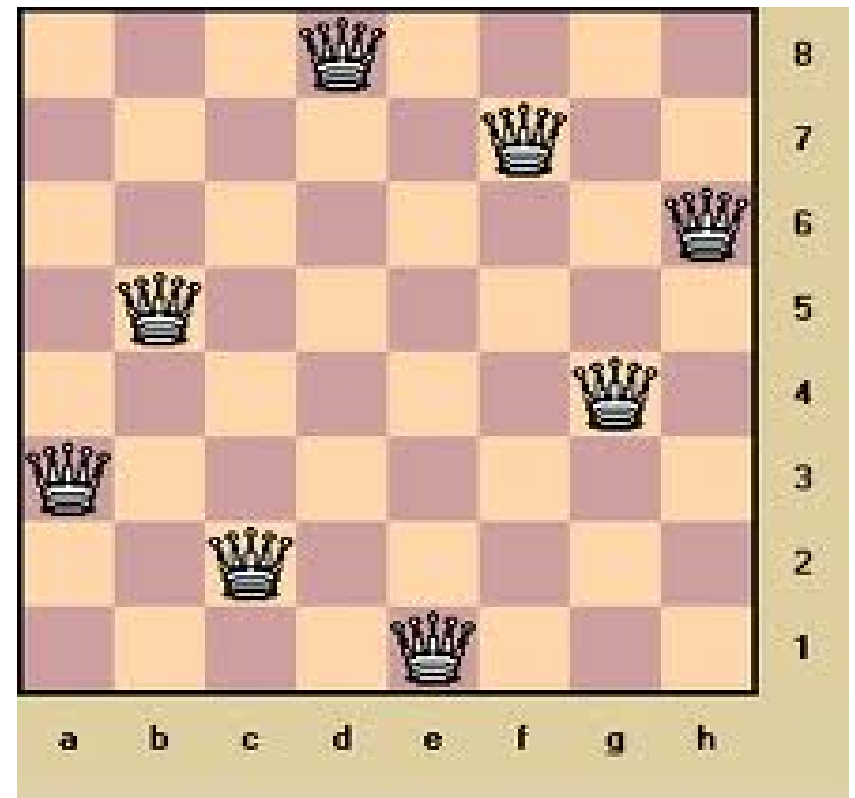
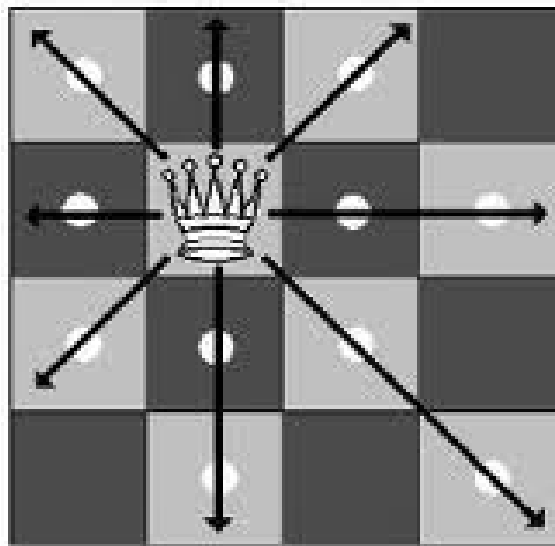


Problema de las 8 reinas

- Colocar 8 reinas en un tablero de tamaño 8x8 sin que se ataquen entre ellas.



Problema de las 8-reinas

Representación solución:

- **vector<int> X[8];**

X[i] representa la fila en la que coloca la reina de la columna i-esima ==> X[i] puede tomar 8 valores.

(antes i representaba la fila en que se colocaba la reina i y X[i] la columna, son representaciones equivalentes)

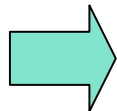
Además, dos reinas no pueden estar en la misma fila

Arbol de Estados: **Arbol de permutaciones**

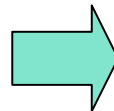
El numero de nodos solución es $8! = 40320$.

4-Reinas

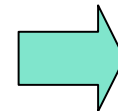
R			



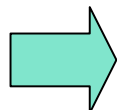
R	R		



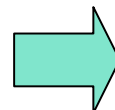
	R		
R	x		



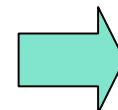
	R		
	x		
R	x		



	R		
	x		
R	x	R	

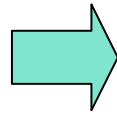


	R		
	x	R	
R	x	x	

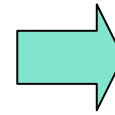


4-Reinas

	R	R	
	X	X	
R	X	X	



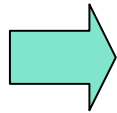
		R	
	R	X	
	X	X	
R	X	X	



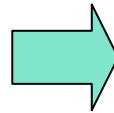
Backtracking

4-Reinas

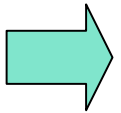
		R	
	R	x	
	x	x	
R	x	x	



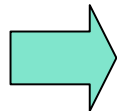
	R		
	x		
	x		
R	x		



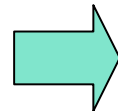
	R		
	x		
	x		
R	x	R	



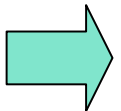
	R		
	x		
	x	R	
R	x	x	



	R		
	x		
	x	R	
R	x	x	R

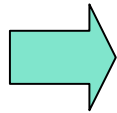


	R		
	x		
	x	R	R
R	x	x	x

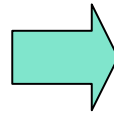


4-Reinas

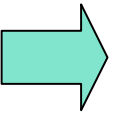
	R		
	x		R
	x	R	x
R	x	x	x



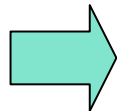
	R		R
	x		x
	x	R	x
R	x	x	x



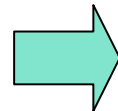
	R		
	x	R	
	x	x	
R	x	x	



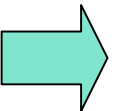
	R	R	
	x	x	
	x	x	
R	x	x	



	R		
	x		
	x		
R	x		

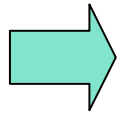


R			
x			

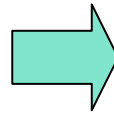


4-Reinas

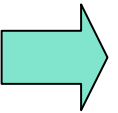
R			
X			



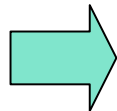
R			
X	R		



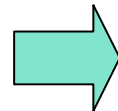
R	R		
X	X		



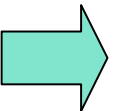
	R		
R	X		
X	X		



	R		
	X		
R	X		
X	X		

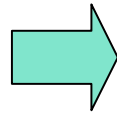


	R		
	X		
R	X		
X	X	R	

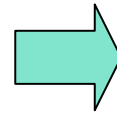


4-Reinas

	R		
	x		
R	x		
x	x	R	R

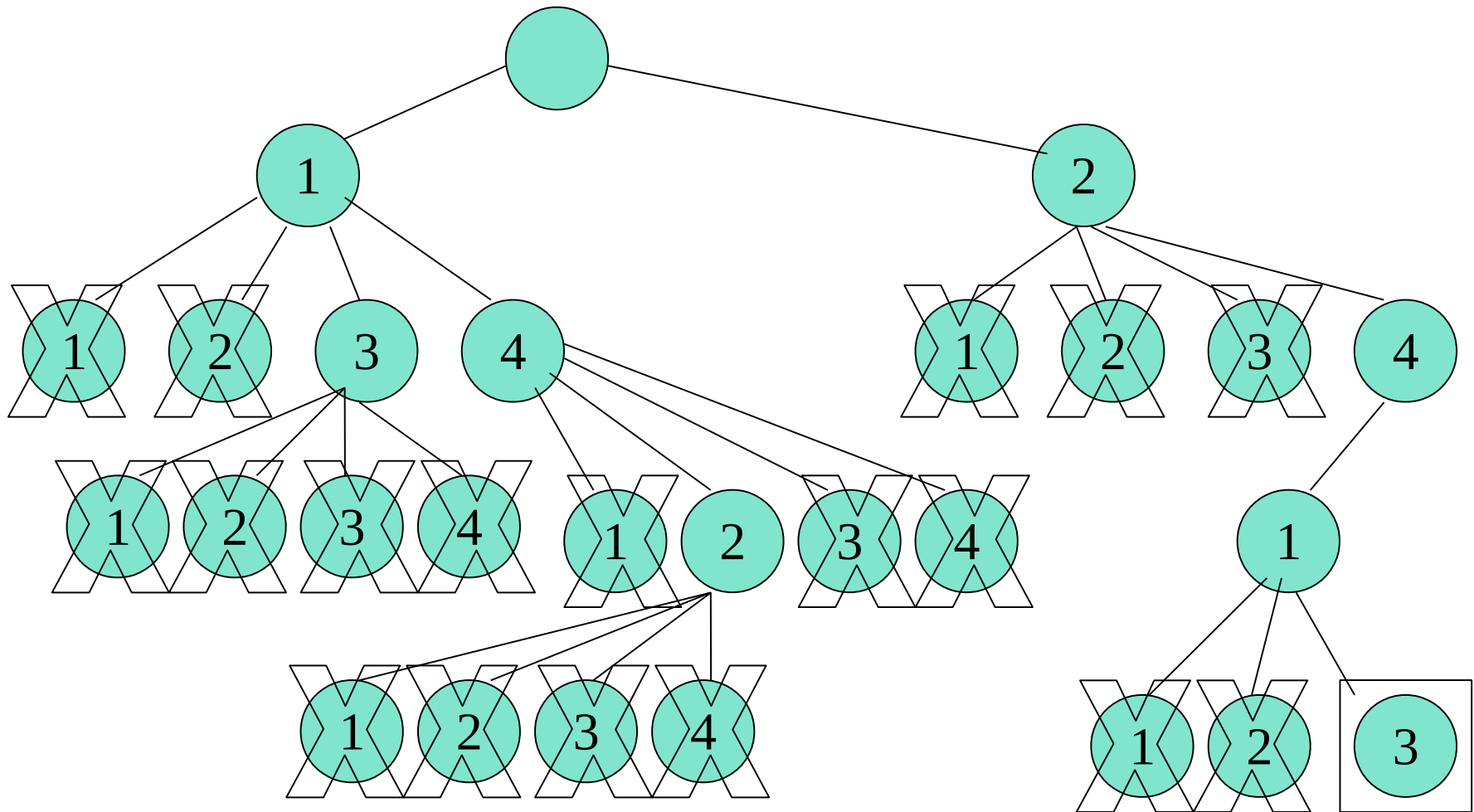


	R		
	x		
R	x		R
x	x	R	x



	R		
	x		R
R	x		x
x	x	R	x

4-Reinas



Funciones de la Clase Solución:

```
void IniciaComp(int k) { X[k]=NULO; } // columna k, fila 0
```

```
void SigValComp(int k) { X[k]++; } // Siguiente fila
```

```
bool TodosGenerados(int k) const {return X[k]==END;}
```

```
bool Solucion::Factible(int k)
```

```
{ // Si X[k]=j, la reina de la columna k, situada en la fila j,
```

