Análisis y diseño de algoritmos

7. Ramificación y Poda

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)

Formalización

- 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:

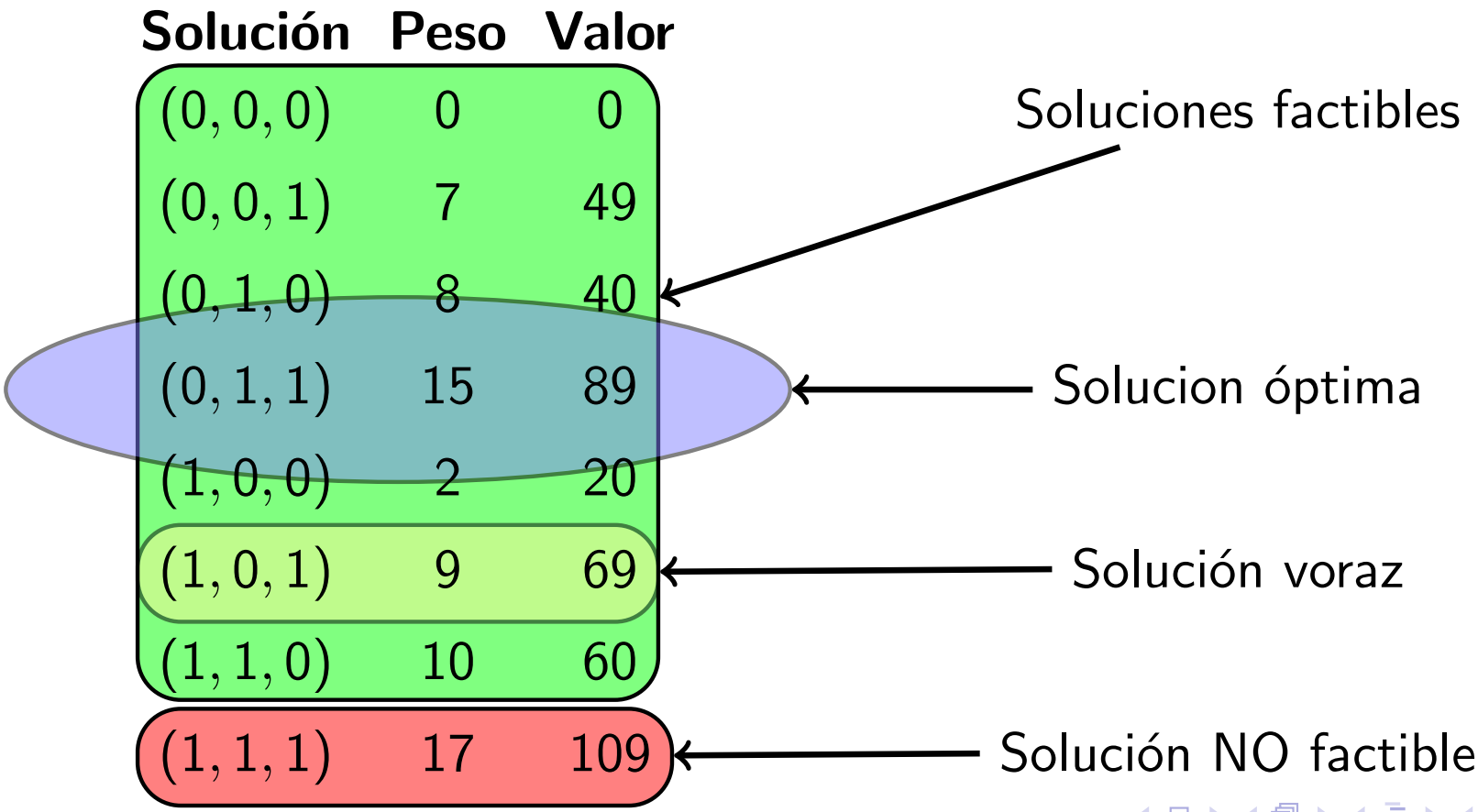
$$\text{máx} \sum_{i=1}^{n} x_i v_i$$

Tipos de soluciones

- Supongamos el siguiente ejemplo:

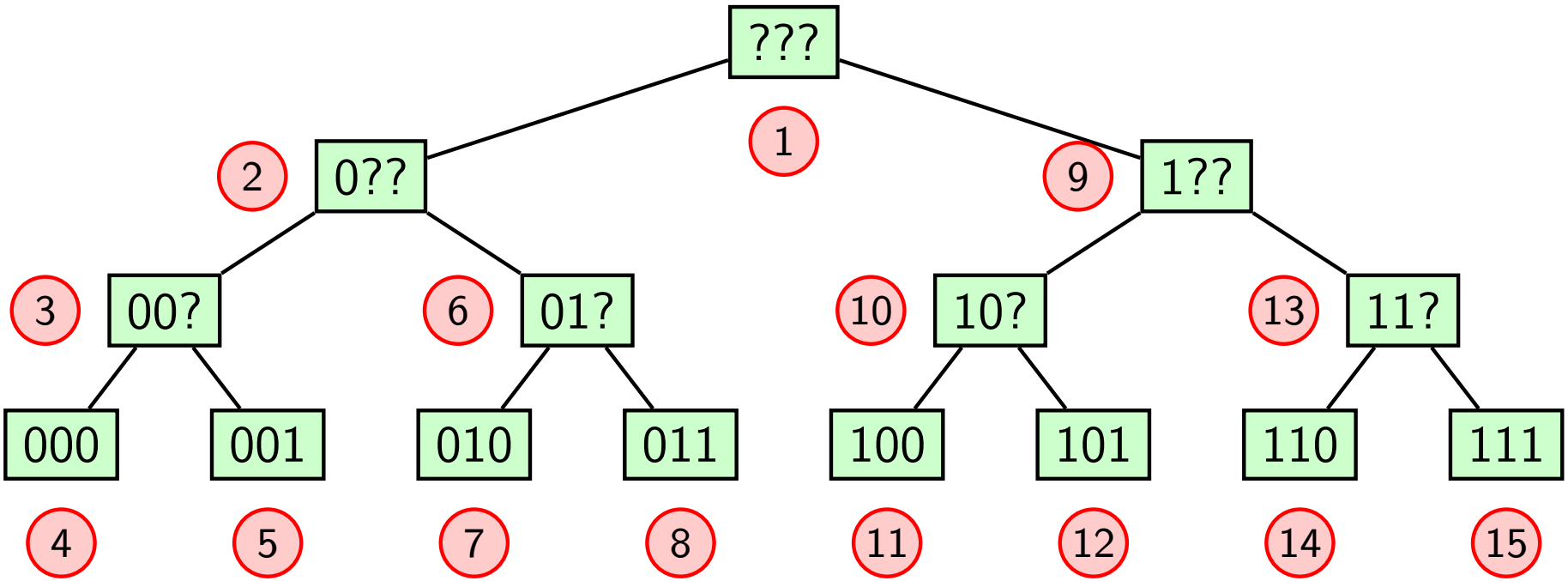
$$W = 16 \qquad w = (2, 8, 7) \qquad v = (20, 40, 49)$$

- Combinaciones posibles (espacio de soluciones):



Solución usando vuelta atrás

- Combinaciones posibles:
  - Supongamos el ejemplo: $W = 16 \quad w = (7, 8, 2) \quad v = (49, 40, 20)$
  - Espacio de soluciones
    - Generación ordenada mediante vuelta atrás
    - Nodos generados: 15
    - Nodos expandidos: 7



¿Es el recorrido importante?

