VNIVERSIDAD NACIONAL
AVTNOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Posgrado en Ciencias (Computación)

Verificación del Número de Configuraciones (19_4) en Paralelo

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

PRESENTA:
Víctor Andrés Hernández Patiño

DIRECTOR DE TESIS
JOSÉ DAVID FLORES PEÑALOSA
Facultad de Ciencias

CO-DIRECTOR DE TESIS
RODOLFO SAN AGUSTÍN CHI
Facultad de Ciencias

Ciudad de México, junio 2016

1. Agradecimientos

Agradezco al Consejo de Ciencias y Tecnología (CONACYT), a nuestra querida casa de estudios, la Universidad Nacional Autónoma de México, al Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas por su apoyo y patrocinio.

En especial doy gracias a David Flores Peñalosa por la revisión y dirección de la tesis. A Rodolfo San Agustín Chi por su asesoría.

Y a todos mis seres queridos, que me han acompañado y animado en esta empresa.

Índice general

Capítulo 1

Introducción

1. Objetivo

Un problema frecuente en la combinatoria es averiguar si existe algún objeto que tenga propiedades dadas. Otro más amplio, pero también recurrente, es enumerar todos los objetos (no isomorfos) que tengan dichas propiedades. El propósito de esta tesis es presentar una técnica para enumerar configuraciones (n_k). En particular, enumerar configuraciones (19_3), (20_3), (19_4) y (20_4).

2. Configuraciones (n_k)

Por una configuración (n_k) nos referimos a un conjunto de n puntos y n líneas tal que cada punto está precisamente en k líneas y cada línea contiene exactamente k de estos puntos. Además, cada par de puntos está en a lo más una línea.

Si $k = 1$ entonces se tienen n líneas y n puntos, tal que cada punto está en exactamente una línea y cada línea contiene exactamente un punto. Por lo que se tendría algo como en la figura 1.

FIGURA 1. Configuración (5_1)

Ahora sea $k = 2$. La figura 2 **no** es una configuración (n_2) porque ambas líneas tienen los mismos dos puntos y en las configuraciones (n_k) se pide que ningún par de puntos puede estar en dos líneas a la vez. Entonces en una configuración (n_2) cada punto está conectado a otros

dos puntos a través de dos líneas. Por lo que la configuración (n_2) más simple posible es un triángulo. Las configuraciones (n_2) serían grupos

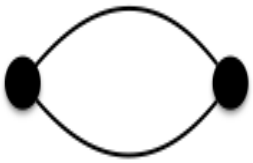

FIGURA 2. Configuración que no es (n_2)

de polígonos cerrados tal que la suma de sus vértices sería n. Una configuración (n_2) podría ser un hexágono o dos triángulos y mientras n sea más grande más opciones aparecerán.

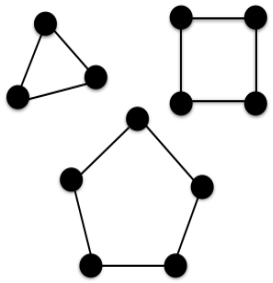

FIGURA 3. Configuracióones (n_2)

Esta tesis se enfocará en las configuraciones (n_3) y (n_4). El ejemplo más simple de estas serían las configuraciones (7_3) (ver la figura 4). Pero ¿Cómo sabemos que las configuraciones (7_3) son las más simples? ¿Existe una configuración (6_3)? Suponga que tenemos una configuración (n_k). Piense en un punto cualquiera en esta configuración, digamos el punto cero. El punto cero tiene que estar en 3 líneas por definición de (n_3). Como cada línea tiene 3 puntos, el punto cero tiene 2 puntos adyacentes en cada una de estas tres líneas. Por otro lado, cada par de puntos sólo puede estar en a lo más una línea. Por lo tanto, esos puntos adyacentes deben ser todos diferentes. Entonces el punto cero tiene 6 puntos adyacentes. Contando al punto cero, la configuración n_3 tiene un mínimo de 7 puntos. Se retomará esto de manera más general y completa en el capitulo 2.

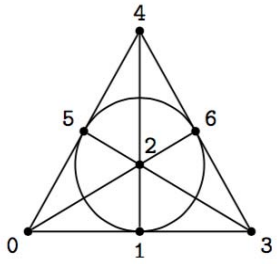

FIGURA 4. Plano de Fano

Resulta que sólo existe una configuración (7_3) y se le llama el plano de Fano. Cualquier otra que se pueda construir resultara isomorfa al plano de Fano ¿Como lo sabemos? Sean $\{0,1,2,3,4,5,6\}$ los puntos de la configuración (7_3). Por lo dicho en el párrafo anterior, tres de las líneas serían $\{0,1,2\}$, $\{0,3,4\}$, $\{0,5,6\}$. El 1 y el 2 también necesitan estar en otras dos líneas sin repetir pares. Por tanto, se tiene algo como $\{0,1,2\}$, $\{0,3,4\}$, $\{0,5,6\}$, $\{1,\ ,\ \}$, $\{1,\ ,\ \}$, $\{2,\ ,\ \}$, $\{2,\ ,\ \}$, en donde hemos dejado en blanco los espacios de las últimas 4 líneas para analizar todas las formas en las que los podemos llenar. Si avanzamos con el 3 y el 4, tenemos $\{0,1,2\}$, $\{0,3,4\}$, $\{0,5,6\}$, $\{1,3,\ \}$, $\{1,4,\ \}$, $\{2,3,\ \}$, $\{2,4,\ \}$. Con el 5 y el 6 hay dos opciones $\{0,1,2\}$, $\{0,3,4\}$, $\{0,5,6\}$, $\{1,3,5\}$, $\{1,4,6\}$, $\{2,3,6\}$, $\{2,4,5\}$ o bien $\{0,1,2\}$, $\{0,3,4\}$, $\{0,5,6\}$, $\{1,3,6\}$, $\{1,4,5\}$, $\{2,3,5\}$, $\{2,4,6\}$, pero estas dos opciones son isomorfas: solo se necesita intercambiar el 5 con el 6 para obtener una a partir de la otra.

En la figura 4 puede verse que las líneas no son necesariamente rectas. Esto es porque en esta tesis se consideran a las configuraciones (n_k) como diseños combinatorios y las líneas son solo subconjuntos de puntos.

La tabla 1 contiene los resultados proveniente del libro de Branko Grunbaum (2009) [**3**, pag 69] de precisamente el número de configuraciones (n_3) calculados previamente con otros métodos. $\#_c(n)$, $\#_t(n)$ y $\#_g(n)$ son el número de configuraciones no isomorfas en un sentido combinatorio, topológico y geométrico respectivamente. Puede apreciarse que conforme n aumenta, el número de configuraciones aumenta exponencialmente. Con $n = 16$ ya tenemos millones de configuraciones y con $n = 19$ se tienen miles de millones.

n	$\#_c(n)$	$\#_t(n)$	$\#_g(n)$
≤ 6	0	0	0
7	1	0	0
8	1	0	0
9	3	3	3
10	10	10	9
11	31	31	31
12	229	229	229
13	2,036		
14	21,399		
15	245,342		
16	3,004,881		
17	38,904,499		
18	530,452,205		
19	7,640,941,062		

TABLA 1. Resultado de trabajos anteriores

3. Metodología

A las configuraciones (n_k) se van a representar con un arreglo de n enteros(integers), llamémosle $\mathscr{V}$. Cada entero representa una línea y cada bit marcado como 1 en ese entero representa un punto perteneciente a esa línea. Ese arreglo se llenará con backtracking y se guardarán todas las combinaciones válidas para poder llevar la cuenta de las no isomorfas. Se usarán diversas técnicas de cómputo y se aprovecharán propiedades de las configuraciones (n_k) para acelerar el proceso.

Para poder trabajar con todas las variaciones sistemáticamente se elige un orden para los puntos y las líneas. Primero se etiquetan los puntos con los números del 1 al n y les aplico el orden usual. A cada fila de $\mathscr{V}$ se les exigió que sus puntos estén ordenados de menor a mayor para evitar representar la misma línea dos veces. Para las líneas se elige el orden lexicográfico con los puntos, ejemplos:

La línea con los puntos (3,6,9) es menor que la que contiene puntos (4, 5, 9), y la línea que contiene los puntos (3,4,7) es menor que la que contiene los puntos (3, 5, 7). Lo que hago entonces es llenar $\mathscr{V}$ en orden. Los puntos en una misma línea deben de estar en orden de menor a mayor y las líneas también tienen que quedar en orden de menor a mayor según el orden que se acaba de establecer.

