Análisis y diseño de algoritmos

6. Vuelta atrás

Juan Morales García,
Victor M. Sánchez Cartagena,
Jose Luis Verdú Mas

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

12 de enero de 2025

Índice

Índice

Ejemplo Introductorio

El problema de la mochila (general)

Dados:

- n objetos con valores v_i y pesos w_i
- una mochila que solo aguanta un peso máximo W

Seleccionar un conjunto de objetos de forma que:

- no se sobrepase el peso límite W (restricción)
- el valor transportado sea máximo (función objetivo)

- **¿Cómo obtener la solución óptima?**
  - Programación dinámica: objetos no fragmentables y pesos discretos
  - Algoritmos voraces: objetos fragmentables
- **– No podemos fragmentar los objetos y los pesos son valores reales –**

Formalización del problema

- Solución: $X = (x_1, x_2, \ldots, x_n) \quad x_i \in \{0, 1\}$
- Restricciones:
  - Implícitas:

$$x_i \in \begin{cases} 0 & \text{no se selecciona el objeto } i \\ 1 & \text{se selecciona el objeto } i \end{cases}$$

  - Explícitas:

$$\sum_{i=1}^{n} x_i w_i \leq W$$

- Función objetivo:

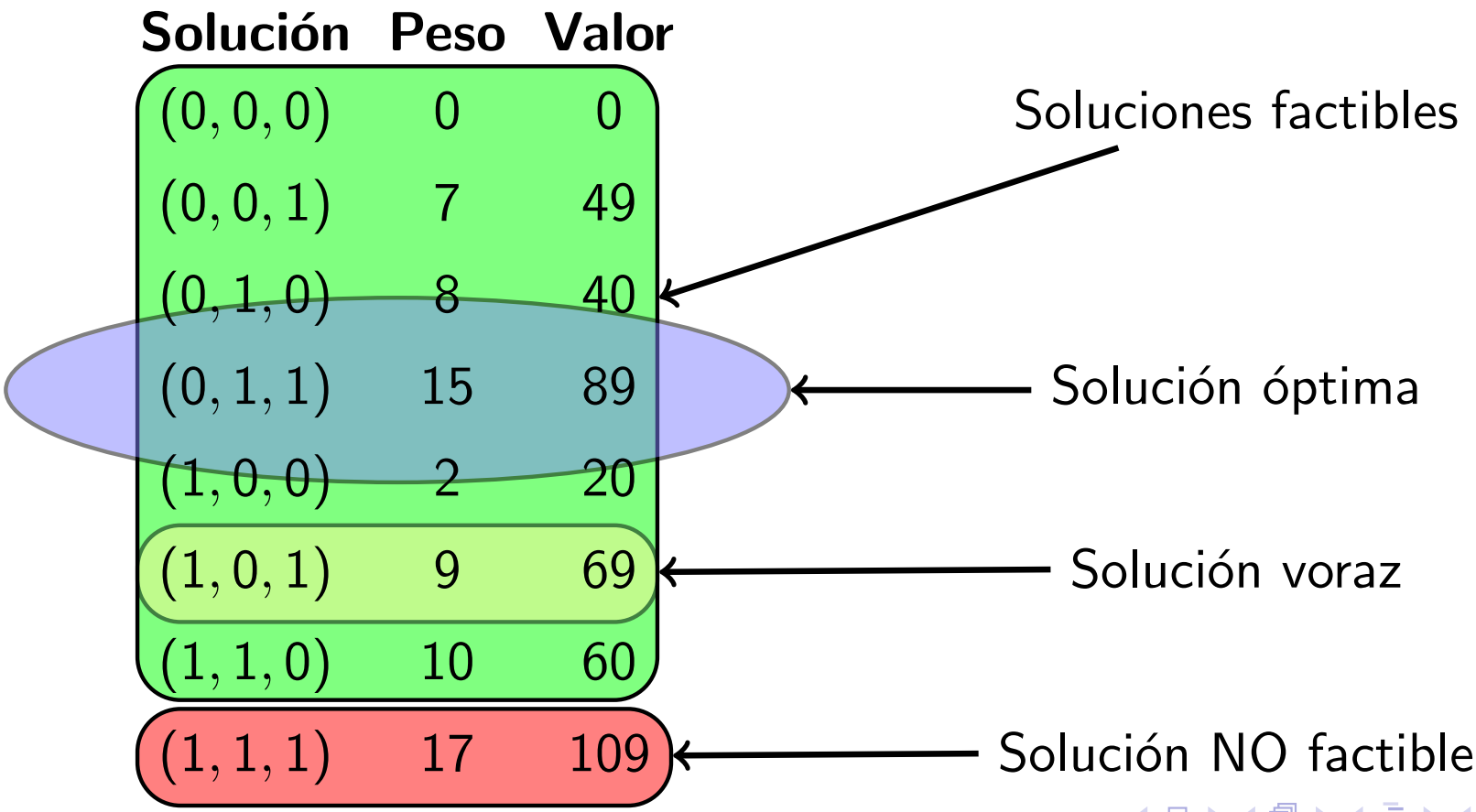
$$\text{máx} \sum_{i=1}^{n} x_i v_i$$

Tipos de soluciones

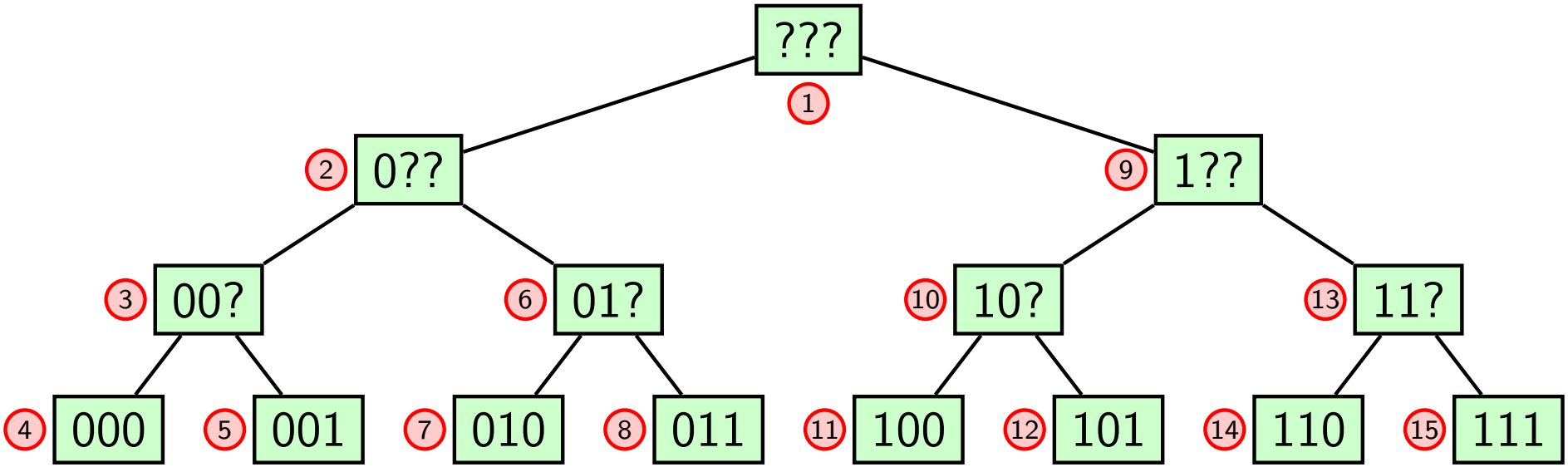
- Supongamos el siguiente ejemplo:

$$W = 16 \qquad w = (2, 8, 7) \qquad v = (20, 40, 49)$$

- Combinaciones posibles (espacio de soluciones):



