Apuntes de Teórico

PROGRAMACIÓN 3

Backtracking

Versión 1.3

Índice

Introducción

Backtracking es una técnica de diseño de algoritmos aplicable a problemas para los cuales se quiere obtener un conjunto de soluciones o la solución óptima según ciertas condiciones. Esta técnica es una alternativa a la "fuerza bruta" logrando las mismas soluciones de una manera más eficiente.

Para poder aplicar Backtracking la solución al problema se debe poder expresar como una tupla $<x_1,x_2,\ldots,x_n>$ donde cada x_i es elegido de un conjunto finito de valores. Estas tuplas deben respetar los criterios establecidos en el problema llamados RESTRICCIONES para ser consideradas soluciones. Estos criterios dividen el ESPACIO DE SOLUCIONES, es decir el conjunto de tuplas candidatas en 3 subconjuntos:

- No solución: son las tuplas que no cumplen las restricciones
- Solución. Son las tuplas que cumplen las restricciones.
- Óptimas. Son las tuplas solución que son óptimas (en el caso de problemas de optimización)

Los algoritmos basados en Backtracking proceden construyendo la tupla solución componente a componente y evaluando si el prefijo de tupla que se tiene en cada momento tiene posibilidades de conducir a una tupla solución, si no existe tal posibilidad se descarta el valor de dicha componente y se analiza con otro valor para la misma. Si no quedan valores para dicha componente se descarta el valor de la penúltima componente y se sigue de esta forma hasta agotar el espacio de soluciones.

Problema de las n reinas

Se trata de ubicar *n* reinas en un tablero de ajedrez de tamaño *nxn* de forma tal que dos reinas cualesquiera no puedan comerse entre sí, es decir que no estén en la misma fila, columna o diagonal.

En el tablero hay n^2 casillas y se deben elegir subconjuntos de *n* casillas. Una posibilidad es escribir un algoritmo que genere todos los subconjuntos de *n* casillas (observar que la cantidad de subconjuntos es $C_n^{n^2}$) y que seleccione aquellos subconjuntos que cumplen las restricciones dadas.

Un algoritmo con estas características es conocido como un algoritmo de FUERZA BRUTA.
Puede observarse que si n = 8 hay $C_8^{64} = 4420000000$ subconjuntos, lo cual hace que el algoritmo deba generar todos estos subconjuntos (si se quieren todas las soluciones) y chequearlos uno a uno.

Si se numeran las filas, las columnas y las reinas de 1 a n se puede asumir, sin perder generalidad, que como cada reina debe estar en una fila diferente entonces la reina i se colocará en la fila i, de esta manera las soluciones al problema de las n reinas pueden ser representadas por una tupla $<x_1,x_2,\ldots,x_n>$ donde x_i es la columna en la cual debe ser colocada la reina i, es decir que la reina i se colocará en la casilla (i, x_i) del tablero. De esta forma para cada x_i de la tupla hay n valores posibles con lo cual la cantidad de tuplas candidatas es n^n. En el caso de n = 8 serían $8^8 = 16$ millones de tuplas. Este enfoque es sustancialmente mejor que el anterior dado que reduce la cantidad de tuplas a generar y es la base para aplicar Backtracking, el cuál como se verá permite reducir aún más la cantidad de tuplas a generar.

Espacio de soluciones

En el ejemplo de las *n* reinas, con *n* = 4, las tuplas candidatas son las tuplas $<x_1,x_2,x_3,x_4>$ donde cada x_i toma valores en el conjunto {1,2,3,4}. Se denomina ESPACIO DE SOLUCIONES al conjunto de todas las tuplas que cumplen lo anterior.

Un algoritmo basado en Backtracking determina la o las soluciones al problema recorriendo sistemáticamente el espacio de soluciones, este recorrido se puede visualizar organizando el espacio de soluciones mediante un árbol llamado ÁRBOL DE SOLUCIONES.
Una visualización, para n = 4, del árbol de soluciones es la que se muestra en la figura 1.

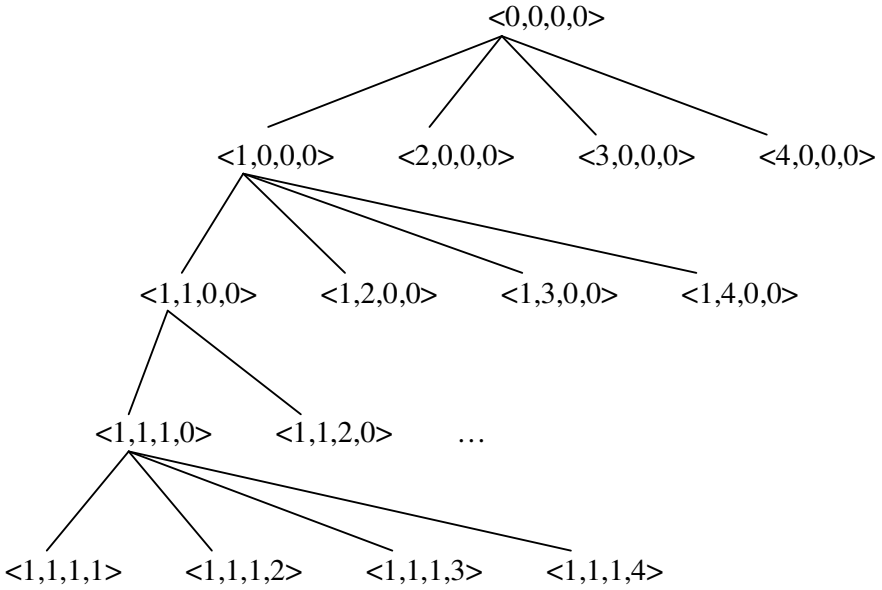


Figura 1

Se utilizará el valor 0 para indicar que no hay una reina colocada en la columna correspondiente.
Cada nodo del árbol de soluciones define un ESTADO DEL PROBLEMA. Un estado del problema *S* es un ESTADO DEL ESPACIO DE SOLUCIONES si el camino desde la raíz (<0,0,0,0>) al estado *S* define una tupla del espacio de soluciones (en el árbol de la figura 1 son las hojas).

