UNIVERSIDAD DE LA HABANA

FACULTAD DE MATEMÁTICA Y COMPUTACIÓN

DAA

AUTOR

Loitzel Ernesto Morales Santiesteban

La Habana, Cuba.
Septiembre, 2024

UNIVERSIDAD DE LA HABANA

FACULTAD DE MATEMÁTICA Y COMPUTACIÓN

DAA

AUTOR

Loitzel Ernesto Morales Santiesteban

La Habana, Cuba.
Septiembre, 2024

Capítulo 1

Appleman and Tree

1.1. Introducción al problema

Appleman tiene un árbol con n vértices. Algunos de los vértices son nodos negros, y el resto son nodos blancos. Appleman quiere saber cuántas formas hay de eliminar k aristas del árbol (donde $0 \leq k < n$) de modo que los $k+1$ componentes conectados resultantes (árboles) contengan exactamente un nodo negro cada uno.

1.2. Solución por fuerza bruta

La primera solución viable para encontrar la solución al problema es mediante fuerza bruta, probando todas las combinaciones posibles de eliminar aristas del árbol y contando aquellas que cumplan la condición que se nos plantea. Esta solución es extremadamente ineficiente que ya que existen 2^E combinaciones donde E es la cantidad de aristas en el árbol, sin embargo la simpleza del algoritmo nos garantizará que la respuesta es correcta, lo cuál podremos usar para comprobar la correctitud de otras soluciones.

1.3. Aproximaciones iniciales

Nuestro objetivo en este problema es encontrar todas las maneras posibles de eliminar aristas de un árbol para crear una configuración donde cada componente conectado resultante tenga exactamente un nodo negro. A esta configuración la

llamaremos **configuración válida**. Basados en esta idea definiremos la siguiente función:

Función de configuración válida: Sea T un árbol con nodos negros y blancos. Definimos la función $CO(T)$ como el número de configuraciones válidas del árbol T.

Definición: Dado un árbol T, la función $CO(T)$ devuelve la cantidad de maneras diferentes en las que se pueden eliminar aristas de T para obtener una configuración válida (es decir, una configuración donde cada componente conectado resultante tenga exactamente un nodo negro).

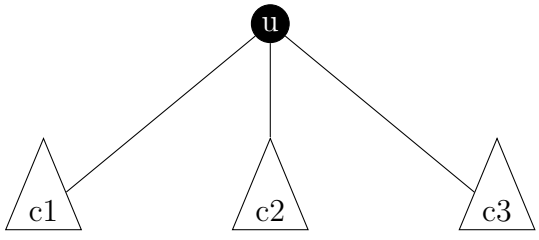
Para mayor claridad, se introduce la siguiente definición, aprovechando la propiedad de que cada componente en una configuración válida debe tener un único nodo negro:

Representante de una componente: Dada una componente C en una configuración válida, definimos el **nodo representante** de C como el único nodo negro contenido en ella.

Con el objetivo de obtener una comprensión profunda del problema y explorar posibles estrategias de solución, se analizarán algunos casos base de manera superficial. Este análisis se enfocará en la identificación de patrones y comportamientos recurrentes que podrían resultar útiles en la búsqueda de una resolución.

En las representaciones que se mostrarán a continuación, los triángulos representarán subárboles y los círculos nodos.

Analizando un caso simplificado



Sea T el árbol completo y $CO_i = CO(c_i)$ para cada subárbol c_i. Entonces, $CO(T) \geq CO_1 \times CO_2 \times CO_3$ debido a que podemos considerar al nodo u como una componente adicional independiente que contiene un único nodo negro.

Sin embargo, estas no son todas las posibles configuraciones válidas. Queda claro que si alguna componente c_i tiene como raíz un nodo negro, habría que eliminar la arista que conecta a dicho nodo con el nodo u, puesto que una componente debe tener un único nodo negro. Pero si existe alguna componente con un nodo raíz blanco, podría agregarse a la componente representada por el nodo u o no, ampliando el número de configuraciones válidas del árbol.

Ampliando la idea anterior, sean W_1, W_2, W_3 las componentes formadas por solamente nodos blancos y que contienen a las raíces de los subárboles c_1, c_2 y c_3 respectivamente. Esto nos permite escoger cualquier cantidad de nodos blancos de cada una de dichas componentes y agregarlos a la componente representada por el nodo u, aumentando la cantidad de configuraciones válidas del árbol por cada una de las combinaciones de nodos blancos agregados.

Basándonos en las ideas de una función de configuración válida y el como componentes formadas solamente por nodos blancos alteran al problema, se propone una solución mediante programación dinámica (DP) para este problema.

