Capítulo 1
Vuelta Atrás[1]

Más vale una retirada a tiempo que una batalla perdida.

Napoleón Bonaparte.

Resumen: En este tema se introduce el esquema algorítmico de *Vuelta atrás*, o *Backtracking*, que es una técnica de carácter general utilizada para resolver una gran clase de problemas, en especial de problemas que exigen un recorrido exhaustivo del universo de soluciones.

1. Motivación

Dado un mapa M y un número $n > 0$ se pide encontrar las formas de colorear los países de M utilizando un máximo de n colores, de tal manera que ningún par de países fronterizos tenga el mismo color.

Una forma de resolver el problema es generar todas las posibles maneras de colorear el mapa y después desechar aquellas en que dos países fronterizos tienen el mismo color. Esta forma de resolver el problema sólo resulta aceptable si el conjunto de países, m, y el conjunto de colores, n, son pequeños, ya que el número de posibles formas de colorear el mapa viene dado por las variaciones con repetición de n elementos tomados de m en m: $VR_n^m = n^m$. En la siguiente tabla se muestra el número de posibilidades que habría que generar para algunos valores de n y m.

n: colores	m: países	nº de posibilidades
3	17	129.140.163
5	17	762.939.453.125
3	50	717.897.987.691.852.588.770.249

Se observa, que si tomamos las 17 comunidades autónomas de España, el número de posibilidades a generar con 5 colores sería superior a los 700 mil millones. Si consideramos las 50 provincias, con sólo 3 colores, el número de posibilidades supera el millón de billones.

[1] Miguel Valero Espada es el autor principal de este tema.
Modificado por Isabel Pita en el curso 2012/13.

En este capítulo se explica como abordar aquellos problemas cuya única forma conocida de resolver es la generación de todas sus posibles soluciones, para obtener de entre ellas la solución o soluciones reales.

2. Introducción

Existen problemas, como el que se presenta en el apartado anterior, para los que no parece existir una forma o regla fija que nos lleve a obtener una solución de una manera eficiente y precisa. La manera de resolverlos consiste en realizar una búsqueda exhaustiva entre todas las soluciones potenciales hasta encontrar una solución válida, o el conjunto de todas las soluciones válidas. A veces se requiere encontrar *la mejor* (en un sentido preciso) de todas las soluciones válidas.

La búsqueda exhaustiva en un espacio finito dado se conoce como *fuerza bruta* y consiste en ir probando sistemáticamente todas las soluciones potenciales hasta encontrar una solución satisfactoria, o bien agotar el universo de posibilidades. En general, la *fuerza bruta* es impracticable para espacios de soluciones potenciales grandes, lo cual ocurre muy a menudo, ya que el número de soluciones potenciales tiende a crecer de forma exponencial con respecto al tamaño de la entrada en la mayoría de los problemas que trataremos. Obsérvense los valores que se proporcionan en el ejemplo de la sección anterior.

El esquema algorítmico de *vuelta atrás* es una mejora a la estrategia de *fuerza bruta*, ya que la búsqueda se realiza de manera estructurada, descartando grandes bloques de soluciones para reducir el espacio de búsqueda. La diferencia principal entre ambos esquemas es que en el primero las soluciones se forman de manera progresiva, generando soluciones parciales, comprobando en cada paso si la solución que se está construyendo puede conducir a una solución satisfactoria. Mientras que en la *fuerza bruta* la estrategia consiste en probar una solución potencial completa tras otra sin ningún criterio.

En el esquema de *vuelta atrás*, si una solución parcial no puede llevar a una solución completa satisfactoria, la búsqueda se aborta y se vuelve a una solución parcial viable, deshaciendo decisiones previas. Este esquema debe su nombre a este salto hacia atrás.

Tomemos como ejemplo el siguiente problema: dadas n letras diferentes, diseñar un algoritmo que calcule las palabras con m letras ($m \leq n$) diferentes escogidas entre las dadas. El orden de las letras es importante: no será la misma solución *abc* que *bac*.

El número de soluciones potenciales o *espacio de búsqueda* es de n^m, que representan las variaciones con repetición de n letras tomadas de m en m. Realizar una búsqueda exhaustiva en este espacio es impracticable.

Antes de ponernos a probar sin criterio alguno entre todas las posibles combinaciones de letras, dediquemos un segundo a evaluar la estructura de la solución. Es evidente que no podemos poner dos letras iguales en la misma palabra, así que vamos a replantear el problema. Trataremos de colocar las letras de una en una, de forma que no se repitan. De esta manera, toda solución del problema se puede representar como una tupla $(x_1, .., x_m)$ en la que x_i representa la letra que se coloca en el lugar i-ésimo de la palabra.

La solución del problema se construye de manera incremental, colocando una letra detrás de otra. En cada paso se comprueba que la última letra no esté repetida con las anteriores. Si la última letra colocada no está repetida, la solución parcial se dice *prometedora* y la búsqueda de la solución continúa a partir de ella. Si no es prometedora, se abortan todas las búsquedas que partan de esa tupla parcial.

De manera general, en los algoritmos de *vuelta atrás*, se consideran problemas cuyas soluciones se puedan construir por etapas. Una solución se expresa como n-tupla $(x_1, .., x_n)$

donde cada $x_i \in S_i$ representa la decisión tomada en la i-ésima etapa de entre un conjunto finito de alternativas.

Una solución tendrá que minimizar, maximizar, o simplemente satisfacer cierta *función criterio*. Se establecen dos categorías de restricciones para los posibles valores de una tupla:

- **Restricciones explícitas**, que indican los conjuntos S_i. Es decir, el conjunto finito de alternativas entre las cuales pueden tomar valor cada una de las componentes de la tupla solución.
- **Restricciones implícitas**, que son las relaciones que se han de establecer entre las componentes de la tupla solución para satisfacer la función criterio.

