El esquema de "Vuelta Atrás"

Miguel Valero Espada es el autor principal de este tema.
Modificado por Isabel Pita. Curso 12-13

Facultad de Informática - UCM

13 de enero de 2014

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.
- Primera aproximación: generar todas las posibles maneras de colorear el mapa y después desechar aquellas en que dos países fronterizos tienen el mismo color.
- Coste: variaciones con repetición de n elementos tomados de m en m: $VR_n^m = n^m$.
- Mapa de España: 17 Comunidades, 50 provincias.

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

Técnica de *vuelta atrás* o *Backtracking*

- Consiste en realizar un recorrido sistemático del universo de soluciones, descartando aquellas *no válidas* para reducir el espacio de búsqueda.
- Se generan soluciones parciales, comprobando en cada paso si la solución que se está construyendo puede conducir a una solución satisfactoria (solución *prometedora*).
- Si una solución parcial no es prometedora, la búsqueda por esta rama se aborta volviendo a una solución parcial prometedora.
- Para volver a una solución anterior se deshacen las decisiones previas: *vuelta atrás*.
- Se puede encontrar:
  - una solución cualquiera,
  - el conjunto de todas las soluciones o
  - *la mejor* (en un sentido preciso) de todas las soluciones válidas.

Ejemplo

- 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 es la misma solución *abc* que *bac*.
- El número de soluciones potenciales o *espacio de búsqueda* es de n^m: variaciones con repetición de n letras tomadas de m en m.
- Esquema *vuelta atrás*.
  - Una solución se representa como una tupla $(x_1, .., x_m)$, donde x_i representa la letra que se coloca en el lugar i-ésimo de la palabra.
  - En cada paso se coloca una nueva letra y se comprueba que no esté repetida con las anteriores.
  - Si la última letra colocada no está repetida, la búsqueda de la solución continúa a partir de ella. Si no, se abortan todas las búsquedas que partan de esa tupla parcial.

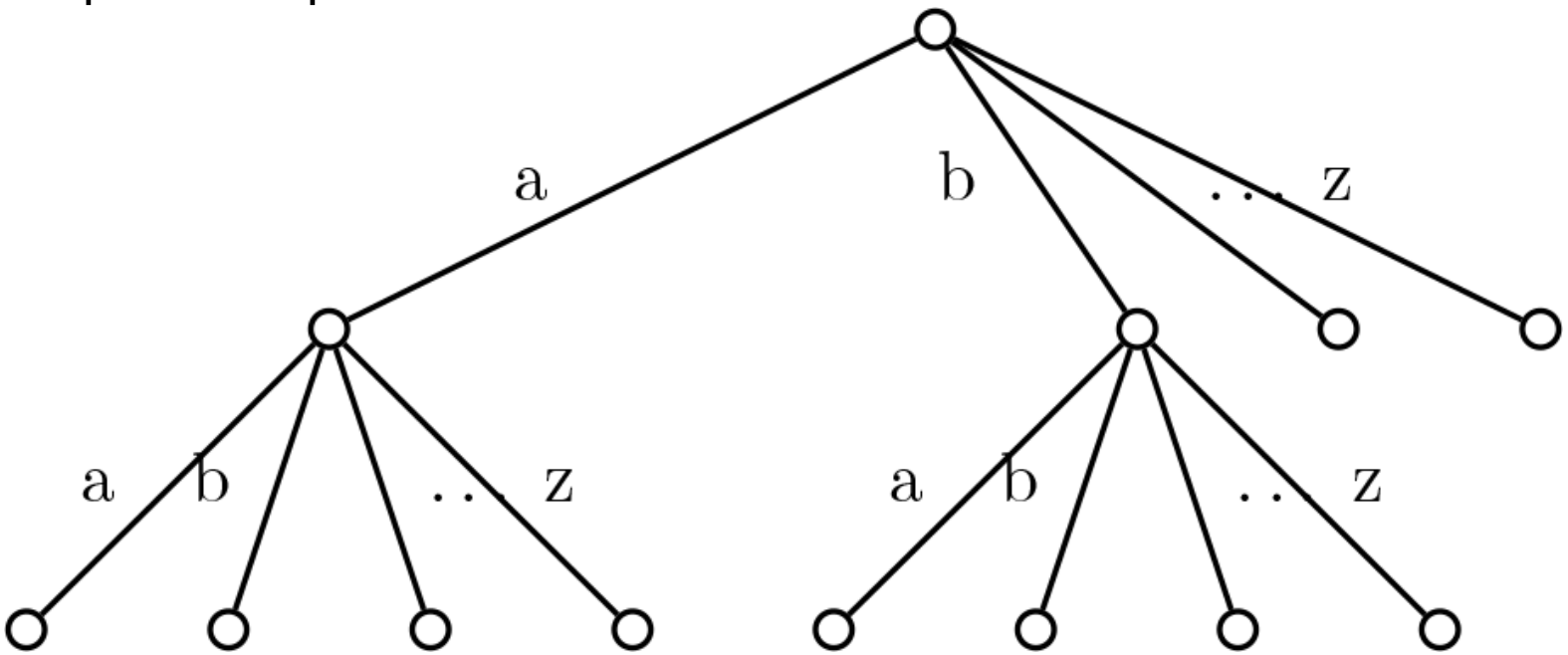
Restricciones para los posibles valores de una tupla