1.4. Solución mediante DP

Para abordar el problema mediante DP definiremos:

Componente blanca: Componente conexa del árbol en la que todos sus nodos son blancos.

dp_black[u]: Cantidad de formas de dividir el árbol con raíz en u en componentes conexas de tal manera que cada componente conexa contenga exactamente 1 nodo negro. Esto es igual a $CO(T)$ donde T es el árbol con raíz en u.

dp_white[u]: Cantidad de formas de dividir el árbol con raíz en u en componentes conexas de tal manera que la componente conexa a la que pertenezca u sea blanca y el resto de componentes conexas contenga exactamente 1 nodo negro.

Calcularemos **dp_black[u]** y **dp_white[u]** de forma recursiva, pero el como se calcula depende del color del nodo u por lo que abordaremos ambos casos de manera separada.

Caso 1: Nodo u es negro

En este caso existe un conjunto de formas de obtener una configuración válida que consiste en dejar el nodo u en una componente con él mismo como único nodo, o lo que es lo mismo, no se adjuntan nodos blancos de ninguno de los subárboles formados por sus hijos. Definiendo $c_1, c_2, \ldots, c_i$ como los nodos hijos de u, esto nos daria un total de $\prod_{i=1}^{n} dp\_black[c_i]$ formas diferentes de obtener una configuración válida con estas características.

El único otro caso a contemplar es aquel en el que se escoja al menos un nodo blanco en $c_1, c_2, \ldots, c_i$ para formar parte de la componente representada por u. Sea c_k un subárbol cualquiera. Si queremos agregar nodos de dicho subárbol a la componente conexa representada por u de tal manera que se mantengan las condiciones del problema, se deberán cumplir tres condiciones:

1. Todos los nodos a agregar a la componente conexa representada por u deben formar parte de una componente conexa, para mantener la conexidad al agregarlo.

2. Todos los nodos deben ser blancos, puesto que u ya es un nodo negro representante de su propia componente conexa.

3. El resto de nodos en c_k deben formar componentes conexas con exactamente un nodo negro.

El número de formas en el que se pueden escoger nodos de tal manera de que se cumpla esta característica es la definición de **dp_white[u]** por lo que existen $\prod_{i=1}^{n} dp\_white[c_i]$ formas de de obtener configuraciones válidas de esta manera.

Simplemente sumando obtenemos que:

$$dp\_black[u] = \prod_{i=1}^{n}(dp\_black[c_i] + dp\_white[c_i])$$
Donde $c_1, c_2, \ldots, c_i$ nodos hijos de u

El caso de $dp\_white[u]$ es simple dado que no existe ninguna manera de tener a u en una componente blanca dado que u es negro, por lo tanto:

$$dp\_white[u] = 0$$

Caso 2: Nodo u es blanco

En este caso si resulta necesario adjuntar el nodo u a alguna de las componentes válidas en los subárboles con raíz en los hijos de u.

Sean $T_1, T_2, \ldots, T_i$ los subárboles con raíz en $c_1, c_2, \ldots, c_i$, y sea T_k un subárbol cualquiera de estos. En este caso, existen $dp\_black[c_k]$ formas posibles de adjuntar el nodo u a una componente válida en T_k, dado que esa es la cantidad de configuraciones válidas de T_k, y estaremos adjuntando u a la componente conexa que contenga a la raíz de T_k.

Es importante destacar que diferentes configuraciones pueden compartir la misma componente conexa que contenga a la raíz, pero estas siguen contando como configuraciones diferentes porque el resto de las componentes cambian. Dado un T_j cualquiera con $j \neq k$, tenemos dos opciones al construir una configuración válida:

- No agregar ningún nodo de T_j a la componente válida que contiene a u: En este caso, el número de configuraciones se multiplica por $dp\_black[c_j]$ al ser árboles independientes.
- Agregar nodos blancos de T_j a la componente que contiene a u: En este caso, el número de configuraciones se multiplica por $dp\_black[c_j]$ dado que estamos seleccionando una componente blanca de T_j para agregar a la componente que contiene a u. (Nótese que u ya está en una componente válida y solo estamos agregando nodos blancos).