Volviendo al ejemplo de las palabras con letras diferentes, hemos visto que la solución será una m-tupla y que cada valor de la tupla es una letra en la palabra. Las restricciones serán:

- **Restricciones explícitas para el problema de las palabras:**
  - $S_i = \{1, \ldots, n\}, 1 \leq i \leq m$. Es decir, cada letra tiene que pertenecer al alfabeto.
- **Restricciones implícitas para el problema de las palabras:**
  - No puede haber dos letras iguales en la misma palabra.
  - Las soluciones son, por tanto, variaciones sin repetición de n elementos tomados de m en m, es decir $\frac{n!}{(n-m)!}$. Si consideramos palabras de 5 letras sobre un alfabeto de 27 letras distintas, el número de posibilidades se reduce de $27^5 = 14.348.907$ a $\frac{27!}{22!} = 9.687.600$.

El espacio de soluciones potenciales a explorar estará formado por el conjunto de tuplas que satisfacen las restricciones explícitas. Este espacio, se puede estructurar como un *árbol de exploración*, donde en cada nivel se toma la decisión sobre la etapa correspondiente.

Se denomina *nodo de estado* a cualquier nodo del árbol de exploración que satisfaga **las restricciones explícitas**, y corresponde a una tupla parcial o una tupla completa. Los *nodos solución* serán los correspondientes a las tuplas completas que además satisfagan **las restricciones implícitas**.

Un elemento adicional imprescindible es la *función de poda* o *test de factibilidad*, que permite determinar cuándo una solución parcial puede conducir a una solución satisfactoria. De tal manera que si un *nodo* no satisface la función de poda es inútil continuar la búsqueda por esa rama del árbol. La *función de poda* permite pues reducir la búsqueda en el árbol de exploración.

Una vez definido el árbol de exploración, el algoritmo realizará un recorrido del árbol en cierto orden, hasta encontrar la primera solución. El mismo algoritmo, con ligeras modificaciones, se podrá utilizar para encontrar todas las soluciones, o una solución óptima.

El árbol de exploración no se construye de manera explícita, es decir no se almacena en memoria, sino que se va construyendo de manera implícita conforme avanza la búsqueda por medio de llamadas recursivas. Durante el proceso, para cada nodo se generarán los nodos sucesores (estados alcanzables tomando una determinada decisión correspondiente a la siguiente etapa). En el proceso de generación habrá distintos tipos de nodos:

Nodos vivos Aquellos para los cuales aún no se han generado todos sus hijos. Todavía pueden expandirse.

Nodo en expansión Aquel para el cual se están generando sus hijos.

Nodos muertos Aquellos que no hay que seguir explorando porque, o bien no han superado el test de factibilidad, o bien se han explorado insatisfactoriamente todos sus hijos.

El recorrido del árbol de exploración se realiza en profundidad. Cuando se llega a un nodo muerto, hay que deshacer la última decisión tomada y optar por otra alternativa (*vuelta atrás*). La forma más sencilla de expresar este retroceso es mediante un algoritmo recursivo, ya que la vuelta atrás se consigue automáticamente haciendo terminar la llamada recursiva y volviendo a aquella que la invocó.

El coste de los algoritmos de *vuelta atrás* en el caso peor es del orden del tamaño del árbol de exploración, ya que en el peor de los casos nos veremos obligados a recorrer exhaustivamente todas las posibilidades. El espacio de soluciones potenciales suele ser, como mínimo, exponencial en el tamaño de la entrada. La efectividad de la *vuelta atrás* va a depender decisivamente de las funciones de poda que se utilicen, ya que si son adecuadas permitirán reducir considerablemente el número de nodos explorados.

Se podrían realizar búsquedas más inteligentes haciendo que en cada momento se explore el nodo más prometedor, utilizando para ello algún tipo de heurística que permita ordenar los nodos en un tipo de datos denominado *cola de prioridad*. Esta estrategia da lugar al esquema conocido como de **ramificación y poda** . Las colas de prioridad y el esquema de ramificación y poda se estudiará el próximo curso.

2.1. Esquema básico de la *vuelta atrás*

El esquema de *Vuelta atrás* en pseudocódigo es el siguiente:

```
vueltaAtras (Tupla & sol,  int k) {
   prepararRecorridoNivel(k);
   while(!ultimoHijoNivel(k)){
       sol[k] = siguienteHijoNivel(k);
       if (esValida(sol, k)){
            if (esSolucion(sol, k))
                tratarSolucion(sol);
            else
                vueltaAtras(sol, k + 1);
       }
   }
}
```

El tipo de la solución `sol` es una tupla de cierto tipo específico para cada problema. En ella se va acumulando la solución. La variable k es la que determina en qué nivel del árbol de exploración estamos.

El método `prepararRecorridoNivel` genera los candidatos para ampliar la solución en la siguiente etapa y depende del problema en concreto. En el cuerpo de la función, iteramos a lo largo de todas las posibles soluciones candidatas, dadas por la función: `siguienteHijoNivel` hasta la última candidata, dada por la función: `ultimoHijoNivel`. Para cada solución candidata, ampliamos la solución con el nuevo valor y comprobamos si satisface las restricciones implícitas/explícitas con la función booleana `esValida`. Esta función implementa la *función de factibilidad* que presentábamos más arriba.

En el caso de que la solución parcial sea válida tenemos dos posibilidades: o bien hemos alcanzado el final de la búsqueda, por lo que ya podemos mostrar la solución final, o bien continuamos nuestra búsqueda mediante la llamada recursiva.