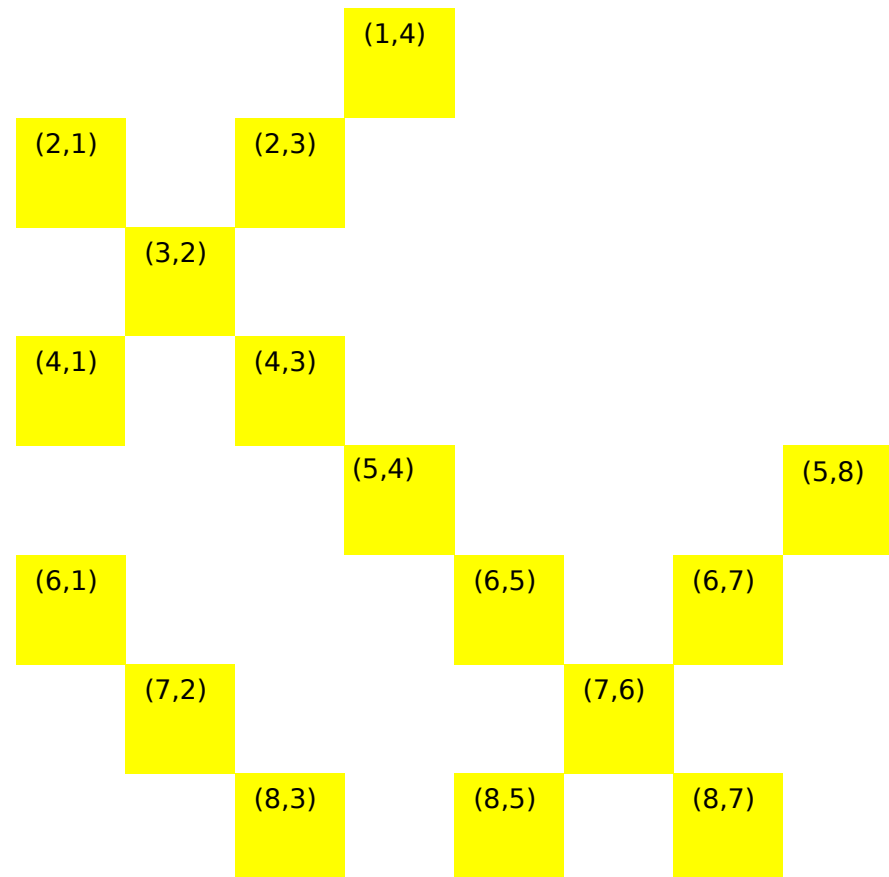
```
// no puede acosar a ninguna de las k-1 anteriores reinas.
```

Por tanto DEBE

- Estar en una columna libre (esta implícito en la formulación de la solución)
 - Estar en fila libre
 - Estar en una diagonal izquierda libre
 - Estar en una diagonal derecha libre
- ```
}
```

# 8 reinas: factibilidad

- ¿Cómo comprobar que dos reinas no están en la misma diagonal?



# 8 reinas: diagonales

- Si las casillas del tablero se numeran como una matriz  $A(1..n,1..n)$ , cada elemento en la misma diagonal que vaya de la parte superior izquierda a la inferior derecha (diagonal derecha), tiene el mismo valor "fila-columna".
- También, cualquier elemento en la misma diagonal que vaya de la parte superior derecha a la inferior izquierda (diagonal izquierda), tiene el mismo valor "fila+columna".
- Si dos reinas están colocadas en las posiciones  $(i,j)$  y  $(k,l)$ , estarán en la misma diagonal si y solo si,

$$i - j = k - l \text{ ó } i + j = k + l$$

- La primera ecuación implica que

$$j - l = i - k$$

- La segunda que

$$j - l = k - i$$

- Así, dos reinas están en la misma diagonal si y solo si

$$|j-l| = |i-k|$$

# 8 reinas: factibilidad

- Si asignamos un valor a la reina de la columna  $k$ ,  $X[k]$ , esa solución será factible si

```
factible(k) {
 For i=1 to k-1 do
 If $X[i] = X[k]$ or $ABS(X[i]-X[k]) = ABS(i-k)$
 return (false)
 return (true)
}
```

Pero esto consume  $O(k)!!$

## Chequear $X[k]=j$ es válido?

- Testear si la fila  $j$ -ésima está libre

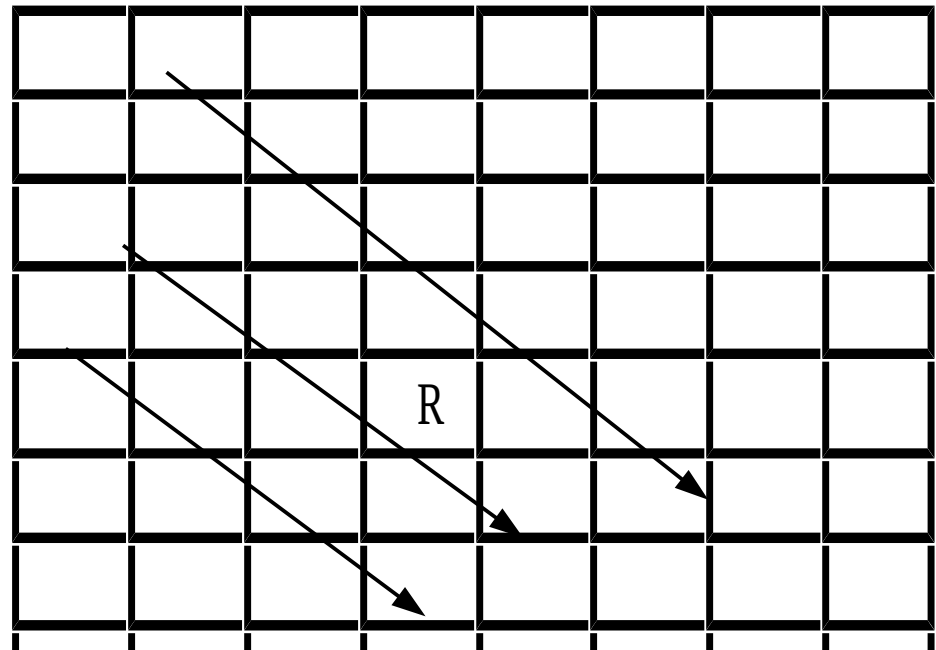
`bool filas[8]={1,1,1,1,1,1,1,1}; // inicio.`

`if (fila[j]==1) fila[j]=0; La fila está libre, la ocupamos`

- Problema diagonal derecha?

Fila-columna es fijo

(Rango -7,...,7)



## Chequear $X[k]=j$ es válido? (cont.)

- Diagonal derecha DD

bool DD[15]={1,1,1,...,1};

if (DD[k-j+7]) DD[k-j+7]=0; // Marcamos ocupada

- Problema diagonal Izquierda (DI)

Fila + Columna es fijo:(Rango 0,...,14) (ó 2..16)

bool DI[15]={1,1,1,...,1};

if (DI[k+j]) DD[k+j]=0; // Marcamos ocupada

bool Solucion::Factible(int k)

{ // Si  $X[k]=j$ , entonces la reina de la columna  $k$ , situada en fila  $j$ ,  
// no puede acosar a ninguna de las  $k-1$  anteriores reinas.

If (fila[X[k]] && DD[k-X[k]+7] && DI[k+X[k]])

{ fila[X[k]]=0;

DD[k-X[k]+7] =0;

DI[k+X[k]]=0;

return 1;

} else return 0;

}



```

Void n_reinas(Solucion & sol, int i)
{
if (i == Sol.size()) Sol.ImprimeSolucion();
else {
 Sol.IniciaComp(i);
 Sol.SigValComp(i);
 while (!Sol.TodosGenerados(i)) {
 if (Sol.Factible(i)){
 n_reinas(Sol, i+1);
 Sol.LiberarPosiciones(i); // tras vuelta atras,
 } // libera posiciones ocup.
 Sol.SigValComp(i);
 }
}
}

```

El algoritmo imprime todas soluciones,....  
 Cómo para al encontrar la 1ª?

```

Void n_reinas(Solucion & sol, int i, bool & solfound)
{
if (i == Sol.size()) {Sol.ImprimeSolucion(); solfound=true;}
else {
 Sol.IniciaComp(i);
 Sol.SigValComp(i);
 while (!Sol.TodosGenerados(i) && !solfound) {
 if (Sol.Factible(i)){
 n_reinas(Sol, i+1,solfound);
 Sol.LiberarPosiciones(i); // tras vuelta atras,
 } // libera posiciones ocup.
 Sol.SigValComp(i);
 }
}
}

```

El algoritmo imprime todas soluciones,....

Cómo para al encontrar la 1ª?

Deteniendo las llamadas recursivas usando un valor booleano

# Solución 8 Reinas

- Nodos posibles

$$1 + \sum_{j=0}^7 \left( \prod_{i=0}^j (8-i) \right)$$

$$= 69281.$$

Se generan 112 nodos  
para alcanzar la  
primera solución.

570 nodos para obtener  
todas.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
|   |   | 1 |   |   |   |   |   |
|   |   |   |   |   | 2 |   |   |
|   | 3 |   |   |   |   |   |   |
|   |   |   |   |   |   | 4 |   |
| 5 |   |   |   |   |   |   |   |
|   |   |   | 6 |   |   |   |   |
|   |   |   |   |   |   |   | 7 |
|   |   |   |   | 8 |   |   |   |

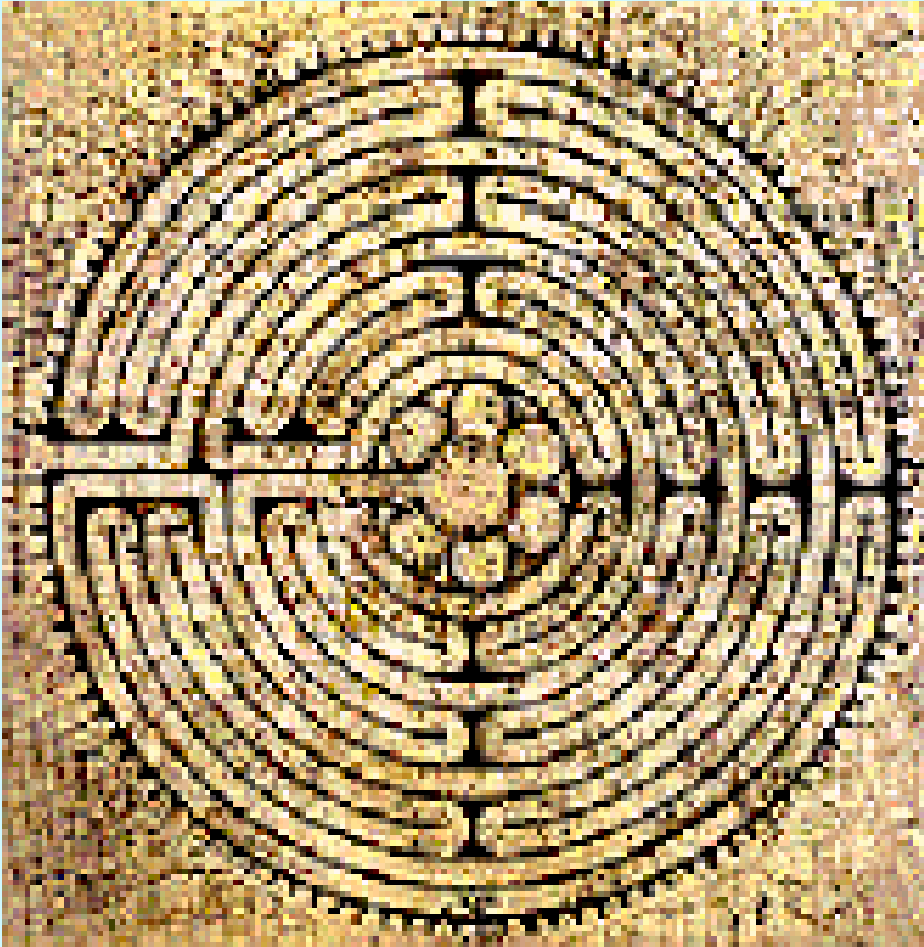
# Solución para las N reinas no recursivo

## Procedimiento NREINAS(n)

{Usando backtracking este procedimiento imprime todos los posibles emplazamientos de n reinas en un tablero nxn sin que se ataquen}

