo. Al menos un proceso termina la llamada, pero quizás más de uno. En la línea 3 los procesos que obtienen $|view| < n'$ invocan realizan una llamada recursiva, cada vez sobre un arreglo compartido diferente, cuando $n' = 1$ el único proceso que invocó el algoritmo regresa *view* tal que sólo lo contiene a él mismo.

Algoritmo 1 IS

$\mathbf{IS}_{n'}(i)$

1: $view \leftarrow \text{WScan}(i)$
2: **if** $|view| = n'$ **then** return $view$
3: **else** return $\mathbf{IS}_{n'-1}(i)$
4: **end if**

Este algoritmo resuelve la tarea del *immediate snapshot*, como se demuestra en [12], por lo que sólo se explicará la recursión lineal que se origina de este algoritmo y se ilustra en la Figura 3.1.

Como se puede observar en la Figura 3.1 un proceso termina en la llamada donde participan n' procesos, y en la última llamada recursiva sólo queda un proceso; del algoritmo se sabe que su vista lo contiene a el mismo, cumple con las propiedades del *immediate snapshot*, por lo que nos garantiza que el algoritmo termina. También se observa cómo las vistas obtenidas por cada proceso cumplen con las tres propiedades del *immediate snapshot*, y es por esta razón que de manera intuitiva se puede demostrar que el algoritmo es correcto. Se pueden tener varias ejecuciones del algoritmo, ya sea concurrente o que parezca una ejecución secuencial, cuando haya procesos con la misma vista se considera que se ejecutaron de manera concurrente y cuando tengan vistas diferentes se considera que se puede dar un orden secuencial de la ejecución, el orden esta dado por contención.

En la Figura 3.2 (a) se muestra el caso donde tres procesos participan y se ven todos a todos por lo que se considera una ejecución concurrente y terminan en la primera llamada. En la Figura (b) se muestra una ejecución donde 2 procesos terminan en la primera llamada, en la segunda llamada nadie termina, se esperan a lo más 2 procesos, pero solo llega uno por lo que la condición de la línea 2 no se cumple y es necesario que el proceso que participa solo realice una tercera llamada para poder terminar.

La etiqueta de la llamada sólo es para dar conocimiento de cuántos procesos se esperan. En este y los siguientes algoritmos se conoce *a priori* el número de procesos que participarán en cada llamada.

3.2. La tarea de renombramiento

En el problema del renombramiento (*renaming*), los procesos inician con nombres distintos, sólo pueden comparar sus nombres y elegir uno de $2n' - 1$ posibles nombres nuevos que llamaremos *casillas*. Por otra parte los nombres iniciales tienen un orden total, es decir $\forall i, j : \ i < j$ o $j < i$.

El renombramiento en [17] se probó que es imposible resolverlo con menos de $2n' - 1$ casillas (*slots*). El Algoritmo 2 resuelve el problema con $2n' - 1$ casillas de manera recursiva, utiliza el *immediate snapshot* y es de complejidad $O(n^2)$.

Los parámetros *Inicio* y *Direccion* del algoritmo son dos números enteros,

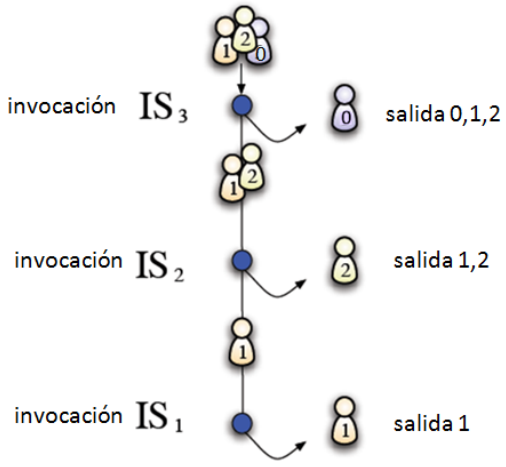


Figura 3.1: Recursión para 3 procesos ejecutando **IS**, tomada de [12].

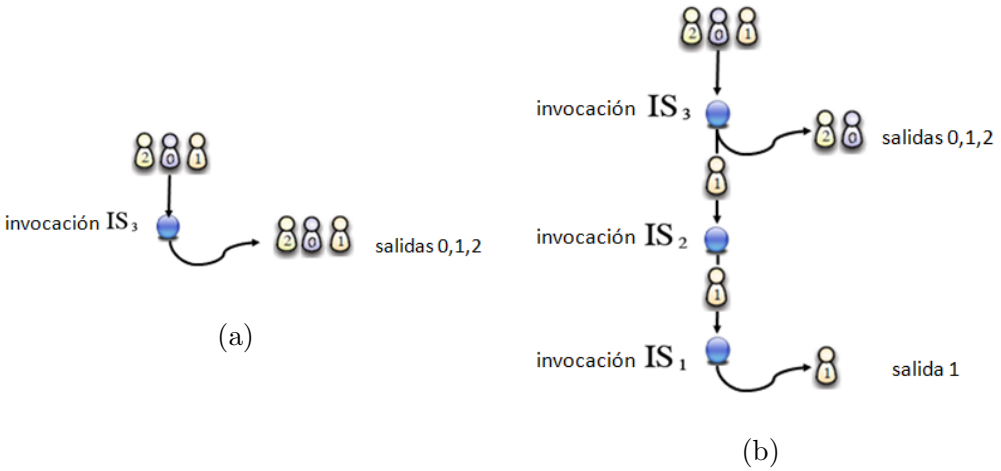


Figura 3.2: En la figura (a) se muestra una ejecución donde los tres procesos se ven todos a todos y terminan en la primera llamada. En (b) se muestra una ejecución donde 2 procesos terminan en la primera llamada, ninguno termina en la segunda y uno termina en la última.

con $Direccion \in \{1, -1\}$. En cada invocación limitan el rango de casillas a $Inicio + [0...2n' - 2]$ si $Direccion = +1$ o a $Inicio + [-(2n' - 2)...0]$ si $Direccion = -1$, es decir, a $Inicio + Direccion * [0...2n' - 2]$. Así el número de casillas es $2n' - 1$ es decir $|Ultimo - Inicio| + 1 = 2n' - 1$, definiendo $Ultimo = Inicio + Direccion * (2n' - 2)$. Como se menciono anteriormente el valor de n' es conocido *a priori* por lo que en el algoritmo la cardinalidad de la vista será menor o igual a n', $|view_i| \leq n'$ para toda i.

La etiqueta inicial del Algoritmo 2, *tag*, se inicializa con $tag = n'$, para indicar que se espera que participen a lo más n' procesos en la primera llamada, y en las siguientes llamadas, *tag* consistirá de una secuencia de enteros, donde el último entero refiere al número de procesos que se esperan en la llamada.

Recursión ramificada En el algoritmo del renombramiento, se utilizan las etiquetas en las llamadas recursivas para que los procesos puedan distinguir entre llamadas, estas etiquetas generan ramificaciones en el árbol de recursión correspondiente a la ejecución del algoritmo. En Algoritmo 2, en la línea 1, el proceso que está ejecutando la llamada, realiza el *immediate snapshot* para saber quienes están participando, en la línea 2 se calcula el posible valor del nuevo nombre que obtendrá el proceso p_i. En la línea 3 se verifica si el identificador del proceso p_i es el máximo de los identificadores que se obtuvieron en la vista, si es así se regresa el nuevo nombre del proceso, en caso contrario, se vuelve a invocar el algoritmo con una nueva etiqueta, en este punto se pueden originar diferentes llamadas a las cuales referiremos como llamadas consecutivas.

Algoritmo 2 isRENAMING

isRENAMING$_{tag}$(*Inicio, Direccion*)

1: $view_i \leftarrow$ **IMMEDIATE_SNAPSHOT**$_{tag}(i)$
2: $Ultimo \leftarrow Inicio + Direccion * (2|view_i| - 2)$
3: **if** $i = max(view_i)$ **then** return *Ultimo*
4: return **isRENAMING**$_{tag,|view_i|-1}$ $(Ultimo - 1, -Direccion)$

El árbol generado de la ejecución del algoritmo realizada por 4 procesos se muestra en la Figura 3.3, los círculos azules representan la ejecución de una llamada y los heptágonos representan un proceso, como se puede observar los procesos p_0 y p_1, denotados únicamente como 0 y 1, ven a todos los participan-

tes; pero, sus identificadores no son los más grandes en sus respectivas vistas, por lo que ambos ejecutan una nueva llamada recursiva $isRENAMING_{4,3}$, donde $4, 3$ es la nueva etiqueta de la llamada recursiva que ejecutarán. Mientras que p_2 ejecuta una llamada diferente $isRENAMING_{4,1}$ y p_3 por verse solo, hace verdadera la condición de la línea 3 y termina. Se nota que en la primera llamada con $n' = 4$, los participantes inician con $Inicio = 0$ y $Direccion = 1$, el rango de donde pueden obtener sus nuevos nombres los procesos es $[0...2n' - 2] = [0...6]$ y cabe destacar que por ser $Inicio = 0$ entonces los procesos terminaran con $Ultimo$ igual a un número par.

Lema 1 *Si* Inicio *es un número par en la ejecución de una llamada, entonces* Ultimo *es par.*

Prueba. Sea $Inicio = 2m$ donde $m \in \mathbb{N}$, del algoritmo tenemos que el nuevo nombre de un proceso se obtiene mediante:
$Ultimo = Inicio + Direccion(2|view| - 2)$
por lo que $Ultimo = 2m + Direccion * 2 * (|view| - 1)$ de lo cual obtenemos que $Ultimo = 2(m + Direccion(|view| - 1))$ por lo tanto $Ultimo$ es un número par. □

Lema 2 *Si* Inicio *es un número impar en la ejecución de una llamada, entonces* Ultimo *es impar.*

Prueba. La prueba se sigue del Lema 1 □

En la llamada recursiva $isRENAMING_{4,3}$ los procesos terminarán con $Ultimo \in [5, 4, 3, 2, 1]$ y por ser $Inicio$ un número impar entonces $Ultimo$ será un número impar del rango, los procesos que terminan en ésta llamada tendrán un nombre diferente a los procesos que terminaron en la anterior. En $isRENAMING_{4,1}$ el rango es igual a [1] y de la misma forma por haber sido invocada por una llamada donde $Ultimo$ fue un número par, ésta llamada fue invocada con $Ultimo - 1$ y sólo participa un proceso por lo que $Inicio$ fue un número impar y el proceso terminó con $Ultimo = Inicio$. Finalmente en la llamada recursiva $isRENAMING_{4,3,1}$ el rango es [2]. Como se puede observar, si X es el conjunto de procesos que terminan en la llamada a y Y es el conjunto de procesos que terminaron en la llamada b, que fue originada en a, los procesos no terminan con el mismo nombre, y los nombres quedan en el rango de nombres originales por que los rangos nuevos quedan anidados.