Por lo que existirán $dp\_black[c_K] \times ((dp\_black[c_j] + dp\_white[c_j])$ solo considerando esos dos árboles y asumiendo que el nodo u se adjunto a una componente conexa en T_k. Esta multiplicación con la suma de los dp_black y dp_white del árbol que estamos considerando se repetirá por cada árbol diferente de T_k. Entonces la forma concisa de calcular el $dp\_black[u]$ es:

$$dp\_black[u] = \sum_{i=1}^{n} \left(\prod_{j=1}^{i-1} C_j \times dp\_black[c_i] \times \prod_{j=i+1}^{n} C_j \right)$$
$$\text{Donde } c_1, c_2, \ldots, c_n, \text{ nodos hijos de u}$$
$$C_j = dp\_black[c_j] + dp\_white[c_j]$$

Para calcular $dp\_white[u]$ podemos analizar que para cada subárbol T_k podemos escoger unir u a una de las componentes blancas en T_k o a una componente válida en T_k, esto es simplemente:

$$dp\_white[u] = \prod_{i=1}^{n}(dp\_black[c_i] + dp\_white[c_i])$$
Donde $c_1, c_2, \ldots, c_n$, nodos hijos de u

Casos base para la recursividad

Dado que estamos trabajando en un árbol los casos bases resultan en las hojas, donde solo tenemos un nodo. En caso de que el nodo sea negro tenemos que:

$$dp\_black[u] = 1 \; dp\_white[u] = 0$$

Y en caso de que el nodo sea blanco tenemos que:

$$dp\_black[u] = 0 \; dp\_white[u] = 1$$

1.5. Complejidad temporal de la solución

Podemos calcular la solución recursivamente visitando cada nodo una sola vez. Esto nos permite calcular $dp\_black[u]$ (donde u es la raíz del árbol) en $O(n)$, donde n es la cantidad de nodos del árbol

Capítulo 2

a-Good String

2.1. Introducción al problema

Dada una cadena de caracteres s de longitud n, compuesta por letras minúsculas del alfabeto latino. Se garantiza que $n = 2^k$ para algún entero $k \geq 0$.

Una cadena s se considera 'c-buena' si cumple al menos una de las siguientes condiciones:

- La longitud de s es 1, y consiste solo del carácter c (es decir, $s_1 = c$).
- La longitud de s es mayor que 1, la primera mitad de la cadena consiste solo en el carácter c (es decir, $s_1 = s_2 = \ldots = s_{n/2} = c$) y la segunda mitad de la cadena (es decir, la cadena $s_{n/2+1}s_{n/2+2}\ldots s_n$) es una cadena "(c+1)-buena".
- La longitud de s es mayor que 1, la segunda mitad de la cadena consiste solo en el carácter c (es decir, $s_{n/2+1} = s_{n/2+2} = \ldots = s_n = c$) y la primera mitad de la cadena (es decir, la cadena $s_1 s_2 \ldots s_{n/2}$) es una cadena "(c+1)-buena".

Por ejemplo, 'aabc' es 'a'-buena, 'ffgheeee' es 'e'-buena.

En un solo movimiento, puedes elegir un índice i entre 1 y n y reemplazar s_i por cualquier letra minúscula del alfabeto latino (cualquier carácter desde 'a' hasta 'z').

Tu tarea es encontrar el número mínimo de movimientos necesarios para obtener una cadena 'a'-buena a partir de s (es decir, una cadena 'c'-buena para $c =' a'$). Se garantiza que la respuesta siempre existe.

2.2. Ideas iniciales

La naturaleza del problema (el tamaño de 2^k de la cadena y las reglas recursiva) indican que puede ser factible resolver el problema mediante **Divide and Conquer**. Se intentará hacer uso de la recursión en el cálculo de cadenas $k{+}1{-}buena$ y búsqueda exhaustiva para garantizar que se encontró el mínimo.

2.3. Solución mediante Divide and Conquer

En vistas de buscar esta solución definamos la siguiente función:

Costo[s, i, j, k]: Cantidad mínima de cambios que hay que realizar a la subcadena s que va de i a j para que sea una cadena k-buena, donde i y j índices y k carácter entre la a y la z.

Cambio[s, i, j, k]: Cantidad mínima de cambios que hay que realizar a la subcadena s que va de i a j para que esté formada solamente por caracteres k, donde i y j índices y k carácter entre la a y la z.

Obsérvese que, ante una cadena s cualquiera, si queremos convertirla en una cadena k-buena, tenemos dos posibilidades:

- Sustituir los valores de la primera mitad de la cadena por k y luego buscar convertir a la segunda mitad en una cadena $(k+1)$-buena.
- Hacer lo opuesto: sustituir los valores de la segunda mitad de la cadena por k y luego buscar convertir a la primera mitad en una cadena $(k+1)$-buena.