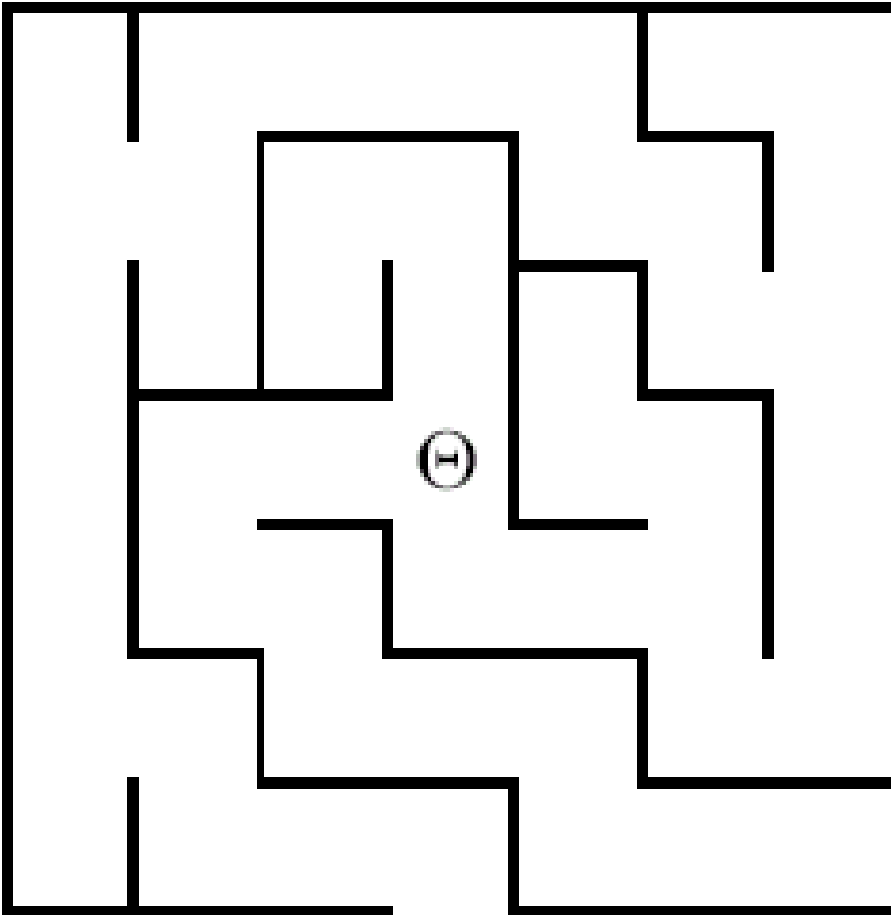
|                                       |                                 |
|---------------------------------------|---------------------------------|
| X(1) = 0; k = 1;                      | {k es la columna actual}        |
| while (k > 0) {                       | {hacer para todas las columnas} |
| X(k) = X(k) + 1;                      | {mover a la siguiente fila}     |
| while ((X(k) <= n) && (!factible(k))) | {puede moverse esta reina?}     |
| X(k) = X(k) + 1;                      |                                 |
| if (X(k) <= n)                        | {Se encontró una posición}      |
| if (k == n)                           | {Es una solución completa?}     |
| print (X)                             |                                 |
| else {k = k + 1; X(k) = 0;}           | {Ir a la siguiente fila}        |
| else k = k - 1;                       | {Backtrack}                     |
| }                                     |                                 |

# Laberintos y Backtracking



Un laberinto puede modelarse como una serie de nodos.  
En cada nodo hay que tomar una decisión que nos conduce a otros nodos.

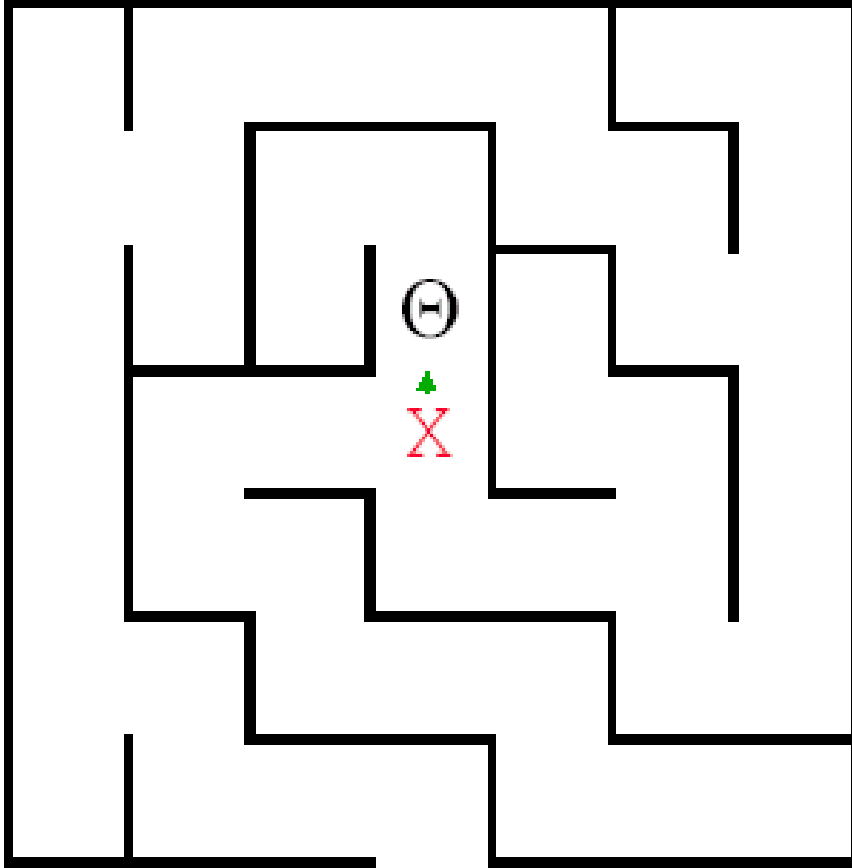
# Un laberinto sencillo



Buscar en el laberinto hasta encontrar una salida. Si no se encuentra una salida, informar de ello.

Hay 4 movimientos posibles.

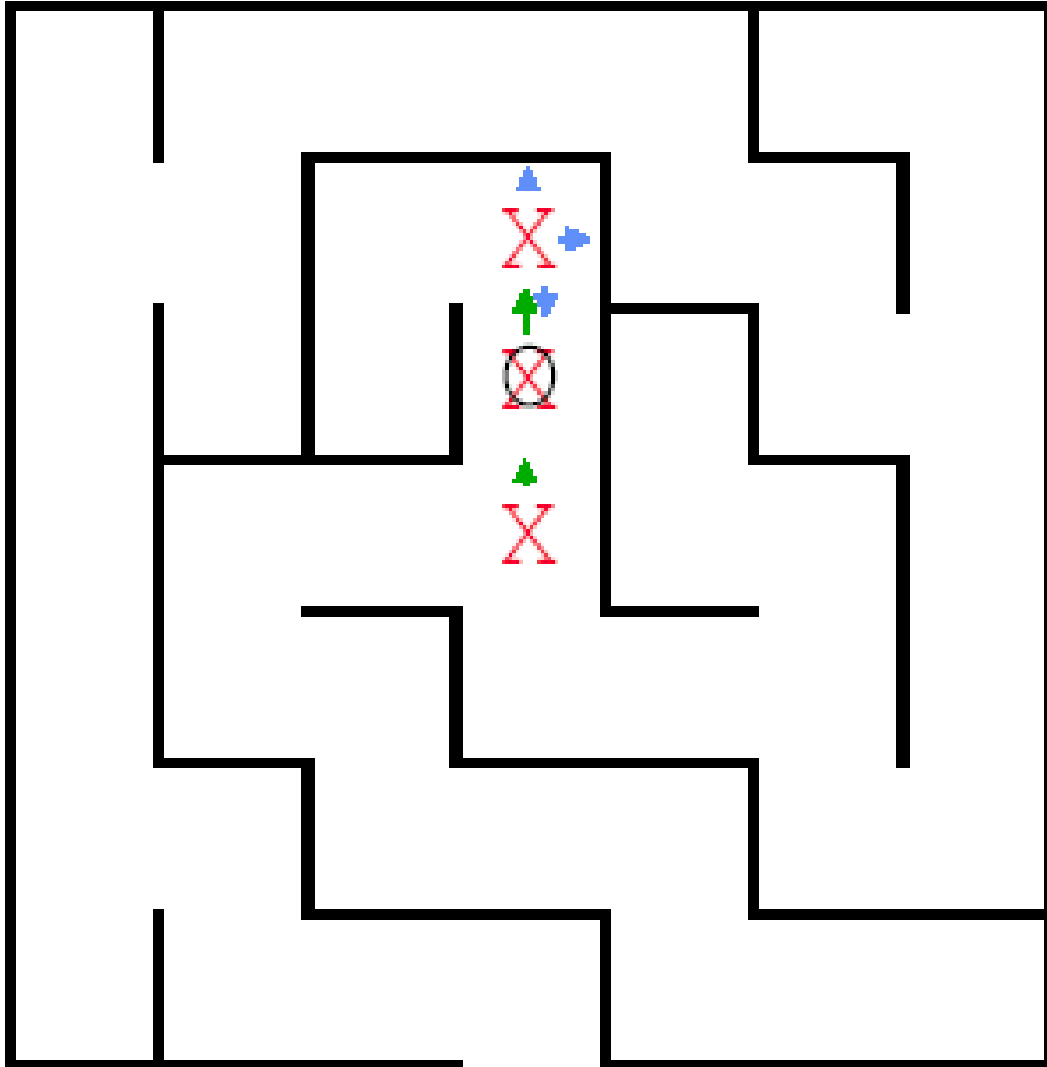
# Backtracking en Acción



Intentamos sucesivamente movernos en cada una de las 4 direcciones.

Inicialmente nos movemos hacia el norte (marcando las posiciones por las que pasamos) .

# Backtracking en Acción

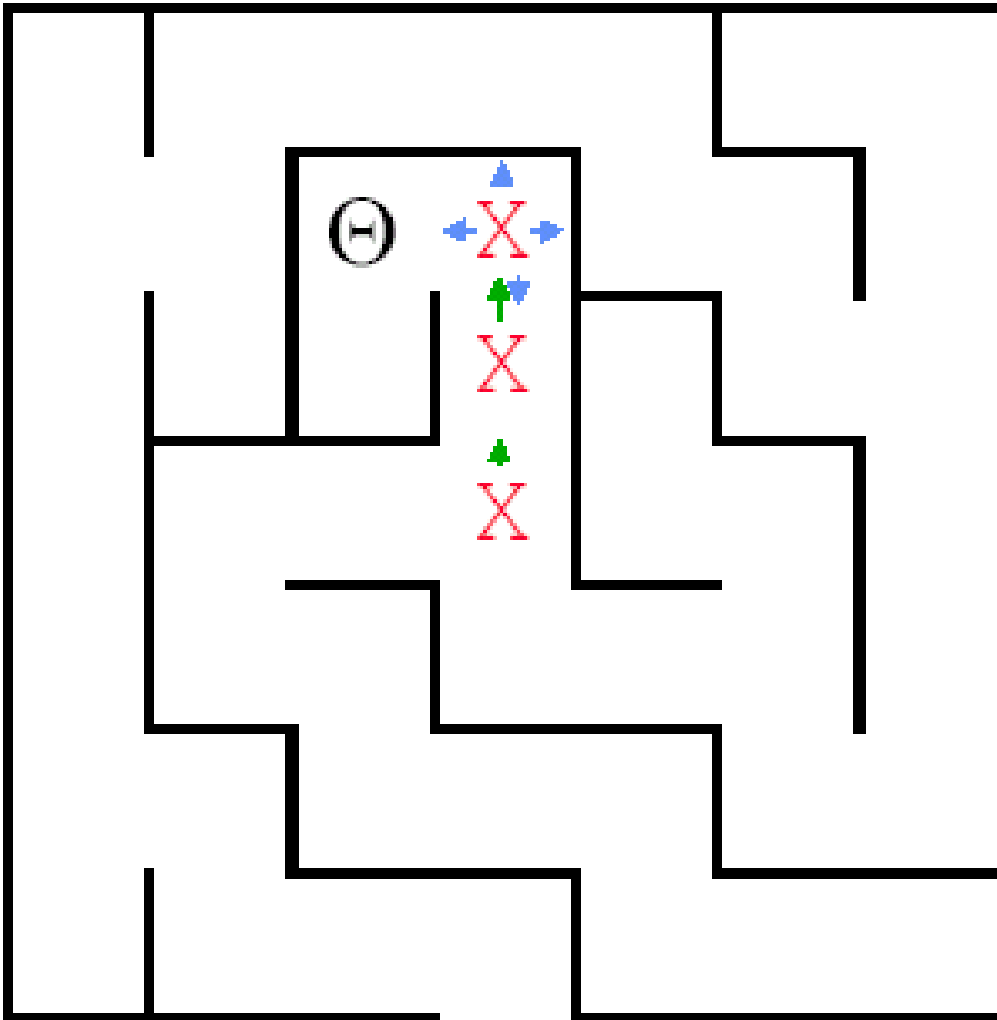


Aquí nos movemos hacia el Norte de nuevo, pero ahora la dirección Norte está bloqueada por un muro.