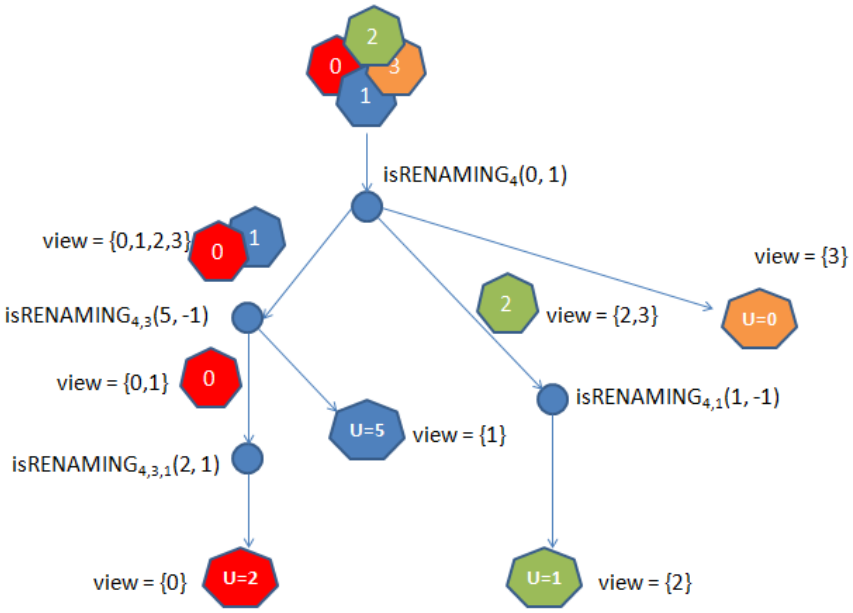


Figura 3.3: Recursión para 4 procesos ejecutando isRENAMING, donde se muestra cómo se ramifica el árbol de recursión. Los círculos azules representan la ejecución de la llamada isRENAMING. *view* muestra la vista al terminar una llamada isRENAMING y conociendo ésta se puede calcular U.

En la Figura 3.4 se da un ejemplo donde 4 procesos se ejecutan prácticamente de manera secuencial, y el valor del nuevo nombre está representado por U en las hojas del árbol, de la misma forma que en la Figura 3.3.

Teorema 2 *El algoritmo 2 resuelve el problema del renombramiento con* $2n' - 1$ *casillas.*

La siguiente prueba se retomará de [12].

Prueba. La prueba se hará por inducción sobre el número de procesos que participan en una llamada recursiva ya que la línea 4 a lo más la ejecutarán $n' - 1$ procesos, es decir en cada nueva llamada recursiva participarán menos procesos hasta que participe únicamente un proceso.

El caso base es cuando sólo un proceso, p_i, ejecuta el algoritmo, $n' = 1$. Como sabemos del algoritmo 1, su vista cumple con autoinclusión por lo que al ser un sólo proceso participando, necesariamente $|view_i| = 1$. Sin pérdida de generalidad supongamos que $Inicio = 0$ y dirección es 1 por lo que el rango es $Inicio + Direccion * [0...2(1) - 2] = 0 + 1[0] = [0]$. Del algoritmo tenemos que $Inicio + Direccion * (2 * |view| - 2)$ por lo que $Ultimo = Inicio$. La condición de la línea 3 se hará verdadera pues $i = max(view_i)$ y el algoritmo termina utilizando $2(n') - 1 = 1$ casillas con $Ultimo = 0$, el cual está en el rango.

Supongamos que el algoritmo es correcto para k procesos, $k < n'$. La hipótesis de inducción es que cuando k procesos invocan $isRENAMING_{tag}(Inicio, Direccion)$, ellos obtienen nombres diferentes en el rango $Inicio + Direccion * [0..2k - 2]$, que en total son $2k - 1$ nombres.

Por demostrar que cuando el algoritmo es invocado por $k + 1 \leq n'$ procesos, $n' > 1$, obtienen nombres diferentes en el rango $Inicio + Direccion * [0..2k - 2]$.

Sin pérdida de generalidad supongamos que inicialmente $Inicio = 0$, número par, y $Direccion = 1$ por lo que el rango de nuevos nombres será $[0...2(k + 1) - 2] = [0...2k]$ y probemos lo siguiente:

Los procesos que terminen en la llamada donde participan $k+1$ procesos obtienen diferentes nombres.

Supongamos por contradicción que existen m procesos, $1 < m \leq k + 1$, que terminan con el mismo nombre, $Inicio$ y $Direccion$ son el mismo valor para todos, sus vistas deben tener la misma cardinalidad, lo cual significa que vieron lo mismo por lo que $Ultimo$ es el mismo valor para los m procesos, pero, la simetría se rompe por medio de los identificadores ya que por

definición los identificadores para cualesquiera dos procesos $x < y$ ó $y < x$, por lo que sólo uno terminará en la línea 3 contradiciendo nuestra suposición. El valor de $Ultimo$ para todos estará dentro del rango pues $|view| \leq k + 1$.

Los procesos que ejecuten la línea 4 lo harán con $Direccion = -1$, $Inicio < 2k$ y será un número impar, por el Lema 1 sabemos que $Ultimo$ fue par en la llamada donde participaron $k + 1$. Por hipótesis de inducción éstas llamadas resuelven el problema para $k < n'$ procesos. Basta con demostrar que las llamadas generadas no regresan el mismo nombre para cualesquiera dos procesos y no toman alguno de la llamada anterior.

Como se menciono anteriormente en la primera llamada, con tag_{k+1}, los procesos obtienen $Ultimo$ siendo un número par, entonces los procesos que terminan obtienen un número par. Los procesos que ejecutan la línea 4 iniciaran una llamada con $Inicio$ impar, por el Lema 2 se tiene que $Ultimo$ en esta nueva llamada será impar, los procesos que terminen en ésta llamada tendrán nombres diferentes a los obtenidos en la llamada tag_{k+1}.

Las llamadas generadas en la línea 4 lo harán con etiquetas tag_i con $0 < i \leq k$. A lo más se generan k llamadas. El valor de i nos dice a lo más cuántos procesos se esperan para participar en la llamada, pero debemos tomar en cuenta que el número de procesos que participan puede ser menor.

Denotemos por X^i el conjunto de procesos que participan en la llamada tag_i y a $Ultimo^i$ el valor con el que se invoca. Recordemos que el rango lo obtenemos mediante $Inicio + Direccion * [0...2|X^i| - 2]$, como esta llamada se genera desde la llamada tag_{k+1} entonces el rango lo definimos como $Ultimo^i - 1 - [0...2|X^i| - 2]$ finalmente tenemos que $[Ultimo^i - 1...Ultimo^i - 1 - (2|X^i| - 2)] = [Ultimo^i - 1...Ultimo^i - 2|X^i| + 1]$.

Ahora que conocemos el rango para cada llamada tag_i basta con demostrar que los intervalos de nombres entre llamadas en el mismo nivel de la recursión no se traslapan. Sin perdida de generalidad consideremos la llamada tag_m tal que m es el valor más grande de procesos esperados a participar en una llamada, intuitivamente si $|X^m| = m$, sería la única y se ocuparía todo el rango. Consideremos otra llamada l tal que $l < m$, entonces tenemos que los rangos para estas llamadas son:

$$[Ultimo^m - 1...Ultimo^m - 2|X^m| + 1]$$
$$[Ultimo^l - 1...Ultimo^l - 2|X^l| + 1]$$

Para demostrar nuestra afirmación basta con probar que $Ultimo^m - 2|X^m| + 1 > Ultimo^l - 1$. De la llamada tag_{k+1} tenemos que $Ultimo^m = Inicio + Direccion * (2|view| - 2)$, dado que $Inicio = 0, Direccion = 1$ y

$|view| = m+1$ entonces $Ultimo^m = (2(m+1)-2) = 2m$, análogamente para $Ultimo^l = 2l$. Claramente $Ultimo^m > Ultimo^l$, el valor de l esta delimitado por $1 \leq l \leq m - |X^m|$, si $|X^m| = m$ no existiría otra llamada, por lo que a lo más $|X^m| \leq m - 1$. Sustituyendo los valores en nuestra desigualdad tenemos que:

$$\begin{aligned} 2m - 2|X^m| + 1 &> 2l - 1 \\ 2m - 2|X^m| &> 2l - 2 \\ m - |X^m| &> l - 1 \end{aligned} \tag{3.1}$$

Los procesos que pueden participar en la llamada tag_m están delimitados por $1 \leq |X^m| < m$, entonces si $|X^m| = 1$, $m - 1 > l - 1$ se cumple y si $|X^m| = m - 1$ entonces

$$\begin{aligned} m - (m-1) &> l - 1 \\ 1 &> l - 1 \\ 2 &> l \end{aligned} \tag{3.2}$$

sabemos que en este caso $l = m - (|X^m| - 1)$ por lo que $l = 1$ entonces tenemos que $2 > 1$ lo cual es verdadero, por lo tanto los intervalos entre llamadas en el mismo nivel de la recursión no se traslapan.

Por hipótesis de inducción estas llamadas tag_i resuelven el problema del renombramiento, y los procesos que terminan en nuestra llamada tag_{k+1} tienen nombres diferentes en el rango original. Por lo que demostramos que el algoritmo resuelve el problema del renombramiento con $2n - 1$ casillas.

□

3.3. La tarea de intercambio

Para explicar esta tarea en [12] se consideran las tareas $tournament_\pi$ y $swap_\pi$.

La tarea *tournament* está definida en [3] de la siguiente manera: *tournament* regresa para un sólo proceso 0 y para cualquier otro proceso regresa algún identificador de los participantes del torneo. Esta la podemos adaptar a: sólo un proceso regresa *tag* y para cualquier otro proceso regresa algún identificador de los participantes del torneo.

En la tarea de *swap*, cada proceso obtiene el id de otro proceso y exactamente un proceso obtiene un identificador inicial que llamamos π ó -1.

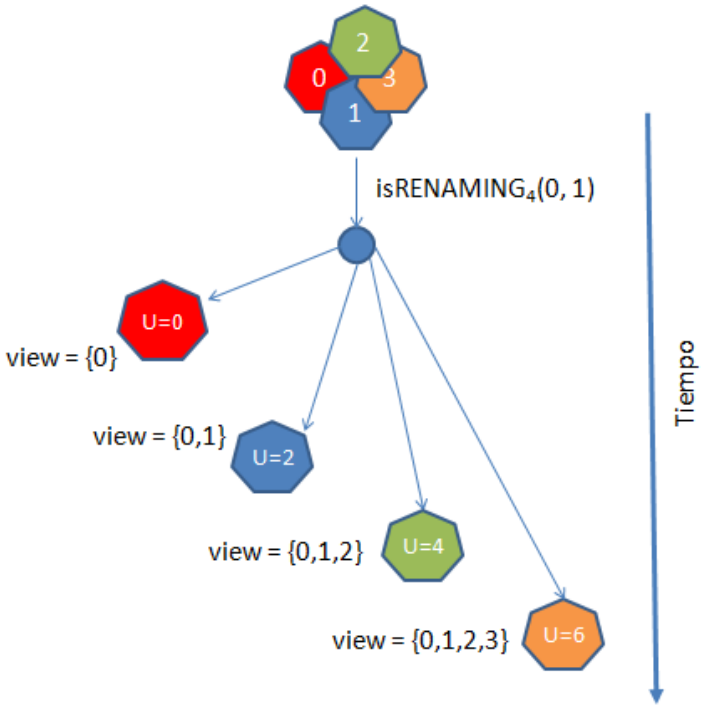


Figura 3.4: Ejecución de isRENAMING donde todos los procesos terminan en la primera llamada, invocando isRENAMING$_4$(Inicio,Ultimo)=isRENAMING$_4$(0,1). U indica el nuevo nombre al final de la ejecución del algoritmo.

Un proceso nunca obtiene de regreso su propio identificador, al final no hay dos procesos con el mismo. La especificación secuencial de la tarea es: cada operación regresa la entrada de la operación que le precede, cada ejecución completa de n' procesos define un orden total.