Se denomina *función criterio* a la función que debe satisfacer, minimizar o maximizar la solución. Viene dada por el enunciado del problema.

- Restricciones explícitas,
  - Indican el conjunto de valores, S_i, que pueden tomar cada una de las componentes de la tupla solución.
  - En el ejemplo cada letra debe pertenecer al alfabeto.
- Restricciones implícitas,
  - Relaciones que se han de establecer entre las componentes de la tupla solución para satisfacer la *función criterio* .
  - En el ejemplo: no puede haber dos letras iguales en la misma palabra.

Espacio de soluciones

El espacio de búsqueda está formado por el conjunto de tuplas que satisfacen las restricciones explícitas. Se puede estructurar como un *árbol de exploración*, en cada nivel se toma la decisión sobre la etapa correspondiente.



- *Nodo de estado*, cualquier nodo del árbol de exploración que satisfaga **las restricciones explícitas**, y corresponde a una tupla parcial o una tupla completa.
- *Nodos solución*, correspondientes a las tuplas completas que además satisfagan **las restricciones implícitas**.
- *Función de poda o test de factibilidad*, permite determinar cuándo una solución parcial puede conducir a una solución satisfactoria.
- 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 algoritmo realiza un recorrido del árbol de exploración en cierto orden, hasta encontrar la primera solución.
- El árbol de exploración no se construye de manera explícita, es decir, no se almacena en memoria.
- El recorrido se realiza 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).
- Cuando se llega a un nodo muerto, hay que deshacer la última decisión tomada y optar por otra alternativa (*vuelta atrás*). Se consigue automáticamente haciendo terminar la llamada recursiva y volviendo a aquella que la invocó
- El mismo algoritmo, con ligeras modificaciones, se podrá utilizar para encontrar todas las soluciones, o una solución óptima.

Coste de los algoritmos de vuelta atrás

- En el caso peor el coste es del orden del tamaño del árbol de exploración.
- El espacio de búsqueda 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. Esta estrategia da lugar al esquema conocido como de ramificación y poda.

Pseudocódigo del esquema de *Vuelta atrás*

Encontrar todas las soluciones a un problema:

```
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);
       }
   }
}
```

Para encontrar una sóla solución basta con añadir una variable booleana *éxito* que haga finalizar los bucles cuando se encuentra la primera solución.

Solución del problema de las palabras

```
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);
            }
        }
    }
}
```

- `prepararRecorridoNivel` no existe porque la iteración es muy simple y se aplica sobre valores numéricos del 0 al $n-1$.
- `esValida` comprueba que la nueva letra que hemos incorporado a la solución no está repetida con las anteriores.

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

- `esSolucion` comprueba que hemos colocado todas las letras.

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

- `tratarSolucion` escribe la solución en la salida estándar:

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

- Llamada inicial :

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

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

- La función `esValida` tiene un coste lineal en función de la entrada. La función se ejecuta para cada llamada recursiva. Podríamos ahorrarnos este coste utilizando lo que se conoce como la técnica de *marcaje*.

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.
- Reduce el coste computacional a cambio de utilizar más memoria.
- 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 del laberinto.

Esquema de *vuelta atrás* con marcaje.

```
VAConMarcaje (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);
                VAConMarcaje(sol, k + 1, marcas);
                desmarcar(marcas, sol, k);
            }
       }
   }
}
```