Un estado del espacio de soluciones es un ESTADO SOLUCIÓN si el mismo cumple las restricciones del problema (en el árbol de la figura 1 son las hojas que representan una configuración del tablero donde 2 reinas cualesquiera no están en la misma fila, columna o diagonal).

Se observa que para n=4 hay 4^4=256 tuplas, sin embargo cuando un estado del problema (tupla en construcción) no cumple las restricciones no tiene sentido seguir generando dicha tupla dado que no conducirá a un estado solución, como se muestra en la figura 2.

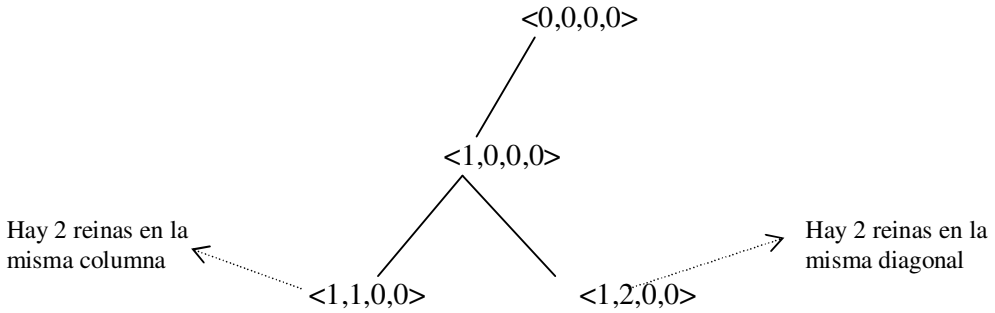


Figura 2.

La visualización en forma de árbol del espacio de soluciones permite definir cómo se pueden generar los estados del problema de forma sistemática. En particular Backtracking implementa la generación de los estados de la siguiente manera:

Se comienza con el estado raíz <0,0,0,0> (que representa la situación en la cual no se ha colocado ninguna reina en el tablero) y en un paso posterior, estando en un estado R se expande dicho estado, o sea se generan los hijos de R. Cada vez que se genera un hijo C de R, C se convierte en el estado actual y se lo expande. R volverá a ser el estado actual en expansión cuando el subárbol de raíz C haya sido generado completamente.

Notar que la forma de implementar la generación de los estados corresponde a una generación DFS de los estados del problema. Esto implica que las tuplas se generan componente a componente y que para cada prefijo de tupla generado se verifica si cumple las restricciones, si no las verifica no se sigue generando dicha tupla y se vuelve a al estado anterior. **Backtracking no implementa el árbol de soluciones, solamente genera sus estados.**

A continuación se verá detalladamente como genera Backtracking los estados para el problema de las *n* reinas en el caso *n* = 4 considerando el árbol de soluciones parcial de la figura 3.

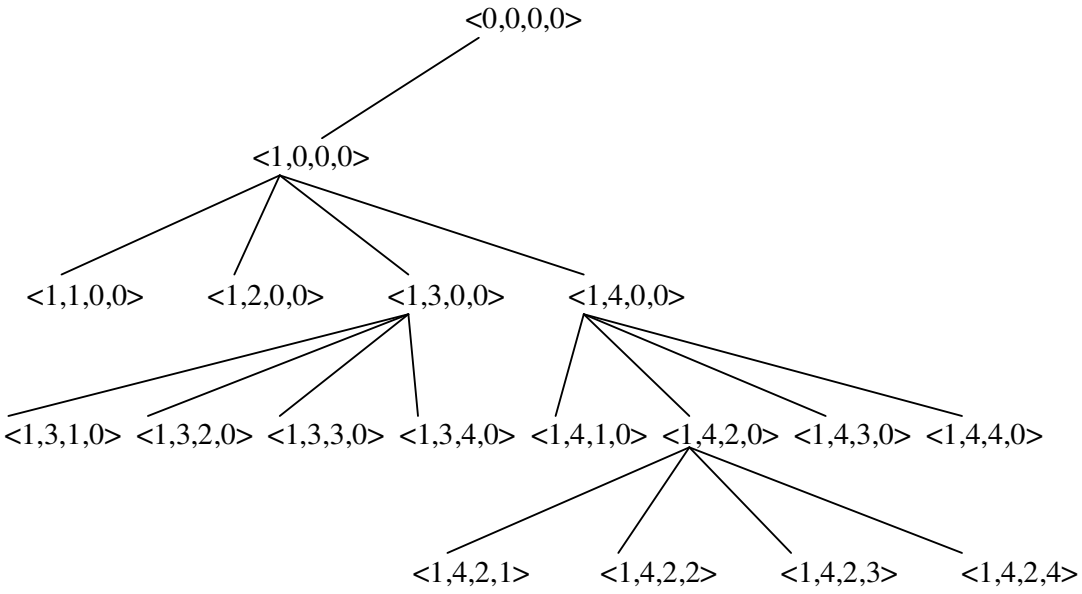


Figura 3.