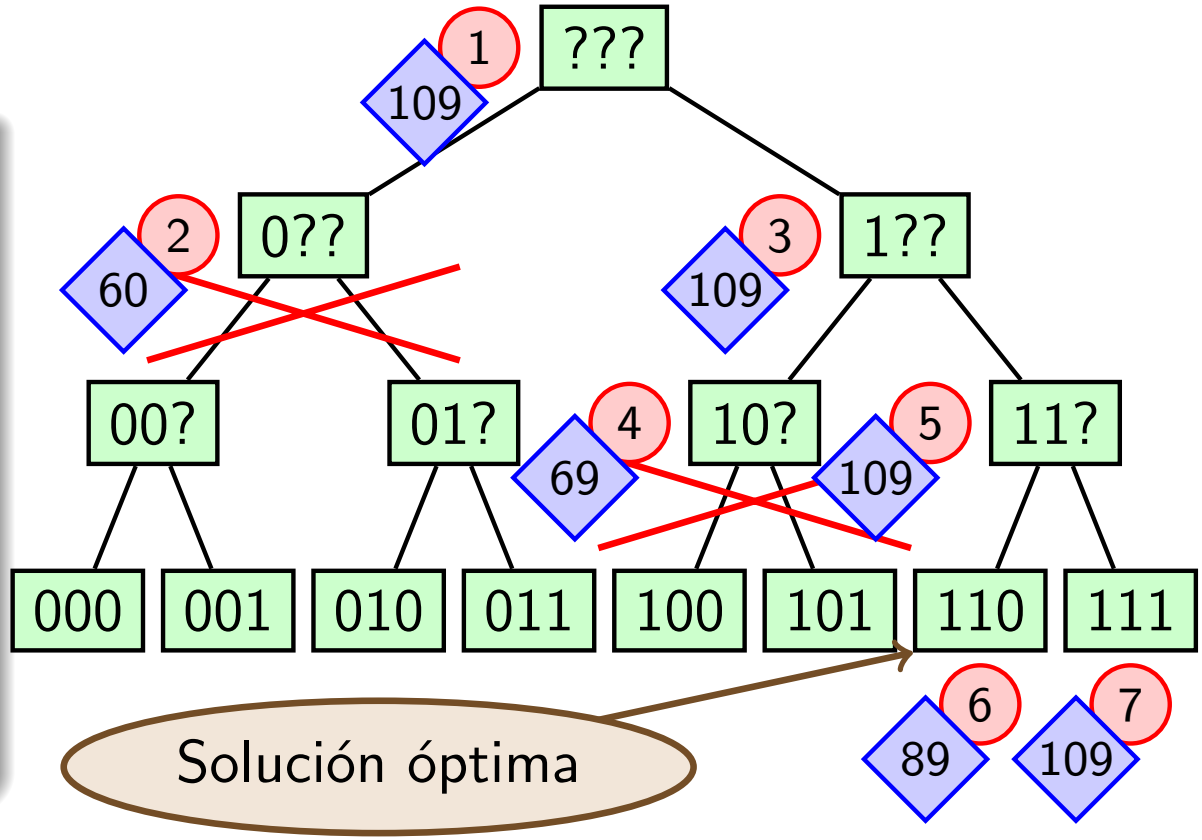
- ¿Podríamos llegar antes a la solución óptima con otro recorrido?
- ¿Cómo?
  - Adecuando el orden de exploración del árbol de soluciones según nuestros intereses. Se priorizará para su exploración aquellos nodo más prometedores
- ... sin olvidar las podas

Ejemplo Introductorio

- Combinaciones posibles:
  - Supongamos el ejemplo: $W = 16 \quad w = (7, 8, 2) \quad v = (49, 40, 20)$

Cota optimista

El valor que resultaría de incluir en la solución aquellos objetos pendientes de tratar (sustituir en cada nodo los '?' por '1'), independientemente de que quepan o no.