La recursión de *SWAP* resulta en un árbol, por lo cual podemos decir que es una recursión ramificada. El Algoritmo 3 resuelve la tarea *swap*, con una complejidad de $O(n^2)$ como se demuestra en [12]. Inicialmente la primera llamada es invocada con $SWAP_{tag}$, con $tag = -1$, en la línea 1 el proceso p_i invoca *TOURNAMENT* con la *tag* inicial, sólo un proceso puede obtener *tag* y lo almacenará en la variable π, dejando su identificador en *TOURNAMENT* donde 1 ó más procesos pueden obtener el identificador del ganador, y otros pueden obtener el identificador de algún otro proceso que haya participado. El proceso que ganó el torneo hará verdadera la condición $tag = \pi$ en la línea 2 y terminará su ejecución, los procesos que obtengan un identificador diferente de -1, ejecutarán la llamada recursiva $SWAP_{\pi}(i)$, donde π es el identificador que obtuvieron del torneo y por el cual participarán en la llamada que ejecuten en la línea 3.

Otro punto importante es que el algoritmo no usa sólo registros de lectura y escritura. En [3] se demostró que las tareas de *tournament* y *swap* requieren usar registros 2-consenso. En la implementación que los autores muestran usan registros *2Pswap*, el cual dadas 2 variables intercambia sus valores de manera atómica. El $2P$ significa que su número de consenso es 2.

Algoritmo 3 SWAP

SWAP$_{tag}(i)$
1: $\pi \leftarrow$ **TOURNAMENT**$_{tag}(i)$
2: **if** $tag = \pi$ return π
3: **else** return **SWAP**$_{\pi}(i)$

En la Figura 3.5 se muestra la ejecución para 3 procesos, se muestra un caso donde la recursión es lineal, solo ocurre si en cada invocación todos ven al ganador de la llamada. El proceso p_2 es quien gana el torneo, por lo cual regresa -1, por lo que ahora el proceso p_0 y p_1 apuntan al proceso 2, es decir, invocan $SWAP_2$ y el proceso que gana en esta llamada del algoritmo es p_1, por lo que regresa 2, finalmente p_0 invoca $SWAP_1$ y por ser el único que compite obtiene el valor 1. En la Figura 3.6 se muestra una ejecución donde el árbol de recursión se ramifica. El proceso 1 gana el torneo con etiqueta

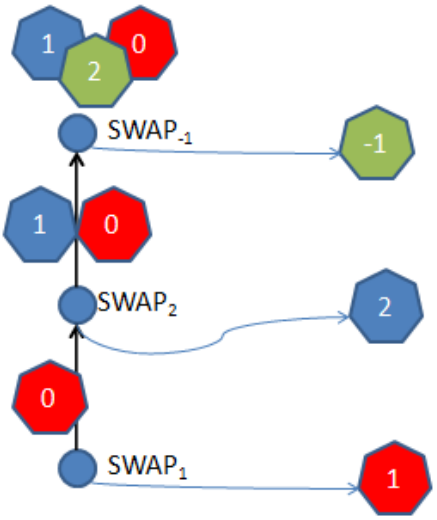


Figura 3.5: Recursión para 3 procesos ejecutando **SWAP**, donde se muestra el árbol de recursión resulta en una trayectoria. Los círculos azules representan la ejecución de la llamada. Las flechas azules indican el proceso que terminó en la llamada SWAP$_{tag}$. Las flechas negras indican la llamada que origino la nueva llamada.

-1, el proceso 2 ve al proceso 1 por lo que ejecuta $SWAP_1$, el proceso 0 ve a 2 por lo que ejecuta $SWAP_2$.

Teorema 3 *El algoritmo SWAP resuelve el problema del intercambio.*

Prueba. El algoritmo termina, ya que en cada invocación siempre regresa un ganador.

El caso base, cuando un proceso compite, sólo hay un ganador, por lo que el algoritmo resuelve el problema para 1 proceso.

Supongamos que el algoritmo resuelve el problema para $k' < n'$ procesos. En el paso inductivo consideremos la ejecución con n' procesos y cuando digamos: *el proceso obtuvo tag*, nos referiremos al proceso que ganó el torneo.

Consideremos la ejecución donde participan n' procesos, por definición del torneo sólo un proceso p_i será el ganador y obtendrá *tag*, inicialmente $tag = -1$, terminará su ejecución y los demás procesos obtendrán el identificador de p_i. Sea W el conjunto de procesos que obtuvo $tag = i$, entonces ejecutarán $SWAP_i$. A lo más $n' - 1$ ejecutaran $SWAP_i$, por hipótesis de inducción $SWAP_i$ resolverá el problema del intercambio para los $n'-1$ procesos,

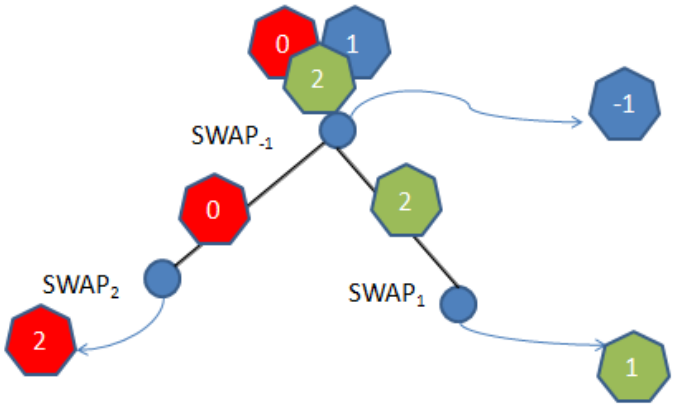


Figura 3.6: Recursión para 3 procesos ejecutando **SWAP**, donde se muestra el árbol ramificado de recursión. Los circulos en azules representan la ejecución de la llamada. Las flechas azules indican el proceso que terminó en la llamada SWAP_{tag}. Las flechas negras indican la llamada que origino la nueva llamada.

de manera similar los procesos que ejecuten $SWAP_x$ con $x \neq i$. Finalmente no más de un proceso puede terminar con x pues este valor solo puede ser obtenido en la llamada $SWAP_x$ y en particular sólo un proceso obtiene -1, en la llamada donde participan los n' procesos. □

En [2] argumentan que el algoritmo SWAP no es linearizable. Los autores en [12] dicen que no requieren de la propiedad de linearización, y como ejemplo mencionan que no necesariamente el ganador es primero.

Una propiedad menos restrictiva es la de ser consistente secuencialmente, donde únicamente nos interesa el orden de programa. Considerando esta propiedad mostraremos una ejecución que no es linearizable.

Recordemos que la definición de linearización, dada en [19] y detallada en [18], que un algoritmo sea linearizable, significa que las llamadas a métodos deben parecer que toman efecto de manera instantánea. Podemos crear una historia que cumpla con la especificación secuencial de la tarea, es decir, podemos establecer un orden total entre las llamadas a métodos de un mismo objeto y además se debe respetar el orden de tiempo real.

Lema 3 *El algoritmo del SWAP recursivo no es linearizable.*

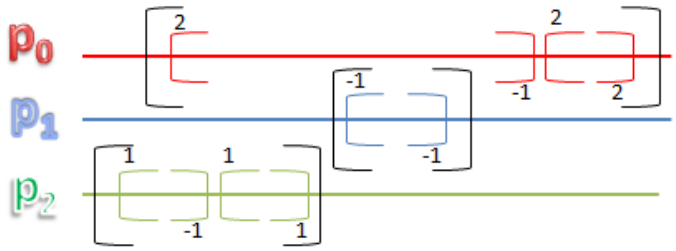


Figura 3.7: Ejecución no linearizable para 3 procesos. Los índices en la parte inferior del intervalo representan el valor por el que se participa en el torneo. Los índices en la parte superior representa el valor que obtuvo el proceso después de ejecutar *tournament*.

Prueba. Para cualquier ejecución del algoritmo SWAP, debemos poder determinar un punto de linearización.

Como en [12] los autores dicen que no se requiere que *tournament* cumpla con linearización, se construyó la ejecución mostrada en la Figura 3.7, y se muestra que es una ejecución válida y no es linearizable.

Las llamadas a $tournament_{-1}$ de los procesos p_0 y p_2 se traslapan, por lo que es válido que p_2 obtenga 0. Al igual para las llamadas a $tournament_{-1}$ de p_0 y p_1. Es válido que p_1 gane el torneo. Del algoritmo podemos notar que una vez que los procesos ganan algún torneo terminan.

No importa donde tratemos poner los puntos de linearización. No es posible determinar un punto en el intervalo de las llamadas tal que $(p_1 : \text{SWAP}) \rightarrow (p_2 : \text{SWAP})$. No se cumple con la especificación secuencial de la tarea *swap*, que dice: cada operación regresa la entrada de la operación que le precede. Por lo tanto el algoritmo SWAP no es linearizable.

□

Capítulo 4

Construcción de Complejos Simpliciales de Manera Recursiva

En esta sección se mostrará cómo los algoritmos distribuidos inducen subdivisiones de complejos o simplemente complejos cromáticos de una manera recursiva. El objetivo de la tesis es mostrar la estrecha conexión entre los algoritmos distribuidos recursivos, y los métodos para subdividir complejos simpliciales.

4.1. Construcción recursiva de subdivisiones cromáticas

Cada simplejo σ^n se puede considerar como un complejo Δ^n considerando σ^n y todas sus caras. En esta sección consideraremos sólo complejos obtenidos a partir de un simplejo.

En un complejo cromático Δ^n los vértices los denotamos por $\mathcal{V}(\Delta^n)$ y son pares (p, V) donde $p \in \Pi$, Π es un conjunto de identificadores $\{0, 1, 2, \ldots, n\}$ y V es la vista (*view*) de p. La vista de p es un conjunto de vértices cromáticos. Sean los vértices (p, V_p) y (q, V_q), se dice que p ve a q si $q \in V_p$. Consideremos un complejo cromático Δ^n que consiste en un $n-$simplejo. La siguiente definición combinatoria, aparece en [21] y se mostró que es una subdivisión de Δ^n. Ésta es completa para un complejo de dimensión n, es decir, cada simplejo está contenido en un $n-$simplejo.

Definición 7 *La subdivisión cromática $\chi(\Delta^n)$ es un complejo simplicial abstracto puro de dimensión n definido como sigue:*

1. *Los vértices de $\chi(\Delta^n)$ son indexados por todos los pares (p, V) donde $V \subseteq \Pi$, y $p \in V$, tal que hay $2^n(n+1)$.*

2. *Los simplejos de dimensión n de $\chi(\Delta^n)$ consisten de todos los conjuntos de vértices $\{(0, V_0), (1, V_1), \ldots, (n, V_n)\}$.*

y satisfacen los siguientes axiomas:

I *Para todo i, $j \in \Pi$, se tiene que $V_i \subseteq V_j$ ó $V_j \subseteq V_i$,*

II *Para todo i, $j \in \Pi$, si $i \in V_j$, entonces $V_i \subseteq V_j$.*

Partiendo de un complejo de entrada Δ^n, obtenemos su subdivisión cromática $\chi(\Delta^n)$. Los vértices de Δ^n están etiquetados de la forma (p_i, V_i), por conveniencia siempre iniciamos con $(i, \{i\})$, $0 \leq i \leq n$. En la interpretación computacional, se considera que Δ^n es la configuración de entrada, es decir el proceso p_i inicia con entrada $\{i\}$.

Recordemos que el n-simplejo tiene $n+1$ vértices. Para probar que el número de vértices de la subdivisión cromática de un n-simplejo es $2^n(n+1)$, nos basaremos en la siguiente observación.