Si aplicamos este razonamiento recursivamente y encontramos el camino de costo mínimo, habremos encontrado la solución. Para esto, podemos simplemente calcular ambas posibilidades y quedarnos con la mejor. Este proceso se puede realizar recursivamente de la siguiente manera:

$$Costo[s,i,j,k] = MIN(Cambio[s,i,\tfrac{j}{2},k] + Costo[s,\tfrac{j}{2}+1,j,k+1], Costo[s,i,\tfrac{j}{2},k] + Cambio[s,\tfrac{j}{2}+1,j,k+1])$$

Donde s cadena, k carácter y k+1 carácter que sucede a k.

2.4. Complejidad temporal de la solución

Para analizar la complejidad de la solución utilizaremos el teorema maestro, primero necesitamos identificar la recurrencia que define nuestro algoritmo.

De la ecuación recursiva que definimos se puede observar que:

- Tenemos dos llamadas recursivas a 'Costo' con un tamaño de problema que se reduce a la mitad ('j/2').
- 'MIN' se realiza en tiempo constante pero 'Cambio' se realiza en $O(n)$ ya que se debe revisar cada carácter

Por lo tanto, la recurrencia nque describe la complejidad de nuestro algoritmo es:

$$T(n) = 2T(n/2) + O()$$

Ahora, podemos aplicar el teorema maestro para determinar la complejidad:

- **a = 2:** El número de subproblemas recursivos.
- **b = 2:** El factor de reducción del tamaño del problema.
- **f(n) = O(n):** La complejidad del trabajo realizado fuera de las llamadas recursivas.

Comparando los valores de a, b y $f(n)$ con las condiciones del teorema maestro, vemos que estamos en el **caso 1**:

$$f(n) = O(n) = O(n^1) \text{ con } 1 = \log_a b = \log_2 2.$$

Por lo tanto, según el teorema maestro, la complejidad de nuestro algoritmo es $\mathbf{O}(nlog(n))$.

2.5. Demostración de Optimalidad mediante adversario

Se demostrará que no existe una solución con una complejidad mejor que $O(nlog(n)$ para este problema utilizando un argumento de adversario.

Argumentación por adversario:

Para demostrar que $O(n \log n)$ es la cota inferior de la complejidad temporal para este problema, usaremos una **prueba por adversario**. La idea es que un adversario puede diseñar la cadena de entrada de manera que cualquier algoritmo eficiente tenga que hacer al menos $O(n \log n)$ operaciones para transformar la cadena en una 'a-buena'.

Estructura del problema

El problema tiene la siguiente naturaleza recursiva:

- Debemos verificar cada mitad de la cadena en cada paso recursivo para ver si es "buena"para un carácter particular.
- Si no es "buena", necesitamos contar cuántos cambios se requieren para convertirla en una cadena "buena".
- Esto debe repetirse recursivamente para cada carácter subsiguiente en el alfabeto.

Argumento adversarial

El adversario puede crear una cadena que necesite la máxima cantidad de operaciones en cada nivel de recursión, de tal manera que cualquier algoritmo que intente resolver el problema necesite al menos $O(n \log n)$ tiempo.

Caso Base

Para una cadena de longitud 1, el problema es trivial: simplemente verificamos si el carácter es igual a 'a'. Si no lo es, se necesita una operación para cambiarlo. Por lo tanto, en este caso el adversario no tiene forma de aumentar la complejidad, ya que solo se necesita una comparación y, si es necesario, una operación de reemplazo.

Caso General

Para cadenas de longitud $n = 2^k$, el adversario puede construir una cadena de manera que sea necesario procesar ambas mitades en cada nivel de la recursión.

Consideremos una cadena de longitud 4:

- El adversario puede hacer que las dos primeras letras de la cadena no sean 'a', lo que nos forzaría a realizar al menos 2 comparaciones y posibles reemplazos.
- Además, las dos últimas letras pueden estar construidas de manera que no sean 'b', forzando comparaciones y cambios adicionales en el siguiente nivel de recursión.

En cada nivel, el adversario puede diseñar las cadenas de manera que:

- La primera mitad de cada subcadena no sea "buena"para el carácter en cuestión.
- La segunda mitad tampoco sea "buena", sino que requiera comparaciones y cambios adicionales.

Paso 3: Número total de operaciones

Para cada nivel de la recursión, el algoritmo necesita procesar las n letras en total, ya que necesita verificar cada mitad de la cadena para cada nivel de "bondad"desde 'a', 'b', etc., hasta el carácter final. Dado que estamos dividiendo la cadena en mitades a lo largo de $\log n$ niveles de recursión, esto resulta en $\log n$ niveles de procesamiento.

- En cada nivel, necesitamos realizar al menos $O(n)$ comparaciones o cambios para procesar las letras.

- Esto debe repetirse para cada uno de los $\log n$ niveles de recursión.

Por lo tanto, la complejidad total es $O(n \log n)$.