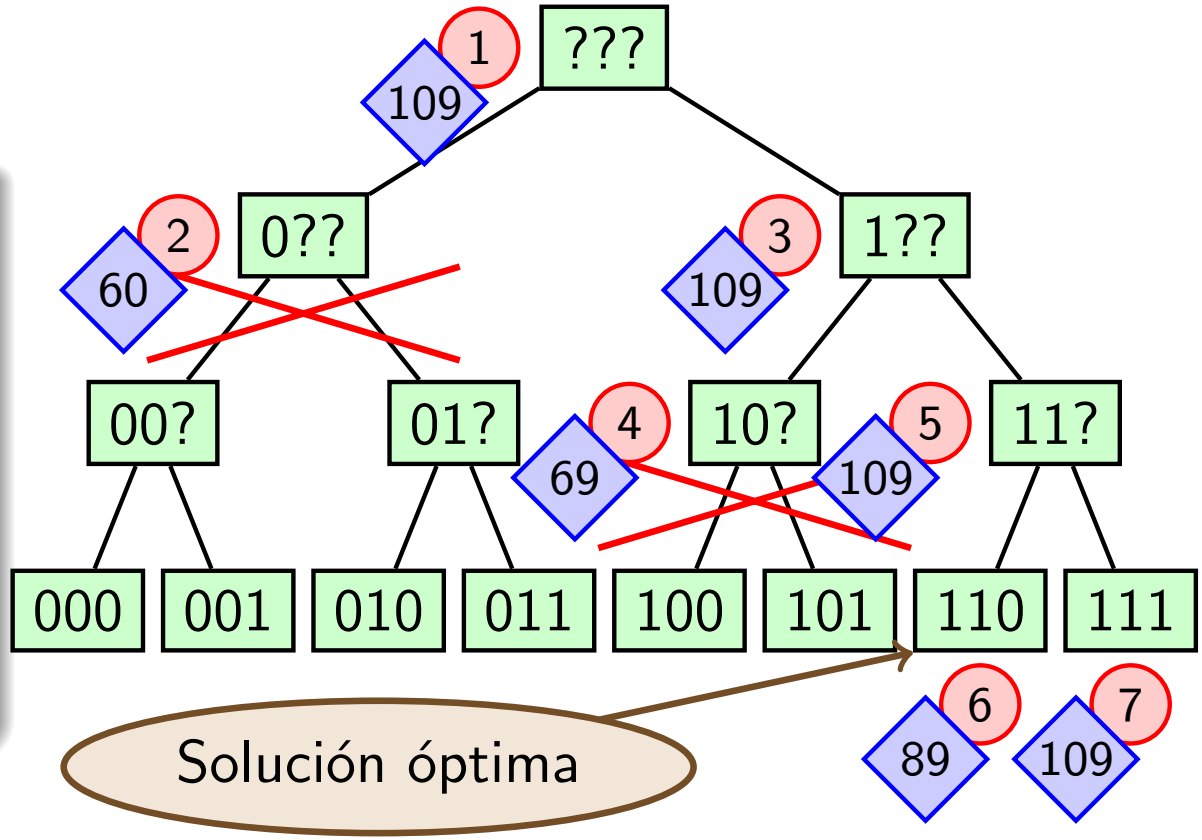


Ejemplo Introductorio

- Combinaciones posibles:
  - Supongamos el ejemplo: $W = 16 \quad w = (7, 8, 2) \quad v = (49, 40, 20)$

Espacio de soluciones

- Orden de expansión priorizando los nodos con mayor cota
- Generados: 7
- Expandidos: 3
- Reducción $\geqslant 50\,\%$



Implementación

```
1  float knapsack( const vector<double> &v, const vector<double> &w, double W ) {
2
3  //    typedef vector<short> Sol;
4  //    typedef tuple<double, double, Sol, int> Node;
5      using Sol = vector<short>;
6      using Node = tuple<double, double, Sol, int>; // acc_v, acc_w, vector x, k
7      priority_queue< Node > pq;                    // A priority_queue is a max-heap
8
9      double best_v = knapsack_d( v, w, 0, W);// updating best current solution
10     pq.emplace( 0.0, 0.0, Sol(v.size()), 0 );           // insert initial node
11
12     while( !pq.empty() ) {
13
14         auto [acc_v, acc_w, x, k] = pq.top();      // structured auto (c++17)
15         pq.pop();
16
17         /* ... next slide ...*/
18
19     }
20
21     return best_v;
22 }
```

Dentro del bucle ...

```
1  /* ... */
2      if( k == v.size() ) {                              // base case
3          best_v = max( acc_v, best_v ) ;
4          continue;
5      }
6
7      for (unsigned j = 0; j < 2; j++ ) {                // expanding
8          x[k] = j;
9
10         double current_w = acc_w + x[k] * w[k]; // updating weight
11         double current_v = acc_v + x[k] * v[k];    // updating value
12
13         if( current_w <= W &&                          // is feasible & is promising
14             current_v + knapsack_c( v, w, k + 1, W - current_w ) > best_v
15         )
16             pq.emplace( current_v, current_w, x, k + 1 );
17     }
18 /* ... */
```

Puede mejorarse?

En este caso, sí: calculando, para todos los nodos visitados, una cota pesimista (cuanto más ajustada mejor):

- Permite mejorar la mejor solución en curso antes de llegar a las hojas
  - y así, conseguir que las podas sean más eficaces.
- En el problema de la mochila resultan fructíferas.
- Pero en ocasiones pueden no serlo debido al tiempo requerido para su cálculo
  - Depende del problema, de lo ajustada que sea la cota y de la eficiencia en su cálculo.
- Con la técnica de vuelta atrás no suelen resultar tan útiles.

Usando podas pesimistas

```
1  /* ... */
2      if( k == v.size() ) {                                  // base case
3          best_v = max( acc_v, best_v ) ;
4          continue;
5      }
6
7      for (unsigned j = 0; j < 2; j++ ) {
8          x[k] = j;
9
10         double current_w = acc_w + x[k] * w[k];            // updating weight
11         double current_v = acc_v + x[k] * v[k];            // updating value
12
13         if( current_w <= W ) {                             // is feasible
14             // pessimistic bound
15             double pes_bound = current_v + knapsack_d( v, w, k+1, W-current_w);
16             best_v = max( best_v, pes_bound);
17
18             double opt_bound = current_v + knapsack_c(v, w, k+1, W-current_w);
19             if( opt_bound > best_v )                          // is promising
20                 pq.emplace( current_v, current_w, x, k+1 );
21         }
22     }
23 /* ... */
```

