La caja fuerte

Un príncipe nigeriano te ha ofrecido el contenido de su caja fuerte si adivinas la combinación

Tienes la siguiente información

- La caja fuerte tiene 3 botones: rojo, verde y azul
- La combinación para abrirlo es de 8 botones

Combinaciones con repetición

Digamos que nuestra combinación es $X = \{x_1, \cdots, x_8\}$

Debemos probar todos los posibles X

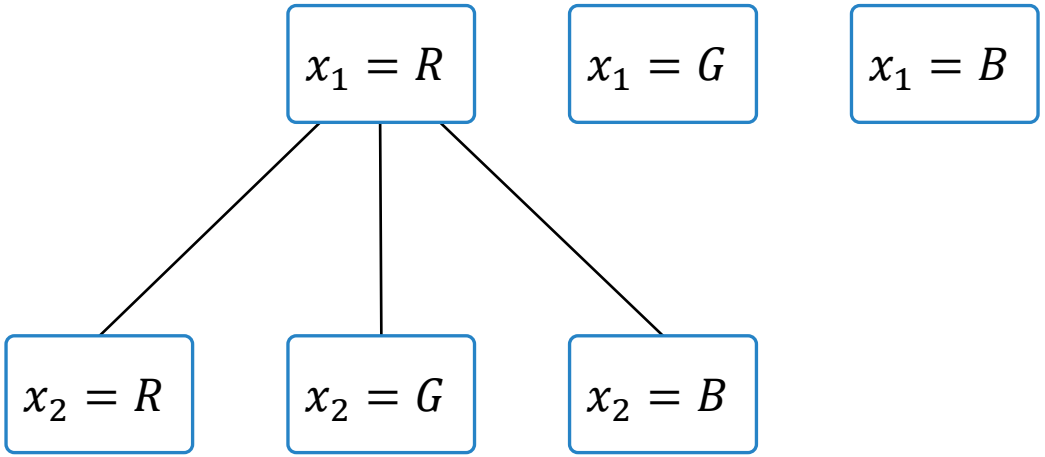
Donde nuestros elementos son los 3 botones $\{R, G, B\}$

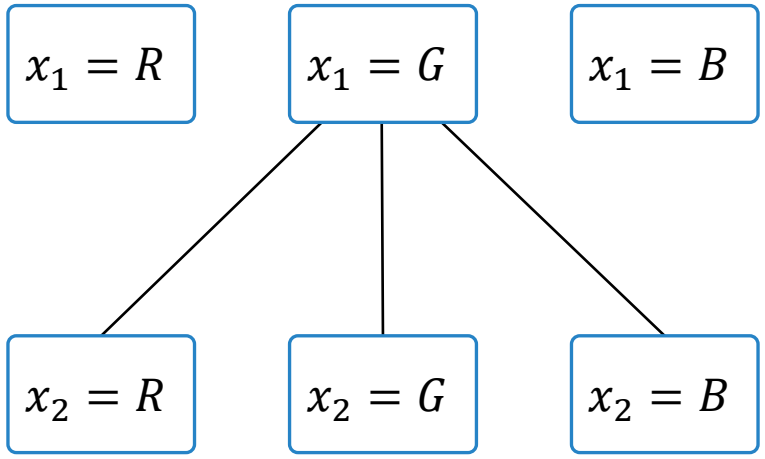
¿Cuantas combinaciones hay que probar en el peor caso?

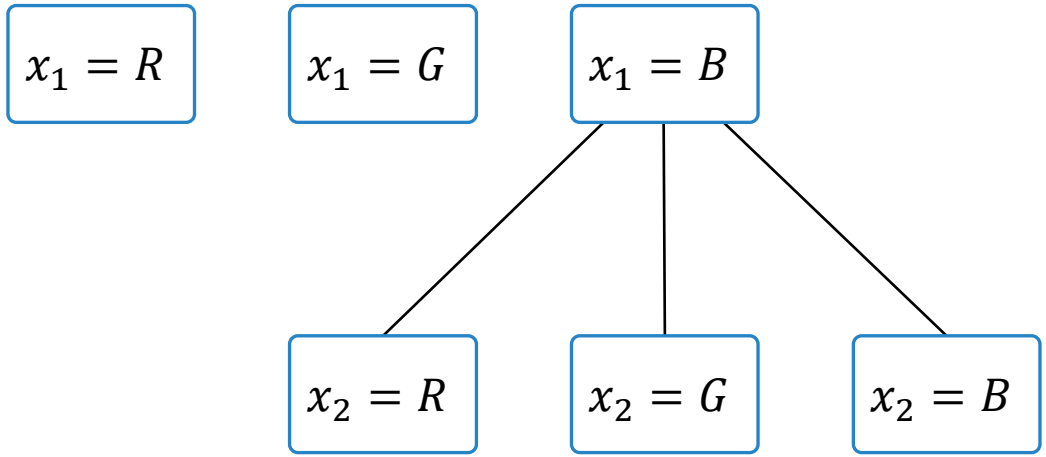
$x_1 = R$

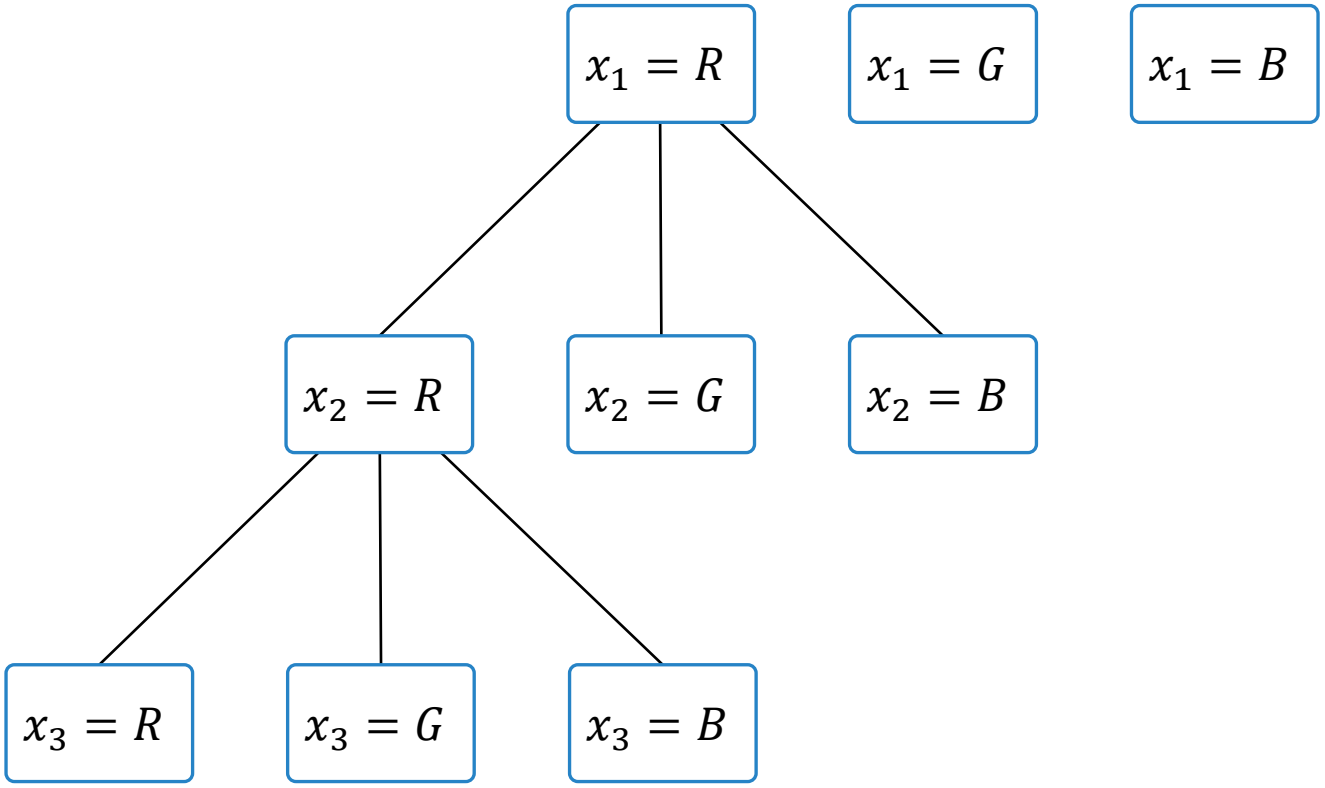
$x_1 = G$

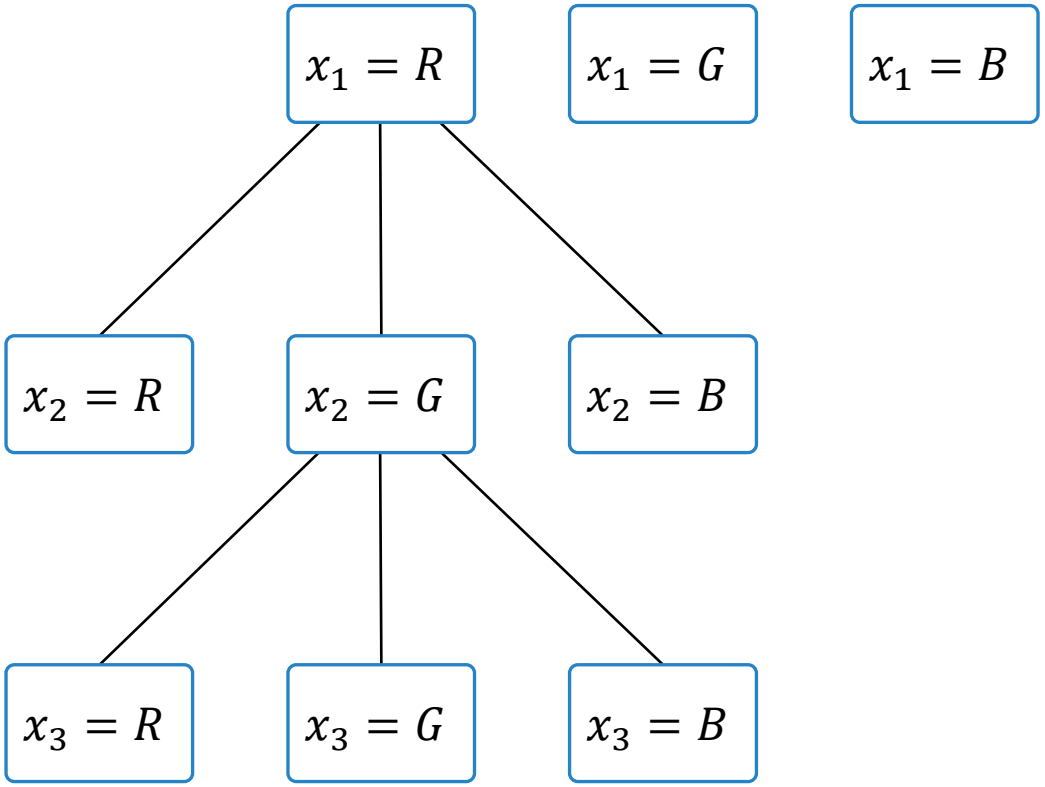
$x_1 = B$

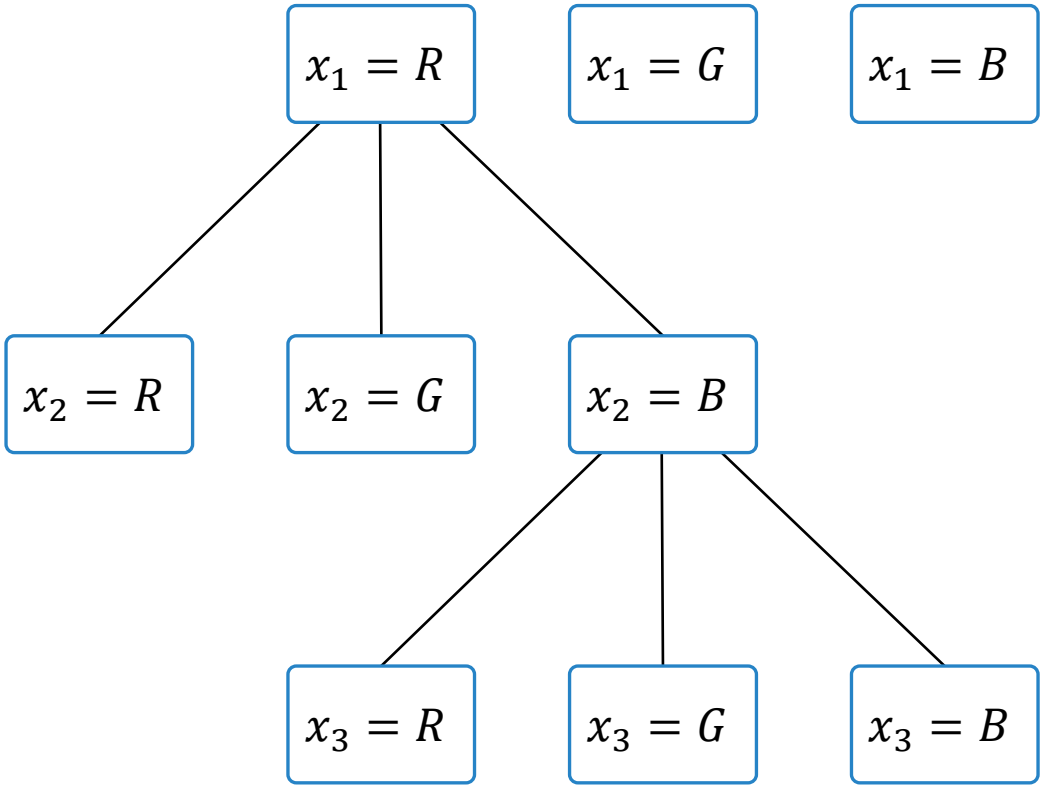

$x_1 = R$
$x_1 = G$
$x_1 = B$
$x_2 = R$
$x_2 = G$
$x_2 = B$


$x_1 = R$
$x_1 = G$
$x_1 = B$
$x_2 = R$
$x_2 = G$
$x_2 = B$


$x_1 = R$
$x_1 = G$
$x_1 = B$
$x_2 = R$
$x_2 = G$
$x_2 = B$


$x_1 = R$
$x_1 = G$
$x_1 = B$
$x_2 = R$
$x_2 = G$
$x_2 = B$
$x_3 = R$
$x_3 = G$
$x_3 = B$


$x_1 = R$
$x_1 = G$
$x_1 = B$
$x_2 = R$
$x_2 = G$
$x_2 = B$
$x_3 = R$
$x_3 = G$
$x_3 = B$


$x_1 = R$
$x_1 = G$
$x_1 = B$
$x_2 = R$
$x_2 = G$
$x_2 = B$
$x_3 = R$
$x_3 = G$
$x_3 = B$