El Este también está bloqueado, por lo que intentamos el Sur. Esa acción descubre que ese punto está marcado.

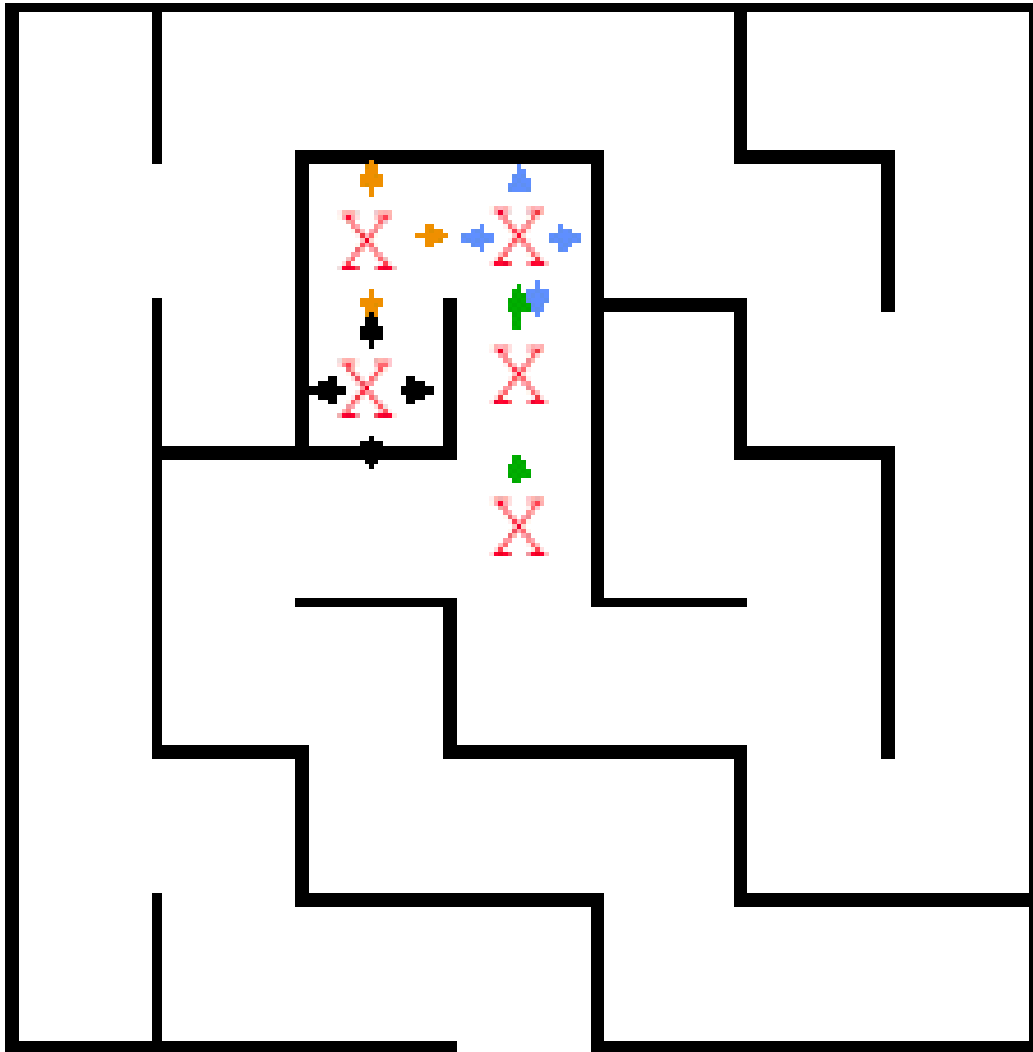


# Backtracking en Acción



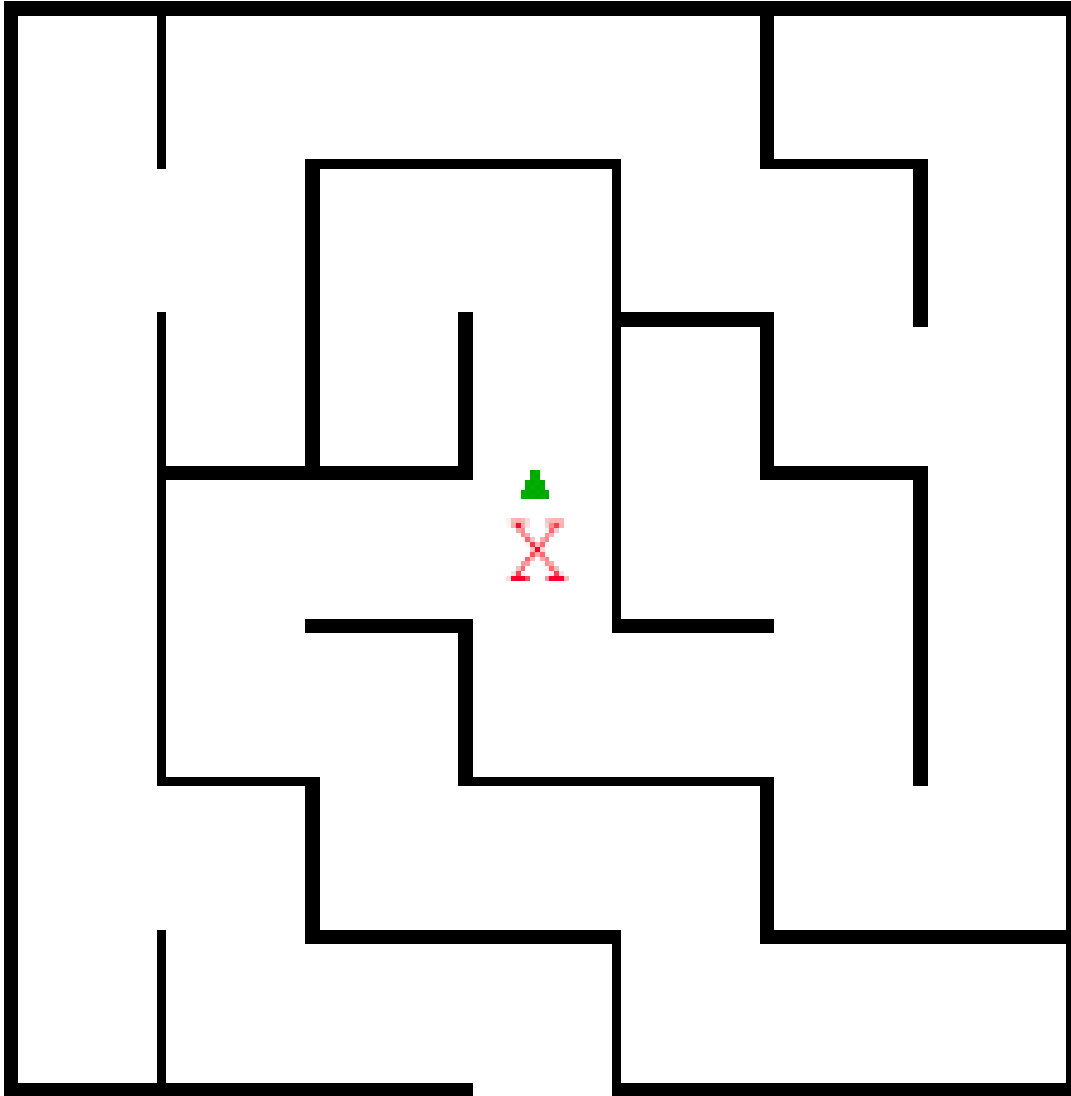
Por tanto el siguiente movimiento que podemos hacer es hacia el Oeste

# Backtracking en Acción



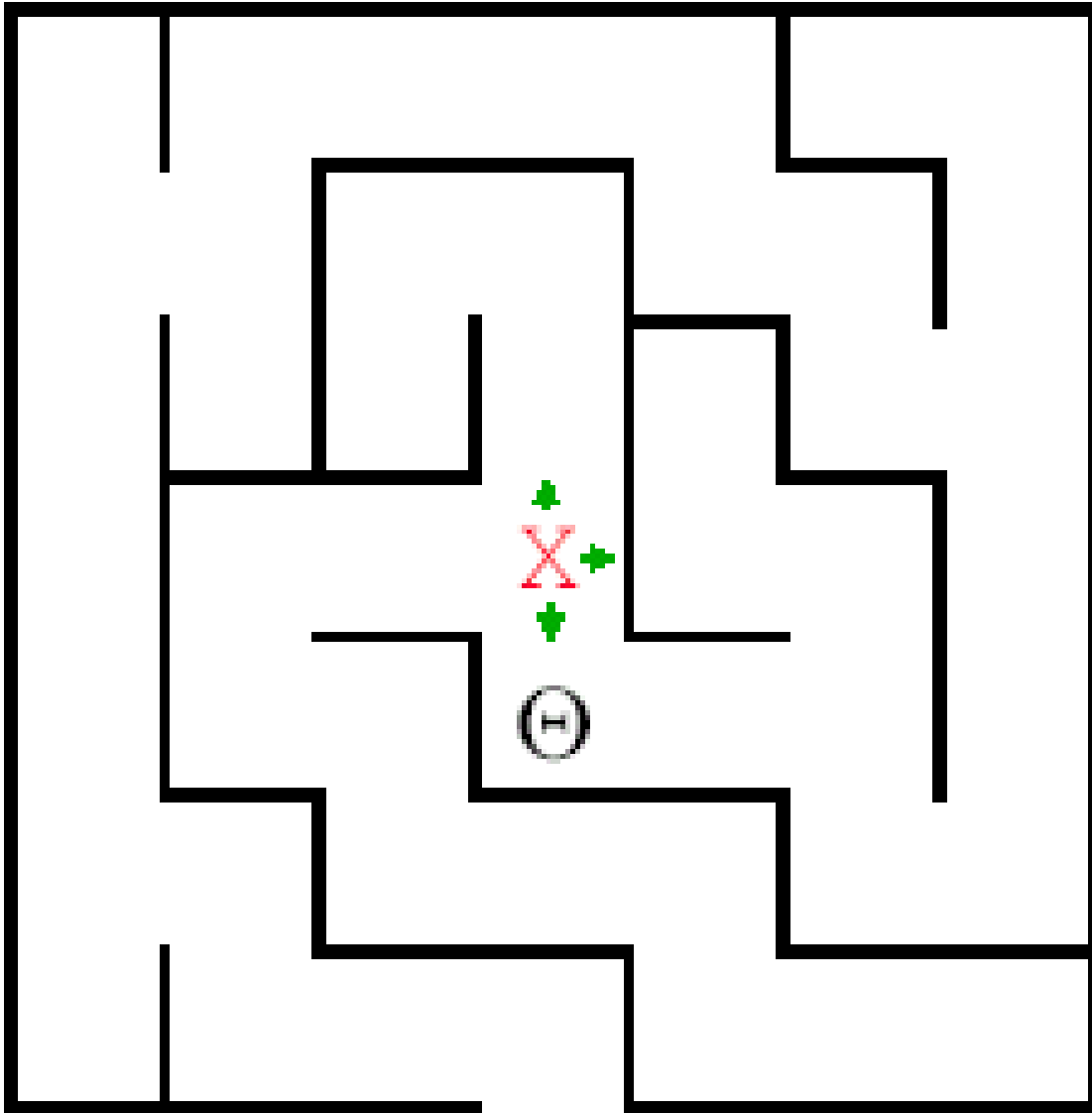
Este camino llega a  
un nodo (final)  
muerto .  
¡Por tanto es el  
momento de hacer  
un backtrack!