Este esquema encontrará todas las soluciones del problema. Si quisiéramos que sólo encontrara una solución bastaría con añadir una variable booleana *éxito* que haga finalizar los bucles cuando se encuentra la primera solución.

2.2. Resolución del problema de las palabras

Veamos cómo se aplica el esquema al problema de las palabras que hemos descrito más arriba. El esquema presentado tendrá pequeñas variaciones en cada problema, hay que tomarlo como una referencia y no como un patrón estricto. En el caso de las palabras la función recursiva en C++ podría ser como sigue.

```
void variaciones(int solucion[], int k, int n, int m){
    for(int letra = 0; letra < n; letra++){
        solucion[k] = letra;
        if(esValida(solucion, k)){
            if(esSolucion(k, m)){
                tratarSolucion(solucion,m);
            }
            else{
               variaciones(solucion, k + 1, n, m);
            }
        }
    }
}
```

La función `prepararRecorridoNivel` no existe explícitamente ya que en este caso la iteración es muy simple y se aplica sobre valores numéricos del 0 al $n - 1$.

La función `esValida` es la encargada de comprobar que la nueva letra que hemos incorporado a la solución no está repetida con las anteriores. La podríamos escribir como sigue:

```
bool esValida(int solucion[], int k) {
  int i == 0;
  while(i < k && solucion[i] != solucion[k]) i++;
  return i == k;
}
```

La función `esSolucion` simplemente comprueba que hemos colocado todas las letras.

```
bool esSolucion(int k, int m){
    return k == (m - 1);
}
```

Cuando encontramos una solución, podemos simplemente escribirla por la salida estándar:

```
void tratarSolucion(int solucion[], int m){
    cout << "Solucion: ";
    for(int i = 0; i < m; i++)
        cout << solucion[i] << " ";
    cout << endl;
}
```

Por último, la llamada inicial será de la siguiente manera:

```
void variaciones(int n, int m){
    int solucion[m];
    variaciones(solucion, 0, n, m);
}

int main()
{
    variaciones(27,5);
    return 0;
}
```

La función `esValida` recorre la lista de letras y comprueba si la última letra insertada en la solución coincide con alguna de las anteriores. Esta operación tiene un coste lineal en función de la entrada. La función se ejecuta muchas veces por lo que el coste de la función repercute negativamente en la ejecución del programa. Podríamos ahorrarnos este coste utilizando lo que se conoce como la técnica de *marcaje*.

3. Vuelta atrás con marcaje

La técnica de *marcaje* consiste en guardar cierta información que ayuda a decidir si una solución parcial es válida o no. La información del *marcaje* se pasa en cada llamada recursiva. Por lo tanto, reduce el coste computacional a cambio de utilizar más memoria. El esquema de *vuelta atrás* con marcaje es el siguiente:

```
vueltaAtrasConMarcaje (Tupla & sol, int k, Marca & marcas) {
   prepararRecorridoNivel(k);
   while(!ultimoHijoNivel(k)){
       sol[k] = siguienteHijoNivel(k);
       if (esValida(sol, k, marcas)){
            if (esSolucion(sol, k))
                tratarSolucion(sol);
            else{
                marcar(marcas, sol, k);
                vueltaAtrasConMarcaje(sol, k + 1, marcas);
                desmarcar(marcas, sol, k);
            }
       }
   }
}
```

El tipo *Marca* depende de cada problema concreto.

Normalmente, *desmarcaremos* después de la llamada recursiva para devolver las marcas a su estado anterior a la llamada. En algunos casos no es necesario *desmarcar*, como ocurre en el ejemplo 6.

En el ejemplo de las palabras, podemos utilizar marcaje para evitar el bucle de comprobación cada vez que aumentamos la solución. Para ello utilizamos un vector de booleanos de tamaño el número de letras del alfabeto considerado. Cada posición del vector indica si la letra correspondiente ha sido ya utilizada. De esta forma, las operaciones de *marcar* una letra como ya utilizada y consultar si una letra ya está utilizada tienen ambas coste constante. La solución con marcaje quedaría como sigue.

```
void variaciones(int solucion[], int k, int n, int m, bool marcas[]){
    for(int letra = 0; letra < n; letra++){
        if(!marcas[letra]){
            solucion[k] = letra;
            if(k == m - 1){
                tratarSolucion(solucion,m);
            }
            else{
               marcas[letra] = true; //marcar
               variaciones(solucion, k + 1, n, m, marcas);
               marcas[letra] = false; //desmarcar
             }
         }
    }
}
```

Observar que al finalizar la llamada recursiva se procede a desmarcar la letra correspondiente a esa llamada recursiva. Los parámetros de entrada `solucion` y `marcas`, modifican su valor de una llamada recursiva a otra. No se utiliza en este caso el paso de parámetros por referencia de forma explicita, debido al tratamiento dado por C++ a los vectores, como punteros a la primera posición de memoria. Esto hace que las modificaciones realizadas en un vector queden reflejas en las sucesivas llamadas recursivas.

4. Ejemplo: problema de las n-reinas

El problema de las 8 reinas consiste en colocar 8 reinas en un tablero de ajedrez sin que se amenacen. Dos reinas se amenazan si comparten la misma fila, columna o diagonal.

El número de soluciones potenciales o espacio de búsqueda teórico es de $\binom{64}{8} = 4.426.165.368$, que representan todas las combinaciones en las que podemos poner 8 reinas en un tablero de 64 casillas. Realizar una búsqueda exhaustiva en este espacio es impracticable, siendo todavía más problemático si ampliamos el tamaño de la entrada, por ejemplo tratando de colocar 11 reinas en un tablero de 11×11 (743.595.781.824 soluciones potenciales).