El nivel i corresponde al valor que toma la variable x_i

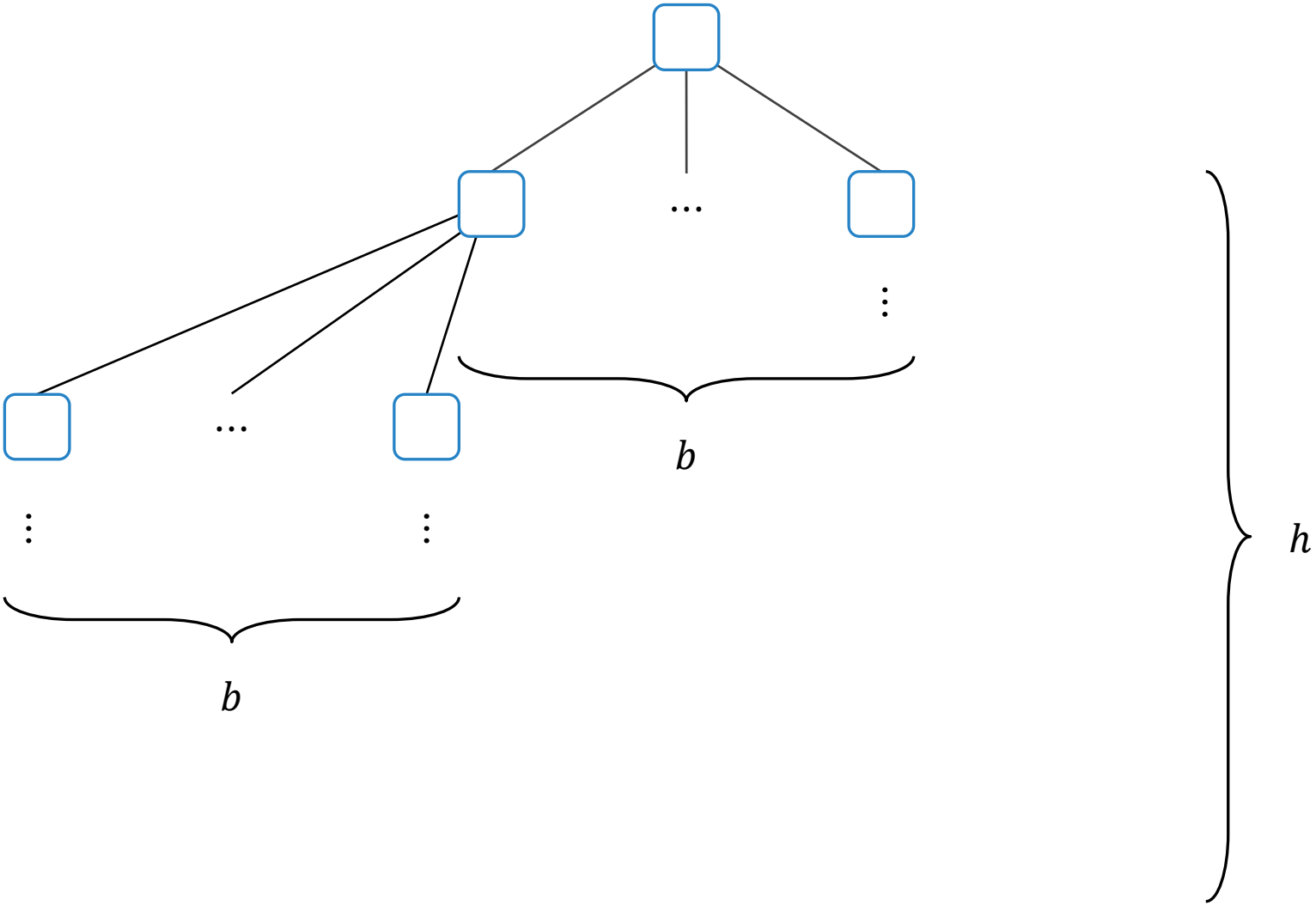
Cada x_i puede tomar 3 valores

Si L_i es la cantidad de nodos del nivel i, tenemos que $L_1 = 3$, y

$$L_i = 3L_{i-1}$$

Por lo tanto,

$$L_8 = 3^8 = 6561$$



El árbol tiene b^h hojas

Combinaciones con restricción

Se nos impone la siguiente restricción:

“No se puede apretar el mismo botón dos veces seguidas”

¿Cuántos X existen ahora?