Generación de todas las combinaciones



Recorrer todas las combinaciones

```
1  void comb(size_t k, vector <unsigned> &x){
2      if( k == x.size() ) {  // It is a leaf
3          cout << x << endl;  // to overload
4          return;
5      }                      // It is not a leaf
6      for ( unsigned j = 0; j < 2; j++) {
7          x[k]=j;             // New alternative
8          comb(k+1, x);                 // expand
9      }
10 }
```

Programa principal

```
1  void comb(size_t n) {
2      vector<unsigned> x(n);
3      comb( 0, x );
4  }
```

Complejidad temporal:
$\Theta(n2^n)$

¿Cómo generar solo soluciones factibles?

- Solo imprimimos las soluciones (hojas) que cumplen:

$$\sum_{i=1}^{n} x_i w_i \leq W$$

Generación de todas las soluciones factibles

Cálculo del peso de la solución x

```
1 double weight( const vector<double> &w, const vector<unsigned> &x){
2     double acc_w = 0.0;
3     for (size_t i = 0; i < w.size(); i++ ) acc_w += x[i] * w[i];
4     return acc_w;
5 }
```

Imprime todas las soluciones factibles

```
1  void feasible(
2      const vector<double> &w, double W,
3      size_t k, vector<unsigned> &x
4  ){
5      if( k == x.size() ) {   // It is a leaf
6          if( weight( w, x ) <= W )
7              cout << x << endl;
8          return;
9      }                       // It is not a leaf
10     for (unsigned j=0; j<2; j++) {
11         x[k]=j;
12         feasible(w,W,k+1,x);       // expand
13     }
14 }
```

Llamada principal

```
1 void feasible(
2     const vector<double> &w,
3     double W
4 ){
5     vector<unsigned> x(w.size());
6     feasible( w, W, 0, x );
7     return;
8 }
```

Complejidad temporal: $\Theta(n2^n)$

Búsqueda del óptimo

- Para encontrar el óptimo hay que:
  - recorrer todas las soluciones factibles
  - calcular su valor y . . .
  - . . . quedarse con el mayor de todos

Búsqueda del valor óptimo

```
1  double value( const vector<double> &v, const vector<unsigned> &x ) {
2      double r = 0.0;
3      for( size_t i = 0; i < v.size(); i++ ) r += v[i] * x[i];
4      return r;
5  }
6  void knapsack( const vector<double> &v, const vector<double> &w, double W,
7      size_t k, vector<unsigned> &x, double &best_v
8  ){
9      if( k == x.size() ) {  // It is a leaf
10         if( weight( w, x ) <= W )
11             best_v = max( best_v, value(v,x) );
12         return;
13     }
14     for (unsigned j=0; j<2; j++) {
15         x[k]=j;
16         knapsack( v, w, W, k+1, x, best_v );      // expand
17     }
18 }
19 double knapsack( const vector<double> &v, const vector<double> &w, double W){
20     vector<unsigned> x(w.size());
21     double best_v = numeric_limits<double>::lowest();
22     knapsack( v, w, W, 0, x, best_v );
23     return best_v;
24 }
```

Búsqueda de la asignación óptima

```
1  void knapsack( const vector<double> &v, const vector<double> &w, double W,
2      size_t k, vector<unsigned> &x, double &best_v, vector<unsigned> &sol
3  ){
4      if( k == x.size() ) { // It is a leaf
5          if( weight( w, x ) <= W ){
6              double current_v = value(v,x);
7              if( current_v > best_v ) {
8                  best_v = curent_v;
9                  sol = x;
10             }
11         }
12         return;
13     }
14     for (unsigned j=0; j<2; j++) {
15         x[k]=j;
16         knapsack( v, w, W, k+1, x, best_v );      // expand
17     }
18 }
19 auto knapsack( const vector<double> &v, const vector<double> &w, double W){
20     vector<unsigned> x(v.size()), sol(v.size());
21     double best_v = numeric_limits<double>::lowest();
22     knapsack( v, w, W, 0, x, best_v, sol );
23     return make_tuple( best_v, sol );
24 }
```

¿Podemos acelerar el programa?

- Sí, evitando explorar ramas que no pueden dar soluciones factibles
- Por ejemplo:
  - hemos construido una solución hasta el elemento k:

$$(x_1, x_2, \ldots, x_k, ?, \ldots, ?)$$

  - si tenemos que:

$$\sum_{i=1}^{k} x_i w_i \geq W$$

$\Rightarrow$ Ninguna expansión de esta solución puede ser factible

Eliminando ramas por peso

```
1  double weight( const vector<double> &w, size_t k, const vector<double> &x ) {
2     double r = 0.0;
3     for( size_t i = 0; i < k; i++ ) r = w[i] * x[i];
4     return r;
5  }
6  void knapsack( const vector<double> &v, const vector<double> &w, double W,
7      size_t k, vector<unsigned> &x, double &best_v
8  ){
9      if( k == x.size() ) {                                  // if it is a leaf
10         best_v = max( best_v, value(v,x));
11         return;
12     }                                                     // if it's not a leaf
13     for (unsigned j = 0; j < 2; j++ ) {
14         x[k]=j;
15         if ( weight(w, k+1, x ) <= W )                     // if it is feasible
16             knapsack( v, w, W, k+1, x, best_v );                    // expand
17   }
18 }
19 double knapsack( const vector<double> &v, const vector<double> &w, double W ) {
20     vector<unsigned> x( v.size());
21     double best_v = numeric_limits<double>::lowest();
22     knapsack( v, w, W, size_t k, const vector<unsigned> &x, best_v );
23     return best_v;
24 }
```

¿se puede mejorar?

- Nótese que el peso se puede ir calculando a medida que se rellena la solución

Calculando el peso de forma incremental

```
1  void knapsack( const vector<double> &v, const vector<double> &w, double W,
2      size_t k, vector<unsigned> &x, double acc_w, double &best_v
3  ){
4      if( k == x.size() ) {                                  // if it is a leaf
5          best_v = max( best_v, value(v,x));
6          return;
7      }
8                                                            // if it is not a leaf
9      for (unsigned j = 0; j < 2; j++ ) {
10         x[k]=j;
11         double current_w = acc_w + x[k] * w[k];             // update weight
12         if ( current_w <= W )                              // if it is feasible
13             knapsack(v, w, W, k+1, x, current_w, best_v);          // expand
14     }
15 }
16 double knapsack( const vector<double> &v, const vector<double> &w, double W ){
17     vector<unsigned> x;
18     double best_v = numeric_limits<double>::lowest();
19     knapsack( v, w, W, 0, x, 0, best_v);
20     return best_v;
21 }
```

- **Peor caso** Todos los objetos caben: $O(n\,2^n)$
- **Mejor caso** Ningún objeto cabe: $\Omega(n)$

¿Se puede mejorar?