# Backtracking en Acción



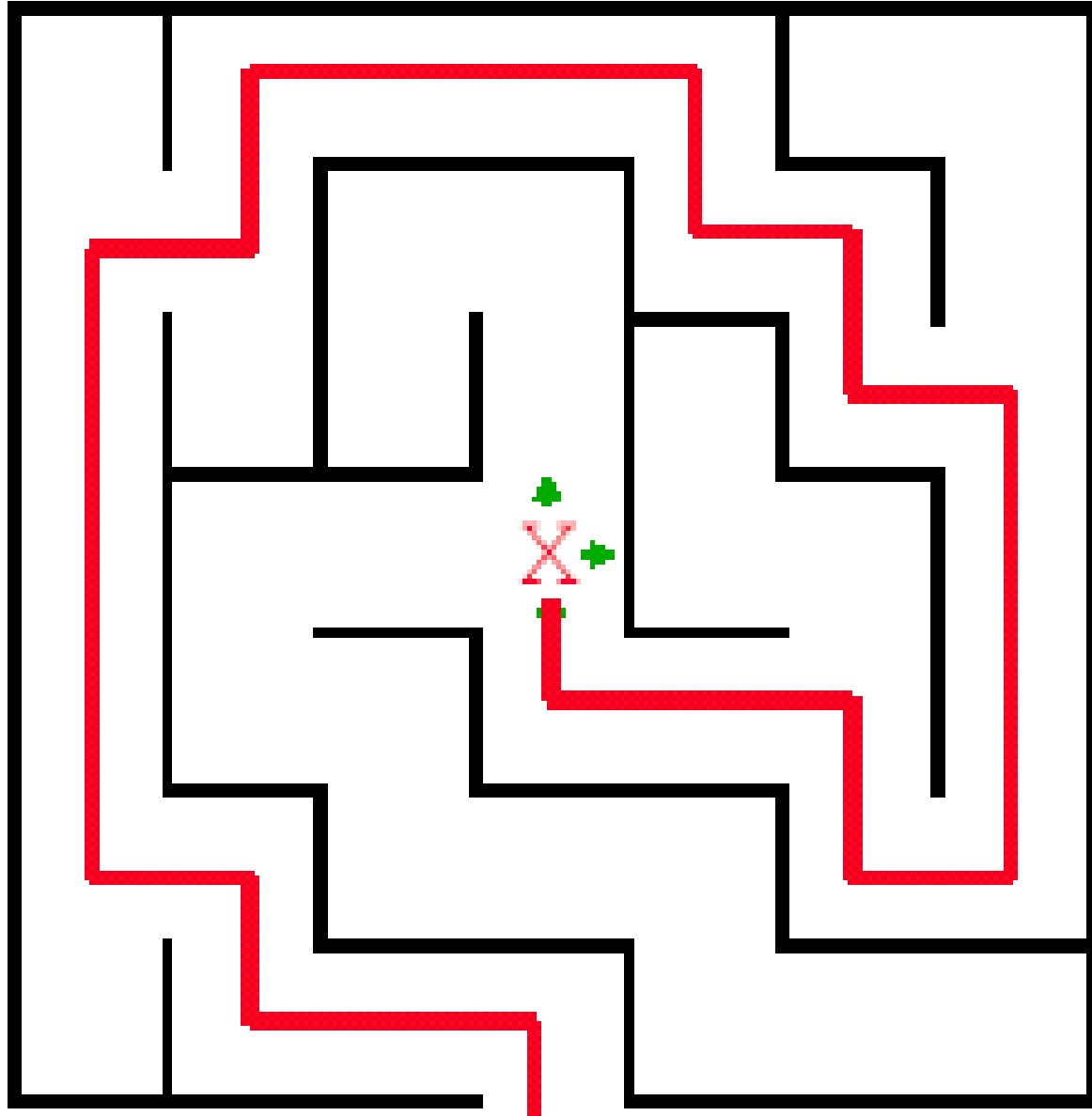
Se realizan  
sucesivas llamadas  
recursivas hasta  
volvernos a  
encontrar aquí

# Backtracking en Acción



Intentamos ahora  
el Sur

# Primer camino que se encuentra



En este problema la longitud del vector solución no es constante, es la lista de posiciones hasta llegar a la salida.

# El problema del coloreo de un grafo

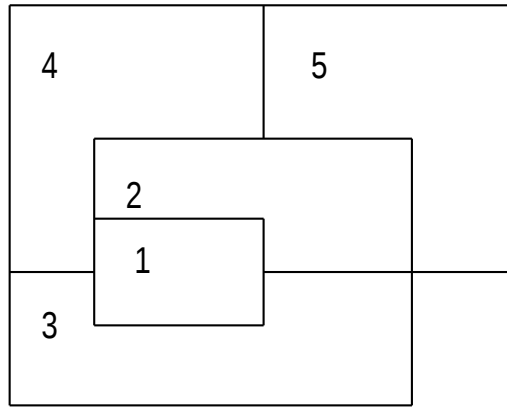
- Sea  $G$  un grafo y  $n$  un número entero positivo. Queremos saber si los nodos de  $G$  pueden colorearse de tal forma que no haya dos vértices adyacentes que tengan el mismo color, y que solo se usen  $n$  colores para esa tarea.
- Este es el problema de la  $n$ -colorabilidad.
- El problema de optimización de la  $n$ -colorabilidad, pregunta por el menor número  $n$  con el que el grafo  $G$  puede colorearse. A ese entero se le denomina Número Cromático del grafo.

# El problema del coloreo de un grafo

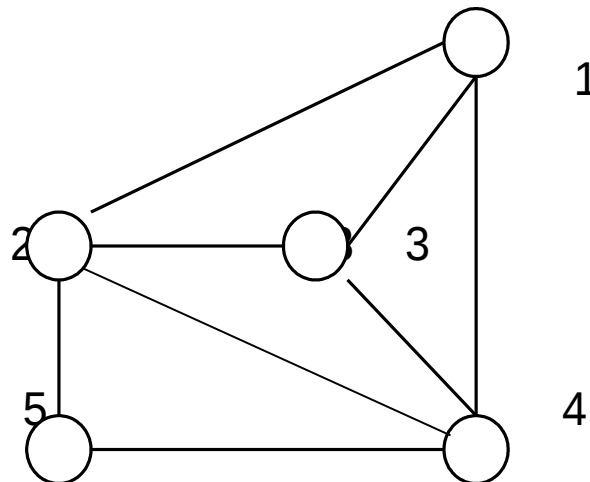
- Un grafo se llama plano si y solo si puede pintarse en un plano de modo que ningún par de aristas se corten entre sí.
- Un caso especial famoso del problema de la  $n$ -colorabilidad es el problema de los cuatro colores para grafos planos que, dado un mapa cualquiera, consiste en saber si ese mapa podrá pintarse de manera que no haya dos zonas colindantes con el mismo color, y además pueda hacerse ese coloreo solo con cuatro colores (sí se puede).
- Este problema es fácilmente traducible a la nomenclatura de grafos.

# El problema del coloreo de un grafo

- El mapa



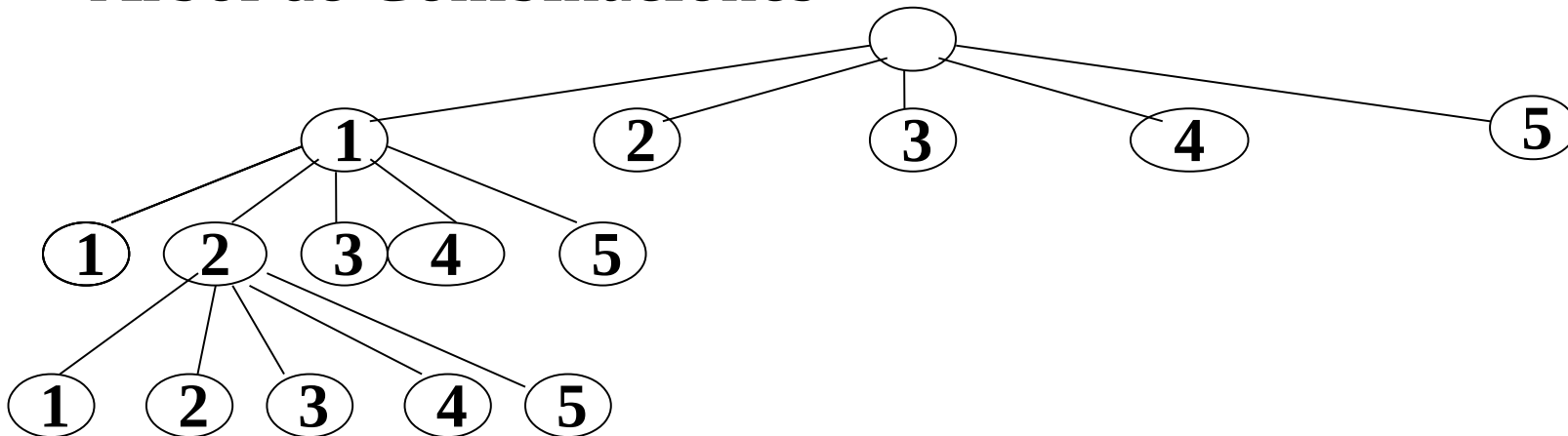
- puede traducirse en el siguiente grafo





# Coloreo: Restricciones

- Restricciones Explícitas
  - $X[i]$  toma valores en  $\{\text{NULO}, 1, 2, 3, 4, \dots, N, \text{END}\}$
- Restricciones Implícitas.
  - $X[i] \neq X[j]$  si  $i$  y  $j$  son países adyacentes
- Arbol de Estados
  - Arbol de Combinaciones

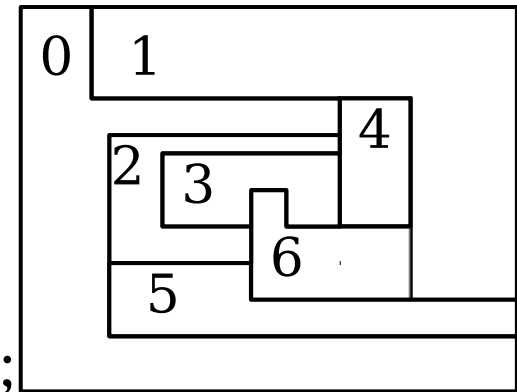


# Coloreo: Estruct. De datos

- Utilizaremos una estructura de datos simple que nos permita chequear para cada pais, todos sus adyacentes
  - `vector<vector<int> > Mapa;`
  - `Mapa[i][j]` es el  $j^{\text{th}}$  pais adyacente al pais `i`
    - `Mapa[5][3]==8` significa que el 3<sup>r</sup> pais adyacente a 5 es 8

# Coloreo: Ejemplo Mapa

```
mapa.resize(7);
mapa[0].push_back(1);
mapa[0].push_back(2);
mapa[0].push_back(4); mapa[0].push_back(5);
```



```
mapa[1].push_back(0); mapa[1].push_back(4);
mapa[1].push_back(6); mapa[1].push_back(5);
mapa[2].push_back(0); mapa[2].push_back(3);
mapa[2].push_back(4);mapa[2].push_back(5);mapa[2].push_back(6);
```

.....