x_1 puede tomar 3 valores, todos los demás sólo pueden tomar 2

Si L_i es la cantidad de nodos del nivel i, tenemos que $L_1 = 3$, y

$$L_i = 2L_{i-1}$$

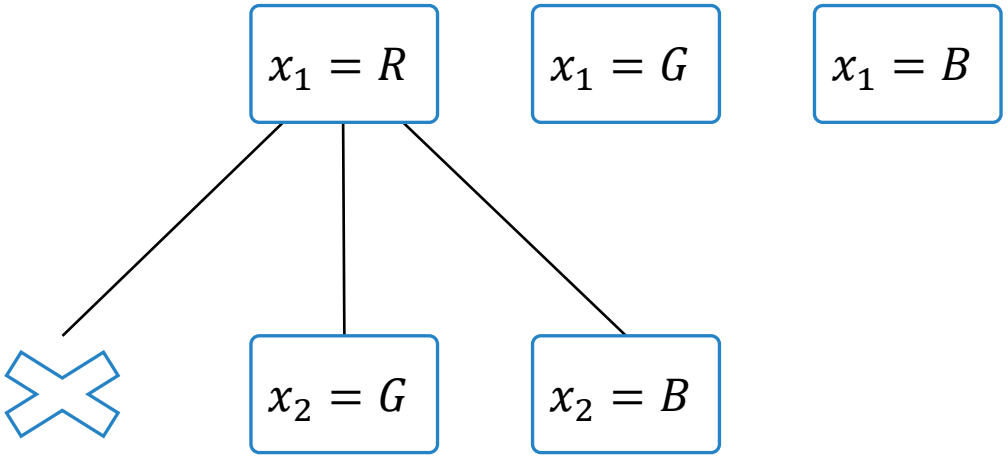
Por lo tanto,

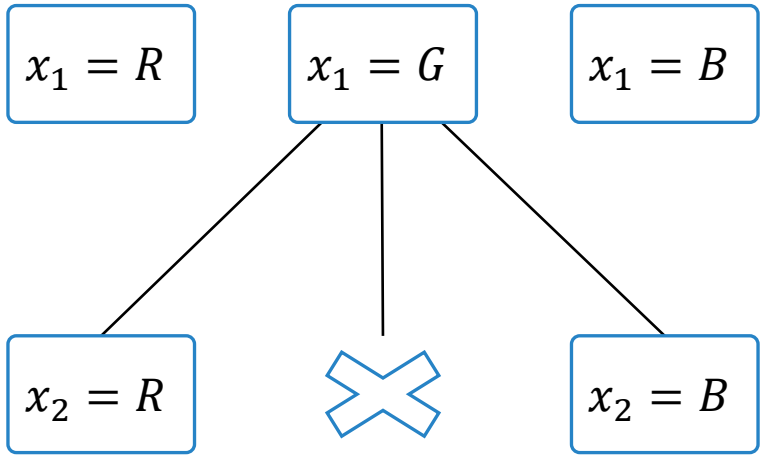
$$L_8 = 3 \cdot 2^7 = 384$$

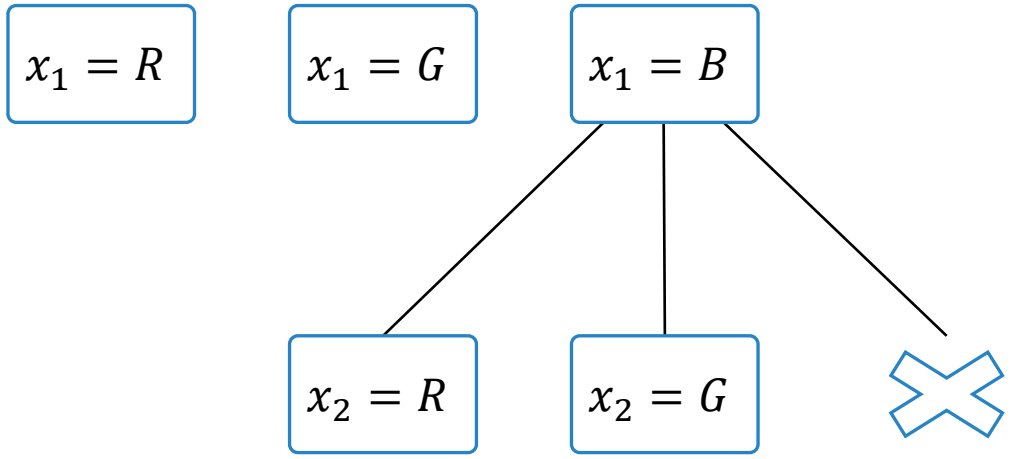
$x_1 = R$

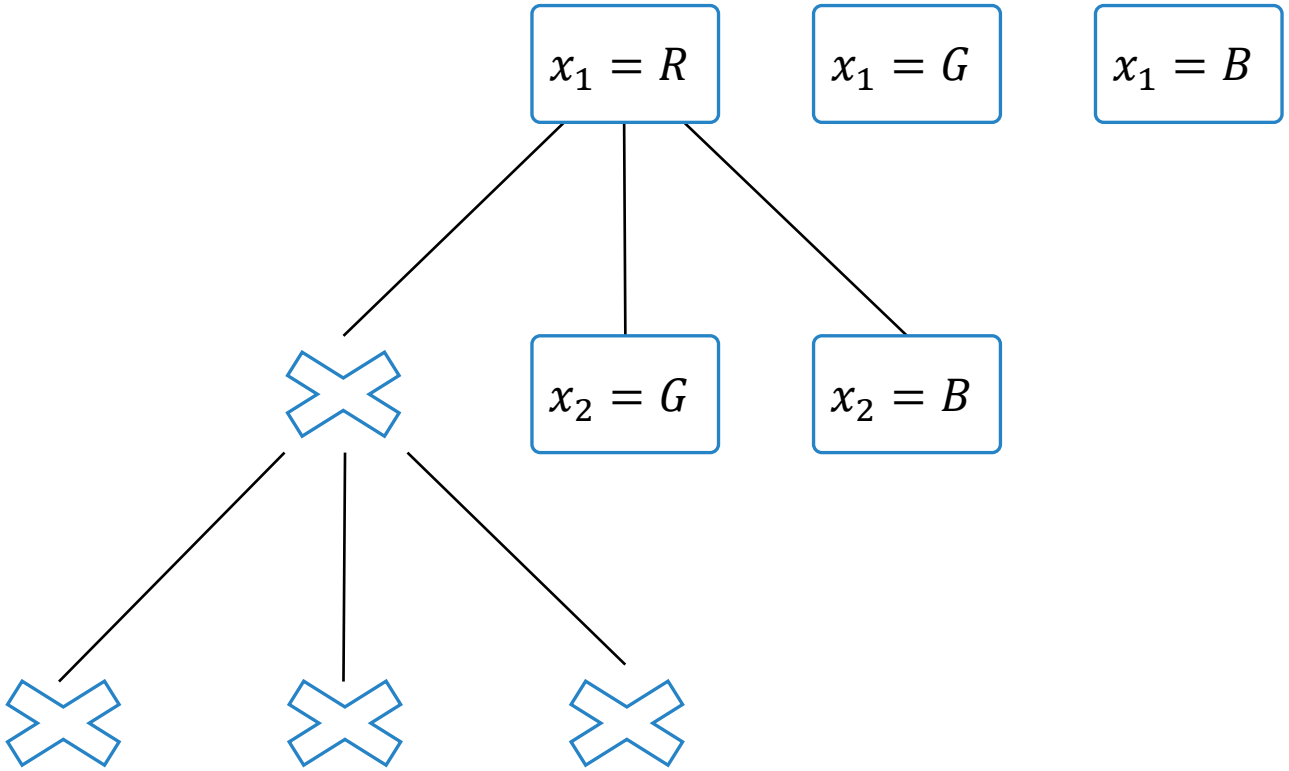
$x_1 = G$

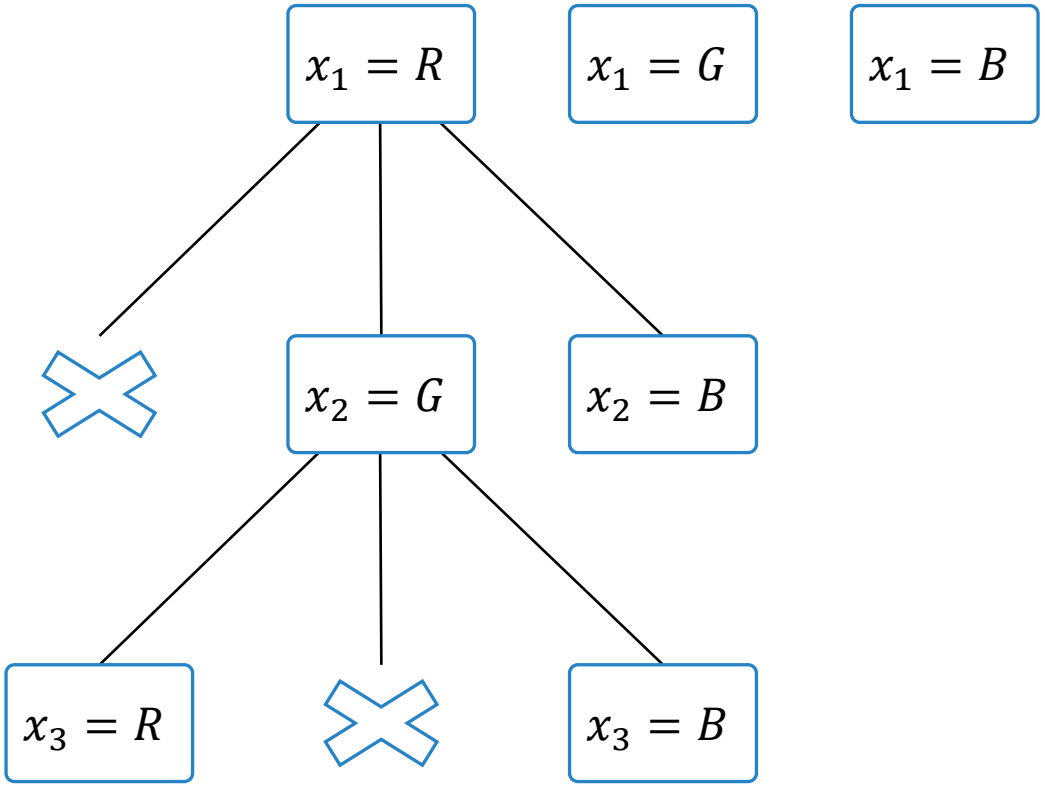
$x_1 = B$

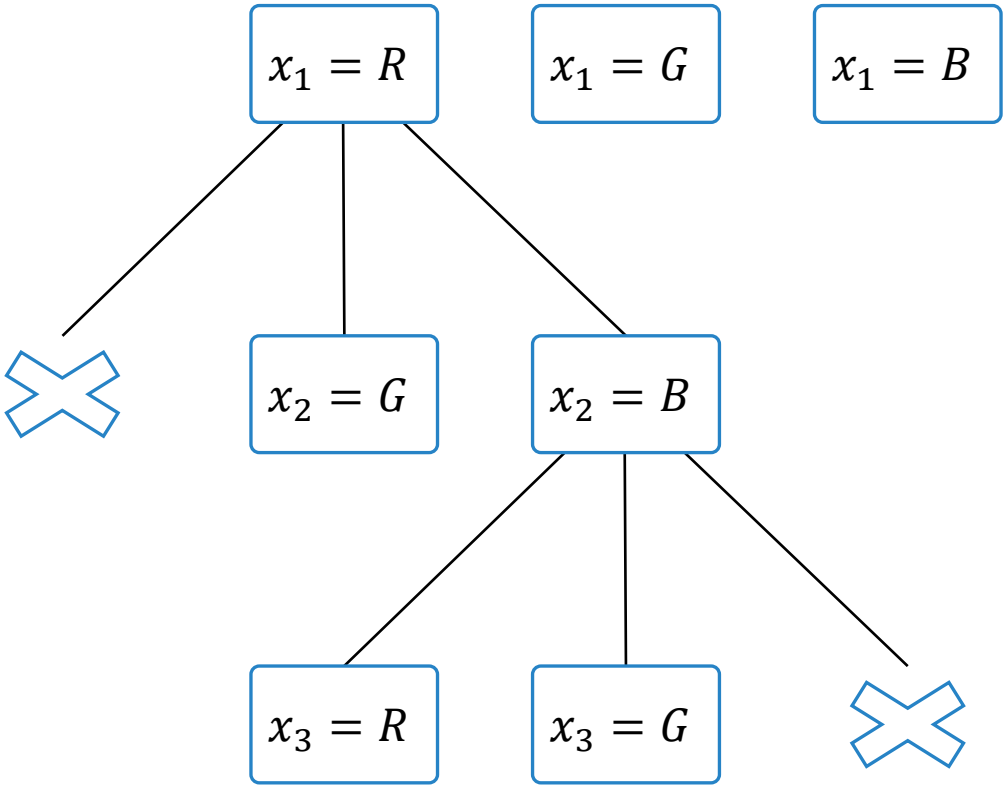

$x_1 = R$
$x_1 = G$
$x_1 = B$
$x_2 = G$
$x_2 = B$


$x_1 = R$
$x_1 = G$
$x_1 = B$
$x_2 = R$
$x_2 = B$


$x_1 = R$
$x_1 = G$
$x_1 = B$
$x_2 = R$
$x_2 = G$


$x_1 = R$
$x_1 = G$
$x_1 = B$
$x_2 = G$
$x_2 = B$


$x_1 = R$
$x_1 = G$
$x_1 = B$
$x_2 = G$
$x_2 = B$
$x_3 = R$
$x_3 = B$


$x_1 = R$
$x_1 = G$
$x_1 = B$
$x_2 = G$
$x_2 = B$
$x_3 = R$
$x_3 = G$

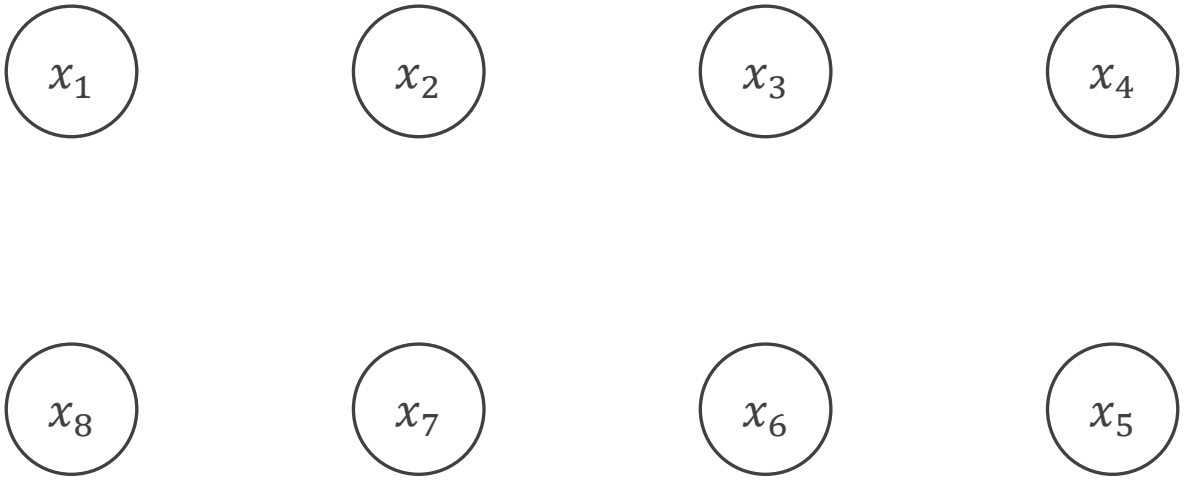
Más restricciones

Se nos imponen las siguientes restricciones:

- $x_4 \neq x_7$
- $x_2 \neq x_8$
- $x_1 \neq x_8$
- $x_2 \neq x_6$

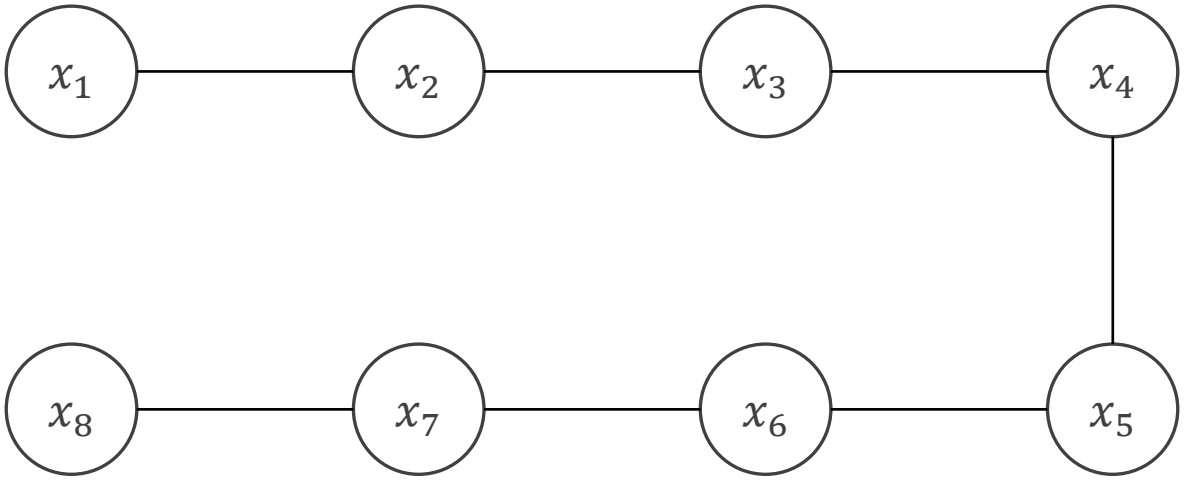
¿Cuántos X existen ahora?

Visualización de las restricciones



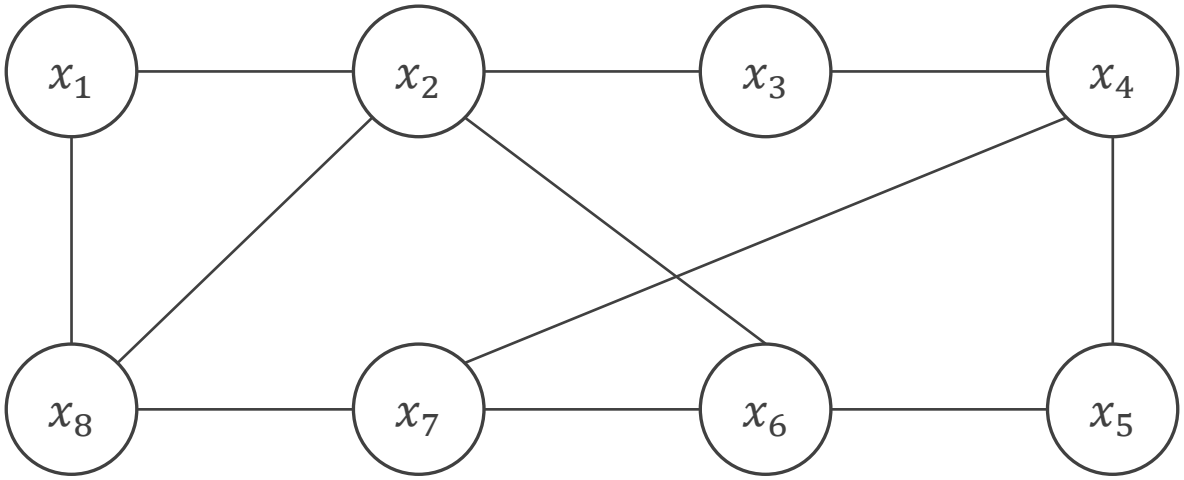
Sin restricciones

Visualización de las restricciones



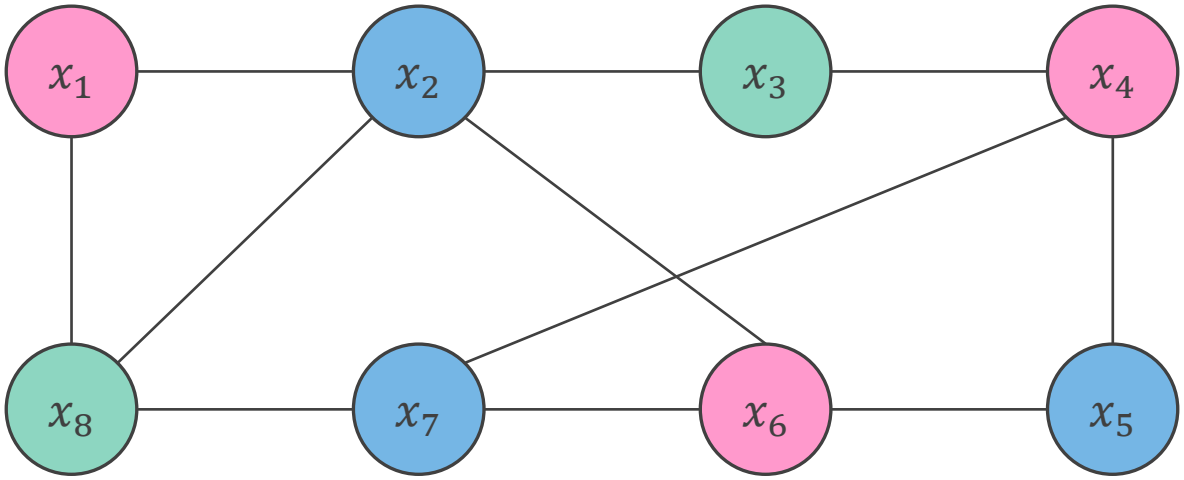
Sin apretar el mismo botón dos veces

Visualización de las restricciones



Agregando el último set de restricciones

Visualización de las restricciones



Un posible X

Coloración de grafos

Dado un grafo no dirigido $\boldsymbol{G(V, E)}$ y k colores, llamamos una **k-coloración** del grafo a una asignación tal que

- Cada nodo tiene asignado un color y sólo uno
- Nodos vecinos tienen colores distintos

¿Cómo podemos encontrar todas las k-coloraciones del grafo?

DFS

```
dfs(V, E):
    for each u in V:
        u.color ← white
    for each u in V:
        if u.color = white:
            dfsVisit(u)
```

```
dfsVisit(u):
    u.color ← gray
    for each v in α[u]:
        if v.color = white:
            dfsVisit(v)
    u.color = black
```

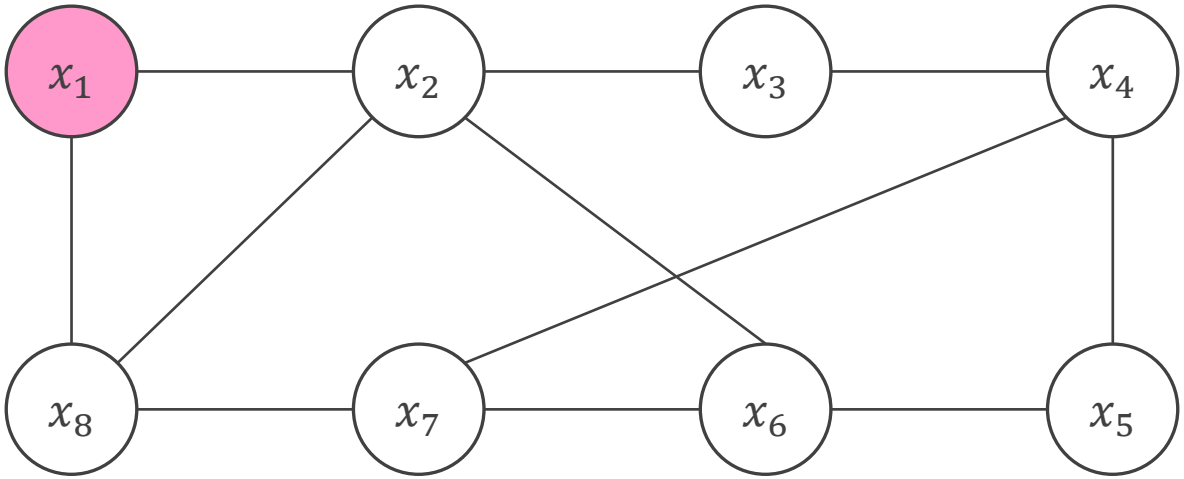
DFS

```
dfs(V, E, k):
    for each u in V:
        u.color ← white
    for each u in V:
        if u.color = white:
            dfsVisit(u, k)
```

```
dfsVisit(u, k):
    u.color ← getColor(u, k)
    for each v in α[u]:
        if v.color = white:
            dfsVisit(v)
```

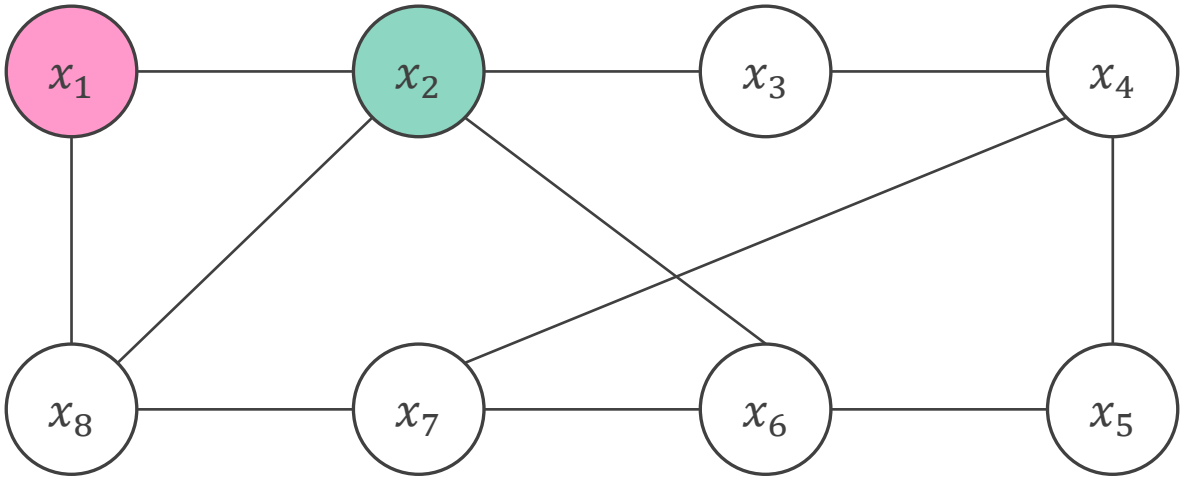
```
getColor(u, k) :
    for c ← 1..k :
        valid ← true
        for v in α[u] :
            if c = v.color:
                valid ← false,  break
        if valid, return c
    return ∅
```