- El valor se puede ir calculando a medida que se rellena la solución

Calculando el valor de forma incremental

```
1  void knapsack(
2      const vector<double> &v, const vector<double> &w, double W,
3      size_t k, vector<unsigned> &x, double acc_w, double acc_v, double &best_v
4  ){
5      if( k == x.size() ) {                                    // if it is a leaf
6          best_v = max( best_v, acc_v);
7          return;
8      }
9                                                               // it is not a leaf
10     for (unsigned j = 0; j < 2; j++ ) {
11         x[k]=j;
12         double current_w = acc_w + x[k] * w[k];               // update weight
13         double current_v = acc_v + x[k] * v[k];               // update value
14         if ( current_w <= W )                                // if it is feasible
15             knapsack( v, w, W, k+1, x, current_w, current_v, best_v);
16     }
17 }
18 double knapsack( const vector<double> &v, const vector<double> &w, double W ){
19     vector<unsigned> x(v.size());
20     double best_v = numeric_limits<double>::lowest();
21     knapsack( v, w, W, 0, x, 0, 0, best_v );
22     return best_v;
23 }
```

¿Podemos acelerar el programa?

- Sí, evitando explorar ramas que no pueden dar soluciones mejores que la que ya tenemos.
- Por ejemplo:
  - hemos construido una solución hasta el elemento k:

$$(x_1, x_2, \ldots, x_k, ?, \ldots, ?)$$

  - ya hemos encontrado una solución factible de valor v_b
  - si tenemos que:

$$\sum_{i=1}^{k} x_i v_i + \sum_{i=k+1}^{n} v_i \leq v_b$$

$\Rightarrow$ Ninguna expansión de esta solución puede dar un valor mayor que v_b

función para añadir el resto

```
1 double add_rest( const vector<double> &v, size_t k ){
2     double r = 0.0;
3     for( size_t i = k; i < v.size(); i++) r += v[i];
4     return r;
5 }
```

poda: cota optimista ingenua

```
1  void knapsack(
2      const vector<double> &v, const vector<double> &w, double W,
3      size_t k, vector<unsigned> &x, double acc_w, double acc_v, double &best_v
4  ){
5      if( k == x.size() ) {                                    // if it is a leaf
6          best_v = max(best_v,acc_v);
7          return;
8      }
9      for (unsigned j = 0; j < 2; j++ ) {                     // it is not a leaf
10         x[k]=j;
11         double current_w = acc_w + x[k] * w[k];               // update weight
12         double current_v = acc_v + x[k] * v[k];                // update value
13         if( current_w <= W &&                                // if is feasible ...
14             current_v + add_rest(v, k+1) > best_v     // ... and is promising
15         )
16             knapsack(v, w, W, k+1, x, current_w, current_v, best_v);
17   }
18 }
19 double knapsack( const vector<double> &v, const vector<double> &w, double W ){
20     vector<unsigned> x(v.size());
21     double best_v = numeric_limits<double>::lowest();
22     knapsack( v, w, W, 0, x, 0, 0, best_v );
23     return best_v;
24 }
```

Podas más ajustadas

- Interesa que los mecanismos de poda "actúen" lo antes posible
- Una poda más ajustada se puede obtener usando la solución voraz al problema de la mochila continua
- la solución al problema de la mochila continua es siempre mayor que la solución al problema de la mochila discreto

$\Rightarrow$ El mejor valor que se puede obtener para una solución incompleta:

$$(x_1, x_2, \ldots, x_k, ?, \ldots, ?)$$

será menor que:

$$\sum_{i=1}^{k} x_i v_i + \texttt{knapsack}_c \left(\{x_{k+1}, \ldots, x_n\}, W - \sum_{i=1}^{k} x_i w_i \right)$$

donde $\texttt{knapsack}_c(X, P)$ es la solución de la mochila continua

Poda optimista basada en la mochila continua

```
1  void knapsack(
2      const vector<double> &v, const vector<double> &w, double W,
3      size_t k, vector<unsigned> &x, double acc_w, double acc_v, double &best_v
4  ){
5      if( k == x.size() ) {                                  // if it is a leaf
6          best_v = max( acc_v, best_v);
7          return;
8      }
9      for (unsigned j = 0; j < 2; j++ ) {                   // it is not a leaf
10         x[k]=j;
11         double current_w = acc_w + x[k] * w[k];             // update weight
12         double current_v = acc_v + x[k] * v[k];              // update value
13         if( current_w <= W &&                              // if it is promising
14             current_v + knapsack_c( v, w, k+1, W - current_w) > best_v
15         )
16             knapsack( v, w, W, k+1, x, current_w, current_v, best_v);
17     }
18 }
19 double knapsack( const vector<double> &v, const vector<double> &w, double W ){
20     vector<unsigned> x(v.size());
21     double best_v = numeric_limits<double>::lowest();
22     knapsack( v, w, W, 0, x, 0, 0, best_v );
23     return best_v;
24 }
```

Partiendo de una solución cuasi–óptima

- La efectividad de la poda también puede aumentarse partiendo de una solución factible muy "buena"
- Una posibilidad es usar la solución voraz para la mochila discreta (`knapsack_d`) de la siguiente forma:

Solución óptima partiendo de un subóptimo.

```
1 double knapsack( const vector<double> &v, const vector<double> &w, double W ){
2     vector<unsigned> x(v.size());
3     double best_v = knapsack_d(v,w,W);
4     knapsack(v, w, W, 0, x, best_v );
5     return best_v;
6 }
```

Análisis empírico

- 25 muestras aleatorias de tamaño $n = 30$

Tipo de poda	**Partiendo de un subóptimo voraz**	**Llamadas recursivas realizadas (promedio)**	**Tiempo medio (segundos)**
Ninguna	–	1054.8×10^6	875.65
Completando con todos los objetos restantes	No	925.5×10^3	0.112
	Si	389.0×10^3	0.072
Completando según la sol. voraz mochila continua	No	2.3×10^3	0.034
	Si	18	0.002

Otras mejoras

También puede ser relevante:

- El orden en el que se explora los objetos
  - i.e.: ordenándolos por valor especifico
- La forma en la que se "despliega el árbol":
  - i.e.: completar la tupla primero con los unos y después con los ceros

Cambiando el orden de exploración de los objetos

```
1  double knapsack( const vector<double> &v, const vector<double> &w, double W ) {
2      vector<size_t> idx( v.size() );                          // index vector
3      iota( begin(idx), end(idx), 0 );
4
5      sort( begin(idx), end(idx),
6          [&v,&w]( size_t i, size_t j) {
7              return v[i]/w[i] > v[j]/w[j];
8          }
9      );
10
11     vector<double> s_v( v.size() ), s_w( w.size() );
12
13     for( size_t i = 0; i < v.size(); i++ ) {
14         s_v[i] = v[ idx[i] ];                                 // sorted values
15         s_w[i] = w[ idx[i] ];                                // sorted weights
16     }
17
18   vector<short> x(v.size());
19   double best_val = knapsack_d( s_v, s_w, 0, W );  // simplified version
20             // we can use a simplified version of knapsack_c in knapsack
21   knapsack( s_v, s_w, W, 0, x, 0, 0, best_val);
22
23   return best_val;
24 }
```