# Clase Solucion:

```
void IniciaComp(int pais)
```

```
{ X[pais] = 0; // Valor NULO }
```

```
void SigValComp(int pais)
```

```
{ // Orden de los valores 0 -> 1 -> 2 -> ...-> N → N+1
```

```
 X[pais]++;
```

```
}
```

```
bool TodosGenerados(int pais) const
```

```
{
```

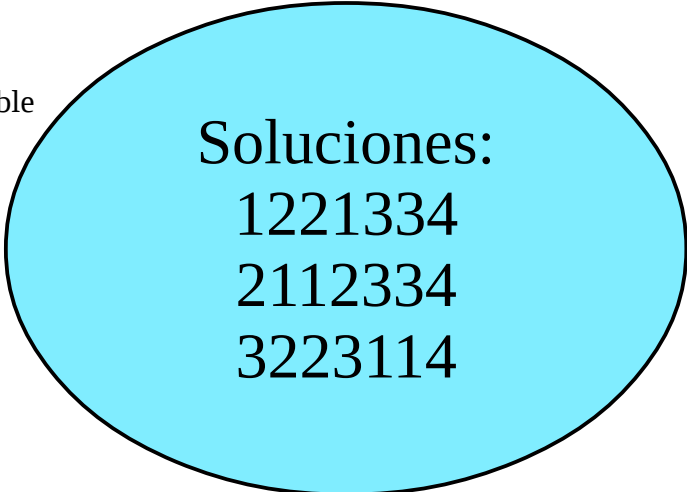
```
 return X[pais]== N+1; //END
```

```
}
```

```
bool Factible(int pais) {
 for (int i = 0; i < M[pais].size(); i++) {
 int ith_adyacente = M[pais][i];
 if (X[ith_adyacente] == X[pais])
 return false;
 }
 return true;
}

void VueltaAtras(int pais){
 if (pais==X.size()) { return;}
 X[pais] = 0;
}
```

```
void back_coloreo(Solucion & Sol, int k) {
 if (k == Sol.Size()) Sol.ProcesaSolucion();
 else {
 Sol.IniciaComp(k);
 Sol.SigValComp(k);
 while (!Sol.TodosGenerados(k)) {
 if (Sol.Factible(k)) {
 back_coloreo(Sol, k+1);
 Sol.VueltaAtras(k+1); } // no es imprescindible
 Sol.SigValComp(k);
 } // While
 } // Else
 }
}
```



Soluciones:  
1221334  
2112334  
3223114

Ojo: Son la misma solución, !!!

# Problema de la Mochila

- Cuál es el árbol de estados?

El espacio de soluciones consiste en las  $2^n$  formas de asignar valores 0 ó 1 a cada objeto (seleccionar o no el objeto)

Es por tanto igual al del problema de la suma de subconjuntos.

Emplearemos un árbol binario: los hijos de un nodo  $(x_1, \dots, x_k)$  son  $(x_1, \dots, x_k, 0)$  y  $(x_1, \dots, x_k, 1)$  (también se podría emplear un árbol combinatorio)

## Funciones de la Clase Solución:

```
void IniciaComp(int k)
```

```
{ X[k] = -1; // valor NULO }
```

```
void SigValComp(int k)
```

```
{ X[k]++; // Siguiete valor del dominio. -1->0->1->2 }
```

```
Bool TodosGenerados(int k) const
```

```
{ return X[k]==2; // END }
```

```
bool Solucion::Factible(int pos) const
```

```
{ //El peso de los objetos seleccionados no sobrepase la capacidad de la mochila
```

```
float ps =0.0;
```

```
for (int k=0; k<=pos ; k++) ps += X[k]*p[k];
```

```
return (ps<=M) ; }
```



Número de objetos: 7      Ejemplo:

Tamaño  $M=11.0$

Beneficio  $=\{30, 20, 36, 40, 48, 6, 2\};$

Pesos  $=\{ 3, 2, 4, 5, 6, 3, 5\};$

Rel B/P  $=\{ 10, 10, 9, 8, 8, 2, 0.4\};$

Nodos Posibles: (tamaño árbol de estados)

255 ( $2^8-1$ ) , de los que 128 ( $2^7$ ) son nodos solución

Nodos Generados: 162, de los cuales 43 son solución

Solución óptima: 98,  $X=\{1,1,0,0,1,0,0\}$

Se podría mejorar el algoritmo?.

# Mejorando el algoritmo ...

## 1) Mejorar la función de Factibilidad:

Una solución no es factible cuando:

- La suma del peso de los objetos seleccionados sobrepasa la capacidad de la mochila (ya considerado)
- Sea N un nodo interno del árbol y sea S la mejor solución que se obtiene al evaluar todos los nodos solución (hojas) que se visitan antes que N

Entonces, podemos podar N si la mejor solución que se podría alcanzar al expandir el subárbol no pueda superar el valor de la mejor solución obtenida hasta el momento, S:  
USAR COTAS

## 2) Expandir primero la rama a priori más prometedora, con la idea de alcanzar antes la solución

Para ello ordenamos los objetos de forma que  $b_i/p_i \geq b_{i+1}/p_{i+1}$

1) TRAS mejorar función factibilidad:

Nodos Posibles: 255, (128 son nodos solución)

Antes: Nodos Generados: 162, (43 son solución).

Ahora: Nodos Generados: 72.

(10 son nodos solución)

(16 se han podado por sobrepasar M)

(11 se podan al no superar mejor solución S)

(35 se han explorado completos)

Solución Optima: 98,  $X=\{1,1,0,0,1,0,0\}$

Problema: Expandimos menos nodos, pero tardamos más en evaluar la factibilidad de los mismos.

2) Expandir **primero la rama más prometedora**, con la idea de alcanzar antes una solución.

```
void IniciaComp(int k) { X[k]=2; }
```

```
void SigValComp(int k) { X[k]-- ; }
```

```
bool TodosGenerados(int k) const {return X[k]==-1;}
```

No tiene costo, y sin embargo mejora considerablemente el rendimiento del algoritmo

Nodos Generados: 34 (**antes 72**).

- 3 (**10**) son nodos solución
- 11 (**16**) se han podado por sobrepasar M
- 4 (**11**) se podan al no superar mejor solución S
- 16 (**35**) se han explorado completos

# Cálculo de cotas....Problema de Maximización (similar para min.)

Necesitamos 2 cotas sobre la ganancia :

- COTA GLOBAL:
  - Representa el valor de la mejor solución estudiada hasta el momento. Inicialmente, se le puede asignar el valor dado por un algoritmo greedy.
  - Es una cota inferior, CI, de la solución optimal:  
 $CI \leq \text{Optimo}.$

# Cálculo de cotas....Problema de Maximización (similar para min.)

- La Cota Global se actualiza siempre que alcancemos una solución, H, (nodo hoja) con ganancia mejor

```
void ActualizaSolucion()
```

```
{ Si ganancia(H) > CI
```

```
 entonces { CI=ganancia(H); MejorSolucion = H;}
```

```
}
```

# Cálculo de cotas.... Maximización

- COTA LOCAL:
  - Se calcula para cada nodo interno,  $N$ , generado, siendo una estimación optimista del mejor valor que se podría alcanzar al expandir el nodo,  $LOptimal(N)$ .
  - Es una cota superior,  $CS(N)$ :  $LOptimal(N) \leq CS(N)$
  - Debe haber un equilibrio entre eficiencia de cómputo y calidad de la cota.
  - Puede incluirse dentro de la función de `Factibilidad(int k)`

# Cálculo de cotas.... Maximización

- PODA? Podamos el nodo N siempre que  $CS(N) \leq CI$
- Podemos asegurar que alcanzamos el óptimo? SI, porque

$$LOptimal(N) \leq CS(N) \leq CI \leq Optimo$$

Cuanto menor sea la cota superior local (más próxima a  $LOptimal(N)$ ) más probable es que podamos podar.

Cuanto mayor sea la cota inferior global (más próxima al verdadero óptimo) mejor



# Cálculo Cota Local ...(max)

- Dado un nodo interno N, tenemos el vector solución que está incompleto:

$$X = [\text{decisiones tomadas}] + [\text{decisiones por tomar}]$$

La Cota Superior  $CS(N)$  dependerá de ambas

$$CS(N) = f(\text{DecTomadas } X) + \text{estimador}(\text{DecPorTomar } X)$$

Interesa que la  $CS(N)$  esté lo más cerca posible de  $L_{\text{optimal}}(N)$ .

Hay que optimizar el estimador.

# Problema de la Mochila: Cota Local

$$X = [\text{decisiones tomadas}] + [\text{decisiones por tomar}]$$

$$X = [1, 1, 0, 0, 1, ?, ?, ?, ?, ?]$$

Sea  $k$  la última decisión.

$f(\text{DecTomadas})$  = suma ganancia objetos  
seleccionados

$$f(\text{DecTomadas}) = \sum_{i=1}^k x[i] * b[i]$$

# Problema de la Mochila: Cota Local

$$X = [\text{decisiones tomadas}] + [\text{decisiones por tomar}]$$

$$X = [1, 1, 0, 0, 1, ?, ?, ?, ?, ?]$$

Sea  $k$  la última decisión.