Para Δ^0 supongamos que su vértice es etiquetado con $(0, \{0\})$. Todas las parejas son $(0, \{0\})$. La subdivisión es el mismo vértice.

Para Δ^1 supongamos que sus vértices son etiquetados con $(0, \{0\}), (1, \{1\})$. Todas las parejas son $(0, \{0\}), (1, \{1\}), (0, \{0, 1\}), (1, \{0, 1\})$.

Para Δ^2 supongamos que sus vértices son etiquetados con $(0, \{0\}), (1, \{1\})(2, \{2\})$, todas las parejas son $(0, \{0\}), (1, \{1\}), (2, \{2\}), (0, \{0, 1\}), (0, \{0, 2\}), (1, \{0, 1\})$, $(1, \{1, 2\}), (2, \{0, 2\}), (2, \{1, 2\}), (0, \{0, 1, 2\}), (1, \{0, 1, 2\}), (0, \{0, 1, 2\})$.

Como se puede observar cada vértice aporta 1 vértice a la subdivisión, cada pareja aporta 2 vértices y cada terna aporta 3 vértices y así sucesivamente. Se puede contar el número de vértices de $\chi(\Delta^n)$ de la siguiente manera.

$$|\mathcal{V}(\chi(\Delta^n))| = \sum_{i=1}^{n+1} iC_i^{n+1} \tag{4.1}$$

Lema 4 *El número de vértices en una subdivisión cromática de un n-simplejo está dado por* $\sum_{i=1}^{n+1} iC_i^{n+1} = 2^n(n+1)$.

Prueba. La prueba se hará por inducción sobre n.

El caso base es cuando $n = 0$ entonces

$\sum_{i=1}^{0+1} iC_i^{0+1} = 2^0(0+1)$;

$1C_1^1 = 1(1)$;

$1 = 1$.

Supongamos que para k se cumple $\sum_{i=1}^{k+1} iC_i^{k+1} = 2^k(k+1)$, con $k < n$.

Probemos para $k+1$.

$\sum_{i=1}^{k+2} iC_i^{k+2}$

Separemos la suma de la siguiente manera

$\sum_{i=1}^{k+1} iC_i^{k+2} + (k+2)C_{k+2}^{k+2} = \sum_{i=1}^{k+1} iC_i^{k+2} + (k+2)$

Aplicando el teorema de Pascal tenemos que

$\sum_{i=1}^{k+1} i(C_{i-1}^{k+1} + C_i^{k+1}) + (k+2) = \sum_{i=1}^{k+1} iC_{i-1}^{k+1} + \sum_{i=1}^{k+1} iC_i^{k+1} + (k+2)$

Por hipótesis de inducción tenemos que

$\sum_{i=1}^{k+1} iC_{i-1}^{k+1} + 2^k(k+1) + (k+2)$

Desarrollando la suma que nos queda

$1C_0^{k+1} + 2C_1^{k+1} + 3C_2^{k+1} + ... + kC_{k-1}^{k+1} + (k+1)C_k^{k+1} + 2^k(k+1) + (k+2) =$

$C_0^{k+1} + (1+1)C_1^{k+1} + (2+1)C_2^{k+1} + ... + ((k-1)+1)C_{k-1}^{k+1} + (k+1)C_k^{k+1} +$

$2^k(k+1) + (k+2)$

Lo reescribimos de la siguiente manera

$1C_1^{k+1} + 2C_2^{k+1} + ... + (k-1)C_{k-1}^{k+1} + kC_k^{k+1} + C_0^{k+1} + C_1^{k+1} + C_2^{k+1} + ... +$

$C_{k-1}^{k+1} + C_k^{k+1} + 2^k(k+1) + (k+2) =$

$\sum_{i=1}^{k} iC_i^{k+1} + \sum_{i=1}^{k} C_i^{k+1} + 2^k(k+1) + (k+2)$

Por hipótesis de inducción tenemos

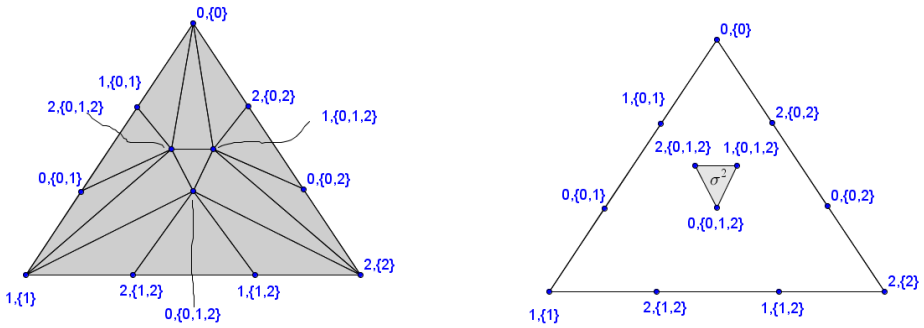
$$(2^k(k+1)-(k+1))+\sum_{i=1}^{k} C_i^{k+1}+2^k(k+1)+(k+2)$$

La suma que queda es conocida como la de coeficientes binomiales por lo que
$(2^k(k+1)-(k+1))+(2^{k+1}-1)+2^k(k+1)+(k+2)=$
$2^k(k+1)-(k+1)+2^{k+1}-1+2^k(k+1)+k+1+1=$
$2^k(k+1)+2^{k+1}+2^k(k+1)=2(2^k(k+1))+2^{k+1}=2^{k+1}(k+1)+2^{k+1}=$
$2^{k+1}(k+1+1)=2^{k+1}(k+2)$ □

Recordemos que si (p,V) es un vértice v, nosotros decimos que p es su nombre y lo denotamos por $nombre(v)$. En general, si σ es un simplejo, se obtienen los nombres de los vértices mediante $nombre(\sigma)$.

Sea el complejo $\Delta^2=(0,\{0\}),(1,\{1\}),(2,\{2\})$. En la Figura 4.1(a) se presenta su subdivisión cromática $\chi(\Delta^2)$ y en la Figura 4.1(b) se muestra el esqueleto del simplejo σ^2 con respecto a $\chi(\Delta^2)$.

Ahora que se recordaron algunas definiciones, se continuará con un algoritmo que construye dichas subdivisiones.



(a) (b)

Figura 4.1: a) Subdivisión cromática $K^2=\chi(\Delta^2)$ de un complejo simplicial $\Delta^2=(0,\{0\}),(1,\{1\}),(2,\{2\})$. b) σ^2 y la colección de simplejos de $Skel^1_{\sigma^2}(K^2)$.

Para poder explorar las propiedades recursivas de los complejos simpliciales como se menciona en [9], se hará mediante el Algoritmo 4 recursivo, que construye una subdivisión cromática. Se puede ver en el algoritmo que el complejo simplicial se construye a partir de un complejo simplicial de dimensión menor. En la línea 3 tenemos la base de la recursión, es decir, cuando

se recibe un complejo puro de dimensión $n = 0$, ya está subdividido de manera cromática. Si la dimensión del complejo no es 0 entonces se procede a subdividir de manera cromática el complejo de dimensión $n > 0$. En la línea 5 se obtiene la subdivisión cromática del esqueleto de dimensión $n-1$ del complejo C^n. En la línea 6 se comienza a iterar sobre los simplejos de dimensión n del complejo C^n. En la línea 7 por cada simplejo σ^n se crea un nuevo simplejo $\tau^n_{\sigma^n}$, dicho simplejo es prácticamente una copia de σ^n con la diferencia de que $V = nombre(\sigma^n)$ es la misma para cada vértice, por lo que las etiquetas de estos vértices las denotamos con (q, V_{σ^n}) en que q es algún identificador de σ^n, es decir $q \in nombre(\sigma^n)$.

Algoritmo 4 Algoritmo para construir una subdivisión cromática en forma recursiva.

1: **ChrSubDiv**(C^n)
2: $K^n = \{\}$
3: **if** $n = 0$ **then** return C^n;
4: **else**
5: $K^{n-1} \leftarrow$ **ChrSubDiv**($Skel^{n-1}(C^n)$)
6: **for each** $\sigma^n \in C^n$ **do**;
7: Sea $\tau^n_{\sigma^n}$ un nuevo simplejo con vértices etiquetados (q, V_{σ^n}) donde $q \in nombre(\sigma^n)$ y $V_{\sigma^n} = nombre(\sigma^n)$
8: Sea $X^{n-1} \leftarrow Skel^{n-1}_{\tau^n_{\sigma^n}}(K^{n-1})$
9: $K^n = K^n \cup$ **ChrJoin** $(\tau^n_{\sigma^n} * X^{n-1})$
10: **end for**
11: return K^n
12: **end if**

En la línea 8 obtenemos el esqueleto subdividido del simplejo σ^n, es decir, de K^{n-1} obtenemos todos los simplejos de dimensión $n-1$, tal que sus etiquetas estén formadas por (p, V), y $p \in nombre(\sigma^n)$. Finalmente aplicamos la función **ChrJoin**$(\tau^n_{\sigma^n} * X_{n-1})$, función que definimos como:

$$\textbf{ChrJoin}(\tau^n_{\sigma^n} * X^{n-1}) = \{\tau^k \cup \delta^m | k \leq n, \delta \in X^{n-1}, m = n-k-1, \\ nombre(\tau^k) \not\subset V_\delta \wedge V_\delta \subset V_\tau\} \tag{4.2}$$

Cabe mencionar que nos referimos a una cara de $\tau^n_{\sigma^n}$ de dimensión k como τ^k. Por ejemplo, la Figura 4.2 ilustra como aplicar la función **ChrJoin** para σ^2 y su esqueleto de dimensión 1 subdividido.

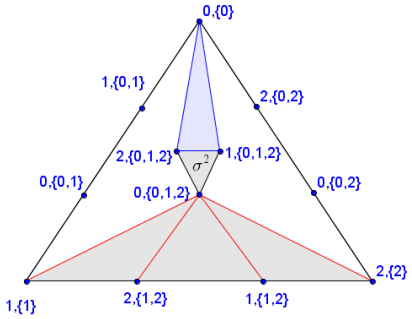


Figura 4.2: Los simplejos en rojo, muestran la unión entre el 0-simplejo $\tau^0 = (0, \{0, 1, 2\})$ y los simplejos de dimensión 1 que cumplen con la definición de **ChrJoin**, $\delta_1 = \{(1, \{1\}), (2, \{1, 2\})\}, \delta_2 = \{(2, \{1, 2\}), (1, \{1, 2\})\}, \delta_3 = \{(1, \{1, 2\}), (2, \{2\})\}$, El simplejo con borde azul fue construido por $\tau^1 = \{(2, \{0, 1, 2\}), (1, \{0, 1, 2\})\}$ y $\delta_4 = (0, \{0\})$. Finalmente, $\tau^2 = \sigma^2$ y $\delta = \{\}$, δ es el simplejo vació. X^{n-1} en este caso es el borde, o todos los simplejos con $|view| \leq 2$.

Teorema 4 *El Algoritmo 4,* ***ChrSubDiv****, construye una subdivisión cromática de un complejo simplicial* Δ^n*.*

Prueba. La invariante que podemos ver en el algoritmo, es que en cada llamada recursiva se tiene el esqueleto subdividido, $Skel^k(\Delta^n)$, por lo que la demostración la haremos por inducción sobre la dimensión k del esqueleto del complejo subdividido. Notemos que cuando $k = n$ tenemos la subdivisión cromática, $\chi(\Delta^n)$

Cuando $k = 0$ tenemos que el $Skel^0(\Delta^n)$, consiste de vértices únicamente. El algoritmo regresa el mismo complejo que por definición es una subdivisión cromática correcta de Δ^0.