Si evaluamos la estructura de la solución vemos que no podemos poner dos reinas en la misma fila. Trataremos de colocar una reina **en cada fila del tablero**, de forma que no se amenacen. De esta manera, toda solución del problema se puede representar como una 8-tupla $(x_1, .., x_8)$ en la que x_i representa la columna en la que se coloca la reina que está en la fila i-ésima del tablero.

La tupla $(4, 7, 3, 8, 2, 5, 1, 6)$ representa el siguiente tablero.

			X				
						X	
		X					
							X
	X						
				X			
X							
					X		

Las restricciones serán:

- **Restricciones explícitas para el problema de las reinas:**

- $S_i = \{1, \ldots, 8\}, 1 \leq i \leq 8$. Es decir, cada columna tiene que estar dentro del tablero.
- Esta representación hace que el espacio de soluciones potenciales se reduzca a 8^8 posibilidades (16.777.216 valores).

▪ **Restricciones implícitas para el problema de las reinas:**

- No puede haber dos reinas en la misma columna, ni en la misma diagonal.
- Al no poder haber dos reinas en la misma columna, se deduce que todas las soluciones son permutaciones de la 8-tupla $(1, 2, 3, 4, 5, 6, 7, 8)$. Por lo tanto el espacio de soluciones potenciales se reduce a 8! (40.320 valores diferentes).

Una función recursiva en C++ que resuelve el problema para n reinas es.

```
void nReinas(int solucion[], int k, int n){
    for(int i = 0; i < n; i++){
        solucion[k] = i;
        if (esValida(solucion, k)){
            if(k == n - 1){
                tratarSolucion(k, n);
            }
            else{
                nReinas(solucion, k + 1, n);
            }
        }
    }
}
```

La función `esValida` es la encargada de comprobar que la nueva reina que hemos incorporado a la solución no amenaza a las anteriores. La podríamos escribir como sigue:

```
bool esValida(int solucion[], int k) {
    bool correcto = true;
    int i = 0;
    while (i < k && correcto){
        if(solucion[i] == solucion[k]
            || abs(solucion[k] - solucion[i]) == k - i))
            correcto = false;
    }
    return correcto;
}
```

Comprobamos que la nueva reina no está en la misma columna que las anteriores, `solucion[i] == solucion[k]` y que no comparten diagonal, `abs(solucion[k] - solucion[i]) == k - i)`. Obviamente nunca puede estar en la misma fila por la manera en que construimos la solución.

Cuando encontramos una solución, podemos simplemente escribirla por la salida estándar:

```
void tratarSolucion(int solucion[], int n){
    cout << "Solucion: ";
    for(int i = 0; i < n; i++)
        cout << solucion[i] << " ";
    cout << endl;
}
```

Por último, la llamada inicial será de la siguiente manera:

```
void nReinas(int n){
    int solucion[n];
    nReinas(solucion, 0, n);
}

int main()
{
    nReinas(8);
    return 0;
}
```

Podríamos reducir el espacio de búsqueda teniendo en cuenta que las soluciones son simétricas. Si hay una solución colocando la primera reina en la casilla 2 también la habrá colocando la reina inicial en la casilla $n-2$. Así pues, podríamos lanzar el método recursivo para el primer nivel sólo para las casillas menores de $n/2$, reduciendo el espacio de búsqueda a la mitad.

La función `esValida` recorre la lista de reinas y comprueba si la última reina insertada en la solución amenaza a las anteriores. Esta operación tiene un coste lineal en función de la entrada. Se puede reducir este coste con la técnica de marcaje.

Podemos utilizar como *marca* una estructura de datos *tablero*. Cada vez que insertamos una reina en la solución *se marcan* las casillas amenazadas por la nueva reina. Para comprobar si una nueva reina está amenazada basta con comprobar si esta marcada la casilla correspondiente del tablero. El problema de esta solución es que marcar en el tablero las casilla que amenaza la nueva reina supone un coste lineal, lo cual sigue resultando poco eficiente.

La solución está en no utilizar un tablero completo para realizar el marcaje, sino dos vectores: uno con las columnas amenazadas y otro con las diagonales amenazadas. El vector de columnas tendrá tamaño n, sin embargo, existen muchas más diagonales. Para poder resolver el problema se deben de numerar las diagonales y considerar cada posición del vector como una de ellas. La modificación y acceso a ambos vectores tendrá coste constante. El problema esta resuelto en detalle en el capítulo 14 del libro (Martí Oliet et al., 2004).

5. Ejemplo de búsqueda de una sola solución: Dominó

Se trata de encontrar una cadena circular de fichas de dominó. Teniendo en cuenta:

- Cada cadena tiene que utilizar las 28 fichas diferentes que contiene el juego de dominó.
- No se puede repetir ninguna ficha.
- Las cadenas tienen que ser correctas, es decir, cada ficha tiene que ser compatible con la siguiente y la cadena tiene que cerrar (el valor de un extremo de la última ficha tiene que coincidir con el otro extremo de la primera). Por ejemplo: $6|3 \rightarrow 3|4 \rightarrow 4|1 \rightarrow 1|0 \rightarrow .. \rightarrow 5|6$ es una cadena correcta.

La solución va a ser una tupla de 29 valores $(x_0, .., x_{28})$ cada x_i es un número del 0 al 6. Es decir, en la solución no guardaremos las fichas, sino los valores de uno de los extremos. En el ejemplo de más arriba la solución tendría la siguiente forma: $(6, 3, 4, 1, 0, .., 5)$. No necesitamos guardar explícitamente los dos extremos de las fichas, ya que cada ficha tiene

que coincidir con la siguiente. Se declara una posición más en la tupla para poder realizar la comprobación de que la cadena es cerrada.