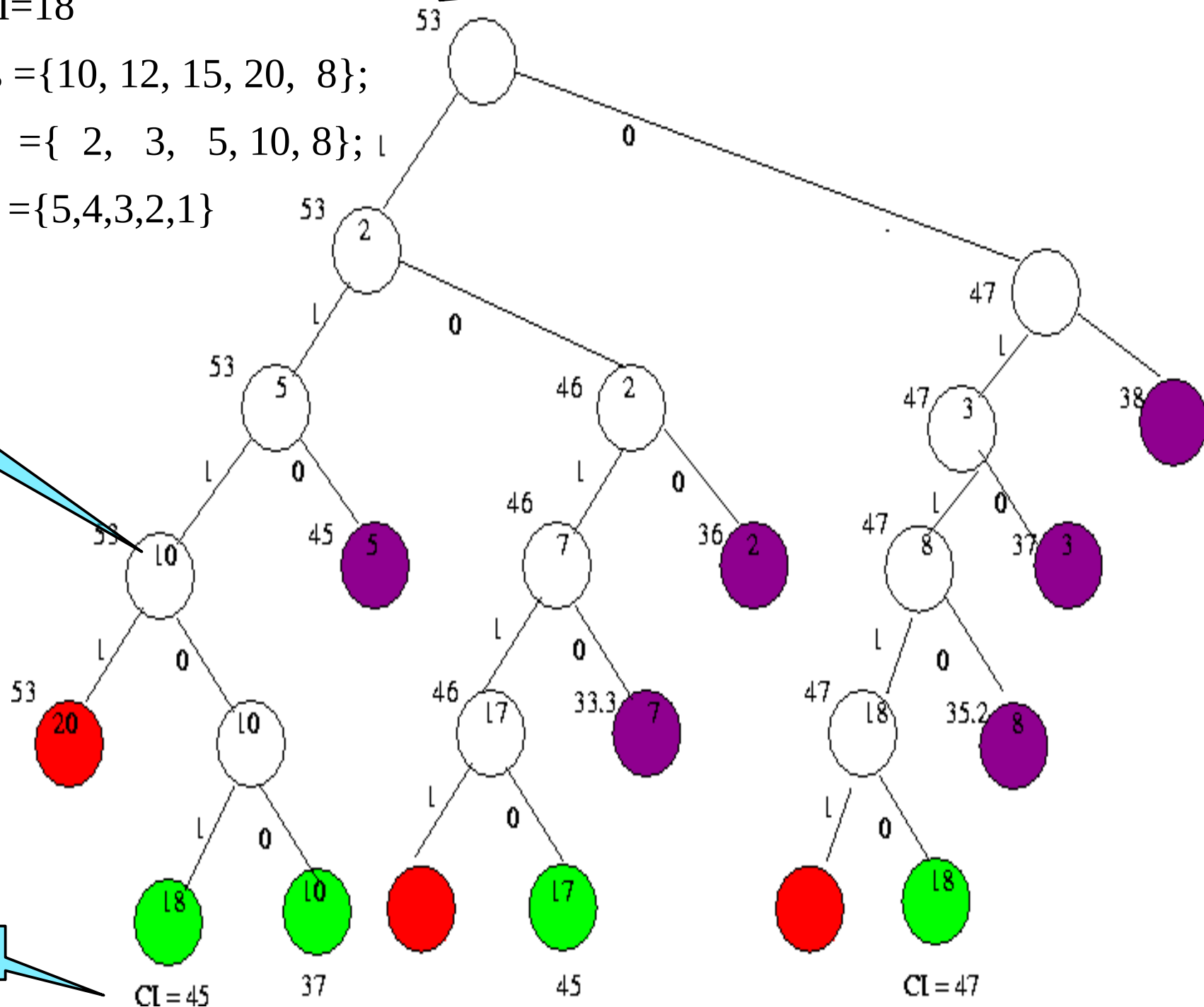
Estimador(decisiones por tomar  $X$ ): Ganancia del algoritmo Greedy No 0/1 para un problema de la mochila con los objetos  $k+1, k+2, \dots, n$  y capacidad  $M'$

$$M' = M - \sum_{i=1}^k x[i] * p[i]$$

$$G/P = \{5, 4, 3, 2, 1\}$$

# Volumen

# Cota Global



## Optimizar en tiempo función factibilidad

- No es necesario volver a realizar todos los cálculos. Asumimos i el objeto que consideramos.

Podemos almacenar inform. (objetos 0,...,i-1).

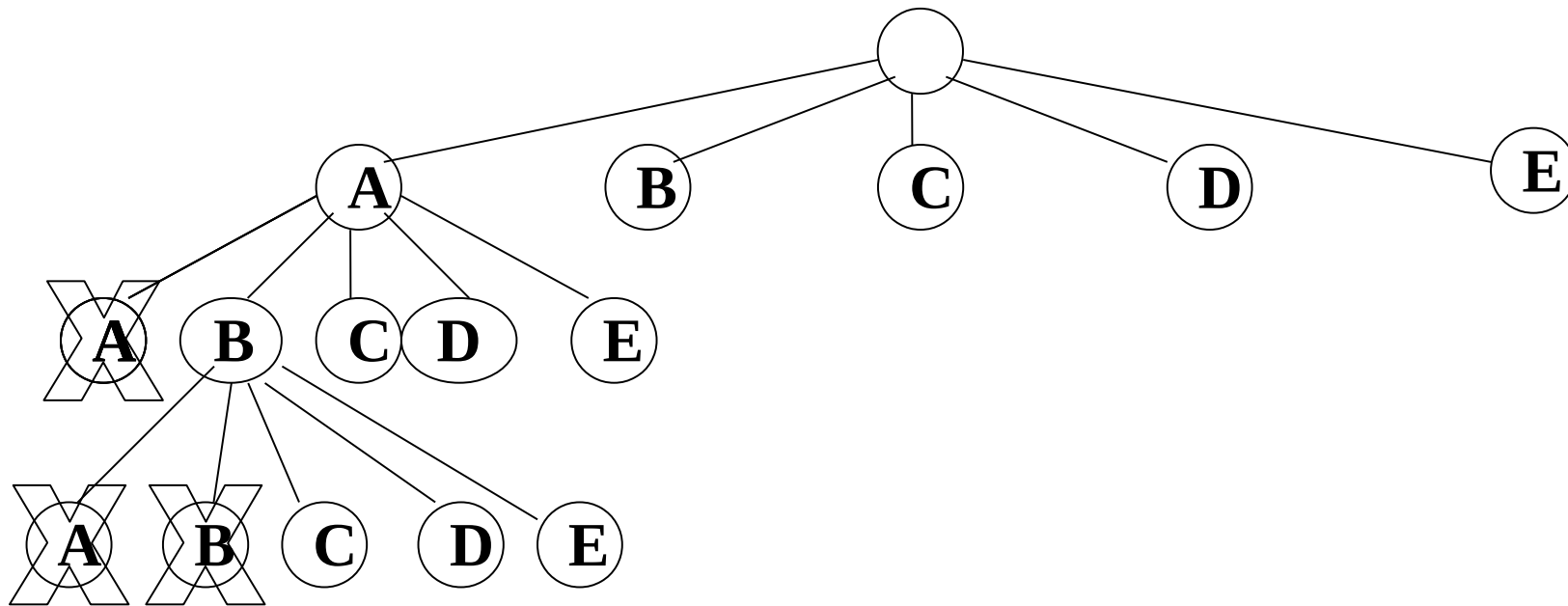
- VolumenOcupado: VO, inicialmente VO=0
- GanaciaAcumulada: GA, inicialmente GA = 0
- Estimador: E inicialmente E = Gr NO 0/1
- Si  $X[i] = 1 \Rightarrow VO += p[i]; GA += b[i];$
- Si  $X[i] = 0 \Rightarrow VO -= p[i]; GA -= b[i];$
- $E = GA + (M-VO)(b[i+1]/p[i+1])$

Todas las operaciones son  $O(1)$ !!!!

# Problema Viajante de Comercio

- Supongamos  $N=5$  ciudades.
  - Solución?  $X=[1,3,0,4,2]$
  - Restricciones Explícitas:  $X[i]$  pertenece a  $\{0,...,N-1\}$
  - Restricciones Implícitas: Una ciudad no puede aparecer dos veces en el recorrido.
  - Espacio de Soluciones: **Permutaciones de  $N$  elementos**

# Viajante de comercio: Arbol de estados



Para evitar repetir varias veces las mismas soluciones, se fija una ciudad como punto de partida (la ciudad 0)

El circuito ABCDEA es lo mismo que BCDEAB o que CDEABC ...

Tenemos pues un espacio de estados de  $(n-1)!$  en vez de  $n!$

- Necesitamos definir cotas para este problema.

- Es un problema de minimización, por tanto:

- **COTA GLOBAL** es una cota superior (CS).

Inicialmente, solución obtenida por Algoritmo Greedy

Se actualiza cada vez que encontremos una solución de menor costo.

- **COTA LOCAL** (estimador) es una cota inferior ( $CI(N)$ ).
    - Esta cota indica que **NO puede existir un ciclo con una distancia menor**. Pero NO implica que necesariamente exista un ciclo con dicha distancia.

Debemos considerar:

- Costo del camino que representa la solución parcial
      - Un estimador (por debajo) del costo para ir al resto de ciudades

- Podemos si  $CI(N) \geq CS$ .



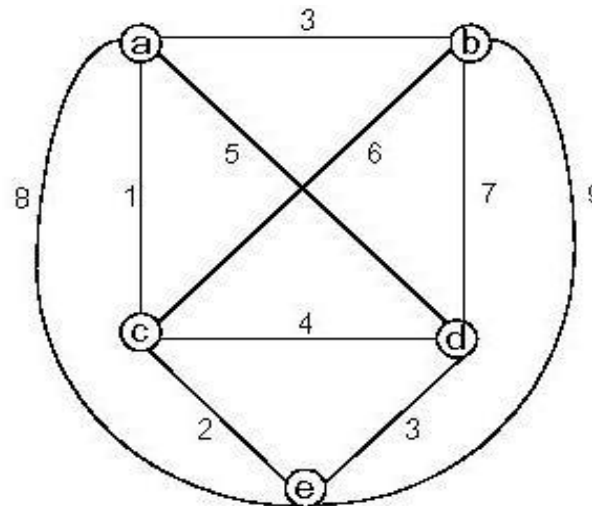
- **COTA LOCAL** (estimador) es una cota inferior (CI(N)).

$$X = [\text{decisiones tomadas}] + [\text{decisiones por tomar}]$$

$$X = [a, d, e, ?, ?]$$

- Costo del camino que representa la solución parcial
- Un estimador (por debajo) del costo para ir al resto de ciudades

Cómo?

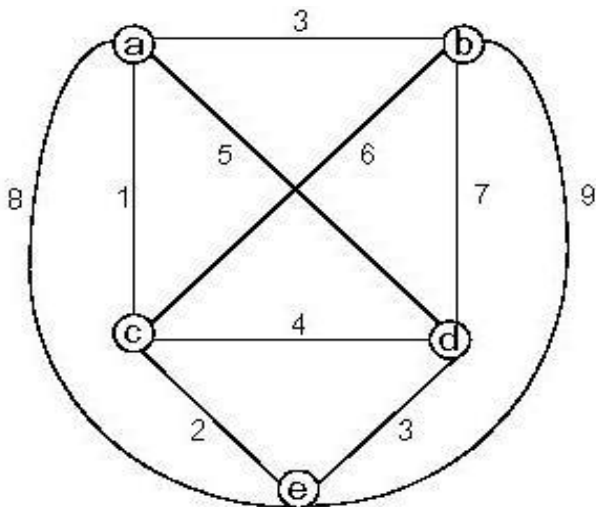


# Viajante de comercio: Cota 1

E1: La selección directa sería encontrar el arco de menor peso en el grafo y multiplicar dicho peso por el número de ciudades que quedan por visitar.