$dfsVisit(x_1, 3)$



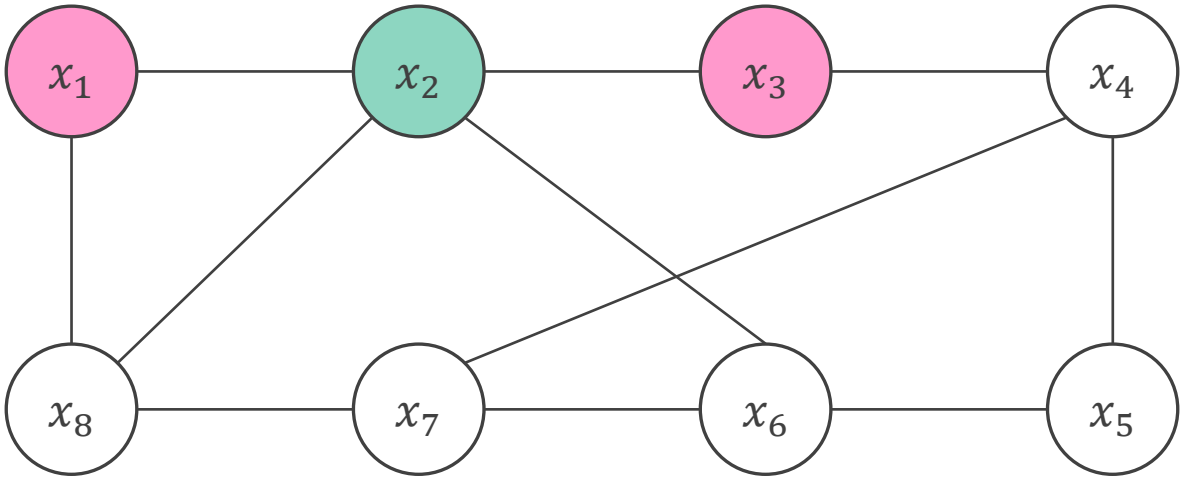
R está disponible

$dfsVisit(x_2, 3)$



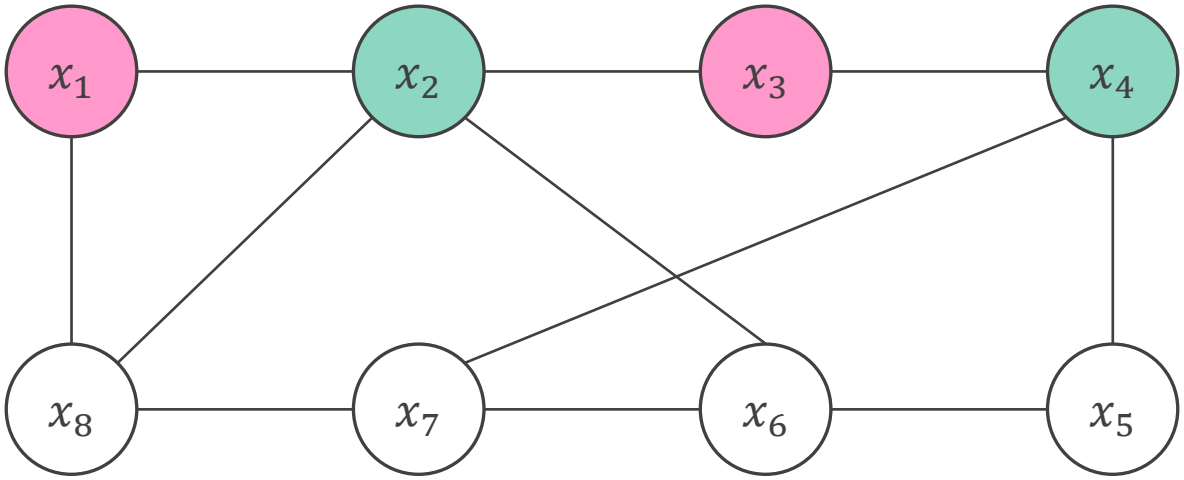
R está ocupado. Elegimos G

$dfsVisit(x_3, 3)$



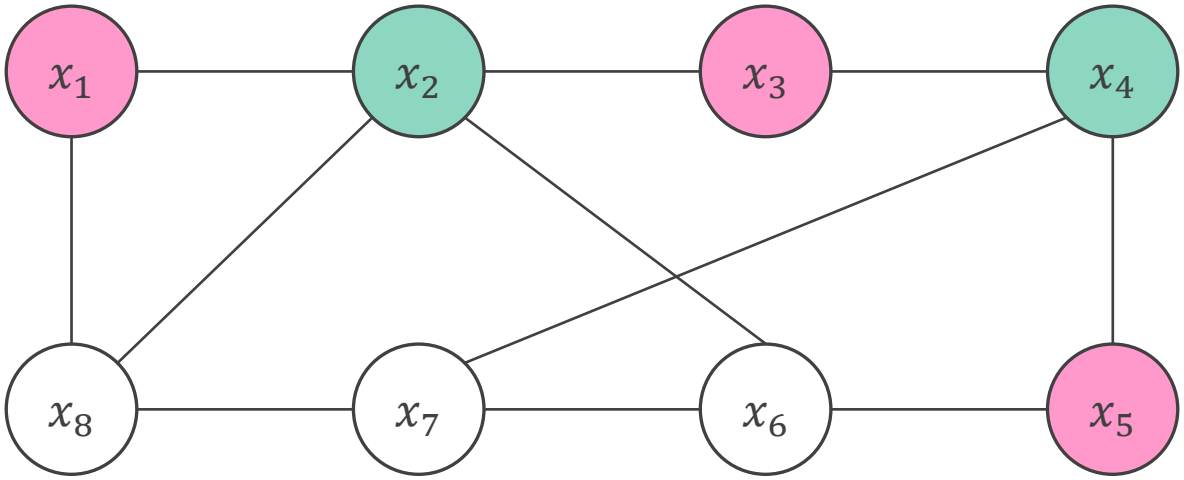
R está disponible.

$dfsVisit(x_4, 3)$



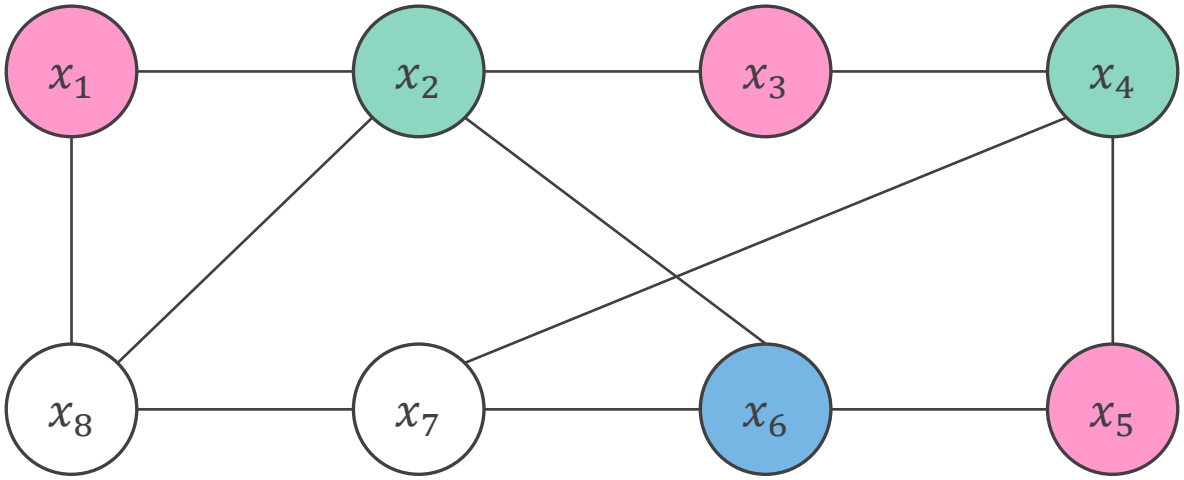
R está ocupado. Elegimos G

$dfsVisit(x_5, 3)$



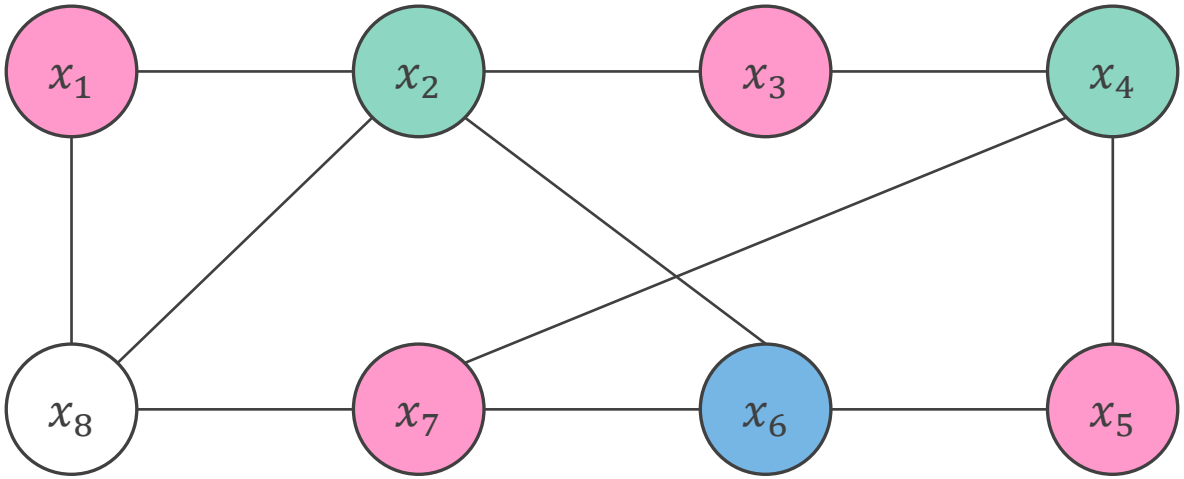
R está disponible.

$dfsVisit(x_6, 3)$



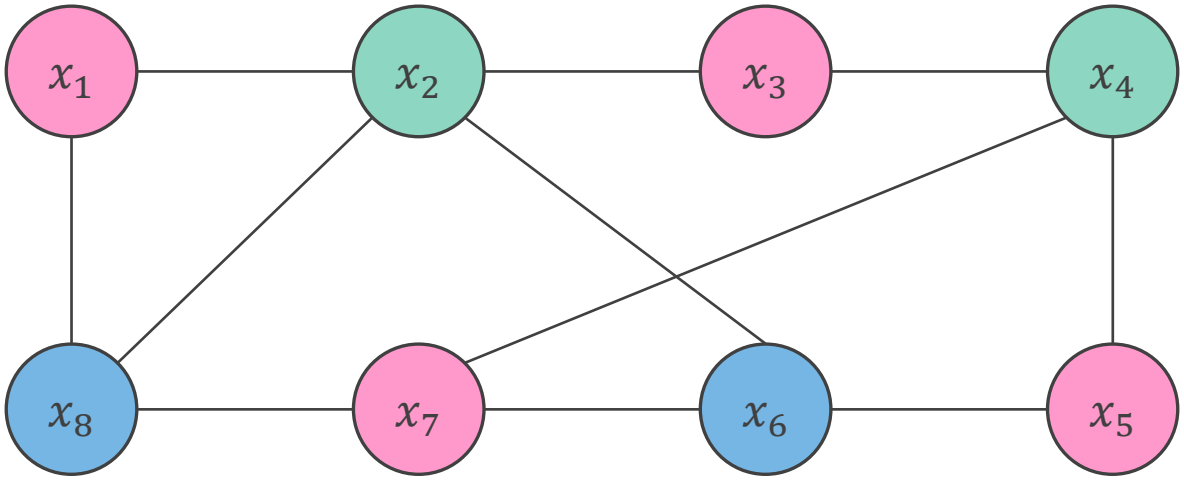
R está ocupado. G también. Elegimos B

$dfsVisit(x_7, 3)$



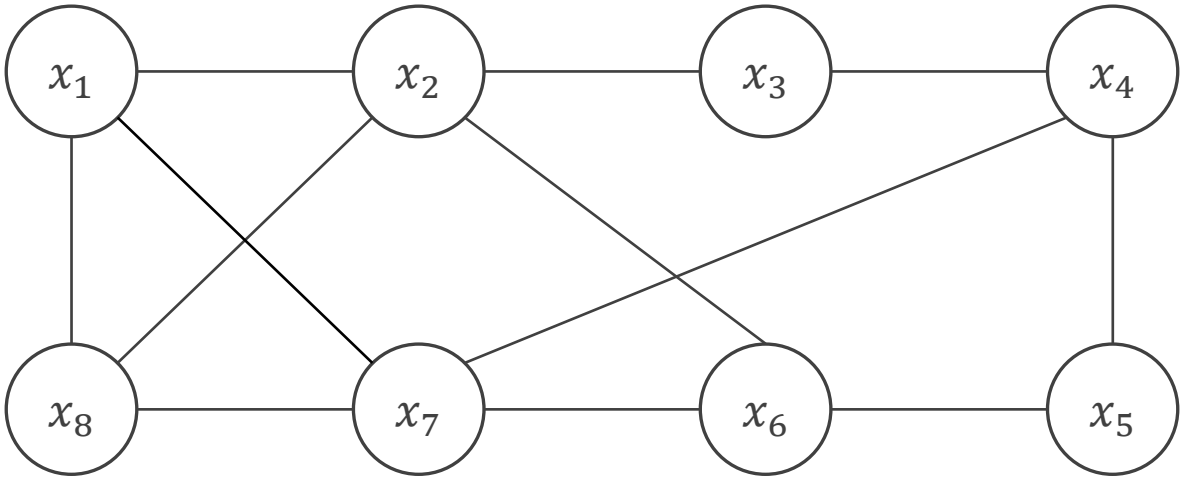
R está disponible

$dfsVisit(x_8, 3)$



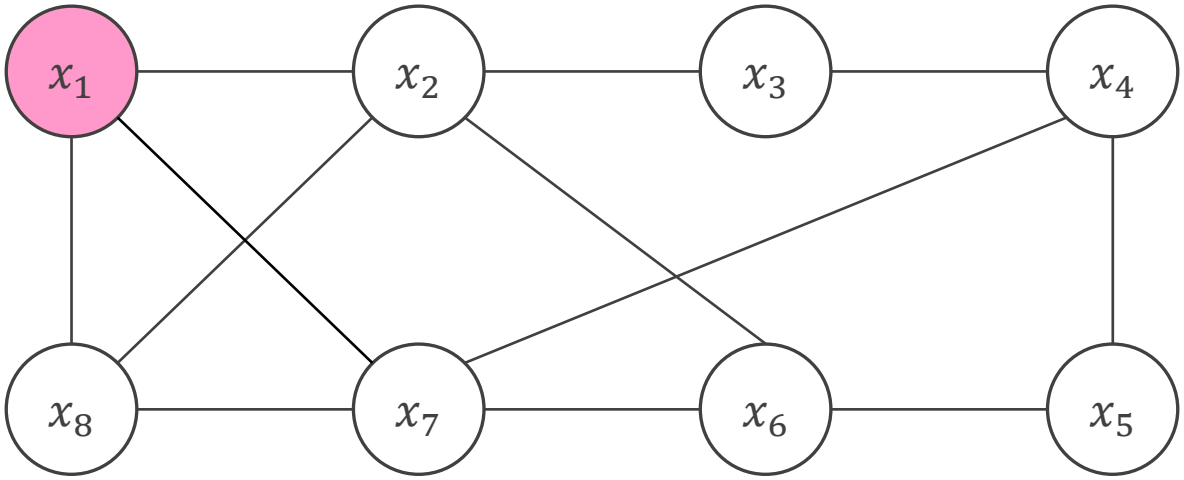
R está ocupado. G también. Elegimos B

Otro grafo



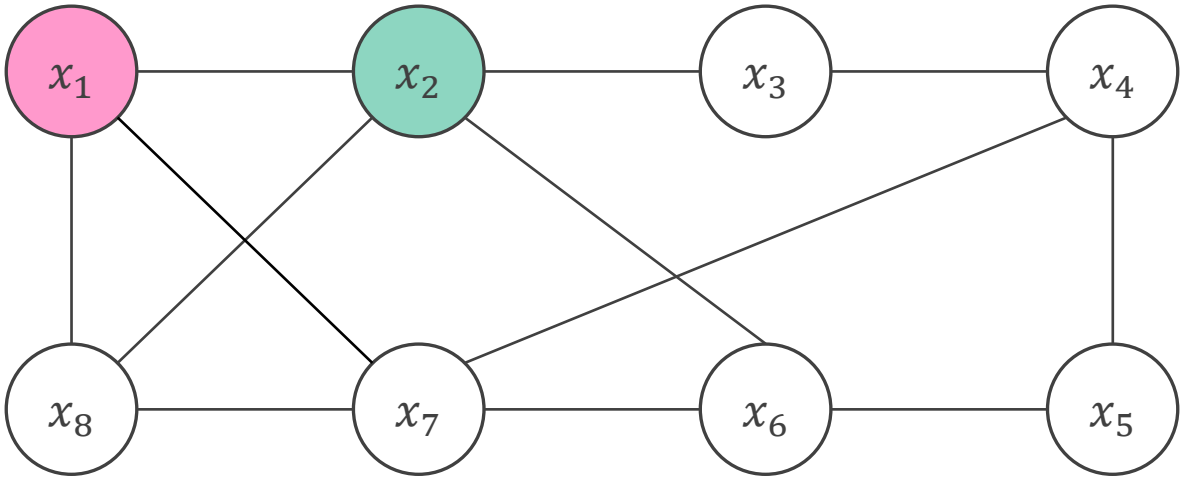
Ahora pintemos un grafo ligeramente diferente