Sea $K^k = \chi(Skel^k(\Delta^n))$. Supongamos que para k con $k < n$, ChrSubDiv($Skel^k(\Delta^n)$) regresa el complejo K^k.

Por demostrar que para $k + 1 = n$ ChrSubDiv($Skel^{k+1}(\Delta^n)$) regresa una subdivisión cromática de $Skel^{k+1}(\Delta^n)$.

Del algoritmo tenemos que K^{k+1} es construido a partir de K^k, por lo que necesitamos analizar que pasa a partir de la línea 6. Para cada simplejo $\sigma^{k+1} \in \Delta^n$, se crea un nuevo simplejo $\tau^{k+1}_{\sigma^{k+1}}$, el cual tiene sus vértices etiquetados de la forma $(q, V_{\sigma^{k+1}})$ donde q esta en los nombres de σ^{k+1} y $V_{\sigma^{k+1}}$ consiste en todos los nombres de σ^{k+1}.

Después en la línea 8 obtenemos X^k, el esqueleto subdividido de $\tau^{k+1}_{\sigma^{k+1}}$, es decir los simplejos en K^k que se pueden relacionar con $\tau^{k+1}_{\sigma^{k+1}}$.

Por definición la función $Skel^k_{\sigma^{k+1}}(K^k)$ regresa los simplejos $\delta^m \in K^k$ tal que los nombres de δ^m estén contenidos en la vista de $\tau^{k+1}_{\sigma^{k+1}}$ y que la V de los vértices de δ^m este contenida en $V_{\tau^{k+1}_{\sigma^{k+1}}}$.

Finalmente se aplica la función *ChrJoin(* $\tau^{k+1}_{\sigma^{k+1}} * X^k$*)*, la cual une cada cara τ^l de $\tau^{k+1}_{\sigma^{k+1}}$, $l \leq k+1$, con un simplejo $\delta^m \in X^k$ tal que $m = (k+1) - l - 1$. Si se toma una cara τ^0 entonces se relacionan con simplejos δ^k tal que los nombres de τ^k no esten contenidos en V_{δ^k} y que V_{δ^k} sea un subconjunto de V_{τ^0}. Los mismo para cada cara τ^1 relacionadas con cada δ^{k-1}. Finalmente cada cara τ^k se relaciona con cada δ^0 como lo indica la función. Así entonces tenemos simplejos de dimensión de a lo más $|\tau^l| + |\delta^m| = |\tau^{k+1}_{\sigma^{k+1}}|$, recordemos que la cardinalidad de un simplejo es el número de vértices que lo forman por lo que $l + 1 + m + 1 = l + 1 + (k+1) - l - 1 + 1 = k + 2$ que es justo $|\tau^{k+1}_{\sigma^{k+1}}|$. Como al final sólo agregamos simplejos de dimensión $k+1$, hemos formado K^{k+1}

La definición de la función *ChrJoin* respeta los axiomas I y II de la definición 7.

Por otra parte tenemos que el algoritmo regresa únicamente vértices en la llamada cero, por lo que tenemos $k+1$ vértices. Por cada simplejo de dimensión k, $k \leq n$, se agregan $k+1$ vértices, y tenemos tantos simplejos de dimensión k como las combinaciones de C^{n+1}_{k+1}, así el número de vértices esta dado por $\sum_{i=1}^{n+1} iC^{n+1}_i$ y por el lema 4 sabemos que el número de vértices en total es $2^n(n+1)$ por lo tanto K^{k+1} es una subdivisión cromática, en particular es la subdivisión cromática del esqueleto de dimensión n de Δ^n y como $Skel^n(\Delta^n) = \Delta^n$ tenemos que el algoritmo ChrSubDiv construye una subdivisión cromática para Δ^n.

□

4.2. Construyendo subdivisiones cromáticas con IS

Como se había visto anteriormente, podemos resolver la tarea del *immediate snapshot* con el Algoritmo 1, ahora se mostrara cómo de manera natural

este algoritmo induce una subdivisión cromática.

Las configuraciones de entrada se consideran como un complejo de la forma Δ^n. Donde cada proceso solo conoce su valor de entrada.

Anteriormente se describió detalladamente el algoritmo IS, ahora enfoquémonos en la línea 2, si los $n' = n+1$ participantes de la llamada recursiva se ven todos a todos, entonces justo en esta línea se formará un simplejo $\tau^n_{n'}$, con la característica de que todos los procesos tendrán la misma vista. Se consideran los procesos como vértices etiquetados de la forma (p_i, V), justo como se hizo anteriormente. Por ejemplo, para tres procesos con identificadores $\Pi = \{0, 1, 2\}$ que invocan $\text{IS}_3(i)$, podemos obtener el simplejo mostrado en la Figura 4.3

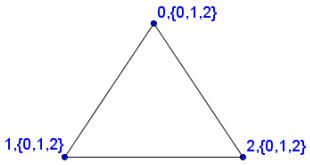


Figura 4.3: Simplejo $\tau^2_{n'}$ de las posibles vistas de los procesos cuando todos terminan en la línea 2, en la primera llamada. En estas vistas todos los procesos se ven a ellos mismos y a todos los demás. Cada vértice representa un proceso y es etiquetado con su *id* y su *vista*, donde la vista es un conjunto de ids que cada proceso ve. Por ejemplo podemos ver que un vértice es etiquetado *(id,view)*, con $1, \{0, 1, 2\}$ donde 1 es el *id* y $view = \{0, 1, 2\}$.

Como no siempre todos los procesos terminan en la línea 2 del Algoritmo IS y no hay manera de saber cuales terminarán en la llamada recursiva donde participan k', $\forall k' \leq n'$, necesitamos considerar todos los posibles simplejos τ para n' procesos de dimensión k, $\tau^k_{n'}$. Podemos construir un complejo con la unión disjunta de estos simplejos. Llamamos a este complejo $U^k_{n'} = \bigcup \tau^k_{n'}$ donde k es la dimensión del complejo y n' es el total de procesos que pueden participar. Por ejemplo, para tres procesos p_0, p_1, p_2 los simplejos en la llamada recursiva donde participan $k' = 2$ son:

$$\begin{aligned} U^1_3 = \{(p_0, \{p_0, p_1\}), (p_1, \{p_0, p_1\})\} \bigcup \\ \{(p_0, \{p_0, p_2\}), (p_2, \{p_0, p_2\})\} \bigcup \\ \{(p_1, \{p_1, p_2\}), (p_2, \{p_1, p_2\})\} \end{aligned}$$

Ahora se puede saber cuantos simplejos $\tau^k_{n'}$ pueden ser obtenidos en la llamada recursiva para k' procesos calculando $\binom{n'}{k'}$, donde $n' = n+1$ es el número total de procesos y $k' = k+1$ es el número de procesos que participan en la llamada recursiva.

En la última llamada recursiva, $\mathrm{IS}_1(i)$, solamente tenemos el conjunto de procesos Π con sus valores iniciales en su vista. Por lo tanto, podemos representar todas las posibles ejecuciones con $U^0_{n'}$, el cual contiene solamente vértices. Como se explicó anteriormente $U^0_{n'} = \bigcup \tau^0_{n'}$.

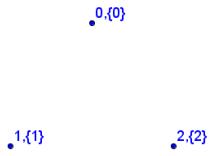


Figura 4.4: Complejo $U^0_{n'}$ de las posibles vistas de los procesos en la llamada recursiva $\mathrm{IS}_1(i)$ para $n' = 3$

Lema 5 *Consideremos el algoritmo IS. En una llamada recursiva para k' procesos se produce un k-simplejo, τ^k, tal que sus vértices son etiquetados con (p_i, V), con $p_i \neq p_j$ si $i \neq j$ y $|V| = k+1$ sii los k' procesos terminan en la línea 2.*

Prueba. $\Rrightarrow$ Sea τ^k un simplejo etiquetado (p_i, V) tal que $p_i \neq p_j$ si $i \neq j$ y $|V| = k+1$. Si consideramos cada vértice como un proceso con identificador p_i y vista $view = V$, entonces tenemos que $|view| = k+1$. Esta es justo la condición de la línea 2.

$\Lleftarrow$ Si en una llamada para k' procesos, éstos terminan en la línea 2 eso implica que $|view_i| = k'$. Entonces todos tienen la misma vista y se tienen k' vértices con etiquetas $(p_i, view)$, $V = view$, por lo tanto podemos construir un simplejo τ^n. □

Se necesita saber qué pasa cuando no todos los procesos terminan en la línea 2, por lo que definiremos el complejo $K^k_{n'}$.

El complejo $K^k_{n'}$ representa todas las posibles ejecuciones hasta la llamada recursiva para k' procesos, y es definido como sigue:

$$\begin{aligned} K^k_{n'} &= ChrJoin(U^k_{n'} * K^{k-1}_{n'}) \\ K^0_{n'} &= ChrJoin(U^0_{n'} * K^{-1}_{n'}) = U^0_{n'} \end{aligned} \tag{4.3}$$

Se tiene que recordar que $U^k_{n'}$ es la unión disjunta de simplejos $\tau^k_{n'}$, entonces por cada $\tau^k_{n'} \in U^k_{n'}$ aplicaremos **ChrJoin**$(\tau^k_{n'} * K^{k-1}_{n'})$.

La definición previa de $K^k_{n'}$ es recursiva. Por ejemplo, si se quiere conocer todas las posibles vistas en la llamada recursiva para 3 procesos, entonces necesitamos conocer $K^1_{n'}$:

$$K^1_{n'} = ChrJoin(U^1_{n'} * K^0_{n'})$$

Sabemos que $K^0_{n'}$ es la base de la recursión. También $K^0_{n'} = U^0_{n'}$, por lo tanto básicamente sólo necesitamos aplicar ChrJoin para los siguientes dos complejos, los cuales se ilustran en la Figura 4.5.

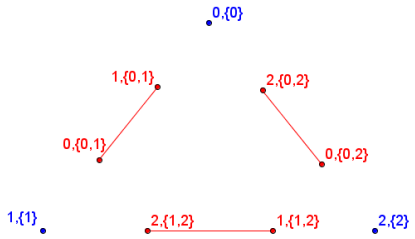


Figura 4.5: $K^0_{n'} = U^0_{n'}$ es representado por los vértices azules y $U^1_{n'}$ es representado por los 1-simplejos rojos.

Los vértices azules forman $K^0_{n'}$ y las aristas rojas forman $U^1_{n'}$. Si aplicamos ChrJoin para estos dos complejos, obtenemos $K^1_{n'}$. Agregamos solamente las aristas azules como se muestra en la Figura 4.6

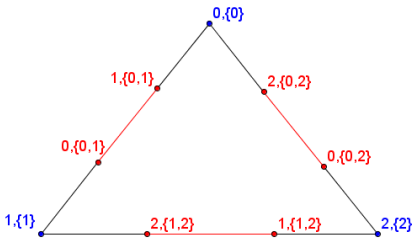


Figura 4.6: $K^1_{n'}$ obtenido a partir del $ChrJoin(U^1_{n'} * K^0_{n'})$

Teorema 5 *El algoritmo IS construye una subdivisión cromática de* Δ^n.