Ordenando por cota optimista

```
1  double knapsack( const vector<double> &v, const vector<double> &w, double W ) {
2    using Sol = vector<short>;
3    //                opt_bound,  acc_v, acc_w,   x,   k
4    using Node = tuple< double, double, double, Sol, unsigned >;
5    priority_queue< node > pq;              // Remind: priority queue is a max-heap
6
7    double best_v = knapsack_d( v, w, 0, W);
8    double opt_bound = knapsack_c(v, w, 0, W);
9
10   pq.emplace( opt_bound, 0.0, 0.0, sol(v.size()), 0 );
11
12   while( !pq.empty() ) {
13
14     auto [ignore, acc_v, acc_w, x, k] = pq.top();
15     pq.pop();
16
17     /* ... Next slide ... */
18
19   }
20   return best_v;
21 }
```

Dentro del bucle

```
1  /* ... */
2      if( k == v.size() ) {  //  base case
3          best_v = max( best_v, acc_v ) ;
4          continue;
5      }
6
7      for (unsigned j = 0; j < 2; j++ ) {
8          x[k] = j;
9
10         double current_w = acc_w + x[k] * w[k]; // updating weight
11         double current_v = acc_v + x[k] * v[k]; // updating value
12
13         if( current_w <= W ) {
14
15             double pes_bound = current_v + knapsack_d( v, w, k+1, W-current_w);
16             best_v = max( best_v, pes_bound);
17
18             double opt_bound = current_v + knapsack_c( v, w, k+1, W-current_w);
19             if( opt_bound > best_v)  // is promising
20                 pq.emplace( opt_bound, current_v, current_w, x, k+1 );
21         }
22     }
23 /* ... */
```

Comparación

Promedio del número de iteraciones para 100 instancias aleatorias del problema de la mochila con 100 objetos.

	optimista	inicializando	pesimista
Vuelta Atrás	4 491	277	253
RyP (por valor)	2 406	229	197
RyP (por cota optimista)	206	122	112

- En cuanto a RyP, en este caso se han comparado dos estrategias de búsqueda:
  1. priorizar los nodos con mayor valor;
  2. priorizar los nodos con mayor cota optimista.

Índice

Algoritmos de búsqueda exhaustiva

- Vuelta atrás y Ramificación y poda
  - Realizan una enumeración parcial del espacio de soluciones mediante la generación de un árbol de expansión (ficticeo)
  - Uso de cotas para podar ramas que no son de interés
    - Hay quien defiende que vuelta atrás no usa cotas (fuerza bruta)
- Ramificación y poda:
  - Es un refinamiento de vuelta atrás
  - Permite definir estrategias de búsqueda (o de exploración):
    - Selección del siguiente nodo a expandir de entre todos los visitados
    - Es lo que lo caracteriza como algoritmo de "búsqueda informada"
    - Muy útil en técnicas de inteligencia artificial: *machine learning*, *pattern recognition*, ...
- Vuelta atrás:
  - No hay estrategia de búsqueda: "búsqueda a ciegas"

Estrategias de búsqueda

- Nodo vivo (o nodo prometedor): aquel con posibilidades de ser ramificado (visitado pero no completamente expandido)
- Lista de nodos vivos (LNV): Estructura de datos donde se almacenan los nodos vivos.
- Estrategia de búsqueda: Forma en la que se recorre LNV para extraer el siguiente nodo a expandir:
  - Estrategia FIFO (recorrido en anchura; se implementa con una cola)
  - Estrategia LIFO (en profundidad; con una pila)
    - ¿vuelta atrás?
  - Estrategias dirigidas o LC *(Least cost)* (primero el mejor; cola de prioridad)
    - Las más utilizadas por los resultados que suelen dar.
    - Es habitual definir varias estrategias de este tipo (LC) para seleccionar la que tenga un mejor comportamiento
- Notar que vuelta atrás es un caso particular de ramificación y poda.
  - Para implementar la estrategia LIFO no es necesario implementar la lista de nodos vivos.