$dfsVisit(x_1, 3)$



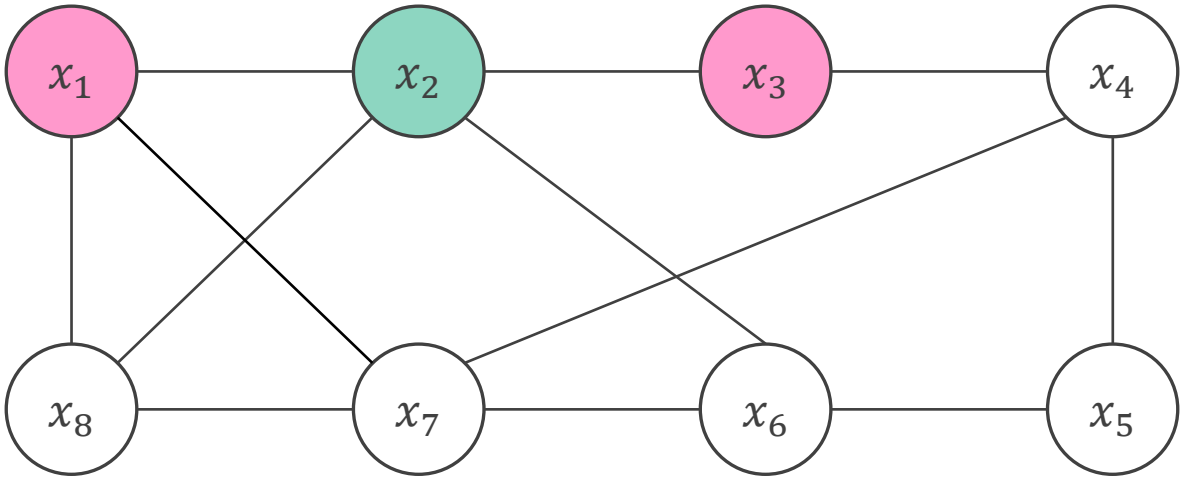
R está disponible

$dfsVisit(x_2, 3)$



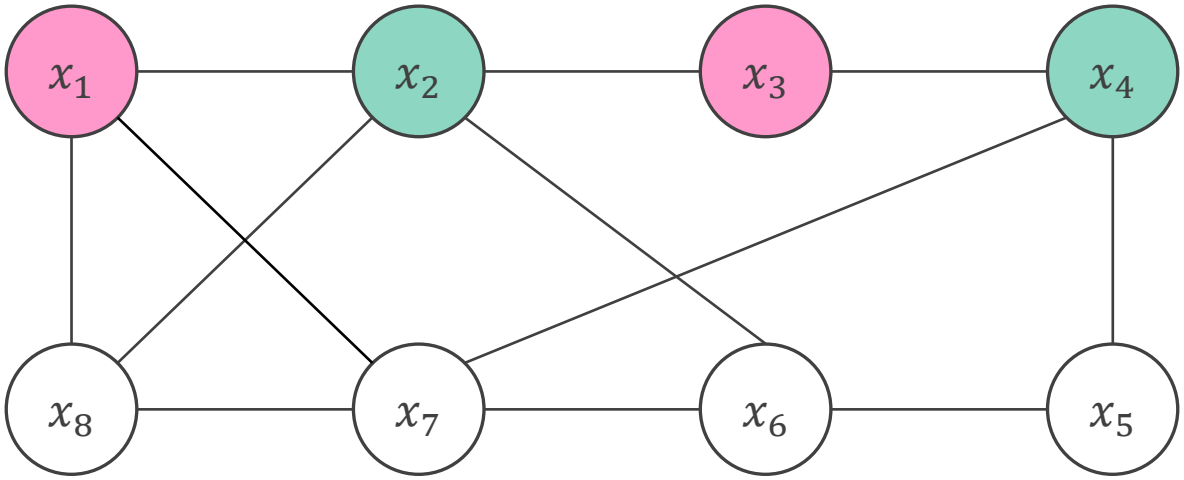
R está ocupado. Elegimos G

$dfsVisit(x_3, 3)$



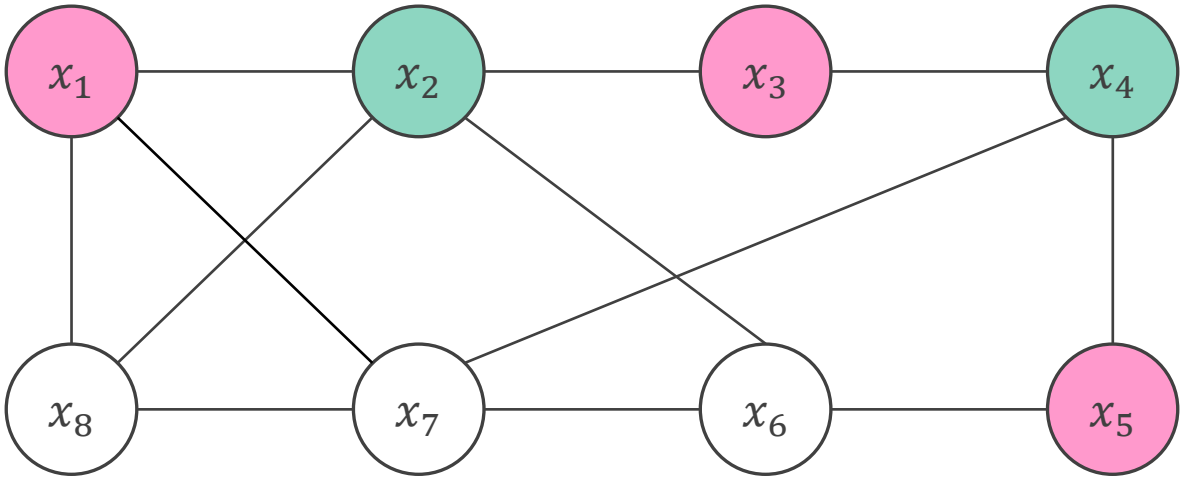
R está disponible.

$dfsVisit(x_4, 3)$



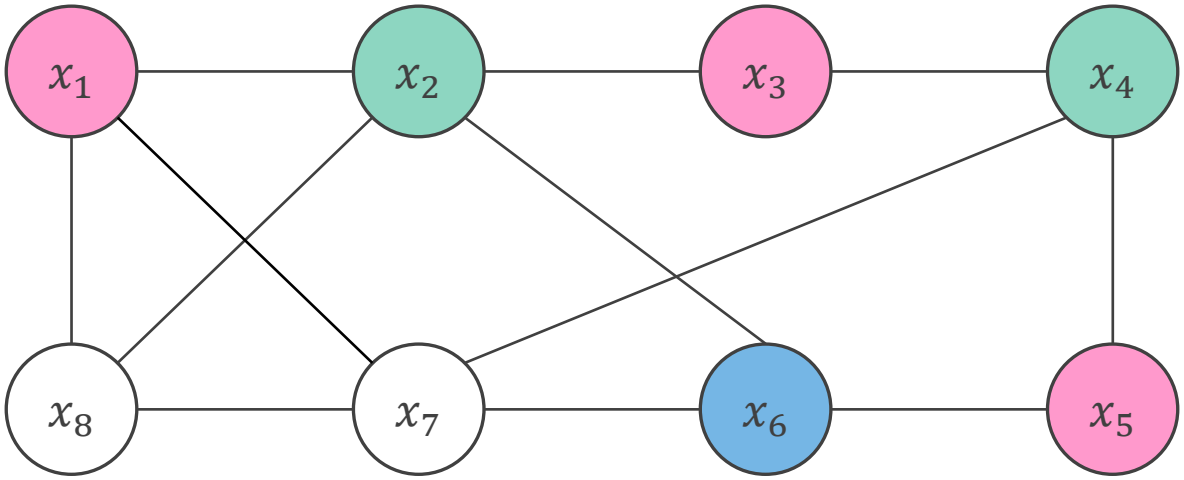
R está ocupado. Elegimos G

$dfsVisit(x_5, 3)$



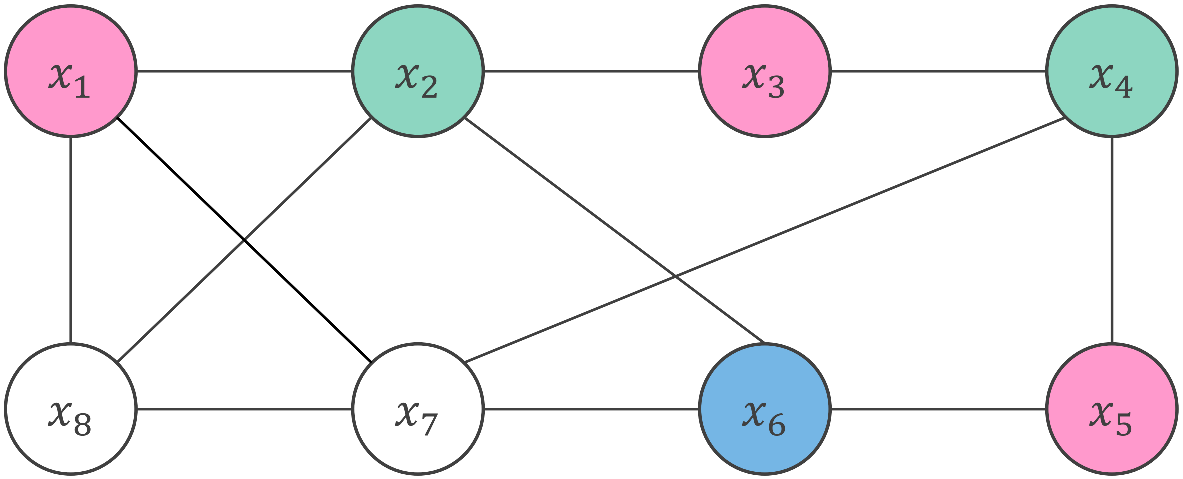
R está disponible.

$dfsVisit(x_6, 3)$



R está ocupado. G también. Elegimos B

$dfsVisit(x_7, 3)$



!!

Traicionados por DFS

Si nos equivocamos, no hay nada que hacer, ya que DFS recorre cada nodo solo una vez.

Esto nos permite quizás encontrar una coloración, pero no todas.

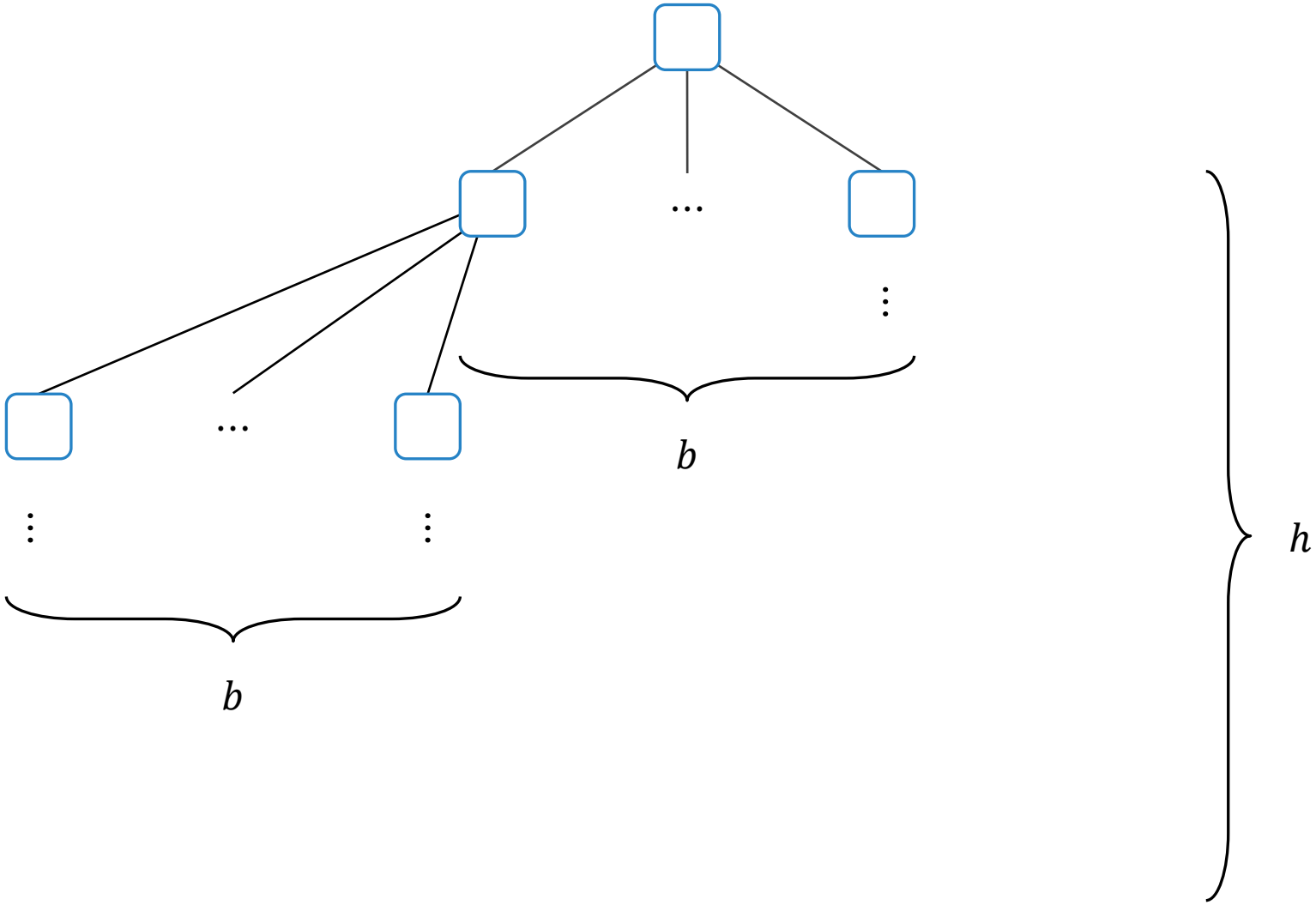
¿Qué podemos hacer?