Con este orden se evita trabajar con la misma configuración dos veces. Además, produce propiedades que ayuda a acelerar el backtraking. Pues se disminuye el número de soluciones parciales que se deben procesar. Por ejemplo digamos que se va llenando el arreglo $\mathscr{V}$, y las líneas que quedan por llenar están todas vacías. El punto de menor valor que aún no ha sido colocado k veces en $\mathscr{V}$ tendrá que colocarse tarde o temprano y por el orden que se definió la línea mas chica de las líneas que aún no se han rellenado tendrá ese punto forzosamente. Por lo que al empezar a llenar una línea se comienza con el punto de menor valor que todavía no se a utilizado k veces. El punto que sigue ya puede ser cualquiera de los demás puntos, pero una vez seleccionado los que siguen en la misma línea tienen que ser mayores a su anterior para conservar el orden, reduciendo las opciones y haciendo el backtraking mas chico.

Pero, aún no se rechazan isomorfismos. Tomemos el punto 0 para (n_3). Sabemos que 0 tendrá que estar en tres líneas. En esas tres líneas los puntos con los que comparte la línea no se pueden repetir porque entonces se repetiría un par en dos líneas. Básicamente las líneas con el 0 lucirían así (0,1,2)(0,3,4)(0,5,6). Cualquier cambio que le haga a los compañeros del 0 produciría un resultado isomorfo. Por lo que siempre tengo estas tres líneas en $\mathscr{V}$. Esto descarta a un enorme grupo de configuraciones. Para detectar a todos los demás isomorfos se utilizará una herramienta llamada Nauty. Para ahorrar más trabajo ha resultado muy efectivo hacer un corte de isomorfismos con soluciones parciales. ¿En qué fase conviene descartar soluciones parciales isomorfas? ha demostrado ser una pregunta clave muy difícil de responder. Un leve cambio aquí puede tener un enorme impacto en la eficiencia de este algoritmo. Se decidió probar diferentes fases al calcular el número de configuraciones n_3 y n_4 con $n \leq 18$ y usar lo aprendido al calcular (19_3), (20_3), (19_4) y (20_4).

Las soluciones tanto totales como parciales se guardan en varios árboles parcialmente ordenados, para que al buscar isomorfismos sea lo más eficiente posible. Luego se generan arreglos ordenados a partir de esos árboles y al último esos arreglos se fusionan, eliminando soluciones que resulten isomorfas.

Capítulo 2

Conceptos básicos

1. Configuraciones (n_k)

Aunque ya se dio un panorama general de lo que se hizo en esta tesis, falta formalizar la teoría y justificar los métodos. Para ello, es conveniente comenzar con definiciones y conceptos básicos que después se refinarán y especializarán.

Una *configuración* C es una familia de p "puntos" (a veces llamados vértices) y una familia de n "líneas", tales que cada "punto" de la familia es "incidente" con precisamente q "líneas", mientras que cada una de las "líneas" es "incidente" con precisamente k "puntos". El uso de comillas en este párrafo es para indicar que estos objetos pueden ser de cualquier clase, mientras la "incidencia" sea una relación que cumpla con las siguientes condiciones:

1. Es una relación simétrica;
2. una "incidencia" solo puede involucrar un "punto" y una "línea";
3. dos "puntos" (o "líneas") pueden ser "incidentes" con a lo más una "línea" (o un "punto").

A las configuraciones se les puede representar con una tabla, donde cada columna representa una línea y cada número en la columna representa un punto incidente a esta línea. Por ejemplo:

0	0	0	1	1	2	2
1	3	5	3	4	3	4
2	4	6	5	6	6	5

es una representación del Plano de Fano (vea figura 4). Una alternativa es:

l_0	l_1	l_2	l_3	l_4	l_5	l_6
0	0	0	1	1	2	2
1	3	5	3	4	3	4
2	4	6	5	6	6	5

Donde los l_i son las líneas y los elementos debajo son los puntos incidentes a esta.

Un **elemento** de una configuración puede ser una "línea" o un "punto". Además dos puntos incidentes a la misma línea se les llaman **colineales**

Una configuración C con **parámetros** p, q, n, k se denotan por (p_q, n_k). Si los valores particulares de p y n no son importantes, se dice que tenemos una **configuración-**$[q, k]$. Si $q = k$, entonces se simplifica la notación denotándola como una **configuración-**k.

Considere una configuración (p_q, n_k). Cada uno de los p puntos tiene exactamente q incidencias. Por lo tanto, tenemos pq incidencias en total. Por otro lado, cada una de las n líneas tiene exactamente k incidencias. Por lo tanto, tenemos nk incidencias en total. En conclusión, una condición necesaria para tener una configuración (p_q, n_k) es que $pq = nk$.

Hay más condiciones necesarias. Cada "punto" de una configuración (p_q, n_k) es "incidente" con q "líneas"; cada una de las cuales es "incidente" con otros $k - 1$ "puntos"; y ninguno de estos "puntos"se puede repetir. De esto resulta que $p \geq q(k-1)+1$. Con un argumento similar, tenemos que $n \geq k(q-1)+1$. Para evitar trivialidades, se asumirá que $q \geq 2$ y $k \geq 2$ y esto, a su vez, implica que $p \geq 3$ y $n \geq 3$. Estas condiciones casi siempre son suficientes para tener una configuración (p_q, n_k), pero hay excepciones.

En esta tesis nos interesan las configuraciones (p_q, n_k) con $p = n$ (y, por lo tanto, $q = k$). Para estas configuraciones se tiene la notación simplificada **configuración** (n_k), o **configuración-**k. Es necesario aclarar que las configuraciones-k se consideran aquí como diseños combinatorios. Esto significa que las "líneas" son familias de "puntos". En este caso, el que un "punto" sea "incidente" con una "línea" significa que ese "punto" pertenece a esa "línea".

2. Configuraciones Parciales

Una manera de construir configuraciones (n_k) es a través de configuraciones parciales.

Una *configuración parcial* P es una familia de "puntos" y una familia de "líneas"tal que, para enteros positivos p, q, n y k cada uno de los

p "puntos"que pertenecen a la familia es "incidente" con a lo más q de las n "líneas", mientras que cada una de estas "líneas" es "incidente" con a lo más k "puntos". Si $p = n$ (y por lo tanto $q = k$) entonces tenemos una configuración parcial (n_k)

La idea es que a partir de una configuración parcial se puedan construir una o más configuraciones-$[q, k]$. Aunque también puede verse como configuraciones-$[q, k]$ que han perdido líneas, puntos y/o incidencias. Si todos los puntos, líneas e incidencias de una configuracíon parcial P están contenidas en otra configuracíon parcial o una configuración-$[q, k]$ P', diremos que P' es una **extensión** de P o que P es una subconfiguración de P'.

Si a los puntos y las líneas de dos configuraciones o configuraciones parciales C y C' se les puede construir una relación biyectiva τ de punto a punto (y de línea a línea) tal que se preserven las incidencias entonces se dice que C y C' son **isomorfas**. En otras palabras C y C' son **isomorfas** si y sólo si existe $\tau : C \to C'$ biyectiva tal que $\forall x, y \in C, x$ es incidente con y si y sólo si $\tau(x)$ es incidente con $\tau(y)$, además x es un punto si y sólo si $\tau(x)$ es un punto. A esta función se le llama **isomorfismo** entre C y C'.

Ejemplo 2.1. Sea

$$C = \begin{array}{|ccccccc|} \hline l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline 0 & 0 & 0 & 1 & 1 & 2 & 2 \\ 1 & 3 & 5 & 3 & 4 & 3 & 4 \\ 2 & 4 & 6 & 5 & 6 & 6 & 5 \\ \hline \end{array}$$

y sea

$$C' = \begin{array}{|ccccccc|} \hline l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \\ \hline 0 & 0 & 0 & 1 & 1 & 2 & 2 \\ 1 & 3 & 5 & 3 & 4 & 4 & 3 \\ 2 & 4 & 6 & 5 & 6 & 5 & 6 \\ \hline \end{array},$$

entonces C y C' son isomorfos. Y

$$\tau(x) = \begin{cases} l_6 & \text{si } x = l_5 \\ l_5 & \text{si } x = l_6 \\ x & \text{si } x \neq l_5 \text{ y } x \neq l_6 0 \end{cases}$$

es el isomorfismo entre C y C'.