Definición y ámbito de aplicación

- Funcionamiento de un algoritmo de ramificación y poda. Etapas:
  - Se parte del nodo inicial y se inserta en la lista de nodos vivos (LNV)
  - Se asigna una solución pesimista (subóptima, soluciones voraces)
  - Selección
    - Extracción del nodo a expandir de LNV
    - La elección depende de la estrategia empleada
    - Se actualiza la mejor solución con las nuevas soluciones encontradas
  - Ramificación
    - Se expande el nodo seleccionado en la etapa anterior dando lugar al conjunto de sus nodos hijos
  - Poda
    - Se eliminan (podan) nodos que no contribuyen a la solución
    - El resto de nodos se añaden a LNV
  - El algoritmo finaliza cuando se agota LNV

Proceso de poda

- 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
  - suele ser una solución no factible
- 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
  - suele ser una solución factible
- Cuanto más ajustadas sean las cotas, mas podas se producirán

Esquema de Ramificación y Poda

```
1  solution branch_and_bound( problem p ) {
2
3      node initial = initial_node(p);                    // supossed feasible
4      solution current_best = pessimistic_solution(initial);   // pessimistic
5      priority_queue<Node> q.push(initial);
6
7      while( ! q.empty() ) {
8          node n = q.top();
9          q.pop();
10
11         if( is_leaf(n) ) {
12             if( is_better( solution(n), current_best) )
13                 current_best = solution(n);
14             continue;
15         }
16
17         for( node a : expand(n) )
18             if( is_feasible(a) && is_promising( a, current_best))
19                 q.push(a);
20     }
21
22     return current_best;
23 }
```

Esquema de Ramificación y Poda

- Funciones:
  - `initial_node(p)`: obtiene el nodo inicial para la expansión
  - `pesimistic_solution(n)`: devuelve una solución aproximada (factible pero no la óptima)
  - `is_leaf(n)`: mira si `n` es una posible solución
  - `solution(n)` devuelve el valor del nodo `n`
  - `expand(n)`: devuelve la expansión de `n`
  - `is_feasible(n)`: comprueba si `n` cumple las restricciones
  - `is_promising( n, current_best )`: mira si a partir del nodo `n` se pueden obtener soluciones mejores que `current_best` (normalmente se obtiene mediante una solución optimista)

Podando con cotas pesimistas

```
1  solution branch_and_bound( problem p ) {
2      node initial = initial_node(p);                    // supossed feasible
3      solution current_best = pessimistic_solution(initial);  // pessimistic
4      priority_queue<Node> q.push(initial);
5
6      while( ! q.empty() ) {
7          node n = q.top();
8          q.pop();
9
10         if( is_leaf(n) ) {
11             if( is_better( solution(n), current_best) )
12                 current_best = solution(n);
13             continue;
14         }
15         for( node a : expand(n) )
16             if( is_feasible(a) ) {
17                 if( is_better( pessimistic_solution(a), current_best ) )
18                     current_best = pessimistic_solution(a);
19                 if( is_promising (a, current_best ))
20                     q.push(a);
21             }
22     }
23     return current_best;
24   }
```

Estadísticas (I)

```
1  solution branch_and_bound( problem p ) {
2      node initial = initial_node(p);
3      solution current_best = pessimistic_solution(initial);
4      priority_queue<Node> q.push(initial);
5
6      while( ! q.empty() ) {
7          node n = q.top();
8          q.pop();
9
10         if( !is_promissing(n,current_best) ){
11             discarded_promissing_nodes++;
12             continue;
13         }
14
15 /* ... next slide ... */
16
17     return current_best;
18   }
```

Estadísticas (II)

```
1  ...
2          if( is_leaf(n) ) {
3              completed_nodes++;
4              if( is_better( solution(n), current_best) ) {
5                  current_best_updates_from_completed_nodes++;
6                  current_best = solution(n);
7              }
8              continue;
9          }
10       expanded_nodes++;
11         for( node a : expand(n) ) {
12             visited_nodes++;
13             if( is_feasible(a) ) {
14                 if( is_better( pessimistic_solution(a), current_best ) ) {
15                     current_best_updates_from_pessimistic_bounds++;
16                     current_best = pessimistic_solution(a);
17                 }
18                 if( is_promising (a, current_best )) {
19                     explored_nodes++;
20                     q.push(a);
21                 } else no_promissing_discarded_nodes++;
22             } else no_feasible_discarded_nodes++;
23         }
24 ...
```

Características

- La estrategia puede proporcionar:
  - Todas las soluciones factibles
  - Una solución al problema
  - La solución óptima al problema
  - Las n mejores soluciones
- Objetivo de esta técnica
  - Mejorar la eficiencia en la exploración del espacio de soluciones
- Desventajas/Necesidades
  - Encontrar una buena cota optimista (problema relajado)
  - Encontrar una buena solución pesimista (estrategias voraces)
  - Encontrar una buena estrategia de exploración (cómo ordenar)
  - Mucho mayor requerimiento de memoria que los algoritmos de Vuelta Atrás
  - Las complejidades en el peor caso suelen ser muy altas
- Ventajas
  - Suelen ser más rápidos que Vuelta Atrás