Cambiando el orden de explorar las decisiones (expansión)

```
1  void knapsack( const vector<double> &v, const vector<double> &w,
2                 double W, unsigned k, vector<short> &x,
3                 double weight, double value, double &best_val ){
4
5    if( k == x.size() ) {  // base case
6      best_val = value;
7      return;
8    }
9
10   for (int j = 1; j >= 0; j-- ){  // <== Reversing the order
11
12     x[k]=j;
13     double new_weight = weight + x[k] * w[k]; // updating weight
14     double new_value  = value  + x[k] * v[k]; // updating value
15
16     if( new_weight <= W &&                          // is promising
17       new_value + knapsack_c( v, w, k+1, W - new_weight ) // simplified version
18         > best_val
19     )
20       knapsack( v, w, W, k+1, x, new_weight, new_value, best_val);
21   }
22 }
```

Índice

Vuelta atrás: definición y ámbito de aplicación

- Algunos problemas sólo se pueden resolver mediante el estudio exhaustivo del conjunto de posibles soluciones al problema
- De entre todas ellas, se podrá seleccionar un subconjunto o bien, aquella que consideremos la mejor (la solución óptima)
- *Vuelta atrás* proporciona una forma sistemática de generar todas las posibles soluciones a un problema
- Generalmente se emplea en la resolución de problemas de selección u optimización en los que el conjunto de soluciones posibles es finito
- En los que se pretende encontrar una o varias soluciones que sean:
  - Factibles: que satisfagan unas restricciones y/o
  - Óptimas: optimicen una cierta función objetivo

Vuelta atrás: características

- Se trata de un recorrido sobre una estructura arbórea imaginaria
- La solución debe poder expresarse mediante una tupla de decisiones: $(x_1, x_2, \ldots, x_n) \quad x_i \in D_i$
  - Las decisiones pueden pertenecer a dominios diferentes entre sí pero estos dominios siempre serán discretos o discretizables
  - El número de soluciones posibles ha de ser finito.
- La estrategia puede proporcionar:
  - una solución factible
  - todas las soluciones factibles
  - la solución óptima al problema
  - las n mejores soluciones factibles al problema
- En la mayoría de los casos las complejidades son prohibitivas

Vuelta atrás: Eficiencia

- Cálculos incrementales
- Podas para evitar la exploración completa del espacio de soluciones:
  - Podar ramas que llevan soluciones no factibles
  - Podar ramas que solo llevan soluciones malas (optimización)
    - uso de cotas optimistas
    - inicialización con cotas pesimistas
- Cambios en el orden de exploración/expansión para conseguir rápidamente soluciones casi óptimas

Podas

- Cota optimista:
  - estima, a mejor, el mejor valor que podría alcanzarse al expandir el nodo
  - puede que no haya ninguna solución factible que alcance ese valor
  - normalmente se obtienen relajando las restricciones del problema
  - si la cota optimista de un nodo es peor que la solución en curso, se puede podar el nodo
- Cota pesimista:
  - valor seguro (lo normal es que no sea el mejor) que puede alcanzarse al expandir el nodo
  - debe corresponder con una solución factible que no tiene por qué ser la mejor
  - normalmente se obtienen mediante soluciones voraces del problema
  - se puede eliminar un nodo si su cota optimista es peor que la mejor cota pesimista
  - permite la poda aún antes de haber encontrado una solución factible
- Cuanto más ajustadas sean las cotas, más podas se producirán

Ejemplo de cotas optimista

Problema de la mochila discreta:

- Restricciones:
  - hay un peso límite W
  - los objetos no se pueden partir
- Relajaciones:
  - no hay límite $\Rightarrow$ añadir todos los objetos que quedan
  - podemos partir los objetos $\Rightarrow$ problema de la mochila continua

Vuelta atrás: Esquema general recursivo

Esquema recursivo de *backtracking* (optimización)

```
1  void backtracking( node n, solution& current_best ){
2
3      if ( is_leaf(n) ) {
4          if( is_best( solution(n), current_best ) )
5              current_best = solution(n);
6          return;
7      }
8
9      for( node a : expand(n) )
10         if( is_feasible(a) && is_promising(a) )
11             backtracking( a, current_best );
12
13     return;
14 }
15
16 solution backtracking( problem P ){
17     solution current_best = feasible_solution(n);
18     backtracking( initial_node(P), current_best);
19     return current_best;
20 }
```

Vuelta atrás: Estadísticas

```
1  void backtracking( node n, solution& current_best ){
2
3      if ( is_leaf(n) ) {
4          visited_leaf_nodes++;
5          if( is_best( solution(n), current_best ) )
6              current_best = solution(n);
7          return;
8      }
9
10     for( node a : expand(n) )
11         visited_nodes++;
12         if( is_feasible(a) ) {
13             if( is_promising(a) ) {
14                 explored_nodes++;
15                 backtracking( a, current_best );
16             } else
17                 no_promising_discarded_nodes++;
18         } else
19             no_feasible_discarded_nodes++;
20
21     return;
22 }
```

Índice

Permutaciones

Dado un entero positivo n, escribir un algoritmo que muestre todas las permutaciones de la secuencia $(0, \ldots, n-1)$

- Solución:
  - sea $X = (x_0, x_1, \ldots, x_{n-1}) \quad x_i \in \{0, 1, \ldots, n-1\}$
  - cada permutación será cada una de las reordenaciones de X
  - restricción: X no puede tener elementos repetidos
  - no hay función objetivo: se buscan todas las combinaciones factibles

Ejemplo:
Generar todas las permutaciones de $(0, 1, 2, 3)$:

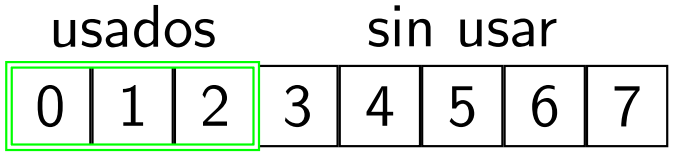
$$(0,1,2,3),(0,1,3,2),(0,2,1,3),(0,2,3,1),(0,3,1,2),(0,3,2,1),$$
$$(1,0,2,3),(1,0,3,2),(1,2,0,3),(1,2,3,0),(1,3,0,2),(1,3,2,1),$$
$$(2,0,1,3),(2,0,3,1),(2,1,0,3),(2,1,3,0),(2,3,0,1),(2,3,1,0),$$
$$(3,0,1,2),(3,0,2,1),(3,1,0,2),(3,1,2,0),(3,2,0,1),(3,2,1,0)$$

```
1  bool is_used( vector<size_t> &x, size_t k, size_t e ) {
2      for( size_t i = 0; i < k; i++ )
3          if( x[i] == e )
4              return true;
5      return false;
6  }
7
8  void permutations( size_t k, vector<size_t> &x ) {
9      if( k == x.size() ) {
10         cout << x << endl;   // Should be overloaded
11         return;
12     }
13
14     for( size_t c = 0; c < x.size(); c++ )
15         if( !is_used( x, k, c ) ) {
16             x[k] = c;
17             permutations( k+1, x );
18         }
19 }
20
21 void permutations( size_t k) {
22     vector<size_t> x(k);
23     permutations( 0, x );
24 }
```