Teorema 2.2. *Si dos configuraciones parciales P_1 y P_2 son isomorfas. Por cada extensión E_1 que se pueda construir a partir de P_1 se puede construir una extensión E_2 a partir de P_2 de tal manera que E_1*

es isomorfa a E_2. Además puede construirse un isomorfismo α entre E_1 y E_2 de tal manera que $\alpha(P_1) = P_2$

DEMOSTRACIÓN. Sea A la familia de puntos que pertenecen a E_1 pero no a P_1, sea A' una familia de puntos isomorfa a A y α su isomorfismo. Sea B la familia líneas que pertenecen a E_1 pero no a P_1, sea B' una familia de puntos isomorfa a B y β su isomorfismo. Sea τ el isomorfismo entre P_1 y P_2. Defino $f : E_1 \to P_2 \cup A' \cup B'$ como:

$$f(x) = \begin{cases} \tau(x) & \text{si } x \in P_1 \\ \alpha(x) & \text{si } x \in A \\ \beta(x) & \text{si } x \in B \end{cases}$$

Definimos $E_2 = P_2 \cup A' \cup B'$ donde para toda $x, y \in E_2$ x es incidente con y si y sólo si $f^{-1}(x)$ es incidente con $f^{-1}(y)$. De esta manera E_2 es isomorfo con E_1 y es una extensión de P_2. □

Este resultado es importante porque se eligió generar todas las configuraciones (n_k) no isomorfas extendiendo configuraciones parciales. Por lo que si dos configuraciones parciales resultan ser isomorfas entonces sólo es necesario extender de todas las maneras posibles una de ellas y descartar la otra inmediatamente y gracias a este teorema sabemos que no se perdió ni una sola configuración.

3. Gráficas de Levi

Una **gráfica de Levi** $L(C)$ de una configuración o una configuración parcial C es una gráfica bipartita con puntos "negros" que corresponden a los putos de C y puntos "blancos" que corresponden a las líneas de C; Dos puntos en la gráfica de Levi $L(C)$ determinan una arista si y sólo si uno de los puntos representa un punto de C y el otro representa una línea incidente con este punto.

TEOREMA 3.1. *Dos configuraciones C y D son isomorfas si y sólo si, $L(C)$ y $L(D)$ son isomorfas.*

DEMOSTRACIÓN. Si C y D son isomorfas entones sea $\tau\colon C \to D$ el isomorfismo entre estas configuraciones.

Sean $l_C\colon C \to L(C)$ y $l_D\colon D \to L(D)$ las correspondencias entre las configuraciones y sus gráficas de Levi. Sean $x, y \in L(C)$. Los puntos x, y y forman una arista si, y sólo si, $l_C^{-1}(x)$ y $l_C^{-1}(y)$ son un punto y una línea incidente a ese punto. Esto es equivalente a que $\tau(l_C^{-1}(x))$ y

$\tau(l_C^{-1}(y))$ sean un punto y una línea incidente a ese punto. Por último, esto es equivalente a que $l_D(\tau(l_C^{-1}(x)))$ y $l_D(\tau(l_C^{-1}(y)))$ formen una arista. Tenemos entonces que $L(C)$ y $L(D)$ son isomorfas.

Si $L(C)$ y $L(D)$ son isomorfas entonces existe un isomorfismo $\alpha\colon L(C) \to L(D)$ entre $L(C)$ y $L(D)$. Entonces se puede hacer un argumento equivalente al anterior para mostrar C y D son isomorfas. □

Este resultado es importante porque la manera elegida de determinar si dos configuraciones son isomorfas es a través de Nauty. Pero Nauty maneja gráficas donde cada línea conecta a los más dos vértices. Las líneas en las configuraciones (n_k) son incidentes con k vértices. Pero las líneas de las gráficas de Levi de cualquier configuracion (n_k) conecta solamente dos vértices. Por lo que el teorema anterior nos permite usar Nauty para determinar si dos configuraciones son isomorfas a partir de sus gráficas de Levi.

4. Estrategia inicial

TEOREMA 4.1. *Existen configuraciones* (n_3) *si y sólo si* $n \geq 7 \ldots$.

DEMOSTRACIÓN. Consideré que $(n_3) = (p_q, n_k)$ con $p = n$ y $q = k = 3$. Por fórmula de la sección 1 tenemos que para que una configuración (n_3) pueda existir se necesita que $n \geq 3(3-1)+1$ entonces $n \geq 7$.

Ahora supongamos que $n \geq 7$. Observe la siguiente tabla.

1	2	3	4	...	$n-5$	$n-4$	n-3	$n-2$	$n-1$	n
2	3	4	5	...	$n-4$	$n-3$	n-2	$n-1$	n	1
4	5	6	7	...	$n-2$	$n-1$	n	1	2	3

Cada fila en la tabla contiene cada número del 1 al n. Son tres filas por lo que cada número esta en la tabla tres veces y como las filas están recorridas ninguna columna contiene mas de una vez el mismo número. Puede observarse en la tabla que si i=1, 2, 3, $n-2$, $n-1$ o n entonces todos los compañeros de i en las columnas son todos diferentes. Sea i tal que $4 \leq i \leq n-3$, i aparece en las tres columnas.

$i-3$	$i-1$	i
$i-2$	i	$i+1$
i	$i+2$	$i+3$

Todos los compañeros de i en las columnas son diferentes. Por lo que ningún par de números se repite en las columnas. Entonces la tabla

se le puede considerarse como una representación de una configuración (n_3). □

5. Configuraciones parciales iniciales

Sea C una configuración (n_k). Suponga 0 un punto en la configuración. 0 va a ser incidente con k líneas, cada una de esas líneas van a ser incidentes con otros $k-1$ puntos. Por la definición de incidencia ningún punto colineal a 0 se va a repetir por lo que 0 tiene $k*(k-1)$ puntos colineales. Como se trabajará con $k=3$ o $k=4$ esto significa que 0 tendrá 6 o 12 puntos colineales, nombrémoslos $\{1,2,\ldots,12\}$ de tal manera que las líneas:

$$\begin{array}{ccc} 0 & 0 & 0 \\ 1 & 3 & 5 \\ 2 & 4 & 6 \end{array}$$

o

$$\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{array}$$

estén presentes en C si $k=3$ o 4 respectivamente. Cualquier configuración (n_3) o (n_4) tendrá una subconfiguración isomorfa a estas últimas por lo que se les llamará **base-3** y **base-4**.

Capítulo 3

Técnicas y herramientas computacionales

Como todo proyecto para poder lograr enumerar las configuraciones (n_k) es necesario aplicar diversas técnicas y aprovechar las herramientas disponibles. A continuación se dará una breve introducción a los conceptos, técnicas y herramientas elegidas y como se usarán para lograr el objetivo.

1. Backtraking

Para muchos problemas combinatorios de interés, la solución o soluciones se pueden representar como una lista $X = [x_0, x_1, \ldots, x_n]$ con cada x_i escogida de de un finito conjunto de posibilidades P_i. El algoritmo de backtraking (también conocido Vuelta Atrás y como Depth-First Search o búsqueda en profundidad) básicamente busca las soluciones entre $P_1 \times P_2 \times \cdots \times P_n$. Los x_i son escogidos uno a la vez de 1 a n, y dependiendo del problema es probable que al haber ya elegidos determinados $x_0, x_1, \ldots x_i$ uno pueda darse cuenta que no se podrán escoger $x_{i+1}, x_{i+2} \ldots x_n$ tal que la lista sea una de las soluciones buscadas. En ese momento no tiene caso seguir trabajando con el conjunto $x_0, x_1, \ldots x_i$ y lo que se hace es elegir inmediatamente un nuevo x_i, si no se puede se elige un nuevo x_{i-1} y si no se puede se va retrocediendo hasta agotar las posibilidades.

La idea es encontrar todas las soluciones de interés en $P_1 \times P_2 \times \cdots \times P_n$ sin tener que probar todas las combinaciones pero con una garantía de que no hay una solución diferente entre las combinaciones que no se revisaron.