Se comienza con el estado raíz <0,0,0,0>. Este estado es el actual y se expande. Primero se genera el hijo <1,0,0,0> que se convierte en el estado actual y se expande; se genera el estado <1,1,0,0>. Este estado no cumple con la restricción de que dos reinas estén en distinta columna por lo que no se expande, lo cual significa que no se generan más tuplas con el prefijo <1,1,0,0>. El algoritmo vuelve hacia atrás al estado <1,0,0,0> y se genera el estado <1,2,0,0> el cual tampoco es expandido por no cumplir la restricción de que dos reinas estén en distinta diagonal. Se vuelve hacia atrás al estado <1,0,0,0> y se genera el estado <1,3,0,0> el cual corresponde a la configuración del tablero:

X			
		X	

y cumple las restricciones.

A continuación se genera el estado <1,3,1,0> que no cumple con la restricción de que dos reinas estén en distinta columna por lo que no se expande. El algoritmo vuelve hacia atrás al estado <1,3,0,0> y se genera el estado <1,3,2,0> el cual tampoco es expandido por no cumplir la restricción de que dos reinas estén en distinta diagonal.

El algoritmo vuelve hacia atrás al estado <1,3,0,0> y se genera el estado <1,3,3,0> el cual tampoco es expandido por no cumplir la restricción de que dos reinas estén en distinta columna.

El algoritmo vuelve hacia atrás al estado <1,3,0,0> y se genera el estado <1,3,4,0> el cual tampoco es expandido por no cumplir la restricción de que dos reinas estén en distinta diagonal.

Al haber eliminado todas las posibilidades para el estado <1,3,0,0> se retrocede nuevamente al estado <1,0,0,0> y se genera el estado <1,4,0,0> el cual como puede observarse en el árbol, tampoco conducirá a una solución, por lo que al haberse examinado todas las posibilidades para el estado <1,0,0,0> sin llegar a una solución se vuelve al estado anterior que es la raíz del árbol (<0,0,0,0>).

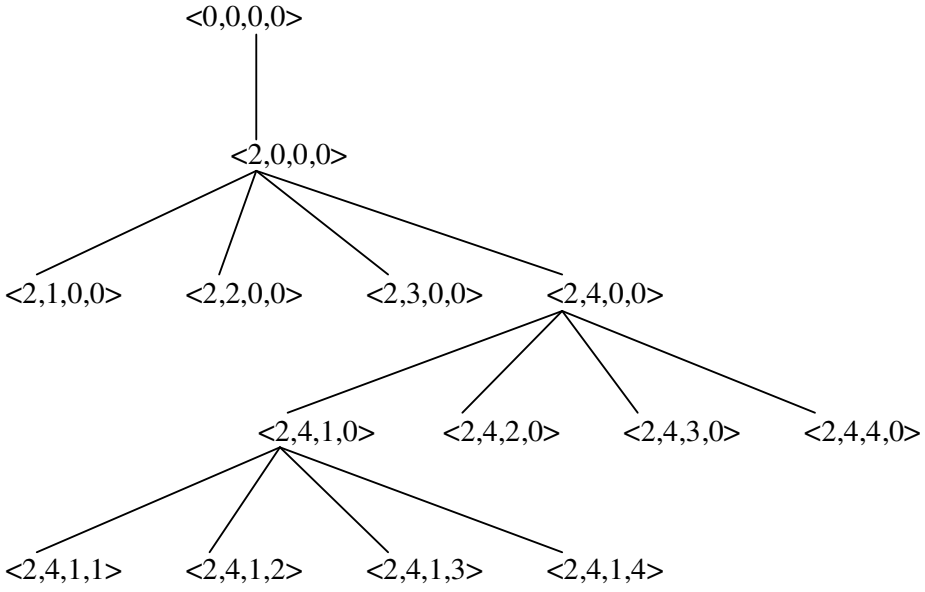


Figura 4.

Se genera el estado <2,0,0,0> como se muestra en la figura 4. De sus hijos solo el <2,4,0,0> representa una configuración del tablero que cumple las restricciones:

	X		
			X

Este estado pasa a ser el estado actual y se genera el <2,4,1,0> que corresponde a la configuración del tablero:

	X		
			X
X			

que también cumple las restricciones.

Luego se generan los nodos <2,4,1,1> que no cumple las restricciones y el <2,4,1,2> que tampoco cumple las restricciones. Posteriormente se genera el estado <2,4,1,3> que corresponde a la configuración del tablero:

	X		
			X
X			
		X	

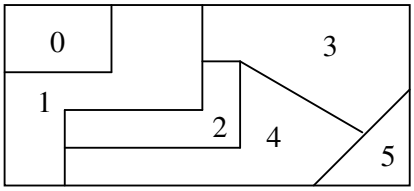
En este momento al haberse colocado todas las reinas en una configuración en la cual no pueden comerse entre sí, se ha llegado a un estado solución el cual corresponde a una tupla solución al problema. El algoritmo puede finalizar si se quería encontrar una solución o continuar si se desean encontrar más soluciones (o todas ellas).

Notar además como la cantidad de tuplas generadas es menor que 256 tuplas (n^4) dado que al verificar que dos reinas estén en distinta columna se generan solamente tuplas que son permutaciones de la tupla <1,2,3,4> dado que los valores de las componentes deben ser distintos entre sí; entonces se generan a los sumo 4! = 24 tuplas, algunas de las cuales tampoco serán generadas al verificarse la restricción de que dos reinas estén en distinta diagonal.