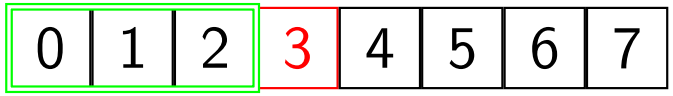
```
1  void permutations( size_t k, vector<size_t> &x, vector<bool>& is_used ){
2      if( k == x.size() ) {
3          cout << x << endl;  // Should be overloaded
4          return;
5      }
6
7      for( size_t c = 0; c < x.size(); c++ ) {
8
9          if( !is_used[c] ) {
10             x[k] = c;
11             is_used[c] = true;
12             permutations( k+1, x, is_used );
13             is_used[c] = false;
14         }
15
16     }
17 }
18
19 void permutations( size_t k ) {
20     vector<bool> is_used(k, false);
21     vector<size_t> x(k);
22     permutations( 0, x, is_used );
23 }
```

Permutaciones usando `swap`

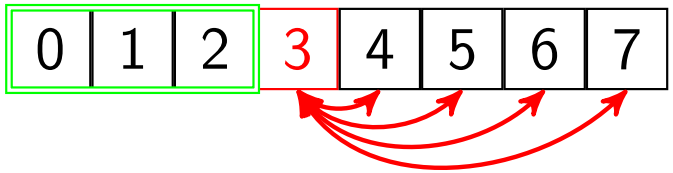
- se separa los elementos ya usados (izquierda) de los sin usar (derecha)



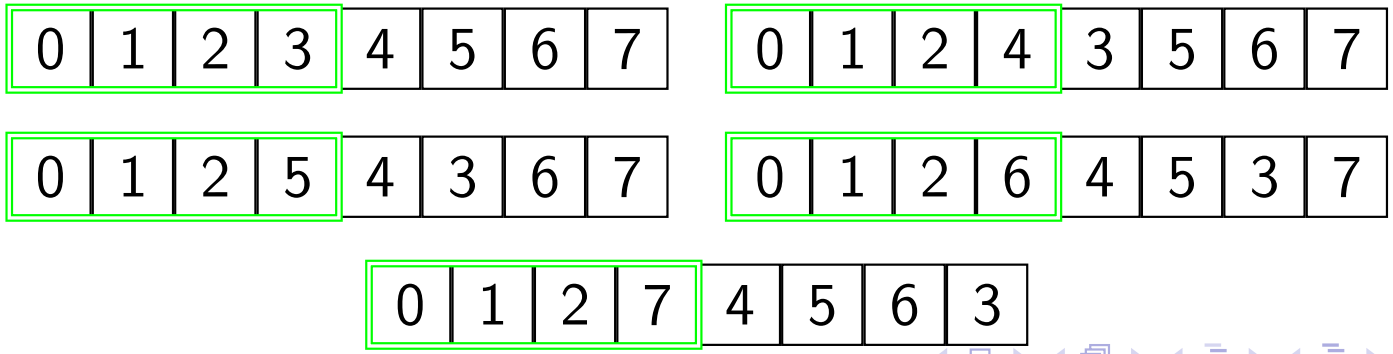
- el pivote es el elemento más a la izquierda de los no usados



- se va intercambiando el pivote con cada uno de los no usados



- para cada uno de los intercambios se integra el pivote en los usados



Restricción (con swap)

```
1  void permutations( size_t k, vector<size_t> &x ) {
2
3      if( k == x.size() ) {
4          cout << x << endl;  // Should be overloaded
5          return;
6      }
7
8      for( size_t c = k; c < x.size(); c++ ) {
9          swap(x[k],x[c]);
10         permutations( k+1, x );
11         swap(x[k],x[c]);
12     }
13 }
14
15 void permutations( size_t k ) {
16     vector<size_t> x(k);
17
18     for( size_t i = 0; i < k; i++ )
19         x[i] = i;
20
21     permutations( 0, x );
22 }
```

El viajante de comercio

Dado un grafo ponderado $g = (V, E)$ con pesos no negativos, el problema consiste en encontrar un *ciclo hamiltoniano* de mínimo coste

- Un *ciclo hamiltoniano* es un recorrido en el grafo que recorre todos los vértices sólo una vez y regresa al de partida
- El coste de un ciclo viene dado por la suma de los pesos de las aristas que lo componen

El viajante de comercio

- Expresamos la solución mediante una tupla $X = (x_1, x_2, \ldots, x_n)$ donde $x_i \in \{1, 2, \ldots, n\}$ es el vértice visitado en i-ésimo lugar
  - Asumimos que los vértices están numerados, $V = \{1, 2, \ldots, n\}, \quad n = |V|$
  - Fijamos el vértice de partida (para evitar rotaciones):
    - $x_1 = 1;\ x_i \in \{2, 3, \ldots, n\} \quad \forall i : 2 \leq i \leq n$
- Restricciones
  - No se puede visitar dos veces el mismo vértice: $i \neq j \rightarrow x_i \neq x_j \quad \forall i, j : 1 \leq i \leq n \quad 1 \leq j \leq n$
  - Existencia de arista: $\forall i : 1 \leq i < n$, $\texttt{weight}(g, x_i, x_{i+1}) \neq \infty$
  - Existencia de arista que cierra el camino: $\texttt{weight}(g, x_n, x_1) \neq \infty$
- Función objetivo:

$$\min \sum_{i=1}^{n-1} \texttt{weight}(g, x_i, x_{i+1}) + \texttt{weight}(g, x_1, x_n)$$

El viajante de comercio

```
1  unsigned round( const graph &g, const vector<size_t> &x ) {
2      size_t d = 0;
3      for( size_t i = 0; i < x.size() - 1; i++ )
4          d += g.dist( x[i], x[i+1]);
5      d += g.dist( x[x.size()-1], x[0] );
6      return d;
7  }
8
9  void solve(
10     const graph g, size_t k, vector<size_t> &x,
11     unsigned &shortest
12 ) {
13     if( k == x.size() ) {
14         shortest = min( shortest, round(g,x));
15         return;
16     }
17
18     for( size_t c = k; c < x.size(); c++ ) {
19         swap( x[k], x[c] );
20         solve( g, k+1, x, shortest );
21         swap( x[k], x[c] );
22     }
23 }
```

El viajante de comercio (con swap)

```
1  void solve(
2      const graph g, size_t k, vector<size_t> &x,
3      unsigned &shortest
4  ) {
5      if( k == x.size() ) {
6          shortest = min( shortest, round(g,x));
7          return;
8      }
9
10     for( size_t c = k; c < x.size(); c++ ) {
11         swap( x[k], x[c] );
12         solve( g, k+1, x, shortest );
13         swap( x[k], x[c] );
14     }
15 }
16
17 size_t solve( const graph &g ) {
18     size_t shortest = numeric_limits<unsigned>::max();
19     vector<size_t> x(g.num_cities());
20     for( size_t i = 0; i < x.size(); i++ )
21         x[i] = i;
22     solve( g, 1, x, shortest ); // fix the first city
23     return shortest;
24 }
```

El viajante de comercio

- La propuesta es inviable por su prohibitiva complejidad: $O(n!)$

Mejoras:

- Cálculo incremental de la vuelta
- Cota optimista:
  - Restricciones:
    - el camino ha de pasar por todas las ciudades
    - el camino ha de ser continuo
  - Relajaciones:
    - no pasar por todas las ciudades $\Rightarrow$ saltar de la última a la primera
    - ir saltando de una ciudad a otra $\Rightarrow$ árbol de recubrimiento mínimo
- Poda basada en la mejor solución hasta el momento (cota pesimista)
  - buscar una solución subóptima
    - algoritmo voraz