Paso 4: Conclusión

El adversario puede construir la cadena de tal manera que, en cada nivel de recursión, ambas mitades de la cadena sean subóptimas y requieran un procesamiento adicional. Este argumento demuestra que cualquier algoritmo necesitaría al menos $O(n \log n)$ tiempo, ya que debe realizar operaciones en cada nivel de la recursión y hay $\log n$ niveles en total.

Resumen

Para una cadena de longitud n, el adversario puede diseñar una secuencia que obligue al algoritmo a hacer $O(n)$ comparaciones en cada uno de los $\log n$ niveles.

Por lo tanto, la **cota inferior** de la complejidad temporal para este problema es $O(n \log n)$.

Capítulo 3

Flow Free

3.1. Introducción al problema

Flow Free es un juego de rompecabezas en el que a los jugadores se les presenta una cuadrícula rectangular que contiene puntos de colores. El objetivo principal es conectar cada par de puntos del mismo color a través de una secuencia de celdas adyacentes, asegurando que los caminos no se crucen y que cada celda de la cuadrícula se visite exactamente una vez.

El problema se puede categorizar en tres versiones distintas:

1. **Primera Versión:** Los jugadores conectan todos los pares de terminales a través de caminos no que no se intercepten sin el requisito de visitar todas las celdas de la cuadrícula.

2. **Segunda Versión:** Los jugadores deben conectar los terminales mientras también visitan cada celda de la cuadrícula, sin tomar pasos extra entre terminales.

3. **Tercera Versión:** Similar a la segunda, pero se permiten tomar pasos extra entre terminales para visitar todas las celdas de la cuadrícula.

La complejidad del problema surge del requisito de que los caminos no se pueden cruzar.

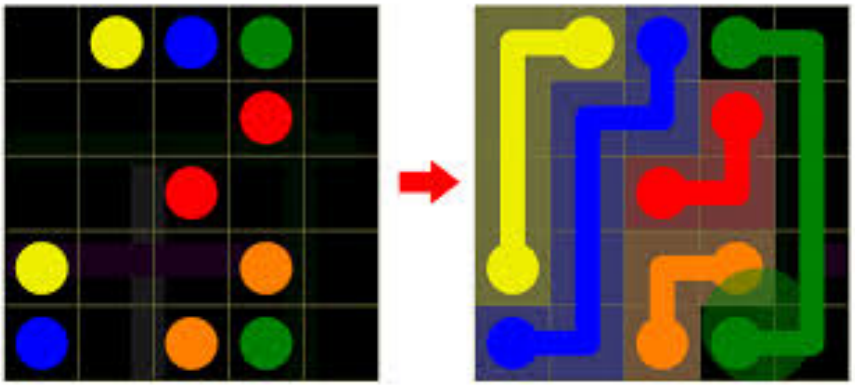


Figura 3.1: Descripción de la imagen: Ejemplo de una instancia de Flow Free antes y despues de resolver.

3.2. Aproximaciones iniciales

Era de mi conocimiento que saber si existe una solución para la segunda versión presentada, en la que se tienen que recorrer todas las casillas de este problema era $NP - Completa$, además parece ser que la complejidad del problema no recae en cubrir todas las casillas, por lo que se tiene la hipótesis de que la primera versión también es $NP - Completa$,. Se trabajara en vista de demostrar o refutar esta hipótesis, para lo que será necesario demostrar o refutar que:

1. Flow Free se encuentra en NP, es decir, es posible verificar su solución en tiempo polinomial
2. Es posible realizar una reducción desde un problema NP-Completo reconocido.

Representación formal del problema

Para proceder con la demostración debemos trasladar el problema a un terreno matemático en el que poder trabajar, para esto se construirá una representación en grafo de cualquier instancia del problema.Se numeraran las celdas empezando por la celda inferior izquierda y terminando en la superior derecha, por ejemplo, en un caso de 4x4 sería:

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

Basándonos en esta numeración construiremos un grafo en donde a cada vértice se le asignará una celda y se asignará una arista entre cada vértices cuyas celdas correspondientes sean adyacentes. Siguiendo el ejemplo anterior tendríamos:

$$V = \{0, 1, \ldots, 15\}$$
$$E = \{\{0, 1\}, \{0, 4\}, \{1, 5\}, \ldots, \{14, 15\}\}$$

Además para cada par de celdas que debamos conectar tendremos una tupla de vértices correspondiente a dichas celdas:

$$T = \{\{0, 1\}, \{0, 4\}, \{1, 5\}, \ldots, \{14, 15\}\}$$

A estos pares le llamaremos *terminales*.