Hay muchas variaciones del backtraking diferentes y depende del problema cual es el más adecuado. En este trabajo se decidió seguir el siguiente algoritmo:

```
S_1 = P_1, i = 1;
mientras i >0 hacer
    mientras S_i ≠ ∅ hacer
        elegir x_i ∈ S_i;
        S_i = S_i − {x_i};
        si [x_1, x_2, ..., x_i] es una solución parcial entonces
            si [x_1, x_2, ..., x_i] es una solución completa entonces
                Guardar solución;
            en otro caso
                i = i + 1;
                calcular S_i;
            fin
        fin
    fin
    i = i − 1;
fin
```

Algoritmo 1: Backtracking

Por ejemplo digamos queremos calcular todas las configuraciones n_3 no isomorfas. Usando la configuración parcial base-3 se le quiere agregar una línea y que siga siendo una configuración parcial que podría ser parte de una configuración (n_3). Básicamente el objetivo es obtener todas las extensiones posibles de la siguiente forma:

$$\begin{array}{cccc} 0 & 0 & 0 & x_1 \\ 1 & 3 & 5 & x_2 \\ 2 & 4 & 6 & x_3 \end{array}$$

Una posibilidad es considerar a $P_1 = \{0, \ldots, n\}$. Determinar S_i como $S_i = P_1 - \{x_1, x_2, \ldots, x_{i-1}\}$ para todo i tal que $1 < i \leq n$. Al seleccionar x_i, en el paso en que hay que determinar si $[x_1, x_2, \ldots, x_i]$ es solución parcial, sólo habrá que determinar dos condiciones:

1. x_i no se encuentra ya tres veces en el resto de la configuración parcial (como el 0 en este caso);
2. Que ningún par (x_j, x_i) con $0 \leq j < i$ se encuentre ya en el resto de la configuración (como el par (1,2) en este caso).

Si queremos una línea incidente con k puntos entonces una lista X tal que $|X| = k$ es considerada una solución completa. Debido a la definición de configuración parcial estas dos condiciones son suficientes para asegurar que todas las soluciones obtenidas son configuraciones

parciales. Llamémosle a este proceso **agregarlinea** el cual genera varias configuraciones parciales con $l + 1$ líneas a partir de una con l líneas.

También es posible aplicar backtracking de manera recursiva (con una función que se se llama a si misma). Como todas las funciones recursivas esta variación tiene las ventajas y desventajas de estas funciones. Por un lado a veces es más fácil programar una función recursiva y resulta ser corta y simple, pero también si se llama a si misma demasiadas veces puede resultar muy inefectiva y necesitar una cantidad excesiva de recursos.

```
BacktrakingRecursivo([x_1, x_2, ..., x_{i-1}], i)
    calcular S_i
    mientras S_i ≠ ∅ hacer
        elegir x_i ∈ S_i
        S_i = S_i − {x_i}
        si [x_1, x_2, ..., x_i] es una solución parcial entonces
            si [x_1, x_2, ..., x_i] es una solución completa entonces
                Guardar solución
            en otro caso
                BacktrakingRecursivo([x_1, x_2, ..., x_i], i + 1)
            fin
        fin
    fin
fin
```

Algoritmo 2: Backtracking Recursivo

2. Breadth-First Search

Una alternativa del backtracking es el algoritmo Breath-first search. Funciona de manera similar al backtracking, en el backtracking se elige un sólo x_i, luego se analizan todas las posibilidades con esa elección, sólo cuando se ya se analizaron todas las combinaciones posibles con el x_i elegido, entonces se selecciona un nuevo x_i. En un Breadth-first search se eligen todos los x_i posibles al mismos tiempo generando varias soluciones parciales a la vez. Luego se toman todas esas soluciones parciales y con cada una se seleccionan todos los x_{i+1} posibles y se repite este proceso hasta generar todas las soluciones buscadas.

Los algoritmos backtracking y Breadth-first search son similares pero con diferentes cualidades, y dependiendo del problema puede haber

una gran diferencia en el rendimiento. En general el algoritmo backtracking necesita mucha menos memoria, pero el algoritmo Breadth-first search es mucho más fácil para la paralelización.

Un plan para poder obtener todos las configuraciones n_k es el algoritmo 3 el cual es una combinación de estos dos métodos.

soluciones1={base-k};
$l = k$;
mientras $l < n$ **hacer**
  soluciones2=$\varnothing$;
  mientras *soluciones1*$\neq \varnothing$ **hacer**
    elegir $C \in$soluciones1;
    soluciones1=soluciones1-$\{C\}$;
    soluciones2=soluciones2$\cup$agregarlinea(C)
  fin
  $l = l + 1$;
  soluciones1=soluciones2
fin

Algoritmo 3: Generar n_k

Otra manera de ver el backtracking y Breadth-First Search es con árboles, ver la sección 6.

3. Orden

El algoritmo anterior es capaz de generar todas las configuraciones que se están buscando pero genera además algunas no necesarias. El orden en el que los puntos aparecen en las columnas no importan por lo que si dejamos el algoritmo como está arriba se generarán por ejemplo:

0	0	0	1
1	3	5	3
2	4	6	5

y

0	0	0	5
1	3	5	3
2	4	6	1

Que en realidad es la misma solución generada dos veces. Evitar esto es muy simple. Una manera es exigir que los puntos estén en orden con los más chicos arriba y los más grandes abajo en la columna. En el algoritmo si ya tenemos la solución parcial $[x_1, x_2, \ldots, x_i]$ entonces $S_{i+1} = \{x_i + 1, x_i + 2, \ldots, n\}$.

Pero puede mejorarse aún más, si $a_1 = n$ claramente no se podrán escoger los a_i restantes. Por lo que el algoritmo es más eficiente si $S_1 = \{0, 1, \ldots, n-k+1\}$ y al seleccionar x_i entonces $S_{i+1} = \{x_i + 1, x_i + 2, \ldots, n-k+i+1\}$. Nótese que si $i+1 = k$ entonces $S_k = \{x_i + 1, x_i + 2, \ldots, n\}$. De esta manera se generan todas las combinaciones donde $x_1 < x_2 < \ldots < x_k$ sin necesitar analizar ninguna que no cumpla esta característica. Para comprender cuanto trabajo se ahorra con esta técnica considere que se puede generar $k!$ variaciones a partir de la misma solución $[x_1, x_2, \ldots, x_k]$. sólo se trabajará en esta tesis con k=3 o 4, pero entonces se tendría 6 o 24 veces más trabajo respectivamente si aceptara cualquier orden.

Pero no sólo es conveniente aplicar un orden para los puntos incidentes de una línea. También es conveniente definir un orden para las mismas líneas. Se tiene el siguiente escenario:

se calcula

0	0	0	1	1
1	3	5	3	4
2	4	6	5	6

y

0	0	0	1	1
1	3	5	4	3
2	4	6	6	5

Estas dos soluciones son claramente isomorfas ya que el isomorfismo sólo tendría que intercambiar las últimas dos líneas. Por el teorema 2.2 sólo necesitamos guardar una de estas dos soluciones antes de seguir agregando líneas. Por escenarios como este es conveniente definir un orden para las líneas como el siguiente:

Sean: $A = \begin{matrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_k \end{matrix}$, $B = \begin{matrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_k \end{matrix}$

Con $a_1 < a_2 < \ldots < a_k$ y $b_1 < b_2 < \ldots < b_k$. Entonces $A > B$ si y sólo si hay un j tal que $a_i = b_i$ para toda $i < j$ y $a_j > b_j$. A esto se le llama **orden lexicográfico**.

TEOREMA 3.1. *El orden lexicográfico es tricotómico y transitivo.*

DEMOSTRACIÓN. Sean tres líneas:

$$A = \begin{matrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_k \end{matrix}, B = \begin{matrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_k \end{matrix}, C = \begin{matrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_k \end{matrix}$$

Si $A \neq B$ entonces hay un i tal que $a_i \neq b_i$. Entonces sea j el más chico con el que esto pasó, entonces $a_j > b_j$ o $a_j < b_j$. Lo que implicaría que $A > B$ o $A < B$ respectivamente. Lo que implica el orden lexicográfico es tricotómico, en otras palabras para todo par de líneas A y B ocurre exactamente uno de los tres siguientes casos:

1. $A = B$
2. $A > B$
3. $A < B$

Si $A < B$ y $B < C$ entonces existen j y l tales que para toda $i < j$ $a_i = b_i$, $a_j < b_j$, para toda $i < l$ $b_i = c_i$, y $b_l < c_l$. Se divide en tres casos.