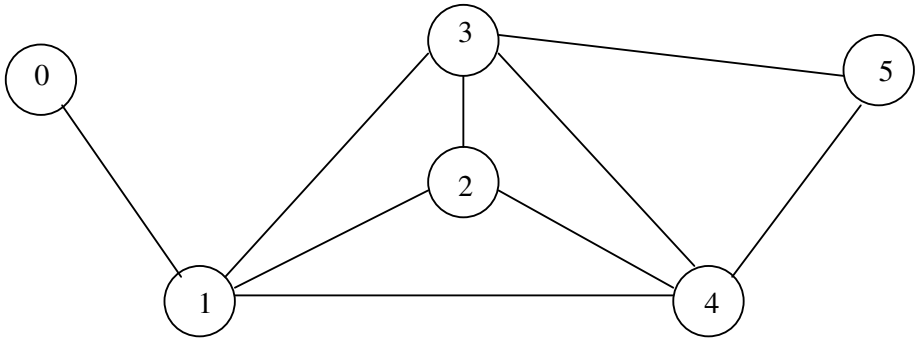
Colorear un mapa

El problema a resolver consiste en colorear un mapa de forma tal que 2 países con frontera no tengan el mismo color.

Por ejemplo, considerando el mapa:



se lo puede representar mediante un grafo no dirigido donde cada vértice representa un país y 2 vértices son adyacentes si los países correspondientes tienen frontera.



Suponiendo que se dispone de k colores numerados de 1 a k la solución se puede expresar como una tupla $<x_o,x_1,\ldots,x_{n-1}>$ donde $x_i \in 1..k$ para $0 \leq i \leq n-1$, con $n = |V|$

La figura 5 muestra una organización en forma de árbol para el espacio de soluciones:

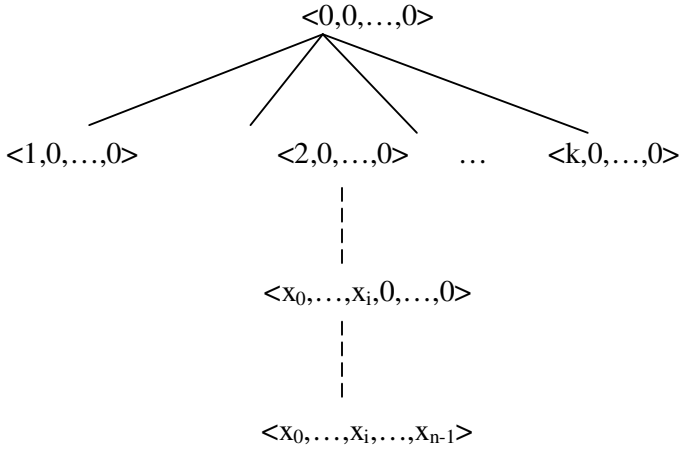


Figura 5.

Se utilizará el valor 0 para indicar que no se ha asignado un color a un vértice. El espacio de soluciones está formado por las hojas del árbol.

Se verán 3 variantes de aplicación de la técnica de Backtracking:

1. Encontrar una solución
2. Encontrar todas las soluciones
3. Encontrar la solución que minimiza la cantidad de colores utilizados

Hallar una solución

En un determinado momento, el algoritmo recibe una tupla en construcción parcialmente coloreada (colores asignados a algunas componentes) y debe verificar si cumple las restricciones del problema. Si las cumple se controla si es solución, si lo es se utiliza una variable booleana para indicar que se ha encontrado una solución y que el algoritmo debe finalizar.

Son necesarios además parámetros para devolver la tupla solución y para indicar a partir de que posición se debe seguir coloreando la tupla en construcción.

```
void colorear(int *tupla, int i, bool &encontre, int *sol)
{
      int color;
      if (verificaRest(tupla, i))
      {
            if (esSolucion(tupla, i))
            {
                  encontre = true;
                  copiar(sol, tupla);
            } else
            {
                  color = 1;
                  while ((color <= k) && (!encontre))
                  {
                        tupla[i] = color;
                        colorear(tupla, i+1, encontre, sol);
                        color++;
                  }
            }
      }
}
```

- `i` es la posición a partir de la cual colorear.
- `tupla` es la tupla que se construye.
- `encontre` indica si se encontró una solución.
- `sol` es la tupla solución que se devuelve.

Notar que la iteración con la llamada recursiva genera la tupla en profundidad, o sea mediante DFS.

La función `verificaRest` evalúa si la tupla `tupla` cumple las restricciones (dos adyacentes no tienen el mismo color). Toma en cuenta solo los elementos de la tupla hasta la posición `i`.

```
bool verificaRest (int *tupla, int i)
{
      int h, j;
      bool ok;
      ok = true;
      j = 0;
      while (ok && (j <= i-1))
      {
            h = 0;
            while ((h < i) && ok)
            {
                  if adyacentes(j,h)
                        ok = tupla[j] != tupla[h];
                  h++;
            }
            j++;
      }
      return ok;
}
```

La función `esSolucion` verifica que se haya llegado a una solución, o sea que todos los vértices (componentes) tengan color asignado.

```
bool esSolucion(int *tupla, int i)
{
      return (i == n);
}
```

La invocación es:

```
bool encontre = false;
int *tupla;
int *sol;
tupla = new int[n]; //n = |V|
sol = new int[n];
for(int i = 0; i < n; i++)
      tupla[i] = 0;
colorear(tupla, 0, encontre, sol);
if (encontre)
{
 …
}
else
{
 …
}
```

Hallar todas las soluciones

En este caso no se controla el hecho de haber encontrado una solución por lo cual el parámetro booleano es innecesario.

```
void colorear(int *tupla, int i, Solucion tuplas)
{
      int color;
      if (verificaRest (tupla, i))
            if (solucion(tupla, i))
                  Agregar(tuplas, tupla)
            else
            {
                  for(color = 1; color <= k; color++)
                  {
                        tupla[i] = color;
                        colorear(tupla, i+1, tuplas);
                  }
            }
}
```

El nuevo parámetro `tuplas` es el conjunto de tuplas solución. El resto de los parámetros son similares al caso anterior.