Prueba. Por inducción sobre la dimensión n. Cuando $n = 0$ la entrada de nuestro algoritmo es Δ^0. Sabemos que Δ^0 consiste de un solo vértice, como

cada proceso se ve así mismo entonces cumple la condición de la línea 2 y termina. A la salida tenemos $(p_i, view_i)$. El vértice cumple con la definición 7. Por lo tanto tenemos una subdivisión de Δ^0.

Supongamos que el algoritmo construye una subdivisión cromática para la entrada Δ^k, con $k < n$.

Por demostrar que el algoritmo construye una subdivisión cromática de la entrada Δ^n.

Si los n' procesos terminan en la misma llamada entonces se tiene un simplejo τ^n. El algoritmo garantiza que al menos un proceso termina, por lo que la siguiente llamada recursiva la se hará con k' procesos. Podemos considerar que la entrada de la nueva llamada es la configuración Δ^k y por hipótesis de inducción IS, forma una subdivisión cromática de este nuevo complejo. Pero se tienen varias configuraciones de entrada para la nueva llamada dadas por $C^{n'}_{k'}$. Siendo así, se tiene $K^k_{n'} = \bigcup_i \chi(\Delta^k_i)$, es decir $K^k_{n'}$ es la unión disjunta de las subdivisiones cromáticas de cada configuración de entrada Δ^k_i. $K^k_{n'}$ contiene todos los simplejos de dimensión k y de menor dimensión, es decir $K^k_{n'}$ es el esqueleto de dimensión k de $\chi(\Delta^n)$.

Por otra parte $K^k_{n'}$ representa todas las posibles ejecuciones en la llamada donde participan k' procesos, finalmente consideremos la llamada donde participan n' procesos, y consideremos la ejecución donde los n' terminan en la linea 2. Por el lema 5 la ejecución representa un simplejo τ^n, para nuestra configuración de n' solo se forma un simplejo τ^n que en particular es un elemento del complejo $U^n_{n'}$. Al aplicar $ChrJoin(U^n_{n'} * K^k_{n'})$, tenemos que es una buena coloración, debido a que *ChrJoin* respeta los axiomas I y II. Finalmente podemos ver que el número de vértices esta dado por la ecuación 4.1 la cual es igual a $2^n(n+1)$, por el 4, por lo tanto tenemos una subdivisión cromática de Δ^n. □

4.3. La topología del renombramiento

En el Algoritmo 2, la llamada de la línea 1, invocación del *immediate snapshot*, nos regresa un simplejo de dimensión n, el proceso o procesos que terminen en la línea 3 obtienen un nuevo nombre y no participan en una nueva llamada recursiva.

Se sabe que al menos un proceso termina, y a lo más $n' - 1$ procesos ejecutarán la línea 4. En esta línea se crean a lo más $n' - 1$ llamadas

$isRENAMING_{tag,N}$, las etiquetas están dadas por $N = |view| - 1$, y cada etiqueta representa el número de procesos que se esperan en la nueva llamada.

Del algoritmo **IS**, sabemos que éste nos induce una subdivisión cromática, por ejemplo: para tres procesos como el de la Figura 4.1, sólo que ahora por cada simplejo σ^2 de $\chi(\Delta^2)$, se vuelve a crear una subdivisión cromática de la siguiente forma:

- Sea $k' = k+1$ el número de procesos con vista de la misma cardinalidad, los que no terminaron en la llamada etiquetada con *tag* son $k' - 1$. Los $k' - 1$ procesos ejecutarán una nueva llamada creando una subdivisión cromática $\chi(\Delta^{k-1})$.
- Los procesos que participen solos en alguna llamada recursiva no modifican la subdivisión cromática.

Por ejemplo, consideremos la subdivisión cromática de Δ^2 mostrada en la Figura 4.7, esta subdivisión representa todas las posibles ejecuciones en la primera llamada de $isRENAMING_3(Inicio, Direccion)$.

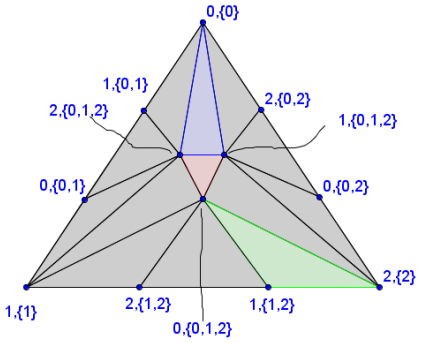


Figura 4.7: Subdivisión cromática de Δ^2, considerando 3 posibles ejecuciones en la primera llamada del renombramiento.

Por ejemplo, consideremos los siguientes simplejos de dimensión 2:
$\sigma = \{(0, \{0\}), (1, \{0, 1, 2\}), (2, \{0, 1, 2\})\}$, de color azul.
$\tau = \{(0, \{0, 1, 2\}), (1, \{0, 1, 2\}), (2, \{0, 1, 2\})\}$, de color rojo.
$\delta = \{(0, \{0, 1, 2\}), (1, \{1, 2\}), (2, \{2\})\}$, de color verde.

El simplejo σ representa una posible ejecución de la primera llamada, en la cual p_0 termina por verse solo, p_2 termina por ser el que tiene el mayor de los identificadores de su vista y finalmente p_1 invocará $isRENAMING_{3,2}(Inicio, Direccion)$, por lo cual σ quedará finalmente definido como:

$$\sigma = \{(0, \{0\}), (1, \{0, 1, 2\}, \{1\}), (2, \{0, 1, 2\})\}.$$

En el caso del simplejo τ el cual representa otra posible ejecución de $isRENAMING_3(Inicio, Direccion)$, tenemos que p_2 termina por ser el que cuenta con el mayor de los identificadores, y que p_1 y p_0 en esta ocasión ejecutan la misma llamada recursiva $isRENAMING_{3,2}(Inicio, Direccion)$, por lo que el *immediate snapshot* de esta llamada subdivide el simplejo $\{(0, \{0, 1, 2\}), (1, \{0, 1, 2\})\}$ en una trayectoria de longitud 3. Donde los posibles casos son: a) p_0 se ejecuta solo y termina por lo que p_1 necesariamente termina, b) p_0 y p_1 se ven, por lo que p_0 volvería a invocar otra llamada recursiva, c) p_1 se ejecuta solo y termina lo cual implica que p_0 vuelve a hacer una llamada recursiva, y así se formaría el complejo dado por:

$$\begin{aligned} C = \{&\{(0, \{0, 1, 2\}, \{0\}), (1, \{0, 1, 2\}, \{0, 1\})\}, \\ &\{(1, \{0, 1, 2\}, \{0, 1\}), (0, \{0, 1, 2\}, \{0, 1\}, \{0\})\}, \\ &\{(0, \{0, 1, 2\}, \{0, 1\}, \{0\}), (1, \{0, 1, 2\}, \{1\})\}\}. \end{aligned}$$

En este caso C es un complejo simplicial puro de dimensión 1, por lo que cada simplejo de dimensión 1 lo asociamos con el simplejo de dimensión 0, $(2, \{0, 1, 2\})$, y de esta manera nos queda subdividido el simplejo τ.

Finalmente, para el caso del simplejo δ tenemos que p_2 termina en la primera llamada por verse solo, p_1 ejecuta una llamada recursiva etiquetada con $tag1$ y p_0 ejecutaría otra llamada recursiva etiquetada con $tag2$, por lo que al final el simplejo queda etiquetado de la siguiente manera.

$$\delta' = \{(0, \{0, 1, 2\}, \{0\}), (1, \{1, 2\}, \{1\}), (2, \{2\})\}$$

Finalmente el complejo simplicial que representaría la ejecución del Algoritmo 2 se muestra en la Figura 4.8.

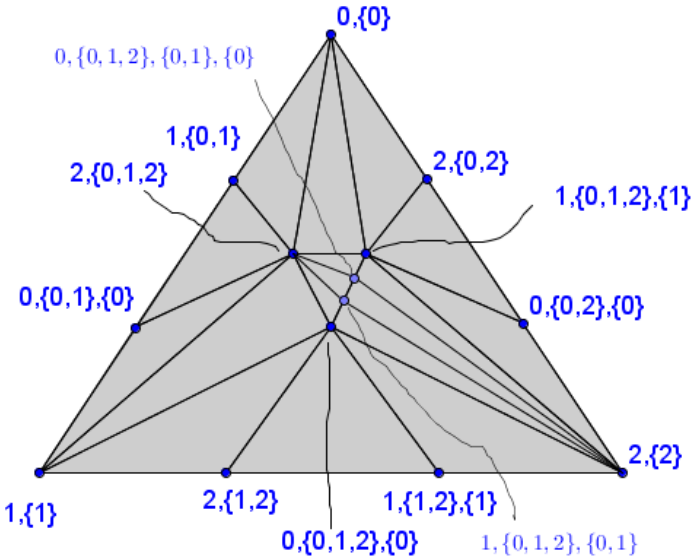


Figura 4.8: Subdivisión cromática para el algoritmo del renombramiento.

En la Figura 4.8, podemos ver que las etiquetas de los vértices nos indican cuantas llamadas recursivas ejecutó cada proceso, por ejemplo: el proceso que más llamadas realizó es el que está etiquetado con $(0, \{0, 1, 2\}, \{0, 1\}, \{0\})$, ésto nos indica que ejecutó las llamadas recursivas $isRENAMING_3$, $isRENAMING_{3,2}$, $isRENAMING_{3,2,1}$ y terminó.

4.4. La topología de la tarea de intercambio.

Como se vio en el capítulo anterior, el Algoritmo 3 resuelve el problema del intercambio, *(swap)*, de manera recursiva. En esta sección se verá que el algoritmo no induce una subdivisión cromática. Retomando los resultados de Herlihy y Shavit en [17], se mostrará que la tarea no se puede resolver únicamente con registros de lectura y escritura, esta demostración será mediante los complejos que son inducidos por los algoritmos que resuelven una tarea únicamente con registros de lectura y escritura.

Como parte de la demostración se referirán a la definición de mapeo simplicial y al teorema principal dado por Herlihy y Shavit en [17].

Definición 1 *Sean K y L dos complejos, posiblemente de diferente dimensión. Un mapeo de vértices $\mu : Skel^0(K) \to Skel^0(L)$ lleva vértices de K a*

vértices de L. μ es un mapeo simplicial si también lleva simplejos de K a simplejos de L.

Teorema 6 (Teorema 3.1 de [17]) *Una tarea de decisión $\langle \mathcal{I}, \mathcal{O}, \Delta \rangle$ tiene un protocolo* libre de espera *usando memoria de lectura y escritura si y solo si existe una subdivisión cromática χ del complejo C y un mapeo simplicial que preserve colores $\mu : \chi(C) \to O$.*

También se hablará de mapeos portadores, mapeos de decisión y protocolo. Estos se definen en [13] de la siguiente manera:

Definición 8 *Sean G y H complejos simpliciales, un mapeo portador $\phi : G \to 2^H$ lleva cada simplejo de $\sigma \in G$ a un subcomplejo $\phi(\sigma)$ de H tal que para todo $\sigma, \tau \in G$, si $\sigma \subseteq \tau$ entonces $\phi(\sigma) \subseteq \phi(\tau)$.*

Definición 9 *Un protocolo para n' procesos es una terna $(\mathcal{I}, \mathcal{P}, \Xi)$ donde:*

- *$\mathcal{I}$ es un complejo puro de dimensión n cromático.*
- *$\mathcal{P}$ es un complejo puro de dimensión n cromático.*
- *$\Xi : \mathcal{I} \to 2^{\mathcal{P}}$, es un mapa portador estricto, es decir lleva simplejos de dimensión n a simplejos de dimensión n, tal que $\mathcal{P} = \bigcup_{\sigma \in \mathcal{I}} \Xi(\sigma)$.*

Definición 10 *Un mapeo simplicial también es un mapeo de decisión δ si respeta el mapa portador (especificación de la tarea) Δ, es decir, satisface $\delta(\phi(\sigma)) \subseteq \Delta(\sigma)$.*

Ahora se analizará el Algoritmo 3 y se mostrará el complejo simplicial que induce para dos y tres procesos.