Para evitar fichas repetidas utilizaremos una matriz (7×7) donde marcaremos las fichas usadas. Hay que tener en cuenta que si marcamos la casilla (i, j), habrá que marcar la simétrica (j, i), ya que se trata de la misma ficha.

El problema pide que se encuentre una sola solución, no todas las que existan, así que vamos a tener que abortar la búsqueda en el momento que aparezca la primera. Para ello utilizaremos una variable de control `exito`.

La función principal será:

```
void domino(int sol[], int k, int n, bool marcas[][], bool &exito){
    int i = 0;
    int m = (n * n + n) /2;
    while (i < n && !exito){
        if(!marcas[sol[k-1]][i]){
            sol[k] = i;
            if(k == m) {
              if (sol[0] == sol[k]){
                tratarSolucion(sol,m);
                exito = true;
              }
            }
            else {
                marcas[sol[k-1]][i] = true;
                marcas[i][sol[k-1]] = true;
                domino(sol, k + 1, n, marcas, exito);
                marcas[sol[k-1]][i] = false;
                marcas[i][sol[k-1]] = false;
           }
        }
        i++;
    }
}
```

donde n es el número de valores posibles de las fichas, en nuestro caso $n = 7$ ya que las fichas toman valores del 0 al 6, la matriz de marcas se declara de dimensión $n \times n$ y el vector solución es de tamaño $(n \times n + n)/2 + 1$.

Por tradición en el juego del dominó, siempre se empieza por el doble 6, así que pondremos los dos primeros valores como 6. Podríamos haber utilizado cualquier par de valores.

La llamada principal del programa será de la siguiente manera:

```
int main()
{
    int sol[29];
    bool marcas[7][7];
    for(int i = 0; i < 7; i++)
        for(int j = 0; j < 7; j++)
            marcas[i][j] = false;

    sol[0] = 6;
    sol[1] = 6;
    marcas[6][6] = true;
    bool exito = false;
    domino(sol, 2, 7, marcas, exito);
    return 0;
```

```
}
```

6. Ejemplo que no necesita desmarcar: El laberinto

Podemos representar un laberinto como una matriz booleana L de $n \times n$ de tal manera que se puede pasar por las casillas con *true*. Las casillas con *false* representan los muros infranqueables. Solo nos podemos desplazar a las cuatro casillas adyacentes: arriba, abajo, izquierda y derecha. Se pide escribir un algoritmo que encuentre la salida, asumiendo que la entrada al laberinto está en la casilla $(0, 0)$ y la salida en la $(n-1, n-1)$.

Las posibles soluciones son todas las listas de posiciones del laberinto, de longitud n^2 como máximo, dado que en el peor caso visitaremos todas y cada una de las casillas. Representamos la solución como un vector `solucion`$[n^2]$ de casillas, de tal manera que cada una de las casillas es transitable y cada casilla es adyacente a su siguiente.

Cada nodo del árbol de búsqueda tendrá 4 hijos correspondientes a cada una de las posibles continuaciones (arriba, abajo, izquierda y derecha).

Para controlar que no pasamos dos veces por la misma casilla mantendremos un marcador en forma de matriz `marcas[n][n]`, marcando aquellas casillas por las que hemos pasado. Veamos como queda el algoritmo principal:

```
void laberinto(bool lab[][], casilla solucion[], int k, int n, bool marcas[][]){
    for(int dir = 0; dir < 4; dir++){
        solucion[k] = sigDireccion(dir, solucion[k-1]);
        if(esValida(lab, solucion[k], n, marcas)){
            if(esSolucion(solucion[k],n)){
                tratarSolucion(solucion, k);
            }
            else{
                // marcar
                marcas[solucion[k].fila][solucion[k].columna] = true;
                laberinto(lab, solucion, k + 1, n, marcas);
                //desmarcar
                //marcas[solucion[k].fila][solucion[k].columna] = false;
            }
        }
    }

}
```

En este caso , el algoritmo no *desmarca* las posiciones utilizadas ya que al volver de la llamada recursiva o hemos encontrado la solución o un callejón sin salida, por lo que no nos interesa volver a considerarla.

Para saber si una casilla es válida basta con preguntar que esté dentro de los límites del tablero, que no sea un *muro* y que no esté marcada.

```
bool esValida(bool lab[][], casilla c, int n, const bool marcas[][]){
    return c.fila >= 0 && c.columna >= 0 && c.fila < n && c.columna < n
    && lab[c.fila][c.columna] && !marcas[c.fila][c.columna];
}
```

Para enumerar las cuatro direcciones posibles de movimiento echamos mano de la siguiente función auxiliar:

```
casilla sigDireccion(int dir, casilla pos){
    switch (dir) {
        case 0:
            ++ pos.columna;
            break;
        case 1:
            ++ pos.fila;
            break;
        case 2:
            -- pos.columna;
            break;
        case 3:
            -- pos.fila;
            break;
        default:
            break;
    }
    return pos;
}
```

El problema asume que la salida está en la posición $(n-1, n-1)$, así que la comprobación de la solución será sencilla:

```
bool esSolucion(casilla pos, int n){
    return pos.fila == n - 1 && pos.columna == n - 1;
}
```

Por último, la llamada inicial será:

```
int main()
{
    const int N = ...;
    bool Laberinto[N][N];
    InicializarLab(Laberinto,N);
    bool marcas[N][N];
    for(int i = 0; i < N; i ++)
        for(int j = 0; j < N; j ++)
            marcas[i][j] = false;

    casilla sol[N*N];
    sol[0].fila = 0;
    sol[0].columna = 0;
    laberinto(Laberinto,sol, 1, N, marcas);
    return 0;
}
```

Vemos que fijamos el primer valor de la solución como $(0, 0)$, ya que esa es la posición de la salida. La función `Inicializarlab` inicializa el laberinto poniendo las paredes.