Las funciones `verificaRest` y `esSolucion` son las mismas que en el caso anterior.

Observar que en este caso se examinan todos los posibles colores para cada componente de la tupla.

La invocación es:

```
int *tupla = new int[n];
for(int i = 0; i < n; i++)
      tupla[i] = 0;
Solucion tuplas = NULL;
colorear(tupla, 0, tuplas);
if (tuplas != NULL)
{
 …
}
else
{
 …
}
```

Hallar la solución con mínima cantidad de colores (óptima)

En este caso desea encontrar la solución que utiliza la mínima cantidad de colores. Para esto es necesario llevar la cuenta de la cantidad de colores utilizados en la tupla para poder comparar.

```
struct tupla {
      int *colores;
      int *cantVeces;
      int cantColores;
}
```

El campo `cantColores` guarda la cantidad de colores distintos utilizados en la tupla.
El campo `cantVeces` se utiliza para saber de manera eficiente (sin necesidad de recorrer la tupla) si un color se utilizó una o varias veces, a los efectos de actualizar el campo `cantColores`.

El algoritmo debe buscar todas las soluciones para encontrar la óptima.

```
void colorear (tupla &t, int i, tupla &sol)
{
      int color;
      if (verificaRest (t, i))
      {
            if esMejor(t, sol)
            {
                  if (solucion(t, i))
                        copiar(sol, t); //copia la tupla en sol
                  else
                  {
                        for(color = 1; color <= k; color++)
                              colorear(AsignarColor(t, i, color), i+1, sol);
                  }
            }
      }
}
```

- `t` es la tupla en construcción.
- `i` es la posición a partir de la cual colorear.
- `sol` es la mejor solución encontrada hasta el momento.

Las funciones `verificaRest` y `esSolucion` son las mismas que en el caso anterior.

La función `esMejor` evalúa si el prefijo de tupla en construcción utiliza una cantidad de colores menor que la solución actual.

```
bool esMejor(tupla t, tupla sol)
{
      return (t.cantColores < sol.cantColores);
}
```

La función `asignarColor` asigna un color y actualiza los campos de la estructura.

```
tupla AsignarColor(tupla &t, int i, in color)
{
      t.colores[i] = color;
      t.cantVeces[color]++;
      if (t.cantVeces[color] == 1)
            t.cantColores++;
      return t;
}
```

La invocación es:

```
tupla t, sol;
t.colores = new int[n];
t.cantVeces = new int[k+1];
sol.colores = new int[n];
sol.cantVeces = new int[k+1];
for(int i = 0; i < n; i++)
{
      t.colores[i] = 0;
      sol.colores[i] = 0;
}
for(int i = 0; i <= k; i++)
{
      t.cantVeces[i] = 0;
      sol.cantVeces[i] = 0;
}
t.cantColores = 0;
sol.cantColores = maxint;
colorear(t, 0, sol);
```

Se observa que se agrega una condición que permite podar el árbol: si en un paso cualquiera el prefijo de tupla tiene una cantidad de colores mayor que la solución encontrada hasta el momento no tiene sentido seguir generando la tupla dado que no conducirá a la solución óptima (mínima cantidad de colores).

Notar la diferencia en el caso que en el algoritmo se invocara a la función `esSolucion` antes de invocar a la función `esMejor`. En este caso se asignarían colores a todas las componentes de la tupla y luego se compararía con la mejor solución encontrada hasta el momento.
La forma en la cual se hace en el algoritmo permite comparar un prefijo de tupla (tupla que no tiene necesariamente colores asignados a todas las componentes) con la mejor solución encontrada hasta el momento, en caso que el prefijo utilice una cantidad mayor de colores no se sigue generando la tupla correspondiente.

Formalización

Como paso previo a escribir un algoritmo de Backtracking se requiere formalizar el problema. La formalización permite definir principalmente cual es el espacio de soluciones del problema y cuales son las tuplas del mismo que son solución. Para la formalización se debe especificar cada uno de los siguientes ítems:

- Forma de la solución
  Aquí se describen las características de la tupla que se usará para dar la solución al problema. En particular si la tupla tendrá una cantidad fija de componentes (indicando que representa esa cantidad) o si tendrá una cantidad variable de componentes. Además se indicará la notación que se utilizará, es decir que representa cada componente de la tupla.
- Restricciones explícitas
  Son reglas que afectan a una sola componente de la tupla y deben indicarse para cada componente de la misma. En general estas reglas determinan el dominio de cada componente y por lo tanto determinan el espacio de soluciones.
- Restricciones implícitas
  Son reglas que determinan cuando una tupla del espacio de soluciones es solución. Estas reglas involucran a varias o todas las componentes de la tupla describiendo como se relacionan entre sí para que la tupla sea una solución. La falta de alguna restricción implícita implica que serán consideradas como solución tuplas que en realidad no lo son.
- Función objetivo
  La función objetivo debe especificarse únicamente para problemas de optimización y se refiere a optimizar alguna cantidad.
- Predicados de poda
  Se definirán en la sección "Sobre la implementación de algoritmos de Backtracking".