Se debe recordar que la etiqueta inicial es -1. Si se tiene un sólo proceso, trivialmente se sabe que éste obtendrá la etiqueta $\pi = -1$, y el complejo inducido será únicamente un vértice. Ahora, pensando en que participarán 2 procesos se tienen las siguientes ejecuciones posibles.

$$C^1 = \{(0, \{\pi\}), (1, \{0\}), \{(0, \{\pi\}), (1, \{0\})\}\}$$
$$D^1 = \{(1, \{\pi\}), (0, \{1\}), \{(1, \{\pi\}), (0, \{1\})\}\}$$

De manera gráfica lo podemos ver en la Figura 4.9. Se puede observar cómo sólo un proceso gana la etiqueta π y el otro obtiene un nuevo nombre. Para este problema las etiquetas de los vértices de los complejos son de la

Figura 4.9: Complejo de salida para el problema del intercambio entre 2 procesos.

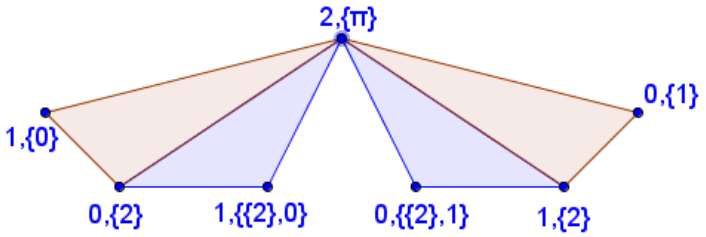


Figura 4.10: Componente conectada por trayectoria, después de ejecutar SWAP donde el proceso 2 gana π.

forma (p_i, V_i), donde V_i es el conjunto de etiquetas visto por cada proceso, de manera que si $V_i = \{v_1, v_2, .., v_k\}$, v_k será la etiqueta obtenida por el proceso en su última llamada recursiva. Nótese que esto sólo se hace por análisis, al inicio de cada llamada recursiva los procesos que participan sólo conocen su identificador y la etiqueta de la llamada.

Como se mencionó anteriormente el algoritmo *SWAP* utiliza registros 2-consenso, esto hace que siempre rompa la simetría de las ejecuciones, es decir, para n' procesos, al menos 1 proceso no verá lo mismo que los otros $n' - 1$ procesos. Como vimos en la tarea de *immediate snapshot* hay ejecuciones en las que n' procesos ven n', es a lo que se llama simetría.

Para construir el complejo, será conveniente hacer énfasis en las ejecuciones en las cuales k' procesos obtienen el mismo *tag*. En el caso en el que obtienen diferente *tag*, del algoritmo sabemos que éstos procesos ejecutan una nueva llamada y terminan. La Figura 4.9 es el ejemplo más básico donde no hay simetría, además, al tener más poder de cómputo con registros 2-consenso, resulta un complejo con dos componentes *conectadas por trayectoria*.

Definición 11 *Un complejo K es* conectado por trayectoria *si hay una trayectoria entre cada dos vértices. Los subcomplejos más grandes de K que son*

conectados por trayectorias *son* componentes conectadas por trayectoria *de* K.

Se continuará con la definición de la tarea *intercambio* para 3 procesos. Sin perdida de generalidad, consideremos el complejo de entrada $\mathcal{I}$ que consiste en el simplejo $\sigma^2 = \{(0,\{0\}),(1,\{1\}),(2,\{2\})\}$ que es portado por Δ de la siguiente manera:

- $\Delta(\{(0,\{0\})\}) = \{(0,\{\pi\})\}$.
- $\Delta(\{(1,\{1\})\}) = \{(1,\{\pi\})\}$.
- $\Delta(\{(2,\{2\})\}) = \{(2,\{\pi\})\}$.
- $\Delta(\{(0,\{0\}),(1,\{1\})\}) = \{(0,\{\pi\}),(1,\{\pi\}),(\{(0,\{\pi\}),(1,\{0\})\}),$
  $(\{(0,\{1\}),(1,\{\pi\})\})\}$.
- $\Delta(\{(0,\{0\}),(2,\{2\})\}) = \{(0,\{\pi\}),(2,\{\pi\}),(\{(0,\{\pi\}),(2,\{0\})\}),$
  $(\{(0,\{2\}),(2,\{\pi\})\})\}$.
- $\Delta((\{(0,\{0\}),(2,\{2\})\}) = \{(1,\{\pi\}),(2,\{\pi\}),(\{(1,\{\pi\}),(2,\{1\})\}),$
  $(\{(1,\{2\}),(2,\{\pi\})\})\}$.
- $\Delta(\{(0,\{0\}),(1,\{1\}),(2,\{2\})\}) = \{(0,\{\pi\}),(1,\{\pi\}),(2,\{\pi\}),$
  $(\{(0,\{\pi\}),(1,\{0\})\}),$
  $(\{(0,\{1\}),(1,\{\pi\})\}),$
  $(\{(0,\{\pi\}),(2,\{0\})\}),$
  $(\{(0,\{2\}),(2,\{\pi\})\}),$
  $(\{(1,\{\pi\}),(2,\{1\})\}),$
  $(\{(1,\{2\}),(2,\{\pi\})\}),$
  $(\{(0,\{\pi\}),(1,\{0\}),(2,\{1\})\}),$
  $(\{(0,\{\pi\}),(1,\{2\}),(2,\{0\})\}),$
  $(\{(1,\{\pi\}),(0,\{1\}),(2,\{0\})\}),$
  $(\{(1,\{\pi\}),(0,\{2\}),(2,\{1\})\}),$
  $(\{(2,\{\pi\}),(0,\{1\}),(1,\{2\})\}),$
  $(\{(2,\{\pi\}),(0,\{2\}),(1,\{0\})\})\}$.

La representación gráfica se puede observar en la Figura 4.12 donde se tiene solo una componente conectada por trayectoria. Para construir el complejo, vamos a partir del formado por dos procesos, además se considerará el subcomplejo obtenido a partir del ganador de la primera llamada. Partiendo del

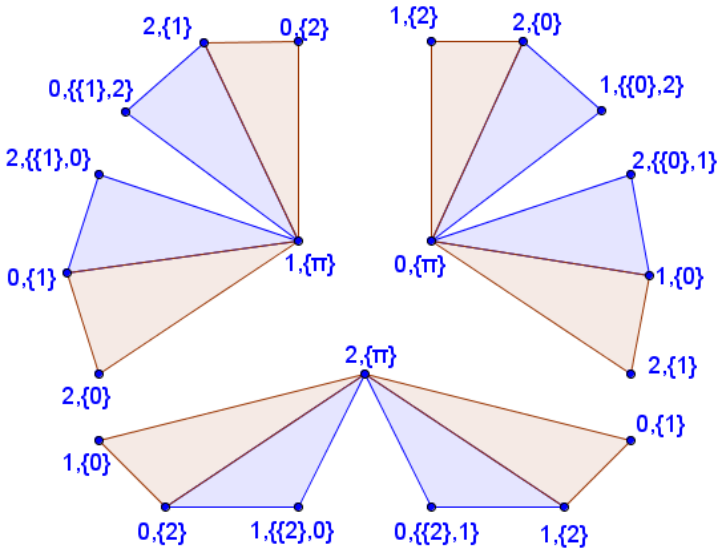


Figura 4.11: Complejo para 3 procesos generado por el algoritmo SWAP.

complejo de la Figura 4.9, agregando otro proceso y suponiendo que gana la etiqueta π entonces se tendrá el subcomplejo mostrado en la Figura 4.10. Los triángulos azules son obtenidos a partir del complejo de 2 procesos, y los triángulos rojos son nuevas ejecuciones posibles. Puede ser el caso que el proceso 2 gane π, 0 vea a 2 y 1 vea a 0. Los triángulos azules resultan de la ejecución de la primera llamada donde 0 y 1 ven lo mismo, es decir al ganador, por lo cual se debe romper la simetría para que ambos terminen con diferentes identificadores.

Se tiene un subcomplejo similar al de la Figura 4.10 por cada par $(i, \{\pi\})$ y sabemos que todas las posibles ejecuciones son representadas por un complejo conectado por trayectoria, pues comparten estados en común, por ejemplo, es posible que el proceso 1 vea 0 cuando 0 fue el segundo ó cuando 0 fue el primero. En la Figura 4.11 se muestra el complejo generado por el algoritmo SWAP para 3 procesos, es importante mencionar que para facilitar la visualización mostramos por separado los subcomplejos, pero los vértices con una misma etiqueta representan solamente a uno.

Nuevamente si se agrega otro proceso al complejo de la Figura 4.11 se tendrían tetraedros y subcomplejos similares para cada par $(i, \{\pi\})$.

Para 3 procesos, como complejo de entrada se tiene un triángulo, éste es

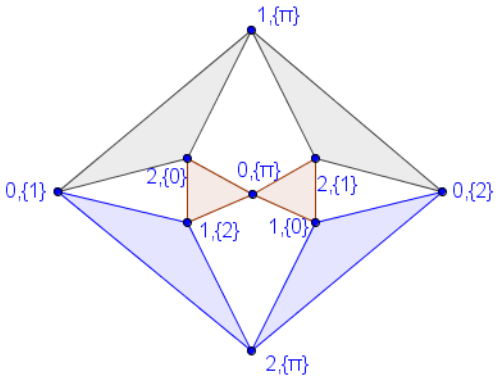


Figura 4.12: Complejo de salida para el problema del intercambio entre 3 procesos.

transformado en el complejo que se muestra en la Figura 4.12. Notemos que un mundo posible, que aparentemente cumple con la especificación de la tarea es en el que los vértices están etiquetados con $(0, \{2\}), (1, \{0\}), (2, \{1\})$, todos tienen un nombre diferente, pero realmente no se considera el 2-simplejo que forman, pues al menos uno debe de tener π, éste lo consideramos como un hoyo.

En la literatura se encuentra que los algoritmos que utilizan registros de lectura y escritura para resolver una tarea generan un complejo sin hoyos, por el Teorema 6, también se puede saber si una tarea se puede resolver en un modelo y si existe un algoritmo que genere un complejo de protocolo que pueda ser mapeado a la especificación de salida.

En la Figura 4.13 podemos ver el complejo generado por el algoritmo que resuelve el *immediate snapshot*, que referiremos ahora como el complejo de protocolo. Si ésta tarea pudiese resolver el *intercambio* entonces cada simplejo del complejo de protocolo debería poderse mapear a un simplejo en el complejo de la especificación del *intercambio*.

Lema 6 *El problema del intercambio no se puede resolver mediante un modelo de immediate snapshot.*

Prueba. Para realizar la prueba basta con demostrar que no existe un mapeo del complejo de protocolo al complejo de salida dado por la especificación de la tarea.