Con esto claro podemos definir cualquier problema Flow Free de NxN casillas como una terna tal que:

$$FlowFree = \{V, E, T\}$$

Donde:

1. **V** conjunto de vértices que representan las celdas
2. **E** conjunto de aristas que representan la adyacencia entre las celdas
3. **T** conjunto de pares de terminales que representan las celdas que deben estar conectadas para representar una solución del problema.

Decimos que sea $S = \{C_1, C_2, \ldots, C_m\}$ donde cada T_i representa un camino que conecta par de terminales en el grafo G siendo m es el número total de pares de terminales, es una solución de $FlowFree(V, E, T)$ **si y solo si** se cumple que:

1. Existe un camino entre cada par de terminales en el grafo G.
2. La intersección de los conjuntos de vértices de estos caminos es el conjunto vacío:
$$\bigcap_{i=1}^{m} V(C_i) = \emptyset$$

Con cada condición tenemos garantizado que:

1. Están conectadas todas las celdas del mismo color.
2. No hay intersecciones entre los caminos que conectan las celdas.

Demostración de que FlowFree se encuentra en NP

Dado que lo dada una solución factible a un problema FlowFree arbitrario solo debe cumplir las condiciones anteriores, solo se deben verificar estas condiciones en la solución. Esto se puede hacer con un simple recorrido a través de esta, donde marquemos los vértices que hemos visto, asegurando siempre la adyacencia entre estos para garantizar que forman caminos. Esta verificación es $O(V)$, por lo que se realiza en tiempo polinomial.

Reducción a FlowFree desde 3-SAT

El próximo paso en la demostración consiste en reducir $3-SAT$ a nuestro problema. $3-SAT$ es un conocido problema NP-Completo en el que se nos da una fórmula booleana en forma **conjuntiva normal (CNF)**, donde cada **cláusula** contiene exactamente tres literales. El objetivo es determinar si existe una asignación de verdad para las **variables** en la fórmula que la satisfaga. Para realizar la reducción, dado un una fórmula $3-SAT$ arbitraria F con variables $x_1, x_2, \ldots, x_n$:

1. Para cada variable x_i que aparezca en A, crearemos un par vértices V_i y V_i'. Estos vértices estarán conectados por 2 aristas distintas T_i y F_i.

2. Por cada cláusula c_j que aparezca en A, crearemos un par vértices C_j y C'_j. Estos vértices estarán conectados por 3 aristas distintas P_{1j}, P_{2j} y P_{3j}.

3. Cada par de estos vértices formara parte del conjunto de terminales T.

4. Sean x_i las variables en la cláusula C_j, si x_i está negada en la cláusula C_j, eliminaremos la arista f_i y la arista P_{ij} y añadiremos el vértice t_ij y las aristas $\{V_i, t_{ij}\}, \{t_{ij}, V'_i\}, \{C_j, t_{ij}\}$ y $\{t_{ij}, C'_j\}$.

5. Sean x_i las variables en la cláusula C_j, si x_i no está negada en la cláusula C_j, eliminaremos la arista f_i y la arista P_{ij} y añadiremos el vértice f_ij y las aristas $\{V_i, f_{ij}\}, \{f_{ij}, V'_i\}, \{C_j, f_{ij}\}$ y $\{f_{ij}, C'_j\}$.

A continuación se declaran algunas definiciones que resultaran útiles en la demostración

T-camino: Se le dice **T-camino** entre los terminales V_k y V'_k al camino entre V_k y V'_k constituido por la arista T_k o al camino constituido por vértices de la forma t_{kj} donde $j = 1 \dots n$ y n cantidad de cláusulas en la instancia del problema $3 - SAT$

F-camino: Se le dice **F-camino** entre los terminales V_k y V'_k al camino entre V_k y V'_k constituido por la arista F_k o al camino constituido por vértices de la forma f_{kj} donde $j = 1 \dots n$ y n cantidad de cláusulas en la instancia del problema $3 - SAT$

Para ilustrar el método que se sigue en la construcción del multigrafo se usará el siguiente ejemplo:

$$(\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$$

Hasta el paso 3 el multigrafo luce de la siguiente manera:

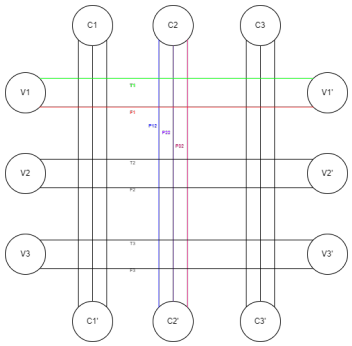


Figura 3.2: Aspecto del grafo hasta la conclusión del paso 3 de la construcción. Se han mostrado los nombres y asignado colores a algunas aristas para mayor comprensión