Las n mejores soluciones

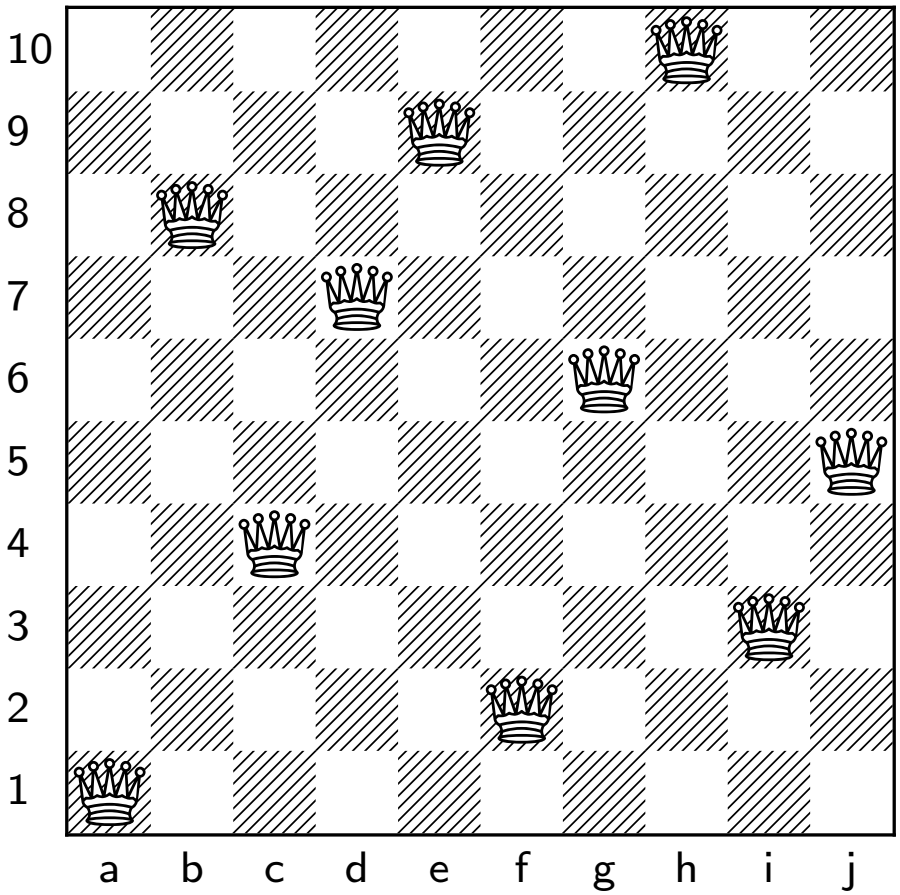
```
1  void solve_nbest(
2      const graph g,
3      size_t k,
4      vector<size_t> &x,
5      priority_queue<size_t> &pq,
6      size_t n
7  ){
8      if( k == x.size() ) {
9          size_t len = round(g,x);
10         if( pq.top() > len ) {   // access to the higher element
11             if( pq.size() == n )
12                 pq.pop();        // extract the higher element
13             pq.push(len);
14         }
15         return;
16     }
17
18     for( size_t c = k; c < x.size(); c++ ) {
19         swap(x[k],x[c]);
20         solve_nbest( g, k+1, x, pq, n );
21         swap(x[k],x[c]);  // not needed
22     }
23 }
```

Las n mejores soluciones

```
priority_queue<unsigned> solve_nbest(
    const graph &g,
    size_t n
){
    vector<size_t> x(g.num_cities());
    for( size_t i = 0; i < x.size(); i++ )
        x[i] = i;

    priority_queue<size_t> pq;
    pq.push(numeric_limits<size_t>::max());

    solve_nbest( g, 1, x, pq, n );

    return pq;
}
```

El problema de las n reinas

- En un tablero de "ajedrez" de $n \times n$ obtener todas las formas de colocar n reinas de forma que no se ataquen mutuamente (no estén ni en la misma fila, ni columna, ni diagonal).



El problema de las n reinas

Solución:

- No puede haber dos reinas en la misma fila,
  - la reina i se colocará en la fila i.
  - El problema es determinar en qué columna se colocará.
- Sea $X = (x_1, x_2, \ldots, x_n)$,
  - $x_i \in \{1, 2, \ldots, n\}$: columna en la que se coloca la reina de la fila i
- Restricciones:
  1. No puede haber dos reinas en la misma fila:
     - implícito en la forma de representar la solución.
  2. No puede haber dos reinas en la misma columna
     - X no puede tener elementos repetidos.
  3. No puede haber dos reinas en la misma diagonal:

     $\Rightarrow \; |i - j| \neq |x_i - x_j|$

El problema de las n reinas (escribir todas las soluciones)

```
1  bool feasible( vector<size_t> &x, size_t k ) {
2      for( size_t i = 0; i < k; i++ ) {
3          if( x[i] == x[k] ) return false;
4          if( x[i] < x[k] && x[k] - x[i] == k - i ) return false;
5          if( x[i] > x[k] && x[i] - x[k] == k - i ) return false;
6      }
7      return true;
8  }
9
10 void n_queens( size_t k, vector<size_t> &x ) {
11     if( k == x.size() ) {
12         cout << x << endl;  // Should be overloaded
13         return;
14     }
15     for( size_t i = 0; i < x.size(); i++ ) {
16         x[k] = i;
17         if( feasible(x, k) ) n_queens( k+1, x );
18     }
19 }
20
21 void n_queens( size_t n ) {
22     vector<size_t> x(n);
23     n_queens(0,x);
24 }
```

El problema de las *n* reinas (escribir sólo una solución)

```
1  void n_queens( size_t k, vector<size_t> &x, bool &found ) {
2
3      if( k == x.size() ) {
4          cout << x << endl;  // Should be overloaded
5          found = true;
6          return;
7      }
8
9      for( size_t i = 0; i < x.size(); i++ ) {
10         x[k] = i;
11         if( factible(x, k) )
12             n_queens( k+1, x, found );
13         if( found ) break;
14     }
15
16 }
17
18 void n_queens( size_t n ) {
19     vector<size_t> x(n);
20     bool found = false;
21
22     n_queens( 0, x, found );
23 }
```

El problema de las *n* reinas (devolver sólo una solución)

```
1  void n_queens(
2      size_t k, vector<size_t> &x, vector<size_t> &sol, bool &found
3  ) {
4      if( k == x.size() ) {
5          sol = x;
6          found = true;
7          return;
8      }
9      for( size_t i = 0; i < x.size(); i++ ) {
10         x[k] = i;
11         if( factible(x, k) )
12             n_queens( k+1, x, sol, found );
13         if( found )
14             return;
15     }
16 }
17
18 vector<size_t> n_queens( size_t n ) {
19     vector<size_t> x(n);
20     vector<size_t> sol(n);
21     bool found = false;
22     n_queens( 0, x, sol, found);
23     return sol;
24 };
```

La función compuesta mínima

Dadas dos funciones $f(x)$ y $g(x)$ y dados dos números cualesquiera x e y, encontrar la función compuesta mínima que obtiene el valor y a partir de x tras aplicaciones sucesivas e indistintas de $f(x)$ y $g(x)$