Esquema de Ramificación y Poda

- Muchas veces, para ver si un nodo es prometedor, se hace comparando la mejor solución obtenida con una solución optimista de nodo.

```
1 bool is_promising( const node &n, const solution &current_best) {
2   return is_better( optimistic_solution(n), current_best );
3 }
```

se pueden hacer podas agresivas cambiándola por:

```
1 bool is_promising( const node &n, const solution &current_best) {
2   return is_significantly_better(optimistic_solution(n), current_best);
3 }
```

¡Cuidado! puede que se pierda la solución óptima

Índice

El viajante de comercio

Dado un grafo ponderado $g = (V, A)$ 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: $\forall i : 1 \leq i < n,\ peso(g, x_i, x_{i+1}) \neq \infty$
  - Existencia de arista que cierra el camino: $peso(g, x_n, x_1) \neq \infty$
- Función objetivo:

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

El viajante de comercio

```
1  unsigned travelling_salesman( const graph &g) {
2      struct Node {
3          unsigned opt_bound, length;
4          vector<short> x;
5          unsigned k;
6      };
7      struct is_worse {
8          bool operator() (const Node& a, const Node& b) {
9              return a.opt_bound > b.opt_bound;
10         }
11     };
12     priority_queue< Node, vector<Node>, is_worse > pq;
13     vector<short> x(g.num_cities());
14     for( unsigned i = 0; i < g.num_cities(); i++ )  x[i] = i;
15     unsigned shortest = pessimistic_bound( g, x, 1);
16     unsigned opt_bound = optimistic_bound( g, x, 1);
17     pq.emplace( opt_bound, 0, x, 1 );
18     while( !pq.empty() ) {
19       Node n = pq.top();
20       pq.pop();
21     /* ... Next slide ... */
22     }
23     return shortest;
24 }
```

Dentro del bucle ...

```
1  /* ... */
2      if( n.k == g.num_cities() ) {
3        shortest = min( shortest, n.length + g.dist(n.x[n.k-1],n.x[0]) );
4        continue;
5      }
6      for( unsigned c = n.k; c < n.x.size(); c++ ) {
7        swap( n.x[n.k], n.x[c] );
8
9        unsigned new_length = n.length + g.dist(n.x[n.k-1],n.x[n.k]);
10       unsigned opt_bound = new_length + optimistic_bound(g,n.x,n.k+1);
11       unsigned pes_bound = new_length + pessimistic_bound(g,n.x,n.k+1);
12
13       shortest = min( shortest, pes_bound);
14
15       if( opt_bound <= shortest )
16         pq.emplace( opt_bound, new_length, n.x, n.k+1 );
17
18       swap( n.x[n.k], n.x[c] );
19     }
20 /* ... */
```

Cambiando la prioridad ...

Por cota optimista

```
1 struct is_worse {
2     bool operator() (const node& a, const node& b) {
3         return a.opt_bound > b.opt_bound;
4     }
5 };
```

Por distancia recorrida

```
1 struct is_worse {
2     bool operator() (const node& a, const node& b) {
3         return a.length > b.length;
4     }
5 };
```

Por distancia media recorrida

```
1 struct is_worse {
2     bool operator() (const node& a, const node& b) {
3         return a.length/a.k > b.length/b.k;
4     }
5 };
```

Algunos ejemplos de ejecución

Promedio del número de iteraciones necesitado para resolver 100 instancias del problema del viajante de comercio con 15 ciudades

- **Cota optimista**: *minumum spanning tree* de las ciudades restantes
- **Cota pesimista**: algoritmo voraz basado en la ciudad más cercana

Algoritmo	**cota opt.**	**dist. recorrida**	**dist. media**
Vuelta atrás	23 478	23 478	23 478
Ramificación y poda	10 798	12 285	11 421
factor de aceleración	2.17	1.91	2.05

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)
- Llamanmos $F(X, k, x)$ 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 recalculos
  - $k < M$, para evitar búsquedas infinitas
  - siempre se puede calcular $F(X, k, x)$
  - $k < v_b$, tupla "prometedora"

Composición de funciones

```
1  const size_t NOT_FOUND = numeric_limits<size_t>::max();
2
3  int composition( unsigned M, int first, int last ) {
4      using Node = tuple< short, int>; // steps, present
5
6      struct is_worse {
7          bool operator() (const Node& a, const Node& b) {
8              return get<0>(a) > get<0>(b);
9          }
10     };
11     priority_queue< Node, vector<Node>, is_worse > pq;
12
13     unsigned best = NOT_FOUND;
14     pq.emplace( 0, first );
15
16     while( !pq.empty() ) {
17         Node n = pq.top();
18         pq.pop();
19
20     /* ... next slide ... */
21
22     }
23     return best;
24 }
```