DFS al rescate, ahora si que si

En lugar de recorrer el grafo de restricciones, podemos recorrer el **árbol de asignaciones**, usando DFS.

Cada arista de este árbol es una asignación, y la cruzamos solo si no viola ninguna restricción.

Si logramos llegar a una hoja, hemos encontrado una combinación válida.



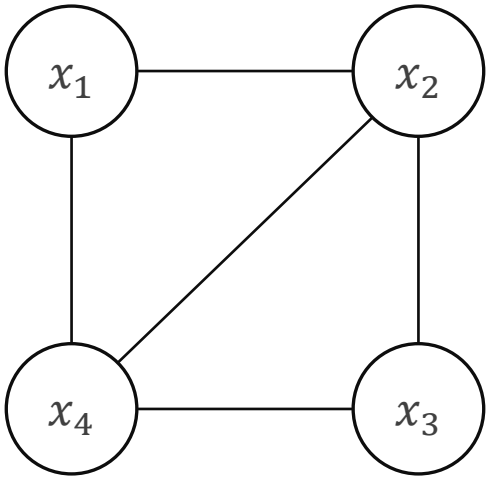
árbol de asignaciones, también conocido como **árbol de búsqueda**

disclaimer

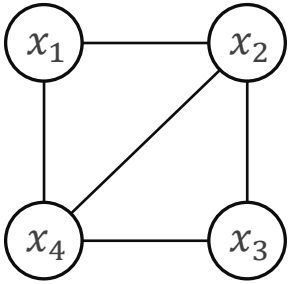
El árbol de asignaciones es una representación del orden en el que probamos las asignaciones / de las asignaciones que no revisamos

No es un árbol que exista en memoria como una estructura de datos

Recorriendo el árbol de búsqueda

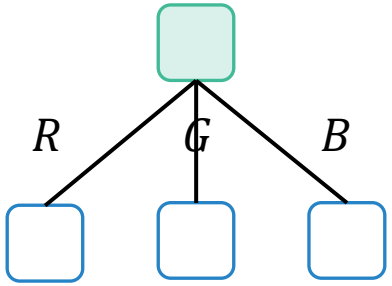


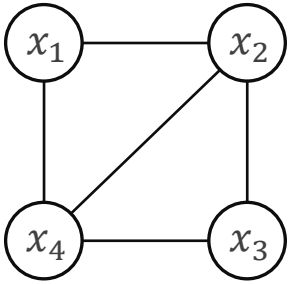
Probemos con este grafo



Inicialmente no hemos asignado nada

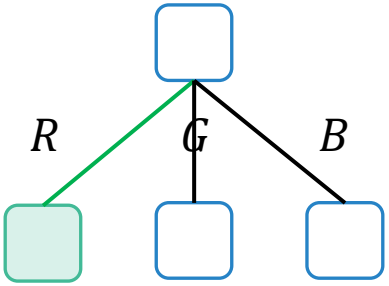
$x_1 =$

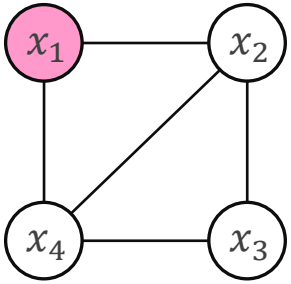




Queremos asignarle un valor a x_1

$x_1 =$

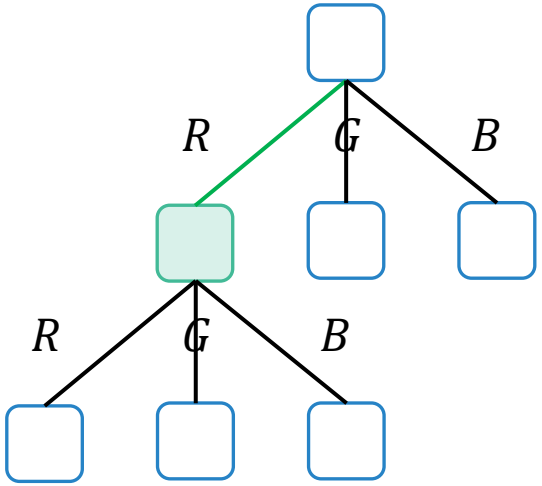


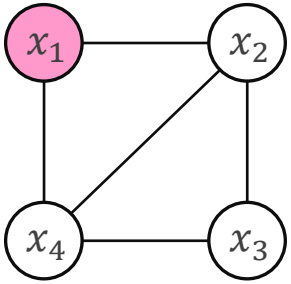


Probamos asignarle R. Como es válido, lo asignamos.

$x_1 =$

$x_2 =$

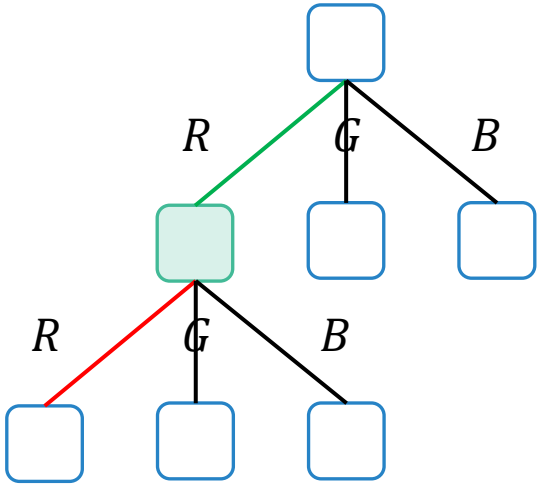


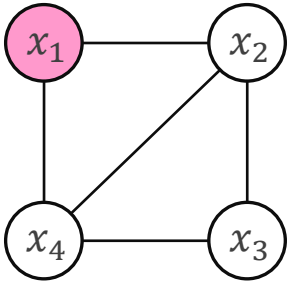


Queremos asignarle un valor a x_2

$x_1 =$

$x_2 =$

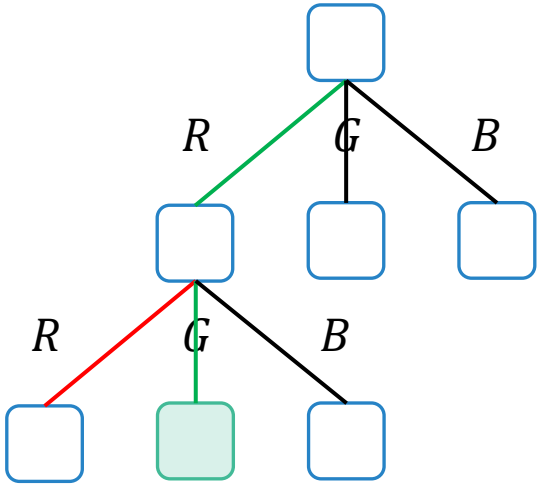


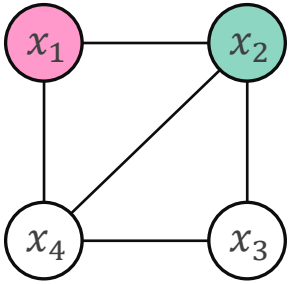


Probamos asignarle R. No es válido, así que probamos el siguiente color.

$x_1 =$

$x_2 =$



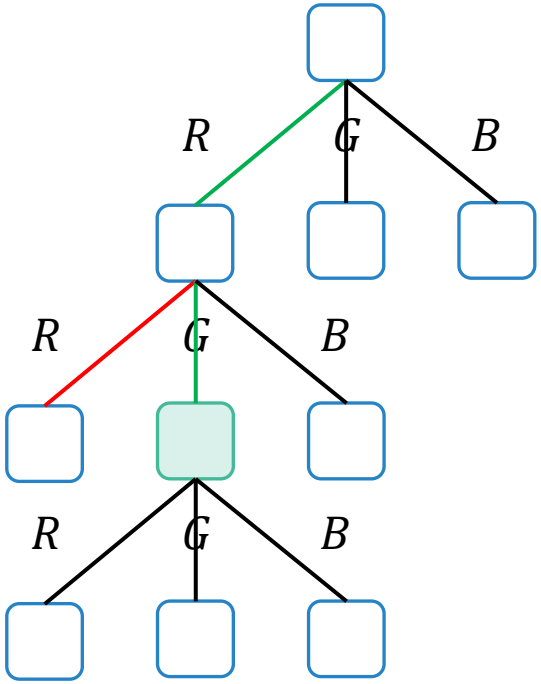


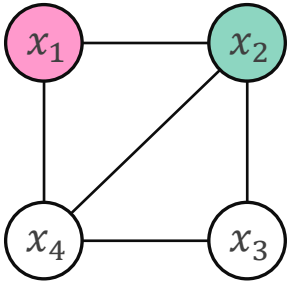
Probamos asignarle G. Como es válido, lo asignamos.

$x_1 =$

$x_2 =$

$x_3 =$



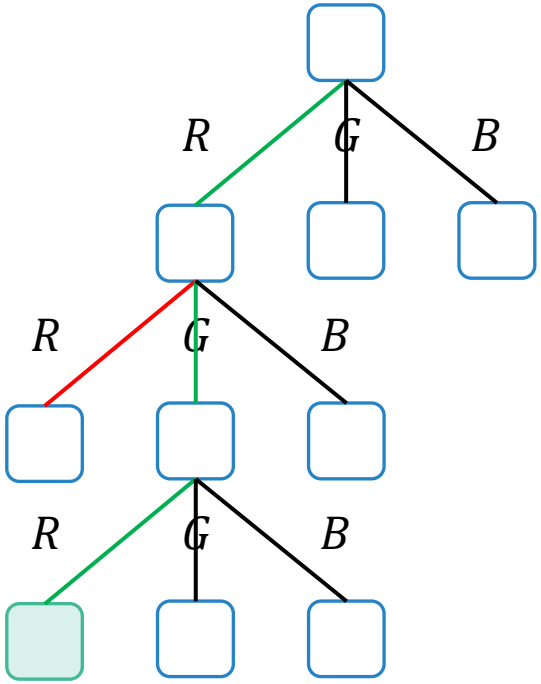


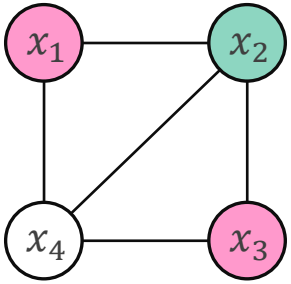
Queremos asignarle un valor a x_3

$x_1 =$

$x_2 =$

$x_3 =$





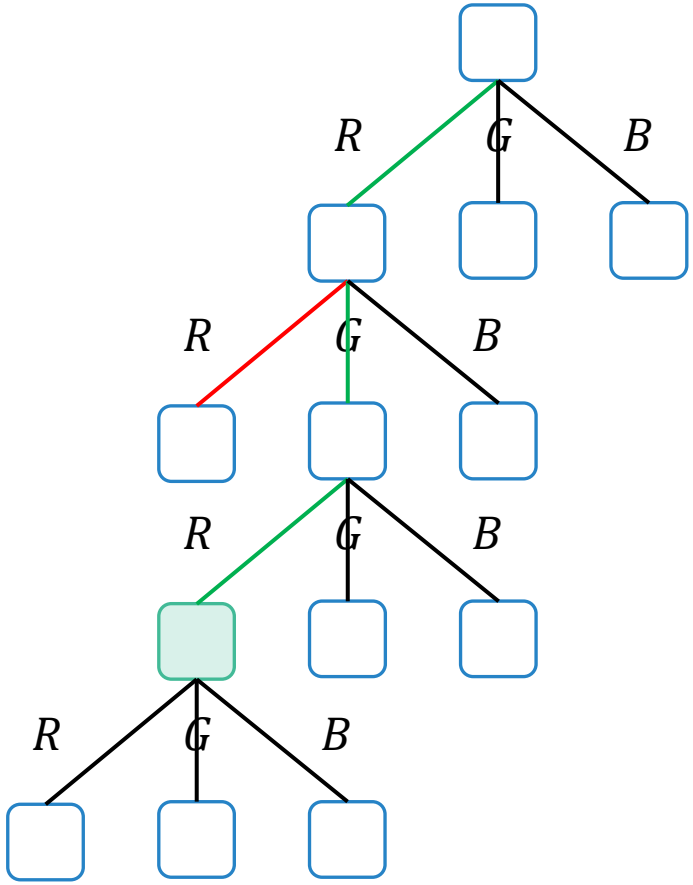
Probamos asignarle R. Como es válido, lo asignamos.