1. $j = l$ en cuyo caso $a_i = b_i = c_i$ para toda $i < j$ y $a_j < b_j < c_j$ lo que implica que $A < C$
2. $j < l$ en cuyo caso $a_i = b_i = c_i$ para toda $i < j$ y $a_j < b_j = c_j$ lo que implica que $A < C$
3. $j > l$ en cuyo caso $a_i = b_i = c_i$ para toda $i < l$ y $a_l = b_l < c_l$ lo que implica que $A < C$

Por lo que si $A < B$ y $B < C$ entonces $A < C$. □

Como el orden definido para las líneas es tricotómico y transitivo es posible tener una configuración o configuración parcial con las líneas ordenadas de menor a mayor. Decimos que estas configuraciones están en orden lexicográfico. En el ejemplo 2.1 se puede ver que las configuraciones cuya única diferencia es el orden de las líneas son isomorfas. Por lo que cada configuración o configuración parcial tiene una representación en orden lexicográfica.

Una configuración (n_k) no puede tener dos líneas con los mismos elementos por lo que son todos diferentes. En cuyo caso todas las líneas de cualquier configuración se pueden ordenar de menor a mayor. Diremos entonces que la configuración está en orden lexicográfico.

El orden lexicográfico también nos permite definir un orden para las configuraciones mismas. Sea $A = A_1A_2 \dots A_m$, sea $B = B_1B_2 \dots B_m$ dos configuraciones parciales en orden lexicográfico con m líneas, siendo

los A_i y sus B_i sus líneas en ese orden. Entonces $A > B$ si y sólo si hay un j tal que $A_i = B_i$ para toda $i < j$ y $A_j > B_j$.

Al exigir que las líneas aparezcan en orden en la configuracioń sólo se trabajará con 1 de las $n!$ combinaciones que aparecerián si se aceptaran en cualquier orden. Combinando ambos tipos de orden, en los puntos incidentes a una línea y las líneas mismas, el tamaño de los S_k se reducen de manera importante. Todos los n puntos tienen que aparecer k veces en una configuración (n_k). Sea C una configuración parcial creada por el algoritmo 1 en algún paso del algoritmo 3. Sea x el punto más chico que no ha aparecido k veces en C. Si x aparece en la siguiente línea a la hora de aplicar el algoritmo 1 la siguiente vez con C, x tendrá que aparecer hasta arriba de la columna que representa esta línea pues es el más chico de los puntos que pueden aparecer en esta línea. Si no aparece en la siguiente línea que le toca a C entonces se creará una extensión de C llamémosle C' con x' hasta arriba de su última línea con $x' > x$. Si se exige que las líneas aparezcan en orden (según se definió el orden para las líneas) entonces sera imposible agregar x a C' pues la línea que contenga a x será más chica que la última de C', lo mismo para cualquier extención de C'. Pero x tiene que aparecer k veces en un momento dado o nunca sera una configuración (n_k). Por lo que se hace imposible construir una configuración (n_k) a partir de C' si se exige que las líneas aparezcan en orden. En conclusión si se exige que las líneas aparezcan en orden $S_1 = \{x\}$ donde x es el punto más chico que no ha aparecido k veces en la configuración parcial que se esté manejando.

$$A = \begin{matrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_k \end{matrix}$$

Sea A la última línea de una configuración parcial C generada por el algoritmo 1. Si siempre se eligió el punto más chico posible para estar en cada una de sus líneas (sólo hasta arriba en la columna que lo representa). Entonces $a_1 \leq x$ donde x es el punto más chico que no aparece k veces en C. Se quiere extender a C con una línea mayor a A que incluya a x. Si $a_1 = x$ por la segunda manera en que una línea es mayor a otra, es necesario no escoger ningún punto menor a a_2. Por lo que $S_2 = \{a_2 + 1, a_2 + 2, \ldots, n - k + 2\}$.

El orden introducido sólo cambia el algoritmo 1. Sean C una configuración parcial que se le va a agregar una línea y $A = [a_1, a_2, \ldots, a_k]$ la última línea de C. El algoritmo se modifica para incorporar las ventajas del orden definido.

x_1 =el punto más chico que no está k veces en C;
si $x_1 = a_1$ **entonces**
  $S_2 = \{a_2 + 1, a_2 + 2, \ldots, n - k + 1\}$
en otro caso
  $S_2 = \{x_1 + 1, x_1 + 2, \ldots, n - k + 1\}$
fin
$i = 2$;
mientras $i > 1$ **hacer**
  mientras $S_i \neq \varnothing$ **hacer**
    elegir $x_i \in S_i$;
    $S_i = S_i - \{x_i\}$;
    si $[x_1, x_2, \ldots, x_i]$ *es una solución parcial* **entonces**
      si $[x_1, x_2, \ldots, x_i]$ *es una solución completa* **entonces**
        Guardar solución;
      en otro caso
        $S_{i+1} = \{x_i + 1, x_i + 2, \ldots, n - k + i + 1\}$;
        $i = i + 1$;
      fin
    fin
  fin
  $i = i - 1$;
fin

Algoritmo 4: agregarlinea

4. Nauty

Definir y aplicar un orden para los elementos en una configuración se evitan muchos isomorfismos pero no todos, por ejemplo se puede tener:

$$\begin{array}{ccccccc} 0 & 0 & 0 & 1 & 1 & 2 & 2 \\ 1 & 3 & 5 & 3 & 4 & 3 & 4 \\ 2 & 4 & 6 & 5 & 6 & 6 & 5 \end{array}$$

$$\begin{array}{ccccccc} 0 & 0 & 0 & 1 & 1 & 2 & 2 \\ 1 & 3 & 5 & 3 & 4 & 3 & 4 \\ 2 & 4 & 6 & 6 & 5 & 5 & 6 \end{array}$$

Ambas configuraciones tiene todos sus elementos en el orden usado pero son isomorfas, ya que por ejemplo $\rho(x) = x$ si $x < 5$, $\rho(5) = 6$ y $\rho(6) = 5$ es un isomorfismo entre estas dos configuraciones.

Nauty es la herramienta escogida para detectar el resto de los isomorfismos. Este programa puede crear un isomorfismo entre dos gráficas o determinar que las dos gráficas no son isomorfas. Hay que tomar en cuenta que Nauty esta diseñado para gráficas donde una línea sólo conecta dos puntos en vez de k, pero por eso se tiene el teorema 3.1 que muestra que determinar que las gráficas de Levi de dos configuraciones n_k son isomorfas es equivalente a determinar que las mismas configuraciones son isomorfas y Nauty puede trabajar perfectamente con las gráficas de Levi por lo que es una herramienta apropiada para trabajar con configuraciones (n_k).

Nauty tiene un método llamado **densenauty** que puede verse como una función $G \to G$ donde G sería el espacio de todas las gráficas que pueden existir. Digamos que le damos a densenauty una gráfica g entonces densenauty nos regresaría una gráfica g_{canon}. Llamemosle la **representación canónica** de g. La idea de estas gráficas es que dos gráficas g y h son isomorfas si y sólo si $g_{canon} = h_{canon}$. ¿Cómo se generan estas gráficas? Eso es algo que se le confía completamente a Nauty.

El plan entonces es, mientras se calculan todas las configuraciones con el algoritmo 4 cada vez que que se encuentre una nueva configuración se le calcula su canon y se le compara con todos los otros canon ya generados. Si no es igual a ningún canon ya encontrado entonces eso significa que verdaderamente es una nueva configuración y se guarda, si no entonces es isomorfa a la configración que generó ese canon y se descarta.

TEOREMA 4.1. *Si en el algoritmo 3, se usa el algoritmo 4 y cuando se han generado dos soluciones parciales isomorfas se guarda la más chica, de todas formas, al final, se generan todas las configuraciones (n_k) no isomorfas.*

DEMOSTRACIÓN. Sea una A una configuración (n_k). Por demostrar: el algoritmo 3 genera una configuración isomorfa a A.