Formalización del problema de las n reinas

- Forma de la solución
  Tupla de largo fijo *n* de la forma $<x_1,x_2,\ldots,x_n>$, donde *n* es la cantidad de reinas. Cada componente x_i representa la columna donde se ubicará la reina i, $\forall i, 1 \leq i \leq n$

- Restricciones explícitas
  $x_i \in \{1..n\} \forall i, 1 \leq i \leq n$

- Restricciones implícitas
  Las reinas deben estar en distintas columnas $x_i \neq x_j \; si \; i \neq j \; \forall i, j \; 1 \leq i, j \leq n$
  Las reinas deben estar en distinta diagonal $abs(x_i - x_j) \neq abs(i - j) \; \forall i, j \; 1 \leq i, j \leq n$

  Notar que la restricción que las reinas estén en distintas filas está considerada en la forma de la solución, por lo cual no es necesario una restricción implícita al respecto.

- Función objetivo
  No corresponde porque no es un problema de optimización.

Formalización del problema de colorear un mapa con la mínima cantidad de colores

- Forma de la solución
  Tuplas de largo fijo n de la forma $<x_0,x_1,\ldots,x_{n-1}>$, donde $n=|V|$. Cada componente x_i representa el color del vértice i, $\forall i, 0 \leq i \leq n-1$

- Restricciones explícitas
  $x_i \in \{1..k\} \forall i, 0 \leq i \leq n-1$ *siendo* $\{1..k\}$ *el conjunto de colores*

- Restricciones implícitas
  Si dos países tienen frontera entonces sus colores deben ser distintos
  $\forall i, j \in V$ *si adyacentes*$(i, j) \Rightarrow x_i \neq x_j$

- Función objetivo
  La función objetivo es f = min(cantColores(t)), donde 'cantColores(t)' es la cantidad de colores distintos utilizados en una tupla solución.

  $$cantColores(t) = \sum_{i=1}^{k} pertenece(t,i)$$

  $$pertenece(t,i) = \begin{cases} 1 & si \quad i \in t \\ 0 & si\,no \end{cases}$$

  $f = \min_{t \in T}(cantColores(t))$, donde T = {t = $<x_0,x_1,\ldots,x_{n-1}>$ / t es solución}

Suma de subconjuntos

Dado un conjunto W finito de reales positivos con |W| = n y un real positivo M, se desea encontrar todos los subconjuntos de W cuyos elementos suman exactamente M.

Por ejemplo:

W = {11, 13, 24, 7}y M = 31
Hay 2 subconjuntos cuyos elementos suman 31: W_1 = {11, 13, 7} y W_2 = {24, 7}

Para definir como se dará la solución se debe decidir como se representarán los subconjuntos. Hay 2 formas simples de representar los subconjuntos, asumiendo que W es un arreglo de *n* elementos:

a) Vector de bits

Se indica si cada elemento de W está en un subconjunto solución, para lo cual se utiliza una correspondencia posición a posición entre los elementos de W y los del subconjunto solución.
Notar que esto lleva a una formulación de tupla de largo fijo, donde para cada elemento de W se debe indicar si está o no en el subconjunto. A continuación se presenta la formalización correspondiente.

- Forma de la solución
  Tupla de largo fijo *n* de la forma $<x_0, \ldots, x_{n-1}>$, donde *n* = |W|. Cada componente x_i representa si el i-ésimo elemento de W está en la tupla.

- Restricciones explicitas
  $x_i \in \{0,1\}\ \forall i, 0 \le i \le n-1$

- Restricciones implícitas
  $\sum_{i=0}^{n-1} x_i * W[i] = M$

- Función objetivo
  No corresponde.

b) Representar los subconjuntos mediante secuencias

Siguiendo con el ejemplo, se pueden representar los subconjuntos solución por:

W_1 → 11→ 13→ 7
W_2 → 24→ 7

Aunque es conveniente manejar los índices de los elementos en el arreglo W y no los valores:

W_1 → 0→ 1→ 3
W_2 → 2→ 3

Notar que en este caso las tuplas que representan los subconjuntos solución tienen largo variable. A continuación se presenta la formalización correspondiente.

- Forma de la solución
  Tupla de largo k variable de la forma $<x_0, \ldots, x_{k-1}>$ con $0 \leq k \leq n$ y $n = |W|$. Cada componente x_i representa el índice de un elemento del conjunto W.

- Restricciones explicitas
  $x_i \in 0..n-1 \; \forall i, 0 \leq i \leq k-1$

- Restricciones implícitas
  $\sum_{i=0}^{k-1} W[x_i] = M$

  $x_i < x_{i+1} \; \forall i, 0 \leq i \leq k-2$ no se desean soluciones con elementos repetidos ni que sean permutaciones de un mismo subconjunto

- Función objetivo
  No corresponde.

Un árbol de soluciones para esta formulación de las soluciones y considerando |W| = 4 es el que se muestra en la figura 6.