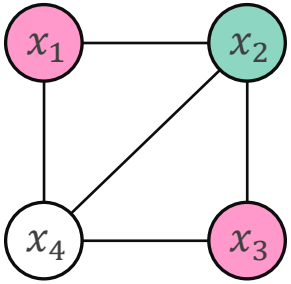
$x_1 =$

$x_2 =$

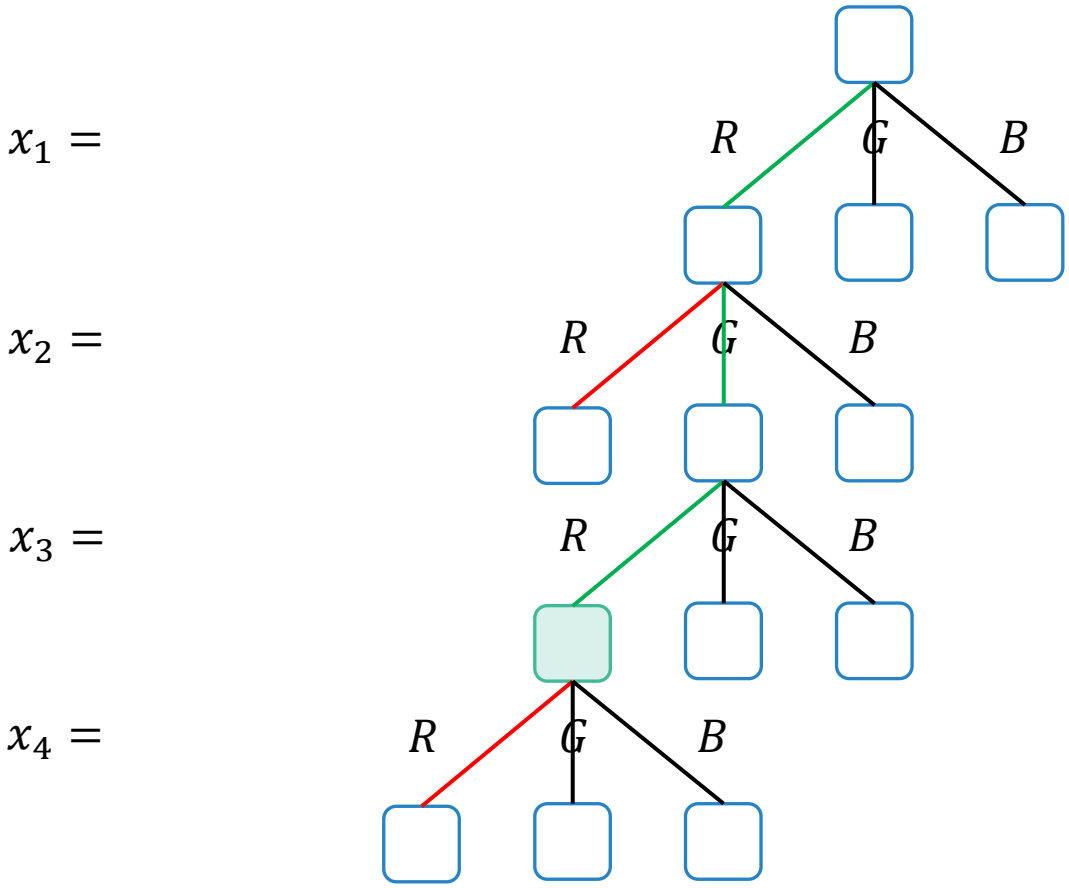
$x_3 =$

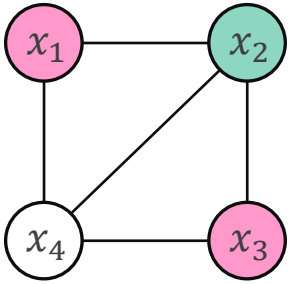
$x_4 =$



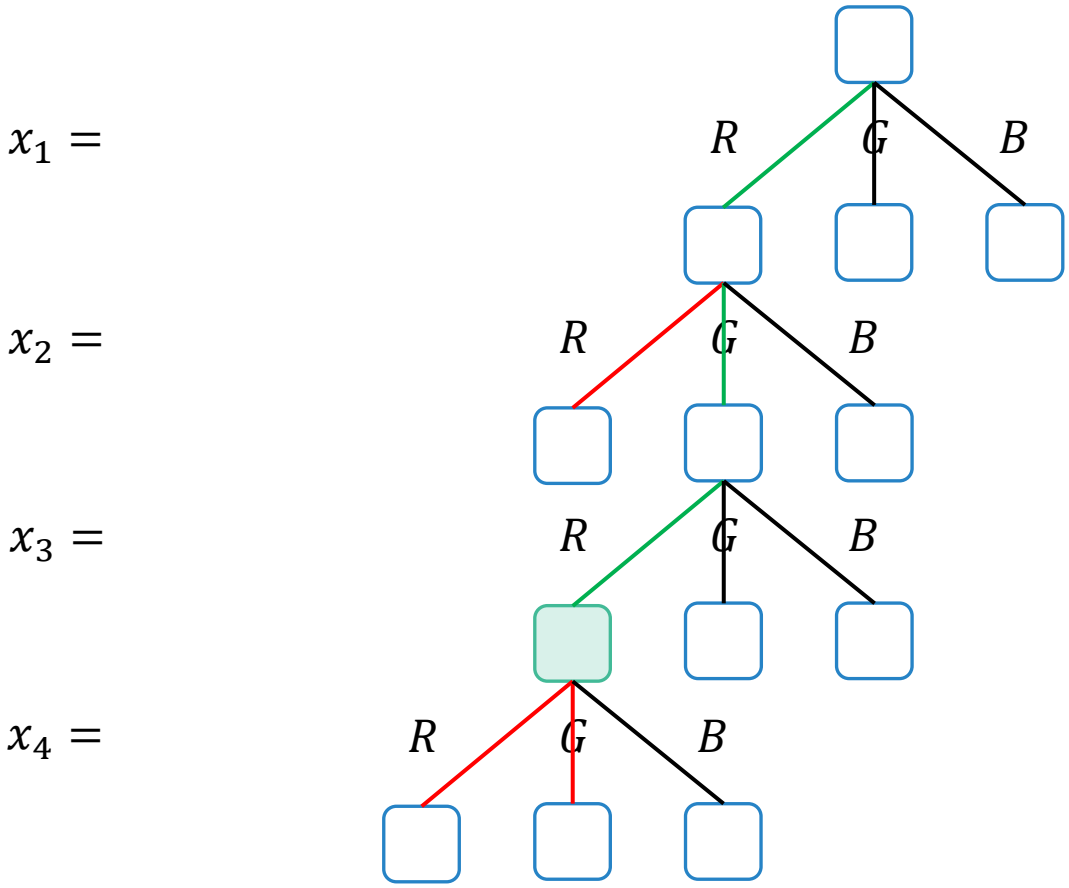


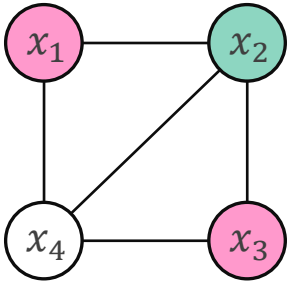
Queremos asignarle un color a x_4



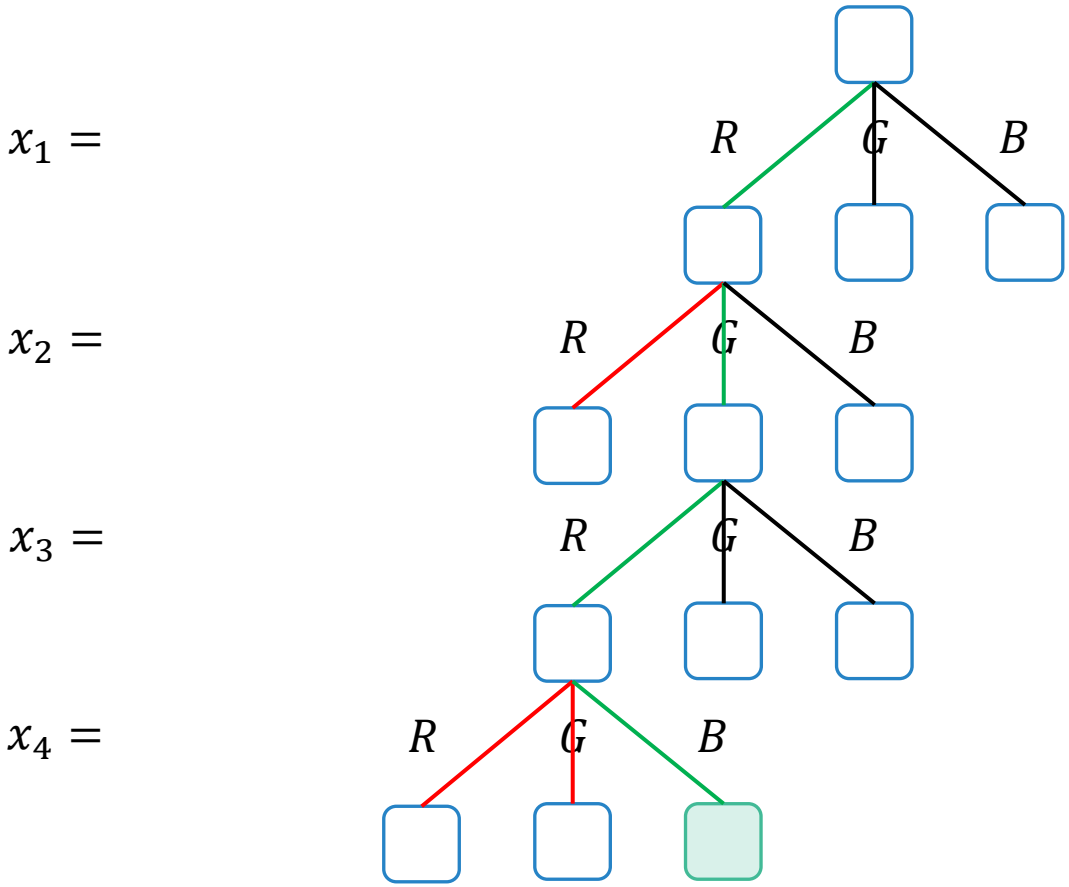


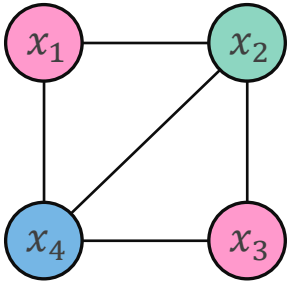
Probamos asignarle R. No es válido, así que probamos el siguiente color.



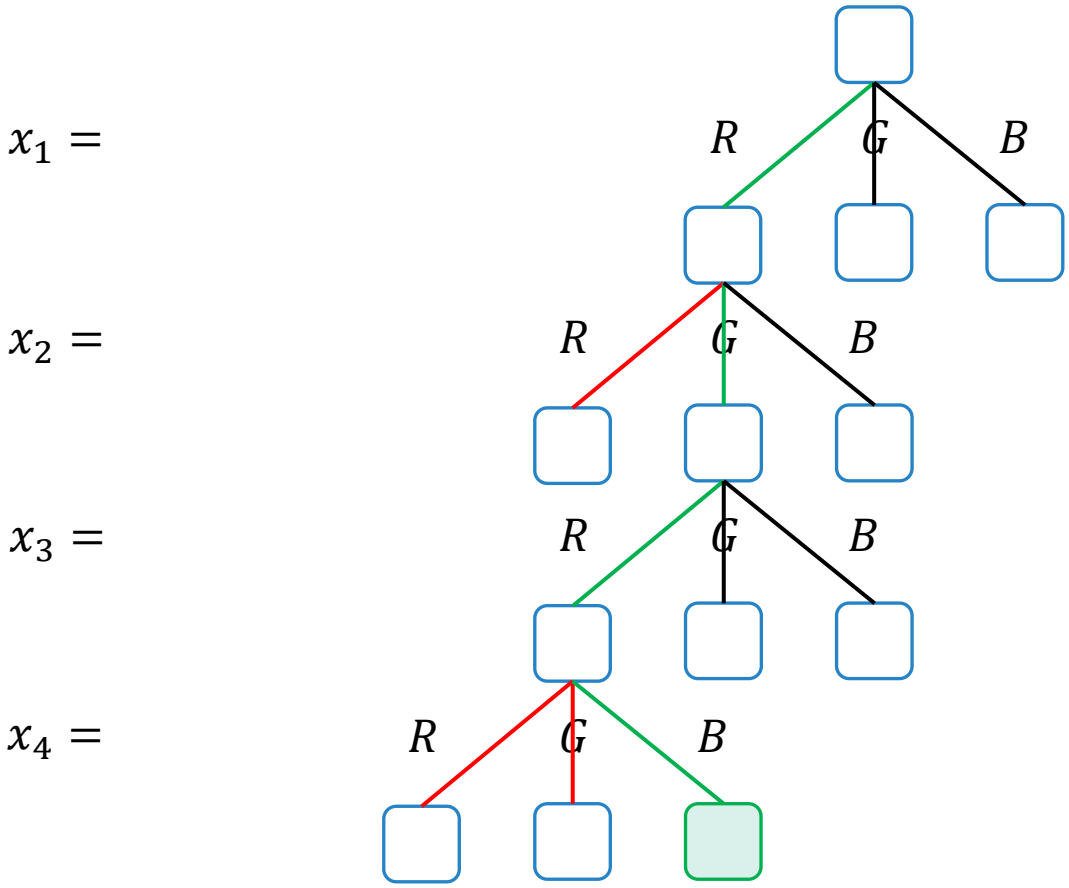


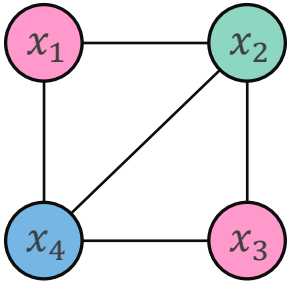
Probamos asignarle G. No es válido, así que probamos el siguiente color.





Probamos asignarle B. Como es válido, lo asignamos.



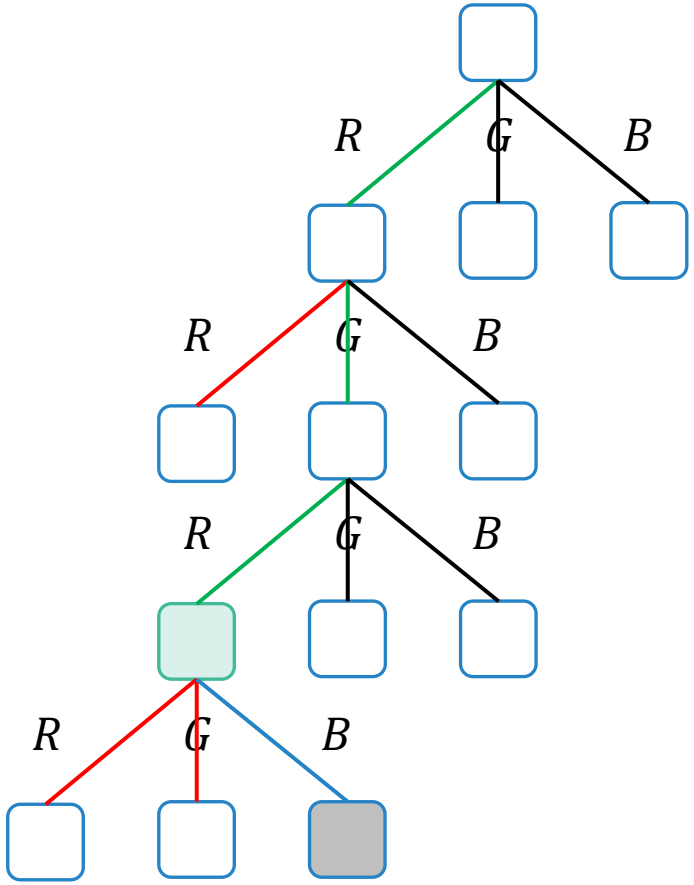


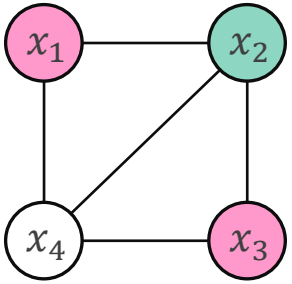
Las aristas verdes representan las asignaciones que conforman esta combinación.