Este algoritmo no tiene porque encontrar el camino óptimo, encuentra simplemente una solución; Sin embargo, *vuelta atrás* se puede utilizar para problemas de optimización como se explica más adelante.

7. Optimización

En muchos casos necesitamos obtener la mejor solución entre todas las soluciones posibles.

Para tratar este tipo de problemas de optimización tenemos que modificar el esquema anterior de forma que se almacene la mejor solución hasta el momento. Así, a la hora de tratar una nueva solución se comparará con la que tenemos almacenada. En general, guardaremos la mejor solución junto con su valor.

7.1. Ejemplo: Problema del viajante

El problema del viajante, en inglés *Travelling Salesman Problem* es uno de los problemas de optimización más estudiados a lo largo de la historia de la computación. A priori parece tener una solución sencilla pero en la práctica encontrar soluciones óptimas es muy complejo computacionalmente.

El problema se puede enunciar de la siguiente manera. Sean N ciudades de un territorio. El objetivo es encontrar una ruta que, comenzando y terminando en una ciudad concreta, pase una sola vez por cada una de las ciudades intermedias y minimice la distancia recorrida por el viajante.

Básicamente hay que encontrar una permutación del conjunto de ciudades $P = \{c_0, .., c_N\}$ tal que la suma de las distancias entre una ciudad y la siguiente sea mínimo, es decir $\sum_{i:0..N-1} d[c_i, c_{(i+1)\,\%N}]$ sea mínimo. La distancia d entre dos ciudades viene dada en una matriz . El tamaño del árbol de soluciones es $(N-1)!$, ya que corresponde a todas las posibles permutaciones, teniendo en cuenta que el principio y el final es el mismo.

El problema tiene considerables aplicaciones prácticas, aparte de las más evidentes en áreas de logística de transporte. Por ejemplo, en robótica, permite resolver problemas de fabricación para minimizar el número de desplazamientos al realizar una serie de perforaciones en una plancha o en un circuito impreso. También puede ser utilizado en control y operativa optimizada de semáforos, etc.

Veamos cómo es la solución para este problema.

```
void viajante(int distancias[][], int solucion[], int coste, int k, int n,
      int solucionMejor[], int &costeMejor){
    for(int i = 0; i < n; i++){
        solucion[k] = i;
        coste += distancias[solucion[k-1]][solucion[k]];
        if(esValida(solucion, k)){
            if(esSolucion(solucion,k)){
                if(coste < costeMejor){
                    costeMejor = coste;
                    copiarSolucion(solucion, solucionMejor);
                }
            }
            else viajante(distancias, solucion, coste, k+1, n,
                          solucionMejor, costeMejor);
        }
        coste -= distancias[solucion[k-1]][solucion[k]];
    }
}
```

Llevamos dos parámetros en los que guardamos la mejor solución hasta el momento y el coste de la misma (*solucionMejor* y *costeMejor*). Cuando encontremos una solución

comprobaremos si es mejor que la que tenemos almacenada, en caso positivo la consideraremos como la nueva mejor solución. El algoritmo de *vuelta atrás* garantiza que al final de la búsqueda la solución encontrada tiene el coste óptimo. El parámetro *coste* acumula el coste de la solución parcial, que se va calculando de forma incremental en cada llamada recursiva, evitando realizar el cálculo en cada llamada. Al finalizar la llamada recursiva debe actualizar su valor, igual que se hace en la técnica de marcaje.

- Podríamos mejorar la búsqueda del camino óptimo utilizando una estimación optimista para realizar podas tempranas. La idea es prever cuál es el mínimo coste de lo que falta por recorrer. Si ese coste, sumado al que llevamos acumulado, supera la mejor solución encontrada hasta el momento, entonces podemos abandonar la búsqueda porque estamos seguros de que ningún camino va a mejorar la solución. Utilizamos una estimación optimista, cuanto menos optimista mejor, para saber si es posible mejorar el resultado.

- Para calcular una estimación optimista es suficiente con encontrar la mínima distancia entre cualquier par de ciudades, y considerar que todos los desplazamientos van a tener esa distancia. Se pueden realizar cálculos más ajustados del coste del camino que queda por recorrer, pero hay que tener en cuenta que el cálculo debe ser sencillo para no aumentar el coste del algoritmo.

Así tendremos que añadir antes de realizar la llamada recursiva, el cálculo del coste estimado. Se calcula considerando que el resto de desplazamientos tienen un coste mínimo. El `costeMinimo` se puede calcular muy fácilmente recorriendo la matriz de distancias; consideramos que se ha procesado al principio de la ejecución y que lo tenemos almacenado en un parámetro. Solo realizaremos la recursión si la solución se puede mejorar.

```
int costeEstimado = coste + (n - k + 1) * costeMinimo;
if(costeEstimado < costeMejor)
   viajante(distancias, solucion, coste, k+1, n, solucionMejor, costeMejor);
```

Para mejorar el coste de la función `esValida` se puede utilizar la técnica de marcaje. En este caso, se declara un vector `usado` de n componentes, donde el valor de cada componente indica si la ciudad correspondiente ha sido visitada.

7.2. Ejemplo: Problema de la mochila

Otro problema clásico de optimización es el problema de la mochila. La idea es que tenemos n objetos con valor $(v_0, .., v_{n-1})$ y peso $(p_0, .., p_{n-1})$, y tenemos que determinar qué objetos transportar en la mochila sin superar su capacidad m (en peso) para maximizar el valor del contenido de la mochila.

Para resolver este problema en cada nivel del árbol de búsqueda vamos a decidir si cogemos o no el i-ésimo elemento. Así pues la solución será una tupla $(b_0, .., b_{n-1})$ de booleanos.