- Ejemplo: Sean $f(x) = 3x$, $g(x) = \lfloor x/2 \rfloor$, y sean $x = 3$, $y = 6$
  - Una transformación de 3 en 6 con operaciones f y g es:

$$(g \circ f \circ g \circ f \circ f \circ g)(3) = 6 \qquad \text{(5 composiciones)}$$

  - La mínima (aunque no única) es:

$$(f \circ g \circ g \circ f)(3) = 6 \qquad \text{(3 composiciones)}$$

La función compuesta mínima

Solución:

- $X = (x_1, x_2, \ldots, x_?) \quad x_i \in \{0, 1\} \begin{cases} 0 \equiv & \text{se aplica } f(x) \\ 1 \equiv & \text{se aplica } g(x) \end{cases}$
  - $(x_1, x_2, \ldots, x_k) \equiv (x_k \circ \cdots \circ x_2 \circ x_1)$
  - El tamaño de la tupla no se conoce a priori
    - Se pretende minimizar el tamaño de la tupla solución (función objetivo)
    - asumiremos un máximo de M composiciones (evitar ramas infinitas)
- Llamamos $F(X, k, x)$ al resultado de aplicar al valor x la composición representada en la tupla X hasta su posición k

$$F(X, k, x) = (x_k \circ \cdots \circ x_2 \circ x_1)(x) = x_k(\ldots x_2(x_1(x))\ldots)$$

- Restricciones:
  - $F(X, k, x) \neq F(X, i, x)\ \forall i < k$, para evitar recálculos
  - $k < M$, para evitar búsquedas infinitas
  - siempre se puede calcular $F(X, k, x)$
  - $k < v_b$, (v_b es la mejor solución actual) tupla "prometedora"

La función compuesta mínima (sin poda)

```
1  const size_t NOT_FOUND = numeric_limits<size_t>::max();
2
3  int F( const vector<short> &x, unsigned k, int init) {
4      for( unsigned i = 0; i < k; i++ )
5          if( x[i] == 0 ) init = f(init);
6          else            init = g(init);
7      return init;
8  }
9  void composition( size_t k, vector<short> &x, size_t M,
10     int first, int last, size_t &best
11 ) {
12     if( k == M ) return;              // base case
13     if( F(x, k, first) == last && k < best ) best = k; // feasible solutions
14
15     for( short i = 0; i < 2; i++ ) {
16         x[k] = i;
17         composition(k+1, x, M, first, last, best);
18     }
19 }
20 size_t composition(size_t M, int first, int last) {
21     vector<short> x(M);
22     size_t best = NOT_FOUND;
23     composition( 0, x, M, first, last, best);
24     return best;
25 }
```

La función compuesta mínima (incremental)

```
1  int F1( unsigned i, int init) {
2      if( i == 0 ) return f(init);
3      else         return g(init);
4  }
5
6  void composition( unsigned k, unsigned M,
7      int current, int last, size_t &best
8  ) {
9      if( k == M ) return;    // base case
10
11     if( current == last && k < best ) best = k;
12
13     for( short i = 0; i < 2; i++ ) {
14         int next = F1(i,current);
15         composition(k+1, M, next, last, best);
16     }
17 }
18
19 size_t composition(unsigned M, int first, int last) {
20     size_t best = NOT_FOUND;
21     composition( 0, M, first, last, best);
22
23     return best;
24 }
```

La función compuesta mínima (incremental con poda)

```
1  void composition( unsigned k, unsigned M,
2      int current, int last, size_t &best
3  ) {
4      if( k == M ) return;    // base case
5
6      if( k >= best ) return;  // bound
7
8      if( current == last ){
9          best = k;
10         return;
11     }
12
13     for( short i = 0; i < 2; i++ ) {
14         int next = F1(i,current);
15         composition(k+1, M, next, last, best);
16     }
17 }
18
19 size_t composition(unsigned M, int first, int last) {
20     size_t best = NOT_FOUND;
21     composition( 0, M, first, last, best);
22
23     return best;
24 }
```

La función compuesta mínima (con poda y memoria)

```
1  void composition( unsigned k, unsigned M,
2      int current, int last,
3      unordered_map<int, Default<size_t, NOT_FOUND>> &best
4  ) {
5      if( k == M ) return;    // base case
6
7      if( k >= best[last] ) return;  // bound
8
9      if( best[current] <= k ) return;
10     best[current] = k;
11
12     for( short i = 0; i < 2; i++ ) {
13         int next = F1(i,current);
14         composition(k+1, M, next, last, best);
15     }
16 }
17
18 size_t composition(unsigned M, int first, int last) {
19     unordered_map<int, Default<size_t, NOT_FOUND>> best;
20     composition( 0, M, first, last, best);
21     return best[last];
22 }
```

Utilidad:

Cambio del valor por defecto del operador []

```
1  template<typename T, T X>
2  class Default {
3  public:
4      Default () : val(T(X)) {}
5      Default (T const & _val) : val(_val) {}
6      operator T & () { return val; }
7      operator T const & () const { return val; }
8  private:
9      T val;
10 };
```

Potencia de las podas

Número de llamadas recursivas para encontrar el mínimo número de composiciones (12) para llegar de 1 a 11

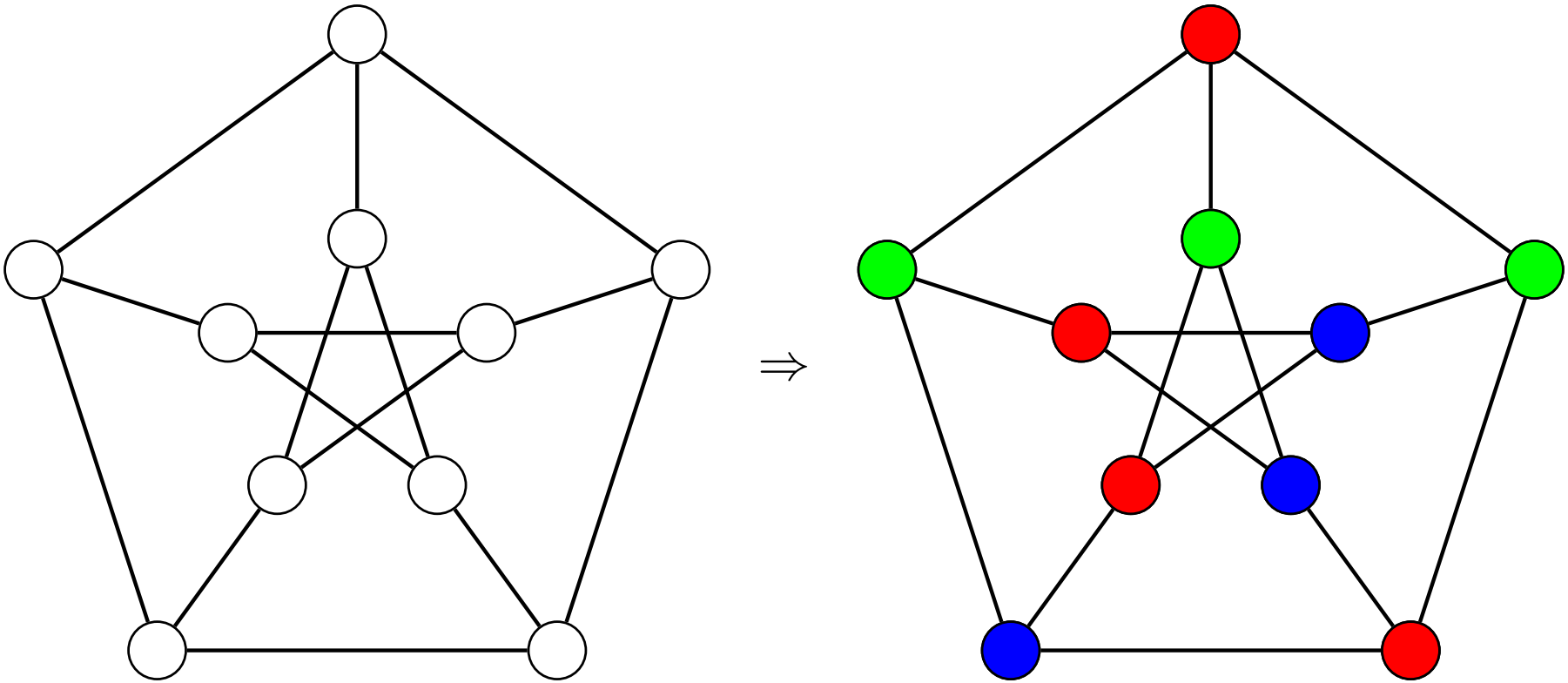
Tipo poda	$M = 15$	$M = 20$	$M = 25$
Básico	65 535	2 097 151	67 108 863
mejor en curso	9 075	40 323	1 040 259
memorización	1 055	7 243	50 669
las dos	565	2 541	16 469