Generación de palabras con marcaje

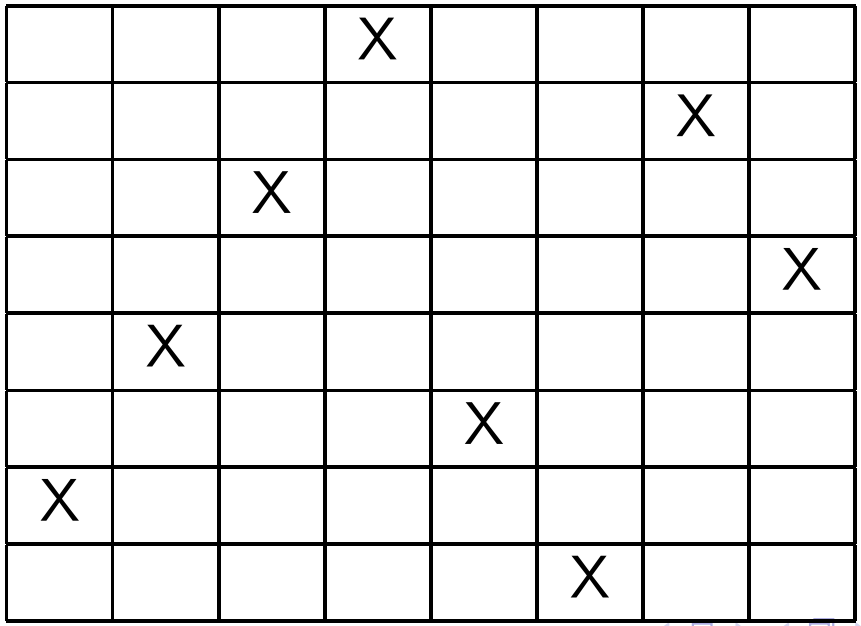
- Utilizamos un vector de booleanos de tamaño el número de letras del alfabeto considerado.
- Cada posición del vector indica si la letra correspondiente ha sido ya utilizada.
- Las operaciones de *marcar* una letra como ya utilizada y consultar si una letra ya está utilizada tienen ambas coste constante.
- Los parámetros `sol` y `marcas`, modifican su valor de una llamada recursiva a otra debido al tratamiento que C++ da a los vectores.

- Algoritmo:

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

Problema de las n-reinas

- Colocar 8 reinas en un tablero de ajedrez sin que se amenacen. Dos reinas se amenazan si comparten la misma fila, columna o diagonal.
- Espacio de búsqueda: $\binom{64}{8} = 4{,}426{,}165{,}368$ soluciones.
- Solución: 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.
- Ejemplo: La tupla $(4, 7, 3, 8, 2, 5, 1, 6)$ representa el tablero.



Las restricciones son:

- 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).

Algoritmo.

```
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);
            }
        }
    }
}
```

- `esValida` comprueba que la nueva reina no amenaza a las anteriores.

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

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

- `tratarSolucion` escribe la solución en la salida:

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

- Llamada inicial:

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

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

Optimizaciones al problema de las reinas

- 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 $n-2$. Reducir el espacio de búsqueda a la mitad ejecutando el método recursivo para el primer nivel sólo para las casillas menores de $n/2$.

Optimizaciones

- `esValida` tiene un coste lineal en función de la entrada. Utilizar la técnica de marcaje.
  - Primera aproximación: Utilizar como *marca* una estructura de datos *tablero*.
    - Al añadir una reina *se marcan* las casillas amenazadas por la nueva reina.
    - Para comprobar si una nueva reina está amenazada se comprueba si esta marcada la casilla correspondiente del tablero.
    - Problema: marcar en el tablero las casilla que amenaza la nueva reina supone un coste lineal
  - Segunda aproximación: utilizar dos vectores: uno con las columnas amenazadas y otro con las diagonales amenazadas.
    - El vector de columnas tiene tamaño n,
    - El nḿuero de diagonales es: $4n - 2$.
    - La modificación y acceso a ambos vectores tiene coste constante.
    - El problema esta resuelto en detalle en el capítulo 14 del libro (Martí Oliet et al., 2013).

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.

- Solución: tupla de 29 valores $(x_0, .., x_{28})$ cada x_i es un número del 0 al 6, que representa el valor de uno de los extremos de la ficha.
- 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. Al marcar la casilla (i, j), hay que marcar la simétrica (j, i), ya que se trata de la misma ficha.
- El problema pide que se encuentre una sola solución. Utilizaremos una variable de control `exito` para abortar la búsqueda cuando se encuentre la primera solución.

```
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++;  }}
```

- Llamada inicial:

```
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;
}
```

- El número de valores posibles de las fichas es $n = 6$. La matriz de marcas se declara de dimensión $n \times n$ y el vector solución es de tamaño $(n \times n + n)/2 + 1$.
- Empezamos por el doble 6. Se podría haber utilizado cualquier par de valores.

Ejemplo que no necesita desmarcar: El laberinto

- Representación del laberinto:
  - Matriz booleana L de $n \times n$.
  - Sólo se puede pasar por las casillas a *true*.
  - Desplazamientos: arriba, abajo, izquierda y derecha.
- Objetivo: Encontrar la salida, asumiendo que la entrada al laberinto está en la casilla $(0, 0)$ y la salida en la $(n-1, n-1)$.
- Espacio de búsqueda: listas de posiciones del laberinto, de longitud n^2 como máximo.
- Solución: vector `sol`$[n^2]$ de casillas, cada casillas es transitable y adyacente a su siguiente.
- Cada nodo del árbol de exploración tiene 4 hijos por cada posible paso: arriba, abajo, izquierda y derecha.
- Marcaje: matriz `marcas[n][n]`, indican las casillas por las que hemos pasado.