Se consideran las siguientes restricciones:

- Deberemos maximizar el valor de lo que llevamos $\sum_{i:0..n-1} b_i v_i$.

- El peso no debe exceder el máximo permitido $\sum_{i:0..n-1} b_i p_i \leq m$.

Una posible solución es:

```
void mochila(float P[], float V[], bool solucion[], int k, int n, int m,
     float peso, float beneficio, int solucionMejor[], int &valorMejor){
    // hijo izquierdo [cogemos el objeto]
    solucion[k] = true;
    peso = peso + P[k];
    beneficio = beneficio + V[k];
    if(peso <= m){
       if(k == n-1){
          if(valorMejor < beneficio){
             valorMejor = beneficio;
             copiarSolucion(solucion, solucionMejor);
          }
       }
       else{
          mochila(P,V,solucion, k+1,n, m, peso, beneficio,
                  solucionMejor, valorMejor);
       }

    }
    peso = peso - P[k];           //desmarcamos peso y beneficio
    beneficio = beneficio - V[k];
    // hijo derecho [no cogemos el objeto]
    solucion[k] = false;
    if(k == n-1){
       if(valorMejor v beneficio)){
          valorMejor = beneficio;
          copiarSolucion(solucion, solucionMejor);
       }
    }
    else{
       mochila(P,V,solucion,  k+1,n, m,peso, beneficio,
               solucionMejor, valorMejor);
    }

}
```

En el hijo de la derecha no tendremos que comprobar si excedemos el peso total, ya que al descartar el objeto no aumentamos el peso acumulado.

Este problema da pie a optimización, ya que podemos calcular una cota superior (una evaluación optimista) del beneficio que podemos obtener con lo que nos resta para rellenar la mochila. Para calcularla, organizamos inicialmente los objetos en los vectores `P` y `V` de manera que estén ordenados por "densidad de valor" decreciente. Llamamos densidad de valor al cociente v_i/p_i. De esta forma, cogeremos primero los objetos que tienen más valor por unidad de peso. Si en un cierto nodo, la tupla parcial es $(b_0, .., b_k)$ y hemos alcanzado un beneficio `beneficio` y un peso `peso`, estimamos el beneficio optimista como la suma de `beneficio` más el beneficio conseguido cogiendo los objetos que quepan en el orden indicado desde el $k+1$ al $n-1$. Si se llega a un objeto j que ya no cabe, se fracciona y se suma el valor de la fracción que quepa. Esta forma de proceder se llama solución *voraz* al problema de la mochila con posible fraccionamiento de objetos y produce siempre una cota superior a cualquier solución donde no se permita faccionamiento.

La poda se produce si el beneficio optimista es **menor** que el beneficio de la mejor solución alcanzada hasta el momento.

8. Para terminar...

Terminamos el tema con la solución al problema del coloreado de mapas planteado en la primera sección. Como en los casos anteriores, lo primero que nos debemos preguntar es sobre la forma de la solución y del árbol de búsqueda.

En este caso:

- Si el mapa M tiene m países, numerados 0 a $m-1$, entonces la solución va a ser una tupla $(x_0, .., x_{m-1})$ donde x_i es el color asignado al i-ésimo pais.
- Cada elemento x_i de la tupla pertenecerá al conjunto $\{0, .., n-1\}$ de colores válidos.

Cada vez que vayamos a pintar un país de un color tendremos que comprobar que ninguno de los adyacentes está pintado con el mismo color. En este caso es más sencillo hacer la comprobación cada vez que coloreamos un vértice en lugar de utilizar *marcaje*.

Así pues la función principal será:

```
void colorear(int solucion[], int k, int n, int m){
    for(int c = 0; c < n; c++){
        solucion[k] = c;
        if(esValida(solucion, k)){
            if(esSolucion(k,m)){
                tratarSolucion(solucion,m);
            }
            else{
               colorear(solucion, k + 1, n,m);
            }
        }
    }
}
```

La función `esValida` será la encargada de comprobar si la solución parcial no vulnera la restricciones de que dos países limítrofes compartan color; para implementarla asumiremos que tenemos acceso a cierto objeto `M` donde se guarda el mapa, y que tiene un método que dice si dos países son fronterizos.

```
bool esValida(int solucion[], int k) {
   int i = 0; bool valida = true;
   while (i < k && valida) {
         if (M.hayFrontera(i, k)  && solucion[k] == solucion[i])
         valida =  false;
         i++;
   }
   return valida;
}
```

A la hora de hacer la llamada inicial podemos asignar al primer país del mapa un color arbitrario.

Notas bibliográficas

Gran parte del contenido de este capítulo está basado en el capítulo correspondiente de (Martí Oliet et al., 2013).

Ejercicios

1. Vamos a realizar una modificación en el problema de las n-reinas. Asumimos que cada casilla del tablero de ajedrez tiene un número asignado. Los número se asignan consecutivamente en orden de izquierda a derecha y de arriba abajo.

   Para un tablero de 4×4 los números serán:

   1 2 3 4

   5 6 7 8

   9 10 11 12

   13 14 15 16

   Se pide hacer algoritmo que resuelva el problema de las n-reinas, de tal manera que la suma de los números de las posiciones donde se colocan las reinas sea máxima.

2. Se pide escribir un programa que decida si se puede escribir una expresión aritmética dados 5 números que sea igual a 23.

   - Las expresiones pueden utilizar los sumas, restas y multiplicaciones.

   El programa debe determinar si es posible o no realizar la expresión, y devolver la expresión oportuna en el caso afirmativo.