Ejemplo: “ a -- d – e .... X .... Y .... (a) “

Conocemos 2 arcos, pero nos falta por conocer 3 .



|               |       |
|---------------|-------|
| A – D         | 5     |
| D – E         | 3     |
| 3 VECES ..... | 1x3 3 |

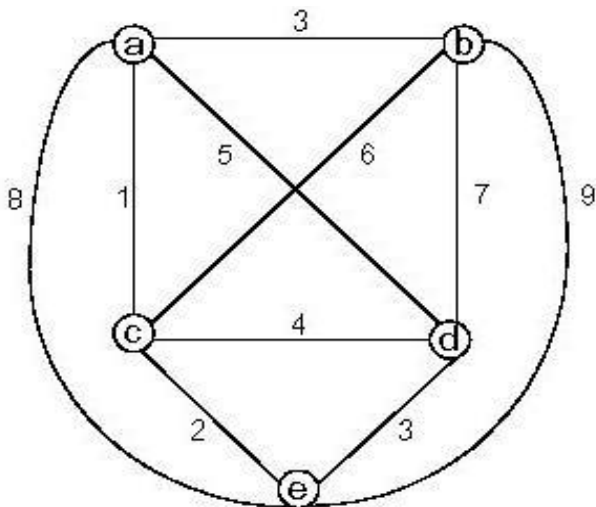
Estimador = 5 + 3 + 3 = 11

# Viajante de comercio: Cota 2

- Ya que un ciclo debe pasar exactamente una vez por cada vértice, un estimador (cota inferior) de la longitud del ciclo se tiene al considerar el MINIMO costo de “SALIR” de cada vértice.
  - Es el minimo valor de todas las entradas no nulas de la matriz de costos (matriz de adyacencia)

Ejemplo: “ a -- d – e .... X .... Y .... (a) “

Conocemos 2 arcos, pero nos falta por conocer 3 .



|            |   |
|------------|---|
| A – D      | 5 |
| D – E      | 3 |
| Salir de E | 2 |
| Salir de B | 3 |
| Salir de C | 1 |

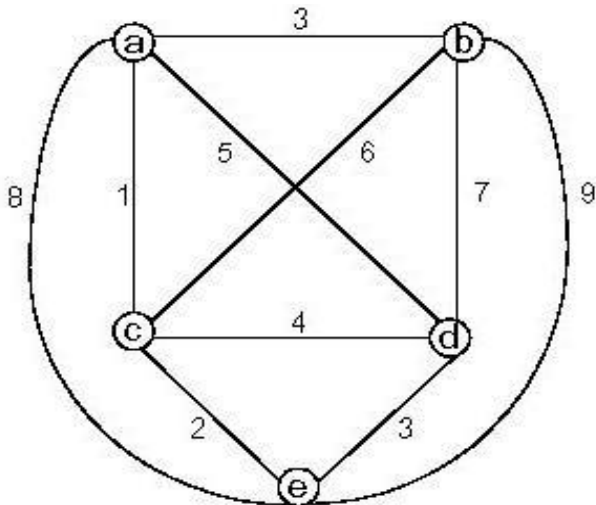
Estimador = 14

# Viajante de comercio: Cota 3

- Ya que un ciclo debe pasar exactamente una vez, un estimador (cota inferior) de la longitud del ciclo se tiene al considerar el MINIMO costo de “SALIR Y ENTRAR” en cada vértice.
  - Para cada arista  $(u, v)$ , podemos considerar la mitad de su peso como el costo de salir de  $u$ , y la otra mitad como el costo de entrar a  $v$ .
  - El costo total de un ciclo = suma de visitar (entrar y salir) un vértice cada vez
  - Cálculo:
    - Para cada vértice, considerar los dos arcos adyacentes con menor coste. Su suma, dividida por 2 es una cota inferior de pasar por cada vértice

Ejemplo: “ a -- d – e .... X .... Y .... (a) “

Conocemos 2 arcos, pero nos falta por conocer 3



.... A – D

$$(1+5) / 2 = 3$$

– D – E

$$(5+3) / 2 = 4$$

– E ....

$$(3+2)/2 = 2.5$$

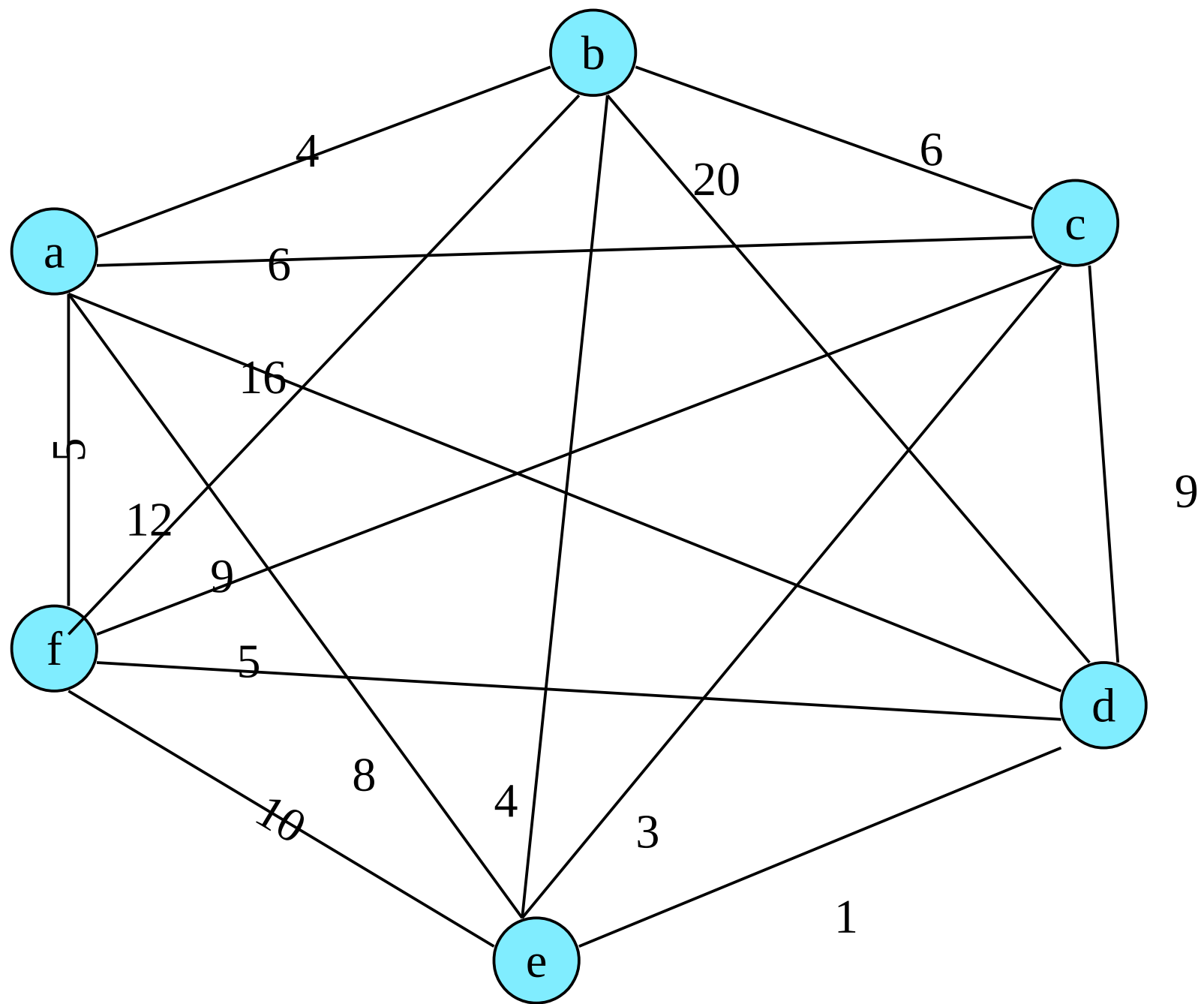
... B ....

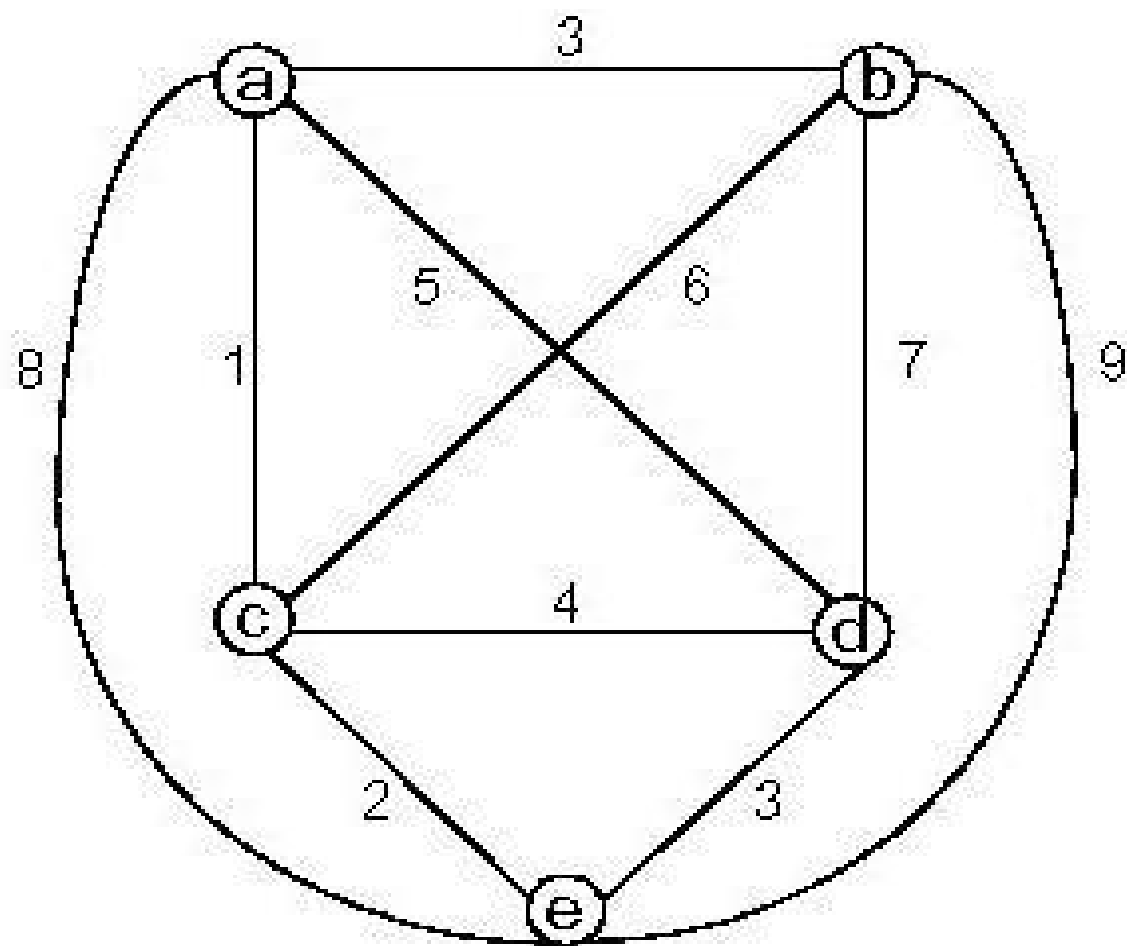
$$(3+6)/2 = 4.5$$

....C ....

$$(1+2)/2 = 1.5$$

Estimador = 15.5





# Eficiencia de Backtracking

- Depende de:
  - El tiempo necesario para generar la siguiente componente de la solución,  $X(k)$
  - El número de  $X(k)$  que satisfacen las restricciones explícitas
  - El tiempo de determinar la función de factibilidad (acota las soluciones)
  - El número de  $X(k)$  que satisfacen la función de factibilidad (número de nodos generados)

# Eficiencia de Backtracking

- Una función de factibilidad es buena sí:
  - permite reducir considerablemente (podar) el número de nodos generados.
  - El tiempo de ejecución es razonable
- Objetivo:
  - Reducir el tiempo global de ejecución
  - Hay que buscar un equilibrio entre esos factores: cuanto mejor (más nodos poda) la función también es mayor su tiempo de ejecución



# Eficiencia de backtracking

- De los 4 factores que determinan el tiempo requerido por un algoritmo backtracking, solo el cuarto, el número de nodos generados, varía de un caso a otro.
- Un algoritmo backtracking en un caso podría generar solo  $O(n)$  nodos, mientras que en otro (relativamente parecido) podría generar casi todos los nodos del árbol de espacio de estados.
- Si el número de nodos en el espacio solución es  $d^n$  o  $n!$ , el tiempo del peor caso para el algoritmo backtracking sería generalmente  $O(p(n)d^n)$  u  $O(q(n)n!)$  respectivamente, con  $p$  y  $q$  polinomios en  $n$ .
- La importancia del backtracking reside en su capacidad para resolver casos con grandes valores de  $n$  en muy poco tiempo.
- La dificultad está en predecir la conducta del algoritmo backtracking en el caso que deseemos resolver.