Sea x_0 cualquier punto de A. Por estructura de las configuraciones (n_k), a_0 será colineal con $k-1$ puntos por cada una de las k en las que

esta y todos estos puntos son diferentes, llamémosles a estos $k*(k-1)$ puntos a_i con $i \in \{1, \ldots, k^2-k\}$. De tal manera que:

$$A_1 = \begin{matrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{k-1} \end{matrix}, A_2 = \begin{matrix} a_0 \\ a_k \\ a_{k+1} \\ \vdots \\ a_{2k-2} \end{matrix} \ldots, A_k = \begin{matrix} a_0 \\ a_{k^2-2k+2} \\ a_{k^2-2k+3} \\ \vdots \\ a_{k^2-k} \end{matrix}$$

Son las k líneas de A que contienen a a_0. Llamémosles a_i con $i \in \{k^2-k+1, \ldots, n-1\}$ a los puntos de A que no son colineales con a_0, llamémosles A_i con $i \in \{k+1, \ldots, n\}$ a las líneas de A que no son incidentes con a_0. Sea $\alpha : A \to B$ como:

$$\alpha(x) = \begin{cases} i & \text{si } x = a_i \\ B_i & \text{si } x = A_i \end{cases}$$

Se define que en B i es incidente con B_j si y sólo si $\alpha^{-1}(i)$ es incidente con $\alpha^{-1}(B_j)$. Entonces B es un isomorfo a A y α es su isomorfismo.Sea $\alpha' : A \to B$ como:

$$\alpha'(x) = \begin{cases} i & \text{si } x = i \\ B'_i & \text{si } x = B_l \text{ y } B_l \text{ es la } i\text{-esima línea más chica de } B. \end{cases}$$

Entonces se obtiene lo siguiente:

$$B_1 = \begin{matrix} 0 \\ 1 \\ 2 \\ \vdots \\ k-1 \end{matrix}, B_2 = \begin{matrix} 0 \\ k \\ k+1 \\ \vdots \\ 2k-2 \end{matrix} \ldots, B_k = \begin{matrix} a_0 \\ k^2-2k+2 \\ k^2-2k+3 \\ \vdots \\ k^2-k \end{matrix}$$

Se define que en B' i es incidente con B'_j si y sólo si $\alpha'^{-1}(i)$ es incidente con $\alpha'^{-1}(B'_j)$. Entonces B' es isomorfa con A y $\alpha' \circ \alpha$ es su isomorfismo. B además está en orden lexicográfico y sus primeras k líneas son la subconfiguración base-k.

Sea $C = C_1C_2 \ldots C_n$ una configuración en orden lexicográfico cuyas primeras k líneas son la subconfiguración base-k. Es posible que el algoritmo 3 genere a C. Si no, es porque para algún $j \leq n$ la configuración parcial $C' = C_1C_2 \ldots C_j$ fue descartada porque alguna configuración parcial $D' = D'_1D'_2 \ldots D'_j$ es menor e isomorfa a C'. C es una extención de C'. Por el teorema 2.2 se sabe que D' tiene alguna extención D isomorfa a C. Además hay un isomorfismo β entre C y D de tal manera que $\beta(C') = D'$. Sea $D_c = D_{j+1}D_{j+2} \ldots D_n$ la configuración parcial $\beta(C_{j+1}C_{j+2} \ldots C_n)$.

Observamos que D puede estar en orden lexicográfico, si $D'_j < D_i$ para toda $i > j$ entonces es posible que el algoritmo 3 genere a D. Si no, sea D_{j+1} la línea más chica de D_c, y sea D'_s la línea más grande de D' que siga siendo más chica que D_{j+1} (base-k es una subconfiguracion de D' y todas sus líneas son más chicas que D_{j+1}). El algoritmo 3 produce la configuración parcial $D'_1D'_2 \dots D'_sD_{j+1}$ ya que produjo y guardo $D'_1D'_2 \dots D'_s$ y, por otra parte, D_{j+1} tiene que tener el punto más chico que no aparece en $D'_1D'_2 \dots D'_s$ además de que $D_s < D_{j+1}$. Si si el algoritmo 3 guarda $D'_1D'_2 \dots D'_sD_{j+1}$ entonces aún puede producir un isomorfismo de D y por lo tanto de C. Si no, sea E' la configuración parcial isomorfa a $D'_1D'_2 \dots D'_sD_{j+1}$ que el algoritmo 3 guardó y sea E la extención de E' isomorfa a D. Se repite este proceso las veces que sea necesario.

El algoritmo 3 produce una cantidad finita de configuraciones y $E < D < C$ por lo que el proceso anterior se tiene que detener en algún momento. □

5. Árboles

A la hora de generar una configuración (n_k) o configuración parcial y generar su gráfica canon se quiere saber si la gráfica canon es nueva o si ya se generó antes, lo que implicaría que la nueva configuración (n_k) o configuración parcial es isomorfa a una de las que ya se han guardado. Pero sería muy impráctico comparar esta gráfica canon con todas las ya generadas una por una, pues se espera generar millones por lo que se acude a los árboles.

En informática los árboles son estructuras de datos constituidos por nodos y sus conexiones. La principal relación que pueden tener dos nodos en un árbol es: **Padre-hijo** un nodo es el padre y el otro es el hijo. Un nodo puede tener varios hijos pero a lo más un sólo padre. Sólo un nodo en todo el árbol puede no tener padre y esa es la **raíz**, si un nodo no tiene hijos se le llama **hoja**, a todos los demás nodos se les llama **rama**. Un **camino** es una secuencia de nodos $n_1, n_2, \dots, n_k$ tal que para todo $1 \leq i < k$, n_i es padre de n_{i+1}. Adicionalmente por cada nodo en el árbol además de la raíz, debe existir un camino entre la raíz y este nodo.

Otros conceptos útiles al trabajar con árboles son:

1. Si existe un camino del nodo a al nodo b se dice que a es **ancestro** de b y que b es **descendiente** de a. Entonces por definición la raíz es ancestro de todos los otros nodos.

2. Sea B un subconjunto de nodos del árbol A. Si existe sr en B tal que sr es el ancestro de todos los otros nodos en B entonces B es un **subárbol** de A. En otras palabras un subárbol B de un árbol A es un subconjunto de A que forma un árbol por si mismo.
3. El **nivel** de un nodo a es el número de nodos en el camino entre a y la raíz.
4. Un Árbol es **binario** si todos sus nodos tienen a lo más dos hijos. Llamémosles **hijo izquierdo** e **hijo derecho**.
5. Sea a un nodo de un árbol binario A y sea B el conjunto de todos los descendientes del hijo izquierdo (derecho) de a, entonces B es es el **subárbol izquierdo (derecho)** de a.
6. Un árbol esta **equilibrado** o **balanceado** si todos los caminos de la raíz a las hojas del árbol difieren en tamaño en a lo más una unidad.

Hay varias clases de árboles. El que se eligió para trabajar es el **árbol binario de búsqueda**. Estos son árboles binarios donde para cada nodo x en el árbol binario de búsqueda A, todos los nodos del subárbol izquierdo de x tienen un valor menor al de x y todos los nodos del subárbol derecho de x tienen un valor mayor al de x.

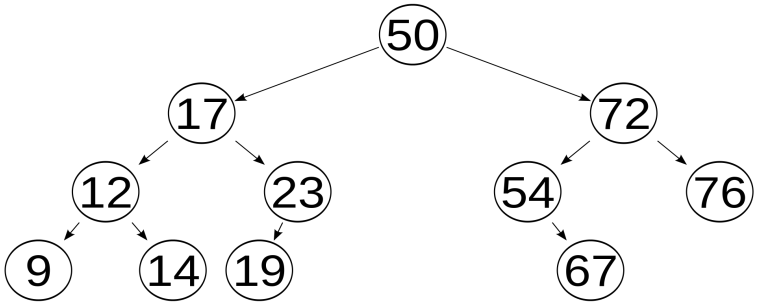

FIGURA 1. árbol binario equilibrado

Digamos que queremos saber si el valor y está en A. Al comparar y con la raíz r de A hay tres posibilidades:

1. $y = r$ en cuyo caso ya se sabe que y está en A.
2. $y > r$ en cuyo caso se puede descartar la posibilidad que y pertenezca al árbol al subárbol izquierdo de r el cual contiene cerca de la mitad de los nodos de A. sólo queda la posibilidad de que y esta en el subárbol derecho de r y si no, entonces y no está en A.

3. $y < r$ en cuyo caso se puede descartar la posibilidad que y pertenezca al subárbol derecho de r el cual contiene cerca de la mitad de los nodos de A. Sólo queda la posibilidad de que y esté en el subárbol izquierdo de r y si no, entonces y no está en A.