3. A partir de un tablero de ajedrez de $n \times n$ posiciones y de un caballo situado en una posición arbitraria (i, j) se pide diseñar un algoritmo que determine un secuencia de movimientos que visite todas las casillas del tablero una sola vez. El último movimiento debe devolver el caballo a su posición inicial.

4. Un restaurante de lujo cuenta con n mesas de tamaños diferentes $[t_0, ..., t_{n-1}]$. El maître recibe cada noche una lista de reservas. No todas ellas se pueden satisfacer, así que debe encontrar una distribución óptima que permita cenar a un número máximo de comensales. Cada grupo está compuesto por g_i miembros.

   - Cuando un grupo no cabe en una sola mesa, el maître puede decidir unir varias mesas.
   - Los grupos deben cenar completos. El maître no puede dejar a parte del grupo sin cenar.
   - El restaurante solo tiene un turno. Las mesas no se pueden reutilizar a lo largo de la noche.

5. Optimizar el problema de la mochila descrito más arriba. Para ello podemos utilizar la siguiente idea para realizar una estimación optimista:

   - Consideramos que tenemos los objetos ordenados, los que tienen mayor valor por unidad de peso son los primeros.
   - Asumimos que los objetos son fraccionables; que podemos coger una determinada parte de cada uno.

   Con estas dos consideraciones podemos realizar una algoritmo de estimación optimista muy sencillo, que siempre escoja una solución óptima, que nunca podrá ser superada por objetos nos fraccionables. Este algoritmo se puede utilizar como cota superior.

6. [Examen Junio, 2012] Implementar una función que encuentre la forma *más rápida* de viajar desde una casilla de salida hasta una casilla de llegada de una rejilla. Cada casilla de la rejilla está etiquetada con una letra, de forma que en el camino desde la salida hacia la llegada se debe ir formando (de forma cíclica) una palabra dada. Desde una celda se puede ir a cualquiera de las cuatro celdas adyacentes.

   Como ejemplo, a continuación aparece la forma más corta de salir de una rejilla de 5×8 en la que el punto de salida está situado en la posición (0, 4) y hay que llegar a la posición (7, 0) y la palabra que hay que ir formando por el camino es `EDA`.

	0	1	2	3	4	5	6	7
0	M	D	A	A	E	E	D	A
1	A	E	E	D	D	A	N	D
2	D	B	D	X	E	D	A	E
3	E	A	E	D	A	R	T	D
4	E	D	M	P	L	E	D	A

   Para la implementación puedes suponer la existencia de dos variables globales `N` y `M` que determinan el tamaño de la rejilla, así como la información de la propia rejilla,

```
char lab[N][M];
```

   También puedes suponer la existencia de una variable global que contiene la palabra a utilizar,

```
Lista<char> palabra;
```

   Las función recibirá el punto origen y el punto destino y deberá determinar la forma más rápida de viajar de uno a otro (si es que esto es posible), dando las direcciones que hay que ir cogiendo (en el caso del ejemplo será E, N, E, E, E, etc.). Si necesitas parámetros adicionales, añádelos indicando sus valores iniciales.

7. [Examen Septiembre, 2012] Los ferrys son barcos que sirven para trasladar coches de una orilla a otra de un río. Los coches esperan en fila al ferry y cuando éste llega un operario les va dejando entrar.

   En nuestro caso el ferry tiene espacio para dos filas de coches (la fila de babor y la fila de estribor) cada una de N metros. El operario conoce de antemano la longitud de cada uno de los coches que están esperando a entrar en él y debe, según van llegando, mandarles a la fila de babor o a la fila de estribor, de forma que el ferry quede lo más cargado posible. Ten en cuenta que los coches deben entrar en el ferry **en el orden en que aparecen** en la fila de espera, por lo que en el momento en que un coche ya no entra en el ferry porque no hay hueco suficiente para él, no puede entrar ningún otro coche que esté detrás aunque sea más pequeño y sí hubiera hueco para él.

   Implementa una función que reciba la capacidad del ferry en metros y la colección con las longitudes de los coches que están esperando (puedes utilizar el TAD que mejor

te venga) y devuelva la colocación óptima de los coches, entendiendo ésta como la secuencia de filas en las que se van metiendo los coches. Supón la existencia de los símbolos `BABOR` y `ESTRIBOR`.

Ejemplo: Si el ferry tiene `5` metros de longitud y en la fila tenemos coches con longitudes (del primero al último) `2.5`, `3`, `1`, `1`, `1.5`, `0.7` y `0.8`, una solución óptima es `[BABOR, ESTRIBOR, ESTRIBOR, ESTRIBOR, BABOR, BABOR]`.

Bibliografía

Y así, del mucho leer y del poco dormir, se le secó el celebro de manera que vino a perder el juicio.

Miguel de Cervantes Saavedra

Brassard, G. y Bratley, P. *Fundamentos de Algoritmia*. Prentice Hall, 1997.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. y Stein, C. *Introduction to Algorithms*. MIT Press, 2nd edición, 2001.

Martí Oliet, N., Segura Díaz, C. M. y Verdejo López, J. A. *Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios*. Ibergarceta Publicaciones, 2012.

Martí Oliet, N., Ortega Mallén, Y. y Verdejo López, J. A. *Estructuras y datos y métodos algorítmicos: 213 Ejercicios resueltos*. Ibergarceta Publicaciones, 2013.

Peña, R. *Diseño de Programas: Formalismo y Abstracción*. Tercera edición. Pearson Prentice-Hall, 2005.

Rodriguez Artalejo, M., González Calero, P. A. y Gómez Martín, M. A. *Estructuras de datos: un enfoque moderno*. Editorial Complutense, 2011.

Stroustrup, B. *The C++ Programming Language, 3rd Edition*. Addison-Wesley, 1998. ISBN 0201889544.