$x_1 =$

$x_2 =$

$x_3 =$



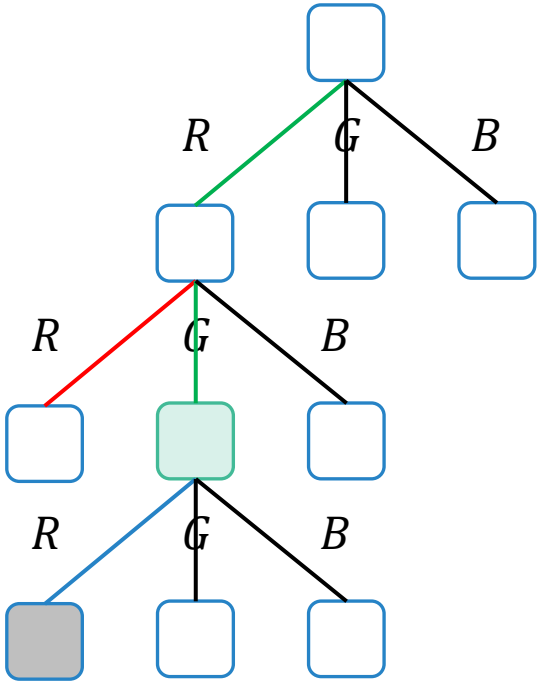


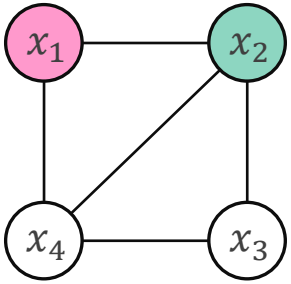
Seguimos probando posibles colores para x_4

$x_1 =$

$x_2 =$

$x_3 =$



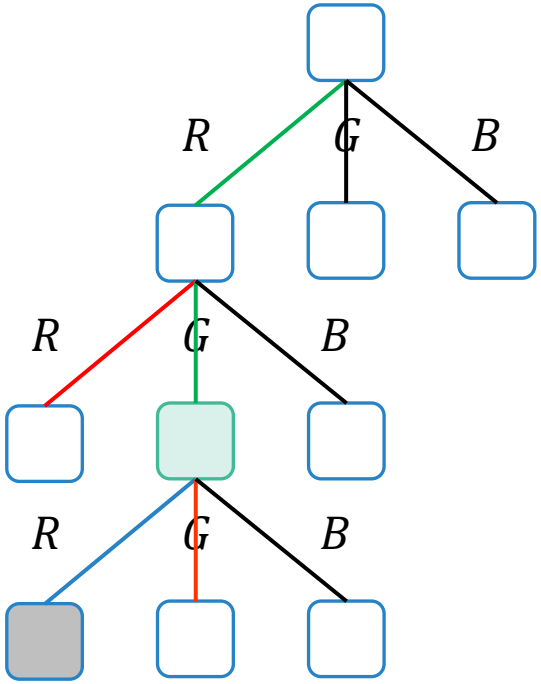


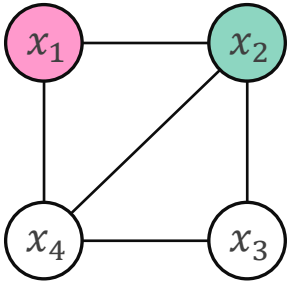
Ya no nos quedan más colores para probar con x_4, volvemos a x_3

$x_1 =$

$x_2 =$

$x_3 =$



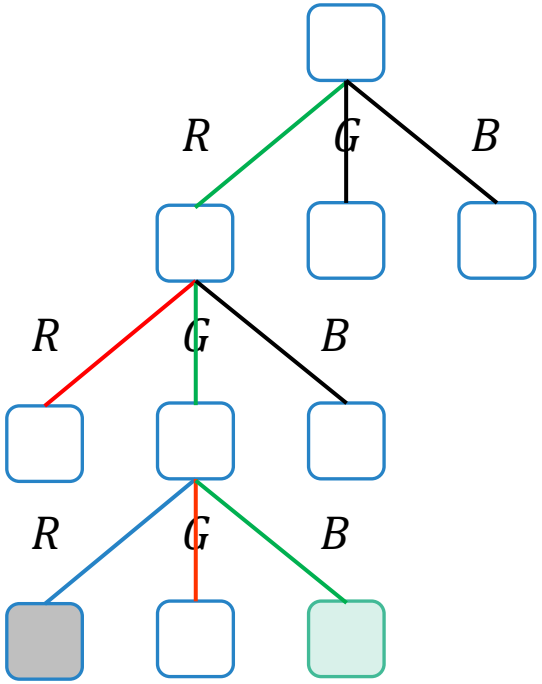


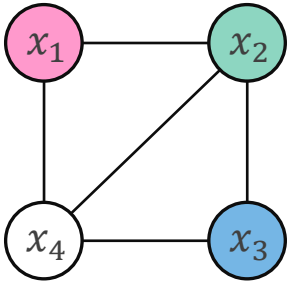
Probamos asignarle G. No es válido, así que probamos el siguiente color.

$x_1 =$

$x_2 =$

$x_3 =$





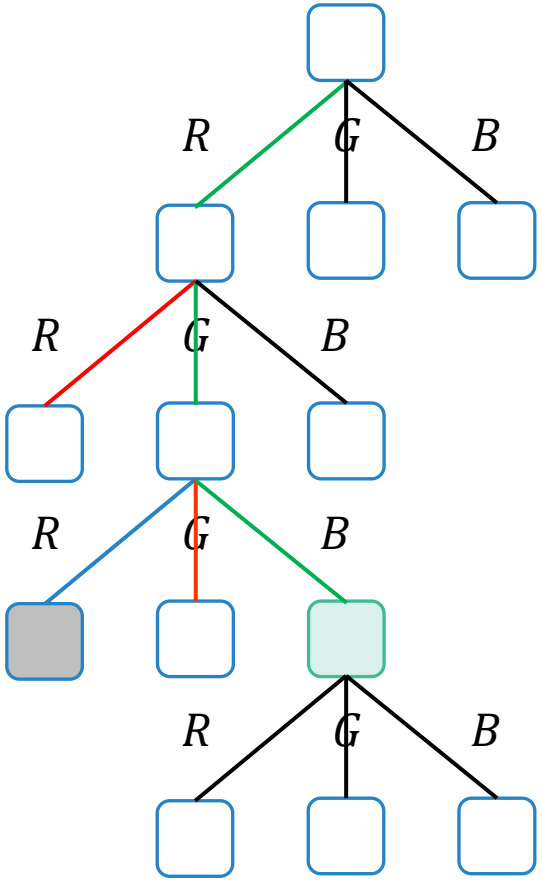
Probamos asignarle B. Como es válido, lo asignamos.

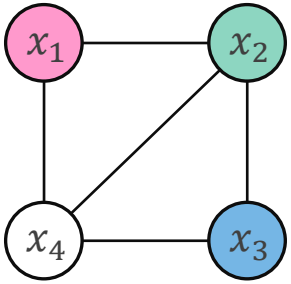
$x_1 =$

$x_2 =$

$x_3 =$

$x_4 =$





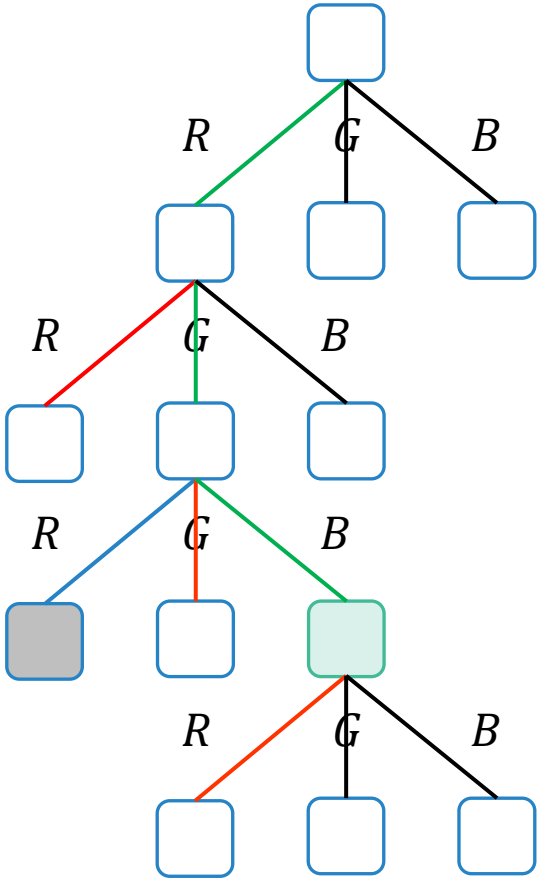
Queremos asignarle un color a x_4

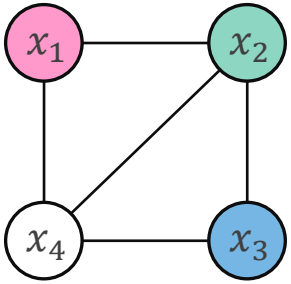
$x_1 =$

$x_2 =$

$x_3 =$

$x_4 =$





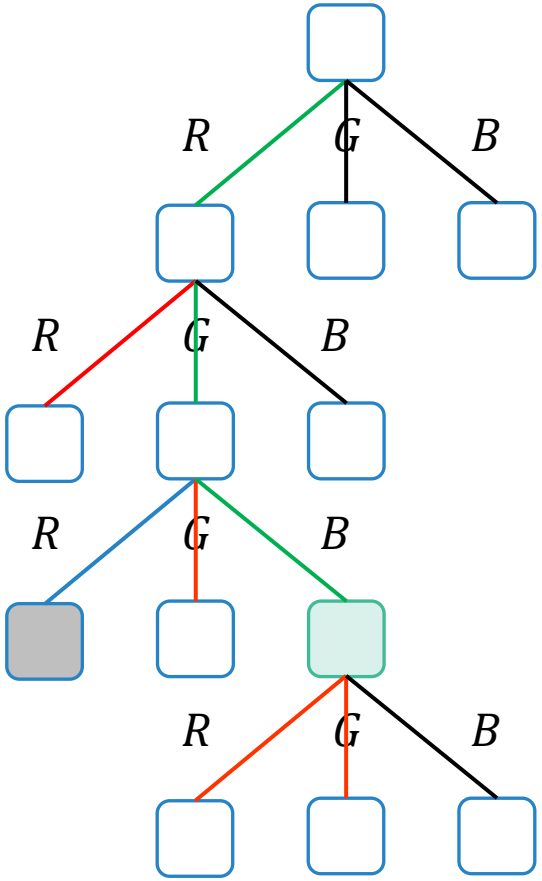
Probamos asignarle R. No es válido, así que probamos el siguiente color.

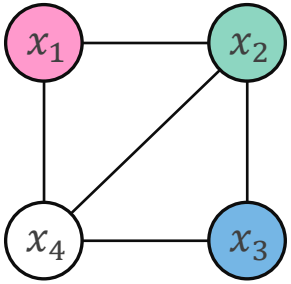
$x_1 =$

$x_2 =$

$x_3 =$

$x_4 =$





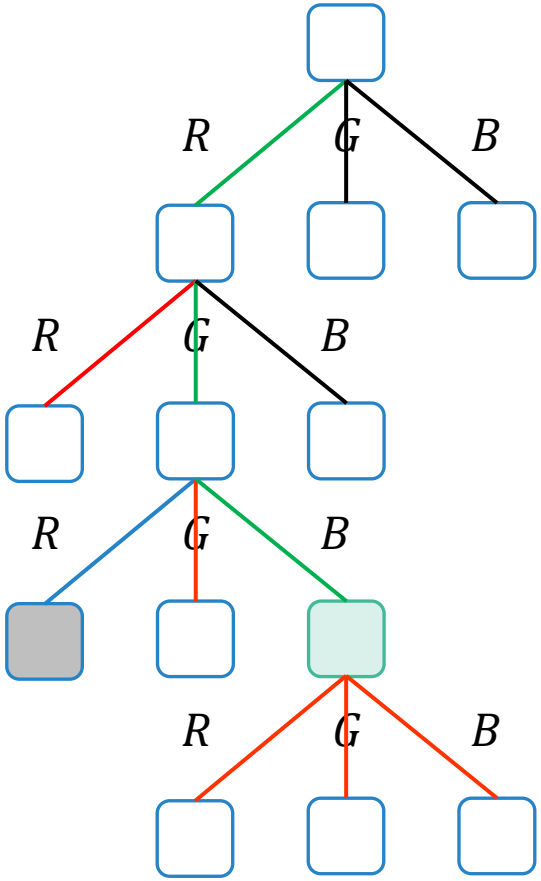
Probamos asignarle G. No es válido, así que probamos el siguiente color.

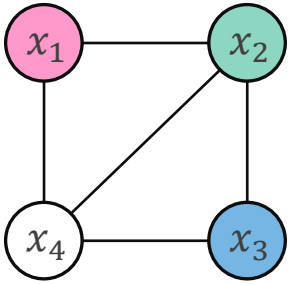
$x_1 =$

$x_2 =$

$x_3 =$

$x_4 =$



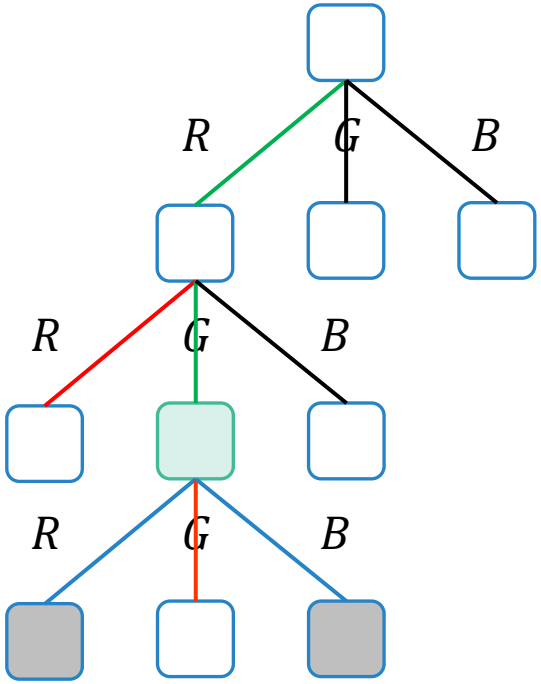


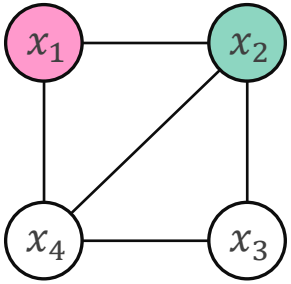
Probamos asignarle B. No es válido, así que probamos el siguiente color.

$x_1 =$

$x_2 =$

$x_3 =$

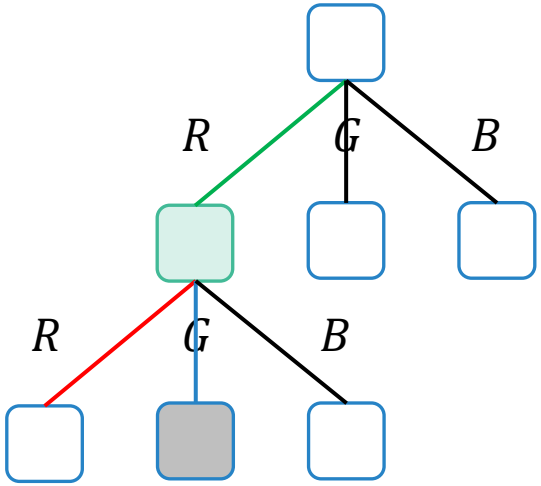


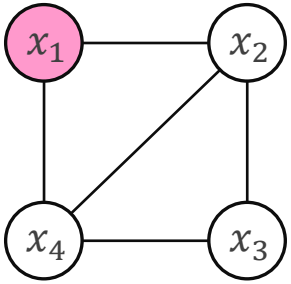


Ya no nos quedan más colores para probar con x_4, volvemos a x_3

$x_1 =$

$x_2 =$





Ya no nos quedan más colores para probar con x_3, volvemos a x_2

ETC

Validez

Una secuencia de asignaciones válidas no necesariamente llega a una solución.

Cuando llegamos a una contradicción, es necesario deshacer la asignación anterior. Este paso se conoce como ***back-track***

```
k − col(n, α, k, x_i):
    if i = n + 1:
        return true
    for c ← 1..k:
        if is valid(x_i, α, c):
            x_i.color ← c
            if k − col(n, α, k, x_{i+1}):
                Tenemos una combinación válida
            x_i.color ← 0
    return false
```

```
is valid(x, α, c):
    for y in α[x]:
        if y.color = c:
            return false
    return true
```

Backtracking

Esto que hicimos ahora es una estrategia algorítmica, conocida como **backtracking**

Es especialmente útil para problemas de **satisfacción de restricciones**

Generalización de BT

Llamamos $\boldsymbol{X}$ al conjunto de **variables**

Los valores que puede tomar cada variable $\boldsymbol{x} \in \boldsymbol{X}$ son el **dominio** de $\boldsymbol{x}$

Las restricciones las abstraemos en un conjunto $\boldsymbol{R}$

```
all − solutions(X, R):
    if is solution(X, R):
        return true
    x = choose unassigned variable(X)
    for v ∈ x.d:
        if is valid(x, v, R):
            assign(x, v)
            if all − solutions(X, R):
                X es una asignación valida
            unassign(x, v)
    return false
```

```
is solvable(X, R):
    if is solution(X, R):
        return true
    x = choose unassigned variable(X)
    for v ∈ x.d:
        if is valid(x, v, R):
            assign(x, v)
            if is solvable(X, R):
                return true
            unassign(x, v)
    return false
```