Si se repite este procedimiento con los subárboles que aún pueden contener a y hasta encontrar a y o hasta encontrar una hoja, entonces se determina si y está en A o no sin tener que comparar y con todos los nodos de A. Si $|A| = N$ y A esta equilibrado entonces el número máximo de comparaciones necesarias es $\log_2(N)$. Digamos que $N = 10000$ entonces $\log_2(N) \approx 13{,}29$, entonces determinar si y esta en el árbol A sólo toma máximo 14 comparaciones en vez de 10 mil. Entonces cada vez que se encuentre un nuevo elemento y se quiera agregar al árbol hay que hacerlo de manera tal que el árbol siga siendo un árbol binario de búsqueda. Esto es de hecho fácil de lograr, una vez que el elemento nuevo se ha buscado y determinado que no está en el árbol, ya se ha hecho la mitad del trabajo. La búsqueda debió de haber acabado en una hoja y sólo se agrega el nuevo elemento como su hijo derecho si es mayor o su hijo izquierdo si es menor.

Hay una ventaja adicional al construir el árbol binario de búsqueda. Puede construirse una lista ordenada en cualquier momento.

```
ListaOrdenada(nodo x)
    si x ≠NULLO entonces
        ListaOrdenada(x.izquierdo)
        Escribir x.valor
        ListaOrdenada(x.derecho)
    fin
fin
```

Algoritmo 5: Lista ordenada a partir de un árbol binario de búsqueda

TEOREMA 5.1. *Si el algoritmo 5 se inicia con la raíz de un árbol binario de búsqueda entonces este escribirá un lista con todos los elementos del árbol en orden.*

DEMOSTRACIÓN. Sea A un árbol binario de búsqueda, supongamos que el algoritmo 5 se inicia con la raíz de A.

Primero se demostrara que el algoritmo 5 escribirá el valor de todos los elementos del árbol. Sea x un nodo de A, si x es la raíz entonces el

algoritmo 5 escribirá su valor pues el algoritmo se inicio con él, si x no es la raíz entonces existe un camino entre la raíz y x. Digamos que este camino es [raíz, $x_0, x_1, \ldots, x_k, x$], como es un árbol binario de búsqueda x_0 forzosamente es el hijo izquierdo o el hijo derecho de la raíz por lo que el algoritmo 5 lo recorrerá y escribirá su valor. Si x_i es hijo de x_{i-1} y este esta siendo recorrido por el algoritmo 5, entonces este recorrerá x_i y escribirá su valor. Por lo que tarde o temprano recorrerá x_k en cuyo caso también recorrerá x y escribirá su valor. Por lo que el algoritmo 5 recorrerá todo el árbol.

Sólo falta demostrar que escribirá todos los valores en orden. El algoritmo 5 primero escribará los valores del subárbol izquierdo de la raíz, por definición de árbol binario de búsqueda todos estos valores son menores, luego escribirá el valor de la raíz y al último se escribirán todos los valores del subárbol derecho de la raiź que por definición todos estos valores son mayores al de la raíz. Entonces primero se escriben todos los valores en el árbol que son menores al valor de la raíz, después se escribe el valor del de la raíz y al último se escriben todos los valores mayores al de la raíz. Repito este razonamiento con cada subárbol que el algoritmo 5 recorre y todos los elementos resultan estar en orden. □

La técnica para crear esta lista se usará en la paralelización

6. Backtracking y Breadth-First Search vistos con árboles

Los algoritmos Backtracking y Breadth-First también pueden verse como maneras de recorrer árboles de manera sistemática.

Con un algoritmo Backtracking se recorren los nodos atravéz de sus hijos hasta que se encuentra con una hoja en cuyo caso se regresa y se elige un hijo que no se haya elegido antes para continuar recorriendo el árbol.

Con un algoritmo Breadth-First Search se recorren los nodos nivel por nivel.

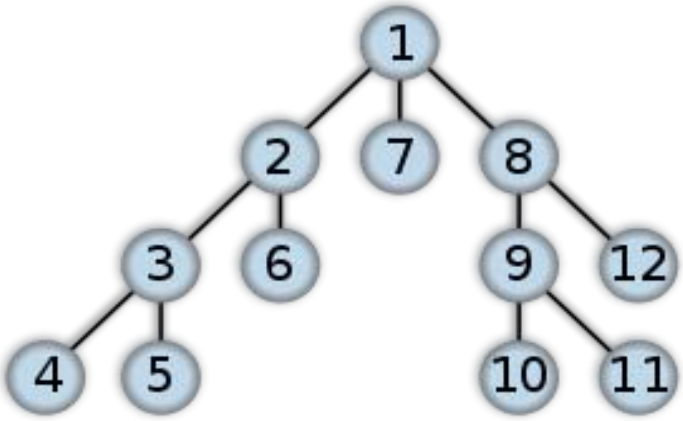

FIGURA 2. Orden en que se recorren los nodos con un algoritmo backtracking

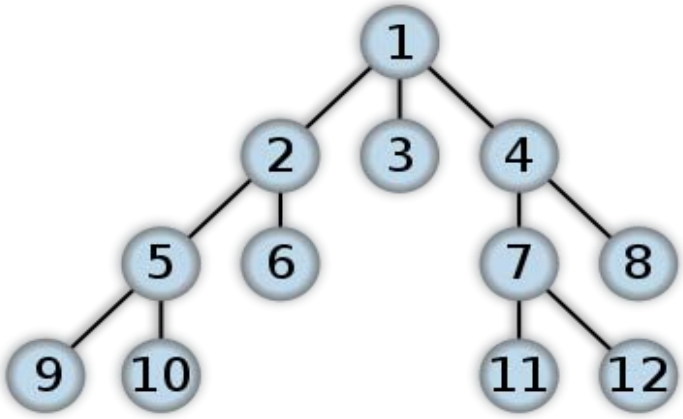

FIGURA 3. Orden en que se recorren los nodos con un Breadth-First Search

Capítulo 4

Paralelismo

En la actualidad, el principal limitante de la velocidad de cálculo de las computadoras es el calor que estas generan. Mientras más rápida sea una computadora, ésta se calentará más y si no se tiene cuidado fácilmente podría dañarse. Por este motivo, en los últimos años no se ha podido aumentar la velocidad de cálculo. Sin embargo, en lo que si se ha podido avanzar es en la capacidad de las computadoras de trabajar en paralelo. Esto es, realizar varios cálculos al mismo tiempo. Para poder resolver problemas aprovechando estos avances hay que asegurarse de adaptar el algoritmo a un ambiente en paralelo. Esto también puede incluir tener varias computadoras trabajando en conjunto para resolver un problema.

1. Plan de trabajo

Para poder aprovechar el paralelismo se necesita un algoritmo adecuado. El algoritmo elegido para poder calcular las configuraciones (n_k) se divide de la siguiente forma:

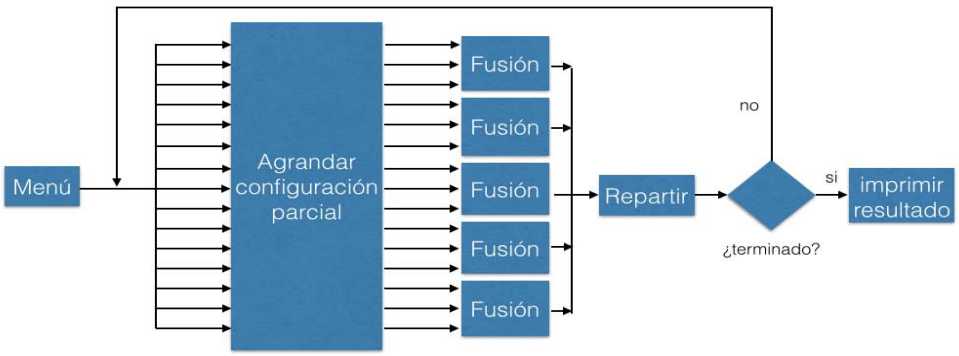

FIGURA 1. Plan de trabajo

Etapas:

1. Menú: En esta etapa el usuario determinara n y k. Además se crearán suficientes configuraciones parciales para iniciar la paralelización los cuales se dividirán en lotes.
2. Agrandar configuración parcial: El número y tamaño de los lotes dependerá de la cantidad de configuraciones parciales, la capacidad de la memoria ram y del número de procesos que la computadora pueda hacer al mismo tiempo. En esta etapa se tomarán todos los lotes de configuraciones parciales y se generarán configuraciones parciales un poco más completas. No necesariamente se precesarán todos los lotes al mismo tiempo pero si varios a la vez. Con cada lote se generará un árbol de configuraciones parciales no isomorfas y sus respectivos etiquetados canónicos. Luego a partir de los árboles se guardarán listas ordenadas en el disco duro con las configuraciones ordenadas según su etiquetado canónico el cual también estara incluido.
3. Fusión: Las listas dejadas por la etapa anterior se generaron de manera hasta cierto punto aislada por lo que es posible que haya una isomorfismos entre dos configuraciones de dos diferentes listas, por lo que quizá se haga trabajo de más después. Para evitar esto, esta etapa fusiona las listas. El hecho de que las listas estén ordenadas según su etiquetado canónico ayuda mucho a que se eliminen isomorfismos. Es posible que haya una gran cantidad de listas por lo que este proceso es en un principio en paralelo.
4. Repartir: Si los diferentes hilos trabajan con diferentes datos o con los mismos pero no los modifican, entonces es fácil la coordinación de estos. Pero si usan los mismos datos y los modifican o escriben sobre el mismo espacio la coordinación de los hilos puede complicarse bastante e incluso si no se tiene cuidado arruinar los datos de manera irremediable. Son frecuentes los casos en que la paralelización no es conveniente y se decidió que esta etapa es uno de esos. A partir de las listas se genera una en la cual se garantiza que no hay dos isomorfas, si las configuraciones ya son (n_k) se detiene el proceso y se imprimen los resultados, pero si no las configuraciones se separan en lotes y se repite el proceso a partir de la etapa 2.

2. GNU Parallel

GNU Parallel es la herramienta seleccionada para integrar el paralelismo. Este funciona con un guíon (o script) de instrucciones las

cuales se ejecutan una por una, excepto las que tiene parallel en las cuales ejecuta varias a la vez dependiendo de la instrucción. También es posible incluir otro guíon o incluso el mismo guíon por lo que es posible la recursión. Debido a la estructura del algoritmo el guíon es sumamente simple por lo que sólo se necesita un mínimo conocimiento de GNU Parallel para este proyecto. Ejemplos:

```
[vhernandez@monstruito scriptproyecto2]$ parallel echo ::: A B C ::: D E F
A D
A E
A F
B D
B E
B F
C D
C E
C F
```

```
[vhernandez@monstruito scriptproyecto2]$ parallel echo ::: A ::: {0..19}
A 0
A 1
A 2
A 3
A 4
A 5
A 6
A 7
A 8
A 9
A 10
A 11
A 12
A 13
A 14
A 15
A 16
A 17
A 18
A 19
```

Básicamente la instrucción funciona como:

$ paralle *programa* ::: *var1* ::: *var2* ::: . . . ::: *varn*

Lo que hace esta instrucción es ejecutar *programa* con todas las posibilidades que las variables puedan tener. No necesariamente todas las posibilidades al mismo tiempo pero si más de una a la vez.

Una variante de esta instrucción es:

$ paralle -j*m* *programa* ::: *var1* ::: *var2* ::: . . . ::: *varn*

La parte de -j*m* esta asignando el numero de trabajos que se harán al mismo tiempo. Ejemplos:

```
[vhernandez@monstruito scriptproyecto2]$ /usr/bin/time parallel -j40 sleep ::: {1..60}
0.18user 0.18system 1:20.38elapsed 0%CPU (0avgtext+0avgdata 44800maxresident)k
0inputs+14232outputs (0major+80237minor)pagefaults 0swaps
[vhernandez@monstruito scriptproyecto2]$ /usr/bin/time parallel -j20 sleep ::: {1..60}
0.20user 0.15system 2:00.80elapsed 0%CPU (0avgtext+0avgdata 44432maxresident)k
0inputs+14368outputs (0major+74352minor)pagefaults 0swaps
```

FIGURA 2. Ejemplo

La misma intrucción es ejecutada 2 veces en la figura 2 con la diferencia de que la primera vez se tiene -j40 y la intrucción tarda 1 minuto y 20 segundos en ejecutarse y la segunda vez es con -j20 y la intrucción tarda 2 minutos en ejecutarse. Eso es porque en uno se ejecutan 40 hilos mientras que en la otra sólo 20. Los hilos se repartirán los trabajos como unidades sin fraccionarlos. Es decir cada trabajo se ejecutará por completo en un sólo hilo. Por lo tanto inevitablemente unos hilos acabarán antes que los otros y estos tendrán que esperar hasta que todos los hilos hayan acabado. Es el trabajo del programador utilizar esta herramienta de la manera más inteligente que pueda para que la paralelización se aproveche al máximo.

3. Ordenamiento por mezcla

En la etapa 3 y 4 en el plan de trabajo se hace lo que se podría considerar como un algoritmo de ordenamiento por mezcla (merge sort). Pues la manera más efectiva de detectar y eliminar repeticiones en un conjunto es ordenando sus elementos. Normalmente este algoritmo parte primero el conjunto que quiere ordenar para luego ordenar cada tramo, pero nada de eso es necesario en este caso pues ya se tienen sublistas ordenadas. Por lo tanto, sólo hay que agregar la parte de generar una sola lista ordenada a partir de n listas ordenadas. Supongamos que tenemos una matriz con apuntadores a las n listas y una manera de saber si ya se recorrió toda la lista como una función *Finlista*.

```
MergeSort(Listas[],n)
    noterminado=Verdadero
    j = 0
    mientras NoTerminado hacer
        maximo=0
        i = 0
        mientras i < n hacer
            si Finlista(Listas[i])==Falso y
            Listas[i].valor>maximo entonces
                maximo=Listas[i].valor
            fin
        fin
        bandera=Verdadero
        i = 0
        mientras i < n hacer
            si Finlista(Listas[i])==Falso y
            Listas[i].valor==maximo entonces
                si bandera entonces
                    ListasNueva[j]=Listas[i]
                    j = j + 1
                    bandera=Falso
                fin
                Listas[i]=Listas[i].siguiente
            fin
        fin
        noterminado=Falso
        i = 0
        mientras i < n hacer
            si Finlista(Listas[i])==Falso entonces
                noterminado=Verdadero
            fin
        fin
    fin
fin
```

Algoritmo 6: Ordenamiento por mezcla

Este algoritmo consta de 3 ciclos adentro de un ciclo principal y supone que las sublistas están cada una ordenadas de mayor a menor y sin repeticiones. El primer ciclo determina cual de todas las sublistas tiene el elemento más grande que no se ha integrado a la lista principal. El segundo ciclo integra el elemento elegido por el primer ciclo

a la lista principal y elimina repeticiones. Y finalmente el último ciclo revisa todas las sublistas para verificar si ya se recorrieron todas para saber si repetir el ciclo principal o no. Este algoritmo es sólo una breve adaptación del clásico ordenamiento por mezcla para este caso.

Capítulo 5

Resultados

n	configuraciones (n3)	configuraciones (n4)
7	1	0
8	1	0
9	3	0
10	10	0
11	31	0
12	229	0
13	2036	1
14	21399	1
15	245342	4
16	3004881	19
17	38904499	1972
18	530452205	971171
19	7597040188	478084719

TABLA 1. Resultados

Se obtuvieron los resultados de la tabla 1 los cuales eran los números que ya se esperaban considerando otras investigaciones de este tema excepto (19_3) y (19_4).

1. Relevancia y contribución del trabajo

1. El estudio de las configuraciones aporta en varios campos: geometría clásica, topología, geometría algebraica, computación y análisis y teoría de números.
2. No ha habido comprobación de el número de configuraciones (19_4).
3. En el artículo "The combinatorial (19_4) configurations"[**1**, pag 1] se mencionan algunas de las aplicaciones de las configuraciones (n_k) y da una idea de la importancia de su enumeración.

Bibliografía

[1] Páez Osuna Octavio, San Agustín Chi Rodolfo, The combinatorial (19_4) configurations. Facultad de Ciencias (UNAM) 22 Marzo 2012.

[2] Velarde Velázquez Carlos Bruno. Patrones de intersección de clases paralelas: Enumeración de diseños resolubles. Facultad de Ciencias (UNAM) Marzo del 2007.

[3] Branko Grunbaum (2009) *Configurations of Points and Lines*. American Mathematical Society

[4] Donald L. Kreher Michigan Technological University y Douglas R. Stinson University of Waterloo. Combinatorial Algorithms Generation, Enumeration and Search

[5] McKay Brendan D. Australian National University, Piperno Adolfo Universitá di Roma nauty and Traces