Sería mejor hacer una búsqueda por niveles...

Índice

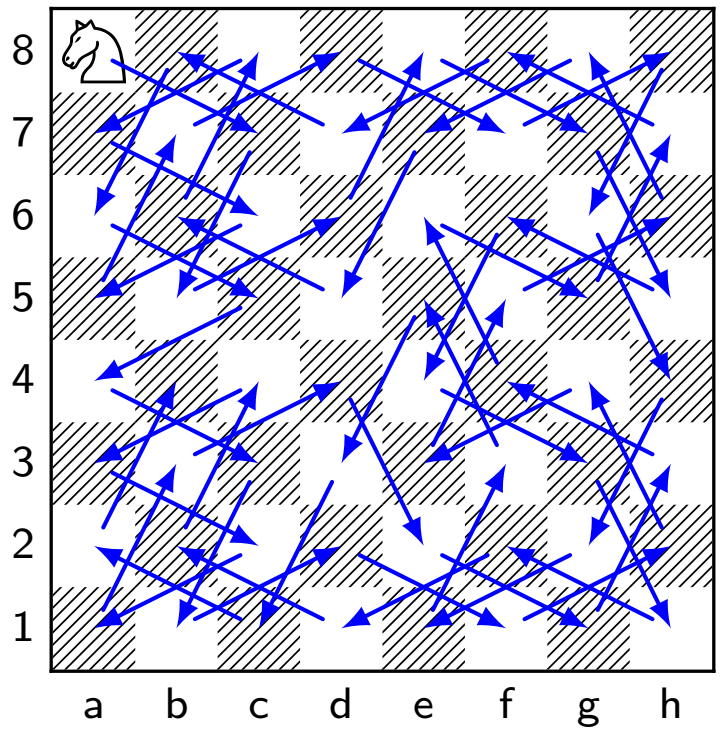
El coloreado de grafos

- Dado un grafo G, encontrar el menor número de colores con el que se pueden colorear sus vértices de forma que no haya dos vértices adyacentes con el mismo color



El recorrido del caballo de ajedrez

- Encontrar una secuencia de movimientos "legales" de un caballo de ajedrez de forma que éste pueda visitar las 64 casillas de un tablero sin repetir ninguna



El laberinto con cuatro movimientos

- Se dispone de una cuadrícula $n \times m$ de valores $\{0, 1\}$ que representa un laberinto. Un valor 0 en una casilla cualquiera de la cuadrícula indica una posición inaccesible; por el contrario, con el valor 1 se simbolizan las casillas accesibles.
- Encontrar un camino que permita ir de la posición $(1, 1)$ a la posición $(n \times m)$ con cuatro tipos de movimiento (arriba, abajo, derecha, izquierda)

La asignación de tareas

- Supongamos que disponemos de n trabajadores y n tareas. Sea $b_{ij} > 0$ el coste de asignarle el trabajo j al trabajador i.
- Una asignación de tareas puede ser expresada como una asignación de los valores 0 ó 1 a las variables x_{ij}, donde $x_{ij} = 0$ significa que al trabajador i no le han asignado la tarea j, y $x_{ij} = 1$ indica que sí
- Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador
- Dada una asignación válida, definimos el coste de dicha asignación como:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} b_{ij}$$

- Encontrar una asignación óptima, es decir, de mínimo coste

La empresa naviera

- Supongamos una empresa naviera que dispone de una flota de N buques cada uno de los cuales transporta mercancías de un valor v_i que tardan en descargarse un tiempo t_i. Solo hay un muelle de descarga y su máximo tiempo de utilización es T
- Diseñar un algoritmo que determine el orden de descarga de los buques de forma que el valor descargado sea máximo sin sobrepasar el tiempo de descarga T. (Si se elige un buque para descargarlo, es necesario que se descargue en su totalidad)

La asignación de turnos

- Estamos al comienzo del curso y los alumnos deben distribuirse en turnos de prácticas
- Para solucionar este problema se propone que valoren los turnos de práctica disponibles a los que desean ir en función de sus preferencias
- El número de alumnos es N y el de turnos disponibles es T
- Se dispone una matriz de preferencias P, $N \times T$, en la que cada alumno escribe, en su fila correspondiente, un número entero (entre 0 y T) que indica la preferencia del alumno por cada turno (0 indica la imposibilidad de asistir a ese turno; M indica máxima preferencia)
- Se dispone también de un vector C con T elementos que contiene la capacidad máxima de alumnos en cada turno
- Se pretende encontrar una solución para satisfacer el número máximo de alumnos según su orden de preferencia sin exceder la capacidad de los turnos

Sudoku

- El famoso juego del Sudoku consiste en rellenar una rejilla de 9×9 celdas dispuestas en 9 subgrupos de 3×3 celdas, con números del 1 al 9, atendiendo a la restricción de que no se debe repetir el mismo número en la misma fila, columna o subgrupo 3×3
- Además, varias celdas disponen de un valor inicial

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Sudoku sin resolver

4	2	*6*	5	*7*	1	*3*	9	*8*
8	*5*	*7*	2	*9*	3	*1*	*4*	6
1	3	*9*	*4*	6	*8*	*2*	7	*5*
9	*7*	1	*3*	*8*	*5*	6	*2*	*4*
5	4	*3*	*7*	*2*	*6*	*8*	1	9
6	*8*	2	*1*	*4*	*9*	7	*5*	*3*
7	9	*4*	*6*	3	*2*	*5*	8	*1*
2	*6*	*5*	8	*1*	4	*9*	*3*	7
3	1	*8*	9	*5*	7	*4*	6	*2*

Sudoku resuelto