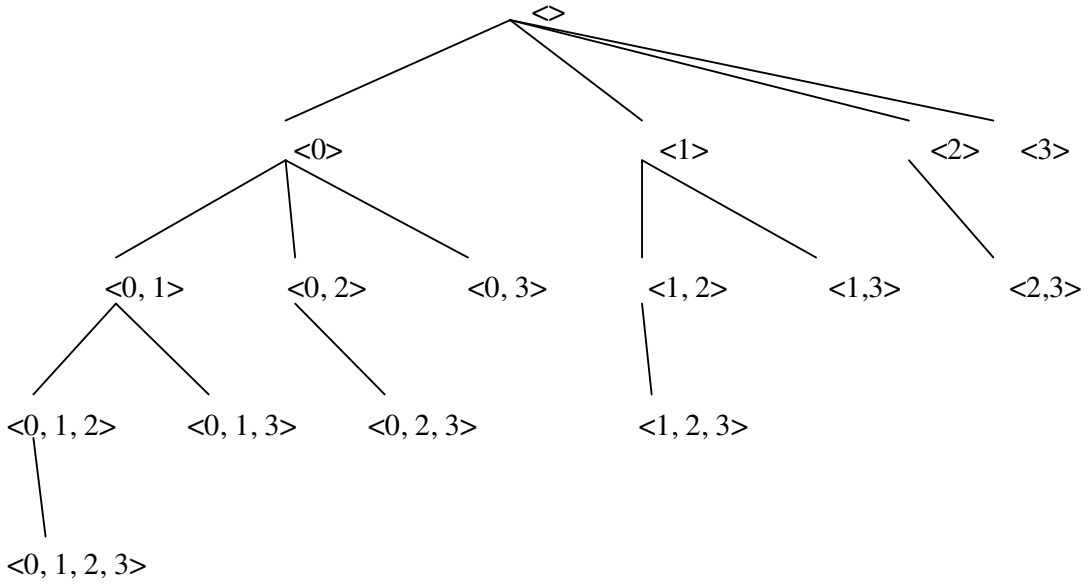


Figura 6.

Observar que en este caso el espacio de soluciones está dado por todos los subconjuntos de W, por lo cual el espacio de soluciones son todos los nodos del árbol.

Sobre la implementación de algoritmos de Backtracking

A partir de la formalización, que es donde se describen las propiedades de las tuplas que constituyen la solución al problema, se realiza la implementación del algoritmo en un cierto lenguaje de programación.

Al momento de la implementación del algoritmo se utilizan las restricciones implícitas o la función objetivo como funciones de poda que se evalúan sobre una tupla en construcción (prefijo de tupla) determinando si las cumple o no. Esto es, porque si una tupla en construcción no cumple alguna restricción implícita o la función objetivo entonces tampoco la cumplirá una vez que se haya terminado de construir.
Notar que se utilizan para detectar lo antes posible que una tupla en construcción no será solución, es decir es una forma de no seguir generando una tupla que finalmente no será solución.

En términos del árbol de soluciones, estas funciones permiten podar un subárbol sin necesidad de llegar a un estado del espacio de soluciones, el cual no será un estado solución porque no verificará las restricciones implícitas o la función objetivo.

En la implementación de colorear un mapa utilizando la mínima cantidad de colores se observa el uso de estas funciones de poda, derivadas de las restricciones implícitas, así como de la función objetivo.

Se debe tener en cuenta que en algunos casos, en la implementación, pueden derivarse de las restricciones implícitas formas de generar valores para una componente del prefijo de tupla (además de lo que indican las restricciones explícitas sobre dicha componente). Esto se verá más adelante en la implementación del algoritmo para el caso de suma de subconjuntos.

Además de las funciones de poda que se derivan de las restricciones implícitas o de la función objetivo pueden existir adicionalmente datos obtenidos del planteo del problema que permitan mejorar aún más la poda (con la intención de mejorar el desempeño del algoritmo en cuanto a la cantidad de tuplas a generar) A dichos datos se les denominará "**predicados de poda**". Y sirven en lo concreto del algoritmo para indicar si una tupla en construcción, que cumple las funciones de poda derivadas de las restricciones implícitas o de la función objetivo, conducirá o no a una tupla solución.

Por ejemplo en el caso del problema de suma de subconjuntos, se tiene el siguiente predicado de poda: si la suma de los elementos del conjunto W que han sido incluidos en la tupla en construcción más la suma de los elementos de W que no han sido considerados aún, no es por lo menos M entonces la tupla en construcción no conducirá a una tupla solución porque no se verificará la restricción implícita de que los elementos de la tupla sumen M (notar que si no se cumple lo expresado anteriormente, a pesar de que se pusieran en la tupla todos los elementos de W no considerados aún, no se alcanzaría M).

El predicado de poda anterior se puede expresar de la siguiente manera, dependiendo de la formulación de la solución del problema:

- Vector de bits
  Sea un prefijo de tupla $<x_0,\ldots,x_k,0,\ldots,0>$, si verifica $\sum_{i=0}^{k} x_i * W[i] + \sum_{i=k+1}^{n-1} W[i] < M \quad \forall k, 0 \le k \le n-1$ entonces no se sigue generando la tupla.

- Secuencia
  Sea un prefijo de tupla <x0,…,xi,0,…,0>, si verifica $\sum_{j=0}^{i} W[x_j] + \sum_{i=x_i+1}^{k-1} W[i] < M$, $0 \le i \le k-1$ y $0 \le k \le n-1$ entonces no se sigue generando la tupla.

Importante:

- **Se deberán indicar los predicados de poda que existan en el problema a resolver. Estos predicados se escribirán en la formalización, dentro de un ítem cuyo nombre será *Predicados de Poda*.**

- Forma de la solución
  Tupla de largo k variable de la forma <x_0, …, x_{k-1}> con $0 \le k \le n-1$ y n = |W|. Cada componente x_i representa el índice de un elemento del conjunto W.

- Restricciones explicitas
  $x_i \in 0..n-1 \; \forall i, 0 \le i \le k-1$

- Restricciones implícitas
  $\sum_{i=0}^{k-1} W[x_i] = M$

  $x_i < x_{i+1} \; \forall i, 0 \le i \le k-2$ no se desean soluciones con elementos repetidos ni que sean permutaciones de un mismo subconjunto

- Función objetivo
  No corresponde.

- Predicados de poda
  Sea un prefijo de tupla <x0,…,xi,0,…,0>, si verifica $\sum_{j=0}^{i} W[x_j] + \sum_{i=x_i+1}^{k-1} W[i] < M$, $0 \le i \le k-1$ y $0 \le k \le n-1$ entonces no se sigue generando la tupla.