Con la conclusión del paso 4 y 5 el multigrafo luce de la siguiente manera:

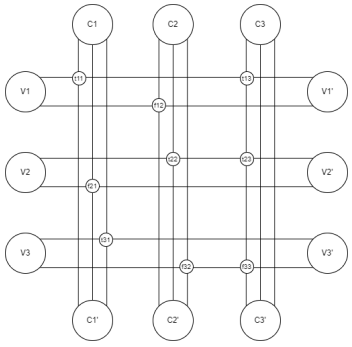


Figura 3.3: Aspecto del grafo al finalizar todos los pasos. Se puede observar como en la çolumna"de la cláusula número 1 se le asigna a la primera variable (V1) la linea "superior"dado que está negada, y lo opuesto para la segunda variable (V2). La misma lógica se sigue en la adición del resto de vértices

Nótese como en cada camino desde C_i a C_i' solo hay a lo sumo un vértice, debido a que solo una variable puede ocupar el lugar de primera, segunda o tercera variable de la cláusula.

Esta construcción nos permite mediante la elección de que camino se escoge comprobar si una cláusula es verdadera o no. Para entender esto informalmente podemos prestar atención a la segunda fila que representa la variable x_2, si quisiéramos conectar V_2 con V_2' mediante el camino superior $(V_2, t_{22}, t_{23}, V_2')$, esto sería el equivalente a

decidir que la variable x_2 será 0, entonces estaremos invalidando el camino C_2, t_{22}, C'_2, al haber usado ya el vértice t_{22}, lo cuál sería el equivalente a establecer que la cláusula número dos no puede hacerse verdadera gracias a la variable x_2 (Debido a que es falsa). Igual lógica siguen las variables negadas, ofreciendo otro posible camino al estar en las aristas ïnferiores", si una de estas variables negadas es 1, bloquearán la posibilidad de la cláusula de ser verdadera gracias a esa variable

Ahora demostraremos formalmente que la representación en grafo de un problema de $FlowTree(V, E, T)$ tendrá una solución **si y solo si** la fórmula $3-SAT$ a partir del cual se construyó es satisfacible.

Demostrando: $3-SAT$ **satisfacible** $\Rightarrow$ **Tiene solución** $FlowTree(V, E, T)$

Encontremos la solución al problema a partir de la resolución de la instancia de 3-SAT.

Para cada variable x_i del problema conectaremos V_i con V'_i mediante un $t-camino$ si el valor de x_i es 1 o mediante un $f-camino$ si el valor de x_i es 0. Por como construimos el grafo no existe manera de que estos caminos compartan vértices, solo existe un único camino entre cualquier V_i y su terminal correspondiente.

Una vez hecho esto para cada terminal C_k debe existir un camino válido hacia su terminal correspondiente C'_k. Demostremos esta afirmación por absurdo.

Sean C_k y C'_k el par de terminales para los cuáles no existe camino válido y x_1, x_2, x_3 las variables de la cláusula que representan, si no existe camino válido entre C_k y C'_k esto significa que cualquier camino entre C_k y C'_k contiene un vértice t_{jk} o f_{jk} con $j = 1, 2, 3$ y que estos vértices ya aparecen en otros caminos. Estos vértices solo pueden aparecer en $t-caminos$ o $f-caminos$ de las variables x_1, x_2 y x_3, pero por por la forma en la que construimos el multigrafo y elegimos los caminos esto significa que estos vértices representan variables con valor 1 que están siendo negadas (regla 3) o con valor 0 sin ser negadas (regla 4) y por ende todos los literales dentro de la cláusula tendrían valor de 0 y su vez la cláusula completa tendría valor de 0 haciendo que esto no fuera una selección de variables que satisfaciera la instancia del $3-SAT$ contradiciendo lo que se había establecido. Por lo tanto queda demostrado por absurdo que debe existir un camino válido entre cada C_k y C'_k y habríamos encontrado una solución para el problema $FlowTree(V, E, T)$.

Demostrando: Tiene solución $FlowTree(V, E, T) \Rightarrow 3 - SAT$ satisfacible

Si existe una solución para el problema $FlowTree(V, E, T)$ entonces existen caminos válidos entre cada V_k y V'_k y cada C_k y C'_k. De igual manera que se demostró en el caso anterior, si el camino que conecta V_k y V'_k es un $t - camino$ podemos asignarle un valor de 1 a la variable y si es un $f - camino$ podemos asignarle un valor de 0. El hecho de que cada terminal C_k conecte con su terminal correspondiente C'_k nos garantizará que en cada cláusula hay al menos un literal con valor de 1 y por ende todas las cláusulas tendrán valores de 1 y habremos encontrado una solución factible para la instancia del problema 3-SAT

Con esto queda demostrada la reducción del $3-SAT$ a $FlowTree(V, E, T)$ y dado que esta reducción se puede hacer en tiempo polinomial (Solo consiste en recorrer una vez el problema) queda demostrado que $FlowTree$ es un problema $NP - Completo$

3.3. Reducción a un problema mas simple

Demostrada la $NP-Completitud$ del problema resultaría conveniente simplificar las reglas del problema en búsqueda de una solución.