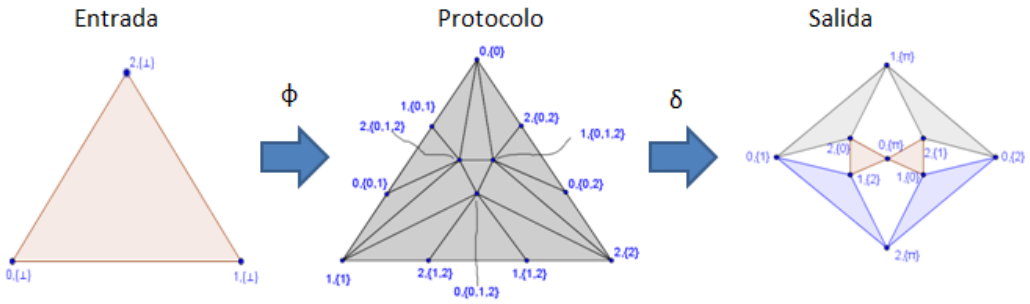


Figura 4.13: Serie de mapeos que deberían de existir para poder resolver el problema del intercambio con IS. ϕ es el mapeo de protocolo, y δ es el mapeo de decisión, donde cada simplejo del protocolo debe estar relacionado con un simplejo del complejo de salida respetando Δ.

Supongamos por contradicción que existe un mapeo de decisión $\delta : \mathcal{P} \to \mathcal{O}$, donde $\mathcal{P}$ es el complejo formado por el algoritmo que resuelve la tarea de *immediate snapshot* para 3 procesos, que nos lleva del complejo de protocolo $\mathcal{P}$ al complejo de salida $\mathcal{O}$ y que preserva nombres, es decir, lleva cada simplejo de $\mathcal{P}$ a $\mathcal{O}$.

Por el Teorema 5 sabemos que un modelo como lo es *immediate snapshot*, que utiliza registros de lectura y escritura, genera una subdivisión cromática la cual no contiene hoyos.

Por el Lema 5 sabemos que en la subdivisión tenemos un simplejo σ tal que la vista en cada vértice es la misma. Este simplejo representa el mundo en el que los procesos se ejecutan de manera concurrente y todos ven lo mismo, es decir el conocimiento local de cada proceso es el mismo. Nuestro mapeo de decisión debe mapear este simplejo a un simplejo en el complejo de salida, tal que los procesos tengan el mismo conocimiento local. El complejo del intercambio con 2 procesos, resulta en un complejo con dos componentes, y en el *immediate snapshot* sería una arista subdividida en 3 aristas, lo cual contradice nuestra suposición de que existe un mapeo $\delta : \mathcal{P} \to \mathcal{O}$, pues δ debe mapear cada simplejo de $\mathcal{P}$ a cada simplejo de $\mathcal{O}$.

□

En la Figura 4.14 se puede ver el complejo para tres procesos derivado de la ejecución y que se puede mapear al complejo de la especificación de salida para el problema del intercambio con tres procesos. Sabemos que el algoritmo SWAP es correcto y otra forma de demostrarlo es mediante la relación de

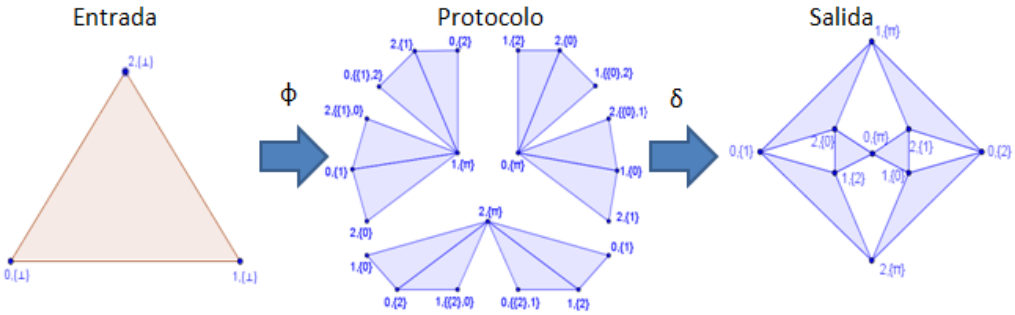


Figura 4.14: Serie de mapeos existen para poder resolver el problema del intercambio con SWAP. ϕ es el mapeo de protocolo, y δ es el mapeo de decisión, donde cada simplejo del protocolo esta relacionado con un simplejo del complejo de salida.

los complejos generados. El mapeo de decisión δ llevaría cada vértice de la forma $(i, \{v_0, v_1, ..., v_k\})$ a un vértice en el complejo de salida $(i, \{v_r\})$, tal que $v_k = v_r$.

Capítulo 5

Conclusiones

En este capítulo se presentan las conclusiones obtenidas de esta Tesis y los posibles trabajos a futuro que se pueden realizar.

5.1. Aportaciones teóricas

Hemos visto que construir una subdivisión cromática recursivamente de manera secuencial y resolver la tarea de *immediate snapshot* recursivamente están estrechamente relacionados. El algoritmo recursivo del *immediate snapshot* es en algún sentido una implementación concurrente del algoritmo de subdivisión. Además, hemos dado la función $ChrJoin$, aprovechando las características recursivas de los complejos, que nos permite construir el complejo inducido por el algoritmo del *immediate snapshot*. También se da la función $Skel^k_{\sigma^k}(C)$ que relaciona un simplejo con los simplejos de un complejo para poderlos relacionar.

En un algoritmo distribuido existen muchas ejecuciones posibles, una por cada traslape de las operaciones de los procesos. Cada una de las ejecuciones construye uno de los simplejos de la subdivisión. Es interesante considerar la interpretación distribuida de la operación unión cromática. La unión cromática entre dos complejos $K^{k-1}_{n'}$ y $U^k_{n'}$ se puede realizar porque las vistas que surgen en el algoritmo distribuido son compatibles. Las vistas en los simplejos del complejo $K^{k-1}_{n'}$, son de los procesos rápidos y pueden ser extendidas a las vistas de los simplejos $U^k_{n'}$ de los procesos lentos que participan después en la ejecución.

Por otra parte se tiene, que si un algoritmo distribuido recursivo se ra-

mifica o es una simple trayectoria, no afecta en la forma de construir los complejos, y apoyándonos de las etiquetas que le demos a éstos complejos, nos ayudan a conocer el universo de ejecuciones cada algoritmo, permitiéndonos un análisis detallado de cada algoritmo.

También se mostró el complejo que induce el problema del intercambio para tres procesos, cómo éste tiene otras características y no puede ser resuelto únicamente con registros de lectura y escritura. Además, se dio una demostración de que el Algoritmo 3 no es linearizable, algo que sólo se menciona en la literatura pero no se da una demostración.

Además, en el problema del intercambio se muestra como el usar registros 2-consenso resulta en un complejo de dos componentes para dos procesos. En general ésto se extiende a que para más procesos no tendremos en una misma ejecución dos procesos que vean lo mismo. Se dio un mapeo de decisión para mostrar que el complejo que resultó del análisis del algoritmo se puede relacionar con el complejo de salida dado por la especificación de la tarea.

Finalmente, se muestra que aprovechando las propiedades recursivas de los complejos simpliciales, el análisis de los algoritmos resulta ser más intuitivo, por ejemplo en el problema del intercambio, y es una herramienta útil para estudiar algoritmos distribuidos recursivos.

5.2. Trabajo a futuro

Estableciendo y conociendo más a detalle la relación entre el cómputo distribuido y la topología, sería interesante investigar si cada algoritmo distribuido recursivo que genera un complejo simplicial puede ser transformado en un algoritmo distribuido iterativo, y si es posible que a partir de estas relaciones surja una estructura de datos general que sirva como base para este propósito.

También se puede estudiar la relación que hay entre las tareas cromáticas y no cromáticas, ya que por el lado de la topología basta con "*aplastar*" simplejos de un complejo cromático para obtener una representación no cromática de un complejo que en cómputo distribuido representaría únicamente los valores de las entradas. Hay que recordar que en las tareas cromáticas siempre es indispensable conocer el par (p_i, v_i) donde p_i es el nombre del proceso y v_i es su valor de entrada.

En [2] se muestra un algoritmo para resolver el problema del intercambio con concurrencia ilimitada. Sería interesante investigar la relación entre la

topología y la concurrencia ilimitada.

Bibliografía

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.

[2] Yehuda Afek, Adam Morrison, and Guy Wertheim. From bounded to unbounded concurrency objects and back. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 119–128, New York, NY, USA, 2011. ACM.

[3] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, PODC '93, pages 159–170, New York, NY, USA, 1993. ACM.

[4] Paul Alexandroff. Simpliziale approximationen in der allgemeinen topologie. *Mathematische Annalen*, 96(1):489–511, 1927.

[5] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, September 1994.

[6] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rudiger Reischuk. Renaming in an Asynchronous Environment. *Journal of the ACM*, July 1990.

[7] Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.

[8] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC '93: Proceedings of*

the twenty-fifth annual ACM symposium on Theory of computing, pages 91–100, New York, NY, USA, 1993. ACM.

[9] Elizabeth Borowsky and Eli Gafni. A Simple Algorithmically Reasoned Characterization of Wait-Free Computations (Extended Abstract). In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 189–198, New York, NY, USA, 1997. ACM.

[10] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[11] Eli Gafni and Sergio Rajsbaum. Distributed programming with tasks. In Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah, editors, *Principles of Distributed Systems*, volume 6490 of *Lecture Notes in Computer Science*, pages 205–218. Springer Berlin Heidelberg, 2010.

[12] Eli Gafni and Sergio Rajsbaum. Recursion in distributed computing. In Shlomi Dolev, Jorge Cobb, Michael Fischer, and Moti Yung, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6366 of *Lecture Notes in Computer Science*, pages 362–376. Springer Berlin Heidelberg, 2010.

[13] M. Herlihy, D. Kozlov, and S. Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Elsevier Science, 2013.

[14] Maurice Herlihy, Sergio Rajsbaum, and Michel Raynal. Computability in distributed computing: a tutorial. *SIGACT News*, 43(3):88–110, August 2012.

[15] Maurice Herlihy, Sergio Rajsbaum, and Michel Raynal. Power and limits of distributed computing shared memory models. *Theoretical Computer Science*, 509(0):3 – 24, 2013. ¡ce:title¿Structural Information and Communication Complexity¡/ce:title¿.

[16] Maurice Herlihy, Sergio Rajsbaum, and Mark R. Tuttle. Unifying synchronous and asynchronous message-passing models. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 133–142, New York, NY, USA, 1998. ACM.

[17] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999.

[18] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[19] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[20] Dmitry N. Kozlov. Combinatorial algebraic topology. In *Algorithms and Computation in Mathematics*, volume 21, pages 7–15. Springer-Verlag, 2008.

[21] Dmitry N. Kozlov. Chromatic subdivision of a simplicial complex. *Homology, Homotopy and Applications*, 14(2):197–209, 2012.

[22] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[23] Jery Matousek. *Using the Borsuk-Ulam Theorem*. Springer, 2008.

[24] James Munkres. *Elements of Algebraic Topology*. Prentice Hall, 2 edition, January 1984.

[25] Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer Publishing Company, Incorporated, 2012.

[26] Michael Saks and Fotios Zaharoglou. Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM J. Comput.*, 29(5):1449–1483, 2000.

[27] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, November 1996.

[28] I. Stojmenovic. Recursive algorithms in computer science courses: Fibonacci numbers and binomial coefficients. *IEEE Trans. on Educ.*, 43(3):273–276, August 2000.

[29] H. Weisman. Implementing shared memory overwriting objects. Master's thesis, Tel-Aviv University, Tel Aviv, Israel, 1994.