Sobre los predicados de poda, se debe tener en cuenta lo siguiente:

- Dependiendo del problema pueden existir o no.
- No determinan si una tupla será solución, es decir el hecho de no controlar este tipo de predicados no genera "falsas" soluciones, porque dicha tupla cuando este completamente construida no verificará alguna restricción implícita o la función objetivo.
- El hecho de controlar estos predicados no asegura una mayor eficiencia del algoritmo de Backtracking, dado que hay que considerar la complejidad del propio predicado.
- Algunas veces, para formular estos predicados, puede ser necesario utilizar datos contenidos en el problema y no solamente la información contenida en el prefijo de tupla.

Otros aspectos relevantes en cuanto a la implementación de algoritmos de Backtracking, están relacionados con los siguientes ítems:

- Uso de variables y/o parámetros para hacer más eficiente la verificación de predicados de poda.
- Uso adecuado del tipo de pasaje de parámetros.

Ambos ítems se verán en detalle en la implementación del algoritmo para el problema de la suma de subconjuntos. Para esta implementación se usará la formulación de tupla de largo variable. Recordar que en este caso los componentes de las tuplas solución tendrán como valores los índices de los elemento del conjunto W.

Para hacer más eficiente la evaluación del predicado de poda se utilizarán los siguientes parámetros:

- `total`: suma de los elementos del conjunto W que han sido incluidos en la tupla en construcción.

- `resto`: suma de los elemento de W no considerados aún para la tupla en construcción.

El predicado de poda, se puede expresarse como:

- `Si total + resto < M` entonces no seguir generando la tupla.

Lo anterior implica que no se deba recorrer en cada paso del algoritmo la tupla para calcular la suma de los elementos contenidos en la misma, ni tampoco calcular la suma de los elementos del conjunto W que no han sido considerados aún.

```
void sumaSub(Tupla t, int ultimo, float total, float resto, Solucion &sol)
{
      if ((total + resto >= M) && (total <= M))
      {
            if (total == M)
            {
                  Agregar(sol, t);
            }
            else
            {
                  for(int proximo = ultimo; proximo < n; proximo++)
                  {
                        resto = resto – W[proximo];
                        sumaSub(Agregar(t, proximo), proximo + 1,
                                total + W[proximo], resto, sol);
                  }
            }
      }
}
```

- `t` es la tupla en construcción.
- `ultimo` indica cual es el índice del elemento de W a considerar para ser agregado a la tupla.
- `total` es la suma de los elementos del conjunto W que han sido considerados para estar en la tupla en construcción (hayan sido incluidos en la tupla o descartados)
- `resto` indica la suma de los elementos de W que no han sido considerados aún.
- `sol` es el conjunto de soluciones encontradas hasta el momento.

La invocación es:

```
Tupla t = NULL;
Solucion sol = NULL;
int ultimo = 0;
int total = 0;
int resto = 0;
for(int i = 0; i < n; i++)
      resto = resto + W[i];
sumaSub(t, ultimo, total, resto, sol);
//si no hay solucion al finalizar, sol == NULL
```

En el algoritmo se observa que para generar los valores de cada componente de la tupla se utiliza la restricción implícita $x_i < x_{i+1} \ \forall i, 0 \le i \le k-2$ para asegurar que no haya repetidos ni obtener tuplas permutadas.
Además se observa que el parámetro `resto` es actualizado antes de la invocación recursiva, esto es porque se desea que cuando se retorne de la invocación recursiva dicho parámetro quede con el valor correcto que indique la suma de los elementos no considerados aún.

Lo expresado en el párrafo anterior tiene que ver con el tipo de pasaje que se usa para cada uno de los parámetros. Para los distintos parámetros se debe tener especial cuidado con:

- el tipo de pasaje de parámetros utilizado
- la forma en como se actualiza su valor en el algoritmo
- si se debe o no conservar el valor del parámetro al retornar de la invocación recursiva

dado que habrá parámetros cuyo valor no debe ser modificado al retornar de una invocación recursiva (caso de los parámetros `t` y `total`) y otros sí (caso del parámetro `resto`).

Por ejemplo, si previamente a la invocación recursiva se modificara el valor de los parámetros `t` y `total` de la siguiente manera:

```
total = total + W[proximo];
Agregar(t, próximo);
```

se deben deshacer los cambios hechos a dichos parámetros al retornar de la recursión para que conserven el valor que tenían previamente a la misma, por lo cual la iteración sería:

```
for(int proximo = ultimo; proximo < n; proximo++)
{
      resto = resto – W[proximo];
      total = total + W[proximo];
      Agregar(t, proximo);
      sumaSub(t, proximo, total, resto, sol);
      //ahora se deshacen los cambios
      total = total – W[proximo];
      Borrar(t, proximo);
}
```

Eficiencia

En general la eficiencia de un algoritmo de Backtracking depende de:

- Tiempo necesario para generar cada valor de una componente.
- Cantidad de valores de cada componente que satisfacen las restricciones implícitas.
- Tiempo necesario para evaluar las restricciones implícitas y los predicados de poda.

Las restricciones implícitas, la función objetivo y los predicados de poda reducen la cantidad de nodos a visitar del árbol de soluciones, aunque esto no garantiza que disminuya el tiempo de ejecución.

Las restricciones implícitas y la función objetivo deben ser chequeadas necesariamente, de otra forma se podrían obtener tuplas que no sean solución. En cambio los predicados de poda no generan problemas si no son chequeados ya que los nodos serán descartados más adelante.
Se debe evaluar, en cada caso, el costo de generar más nodos o evaluar los predicados de poda.

En el peor de los casos se visitarán todos los nodos del árbol de soluciones por lo que el algoritmo tendrá orden exponencial.