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

No *desmarca* porque 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.

- Una casilla es válida si está dentro de los límites del tablero, no es un *muro* y no está marcada.

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

- Cálculo de la siguiente posición:

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

- `esSolucion` comprueba si está en la posición $(n-1, n-1)$.

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

- Llamada inicial:

```
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].col = 0;
    laberinto(Laberinto,sol, 1, N, marcas);
    return 0; }
```

- Este algoritmo encuentra una solución, no tiene porque ser la óptima.

Optimización

- Para obtener la *mejor* solución tenemos que almacenar la mejor solución hasta el momento. Cada nueva solución se compara con la que tenemos almacenada.
- Problema del viajante (*Travelling Salesman Problem*)
  Sean N ciudades. 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.

- Hay que encontrar una permutación del conjunto de ciudades $P = \{c_0, .., c_N\}$ tal que: $\sum_{i:0..N-1} d[c_i, c_{(i+1)\,\%N}]$ sea mínimo.
- La distancia d entre dos ciudades viene dada en una matriz .
- El tamaño del árbol de exploración es $(N-1)!$, ya que corresponde a todas las posibles permutaciones, teniendo en cuenta que el principio y el final es el mismo.
- Aplicaciones prácticas:
  - área de logística de transporte.
  - Robótica, minimizar el número de desplazamientos al realizar una serie de perforaciones en una plancha o en un circuito impreso.
  - Control y operativa optimizada de semáforos, etc.

```
void viajante(int d[][], int sol[], int coste, int k,
int n,int solMejor[], int &cMejor) {
    for(int i = 0; i < n; i++){
        sol[k] = i;
        coste += d[sol[k-1]][sol[k]];
        if(esValida(sol, k)){
            if(esSolucion(sol,k)) {
                if(coste < cMejor) {
                    cMejor = coste;
                    copiarSolucion(sol, solMejor);
                }
            }
            else viajante(d, sol, coste, k+1, n,
                          solMejor, cMejor);
        }
        coste -= d[sol[k-1]][sol[k]];
    }
}
```

- Mejorar la búsqueda del camino óptimo utilizando una estimación optimista para realizar podas tempranas.
- Idea: prever 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 $\Rightarrow$ abandonar la búsqueda.
- Cálculo de una estimación optimista: 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.

- Añadir antes de realizar la llamada recursiva, el cálculo del coste estimado.

```
int costeEst = coste + (n - k + 1) * costeMin;
if(costeEst < cMejor)
   viajante(d,sol, coste, k+1, n, solMejor, cMejor);
```

- El `costeMinimo` se calcula 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.
- Para mejorar el coste de la función `esValida` utilizar la técnica de marcaje. Se declara un vector `usado` de n componentes, donde el valor de cada componente indica si la ciudad correspondiente ha sido visitada.

Problema de la mochila

- Tenemos n objetos con valor $(v_0, .., v_{n-1})$ y peso $(p_0, .., p_{n-1})$, y queremos determinar qué objetos transportar en la mochila sin superar su capacidad m (en peso) para maximizar el valor del contenido de la mochila.
- La solución es una tupla $(b_0, .., b_{n-1})$ de booleanos, cada b_1 indica si se coge el i-ésimo elemento.
- Restricciones:
  - Debemos maximizar el valor de lo que llevamos $\sum_{i:0..n-1} b_i v_i$.
  - El peso no debe exceder el máximo permitido $\sum_{i:0..n-1} b_i p_i \leq m$.

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

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

Optimización problema de la mochila

- Cáculo de una cota superior (una evaluación optimista) del beneficio que podemos obtener con lo que nos resta para rellenar la mochila.
  - 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.
  - Cogeremos primero los objetos que tienen más valor por unidad de peso. Si al decidir sobre el objeto k hemos alcanzado un beneficio b y un peso p, estimamos el beneficio optimista como la suma de b 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. (Solución *voraz*, 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.

Coloreado de mapas

- 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*.

```
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);
            }
        }
    }
}
```

`esValida` comprueba si la solución parcial cumple que dos países limítrofes no compartan color; Asumimos 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 sol[], int k) {
int i = 0; bool valida = true;
while (i < k && valida) {
      if (M.hayFrontera(i, k)  && sol[k] == sol[i])
valida =  false;
      i++;
}
return valida;
}
```

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