Para esto se propone eliminar la restricción de colores en el problema, es decir, una solución será valida si existen caminos no intersecantes para cada par de terminales o en otras palabras, cualquier terminal puede estar conectado a cualquier terminal.

3.4. Solución Polinomial usando Edmond-Karp

Consideremos un tablero o **grid** de tamaño $n \times m$, donde cada celda puede ser representada como un nodo. Nuestro objetivo es encontrar caminos que conecten cada terminal con cualquier otro terminal. Para abordar este problema, podemos modelar el tablero como una **red de flujo** y utilizar el algoritmo de Edmonds-Karp para encontrar el flujo máximo, lo que nos permitirá determinar la cantidad de caminos que conectan los pares de puntos, si la cantidad de caminos es igual a la mitad de la cantidad de terminales, habremos encontrado un camino entre cada par

de terminales y el problema tendrá solución.

El proceso sigue los siguientes pasos:

Modelado de la Grid

Para cada celda (i, j) del tablero crearemos dos nodos:

- Un nodo **in_node** que representa la entrada a la celda.
- Un nodo **out_node** que representa la salida de la celda.

Entre estos dos nodos, se establece una arista con una capacidad de 1, de tal manera que solo se puede "pasar.ª través de la casilla una sola vez

Conexión entre Celdas Adyacentes

Para cada celda (i, j), establecemos conexiones con sus celdas adyacentes. Las posibles conexiones son:

- $(i - 1, j)$: la celda inmediatamente arriba.
- $(i + 1, j)$: la celda inmediatamente abajo.
- $(i, j - 1)$: la celda inmediatamente a la izquierda.
- $(i, j + 1)$: la celda inmediatamente a la derecha.

Estas conexiones entre celdas se modelan como aristas dirigidas entre los nodos **out_node** de la celda actual y los **in_node** de las celdas adyacentes, con una capacidad de 1.

Fuente y Sumidero

Para encontrar caminos específicos entre pares de puntos en el tablero, agregamos un nodo **fuente** (source) y un nodo **sumidero** (sink):

- La fuente se conecta a las celdas de entrada seleccionadas con aristas de capacidad 1.
- El sumidero se conecta a las celdas de salida seleccionadas, también con aristas de capacidad 1.

El objetivo es determinar el flujo máximo desde la fuente hasta el sumidero, lo que corresponde a la cantidad máxima de caminos independientes entre los puntos de entrada y salida.

Luego mediante cualquier algoritmo que nos permita hayar flujo máximo, en nuestro caso Edmond-Karp, se puede hayar el flujo máximo.

La demostración de que mediante este proceso siempre se encontrará la cantidad máxima posible caminos disjuntos para cada par de nodos es bastante trivial.

Asumamos que existen n caminos disjuntos para cada par de nodos en el problema, pero el flujo máximo F resulta ser menor a n. En este caso podemos encontrar un flujo mayor a F simplemente saturando las aristas presentes en cada celda del tablero, por lo que F no era un flujo máximo.

Marcado de Caminos Saturados

Una vez que se ha calculado el flujo máximo, podemos identificar las celdas que forman parte de los caminos saturados. Una arista está saturada si su capacidad residual es 0. Si la arista entre el nodo **in_node** y **out_node** de una celda está saturada, significa que esa celda forma parte de un camino.

Podemos dibujar la grid y marcar las celdas que forman parte de caminos saturados, lo que corresponde a los caminos efectivos en el tablero.

Complejidad

La complejidad del algoritmo Edmonds-Karp es $O(V \cdot E^2)$, donde V es el número de nodos (en este caso, $n \times m \times 2 + 2$ por el uso de nodos *in* y *out*, y los nodos source y sink), y E es el número de aristas en el grafo del tablero.

3.5. Conclusiones

Dado que para el problema simplificado encontramos una solución en tiempo polinomial, podemos afirmar que la complejidad del problema original recae en la restricción de conectar cada terminal a otro terminal, y no en el hecho en si de necesitar encontrar caminos disjuntos, cada camino disjunto debe comenzar y terminar en posiciones concretas.