Dentro del bucle ... (sin podas)

```
1  /*...*/
2      unsigned k = get<0>(n);
3      int present = get<1>(n);
4
5      if( present == last && k < best )  // select the best
6          best = k;
7
8      if( k ==  M )                      //  limit bound
9          continue;
10
11     for( short i = 0; i < 2; i++ ) {
12         int next = F1( i, present );
13         pq.emplace( k+1, next );
14     }
15 /*...*/
```

Dentro del bucle ... (poda: mejor en curso)

poda: mejor en curso

```
1  /*...*/
2      unsigned k = get<0>(n);
3      int present = get<1>(n);
4
5      if( present == last && k < best ) { // select the best
6          best = k;
7          continue;
8      }
9
10     if( k >= best )          // best solution in progress bound
11         continue;
12
13     if( k ==  M )            // limit bound
14         continue;
15
16     for( short i = 0; i < 2; i++ ) {
17         int next = F1( i, present );
18         pq.emplace( k+1, next );
19     }
20 /*...*/
```

Podando con memoria

```
1  int composition( unsigned M, int first, int last ) {
2
3      using Node = tuple< short, int>; // steps, hit
4
5      struct is_worse {
6          bool operator() (const Node& a, const Node& b) {
7              return get<0>(a) > get<0>(b);
8          }
9      };
10     priority_queue< Node, vector<Node>, is_worse > pq;
11     unordered_map<int, Default<unsigned, NOT_FOUND>> best;
12
13     pq.emplace( 0, first );
14     while( !pq.empty() ) {
15         node n = pq.top();
16         pq.pop();
17
18     /* ... next slide ... */
19
20     }
21     return best[last];
22 }
```

Dentro del bucle ...

```
1  /* ... */
2      unsigned k = get<0>(n);
3      int present = get<1>(n);
4
5      if( best[present] <= k )  // update the best
6          continue;
7      best[present] = k;
8
9      if( k >= best[last] )   // (extended) best solution in progress bound
10         continue;
11
12     if( k == M )            // limit bound
13         continue;
14
15     for( short i = 0; i < 2; i++ ) {
16         int next = F1( i, present );
17         pq.emplace( k+1, next );
18     }
19 /* ... */
```

Ejemplo de ejecución

Número de iteraciones para alcanzar el valor 11 desde 1 (con $M = 20$)

Algoritmo	**Vuelta Atrás**	**Ramificación y Poda**
básico	65 535	65 535
mejor en curso	9 073	8 311
memorizando	565	329

Índice

El Puzzle

- Disponemos de un tablero con n^2 casillas y de $n^2 - 1$ piezas numeradas del uno al $n^2 - 1$. Dada una ordenación inicial de todas las piezas en el tablero, queda sólo una casilla vacía (con valor 0), a la que denominamos *hueco*.
- El objetivo del juego es transformar dicha disposición inicial de las piezas en una disposición final ordenada, en donde en la casilla (i, j) se encuentra la pieza numerada $n(i-1)+j$ y en la casilla (n, n) se encuentra el hueco
- Los únicos movimientos permitidos son los de las piezas adyacentes al hueco (horizontal y verticalmente), que pueden ocuparlo. Al hacerlo, dejan el hueco en la posición en donde se encontraba la pieza antes del movimiento. Otra forma de abordar el problema es considerar que lo que se mueve es el hueco, pudiendo hacerlo hacia arriba, abajo, izquierda o derecha. Al moverse, su casilla es ocupada por la pieza que ocupaba la casilla a donde se ha *movido* el hueco

El Puzzle

- Por ejemplo, para el caso n = 3 se muestra a continuación una disposición inicial junto con la disposición final:

1	5	2
4	3	
7	8	6

Disposición Inicial

1	2	3
4	5	6
7	8	

Disposición final

El Puzzle

- Regla: una pieza se puede mover de A a B si:
  - A está al lado de B
  - B es el hueco

- Según se relaje el problema tenemos dos funciones de coste diferentes:
  1. Calcular el número de piezas que están en una posición distinta de la que les corresponde en la disposición final
  2. Calcular la suma de las distancias de Manhattan desde la posición de cada pieza a su posición en la disposición final
- La distancia de Manhattan entre dos puntos del plano de coordenadas (x_1, y_1) y (x_2, y_2) viene dada por la expresión:

$$|x_1 - x_2| + |y_1 - y_2|$$