Análisis y diseño de algoritmos

7. Ramificación y Poda

José Luis Verdú Mas, Jose Oncina, Mikel L. Forcada

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

26-01-2015 (318)

Ejemplo Introductorio

El problema de la mochila (general)

Dados:

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

Seleccionar un conjunto de objetos de forma que:

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

Ejemplo Introductorio

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:

$$\sum_{i=1}^{n} x_i p_i \leq P$$

- Función objetivo:

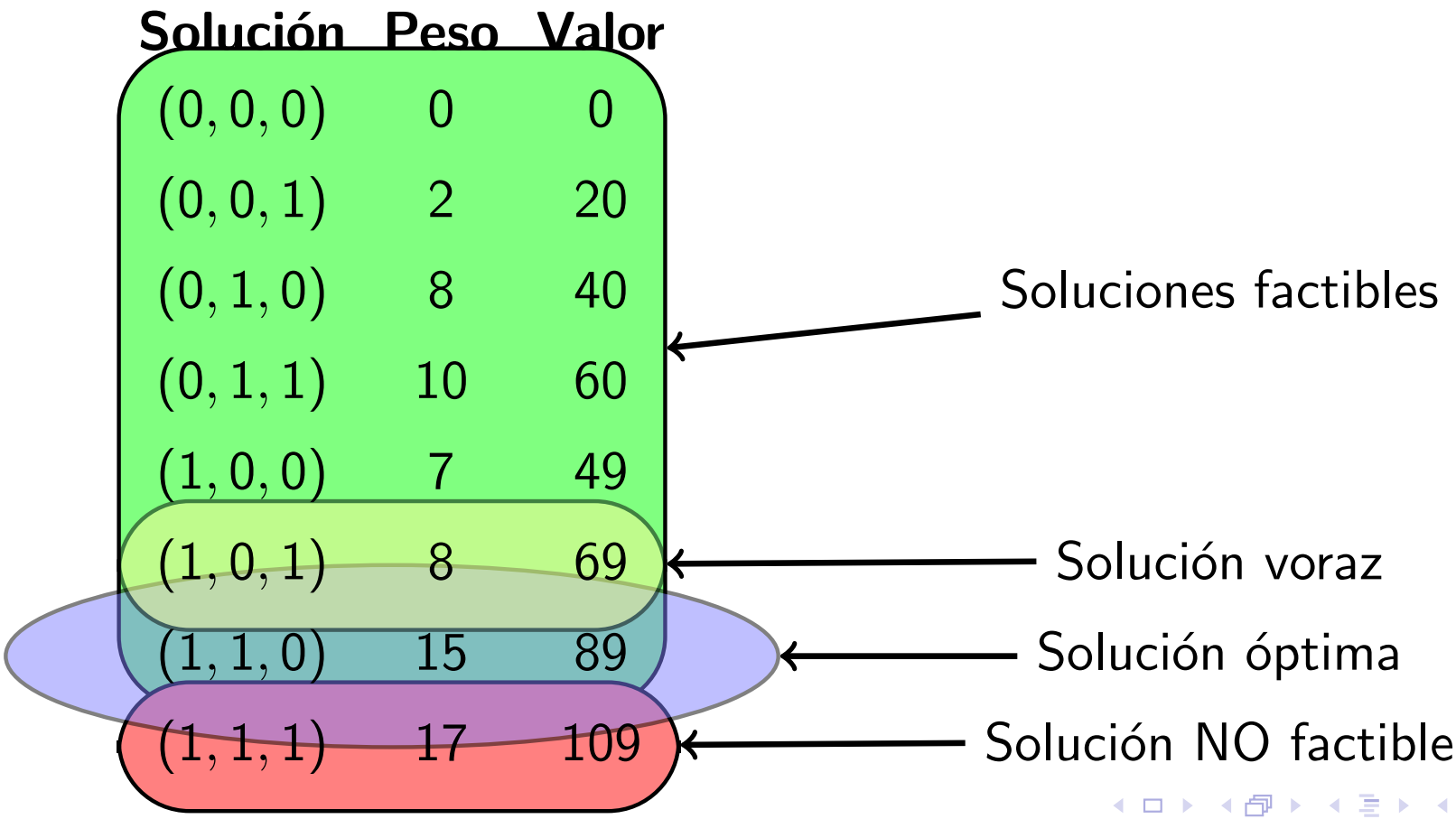
$$\text{máx} \sum_{i=1}^{n} x_i v_i$$

Ejemplo Introductorio

- Supongamos el siguiente ejemplo:

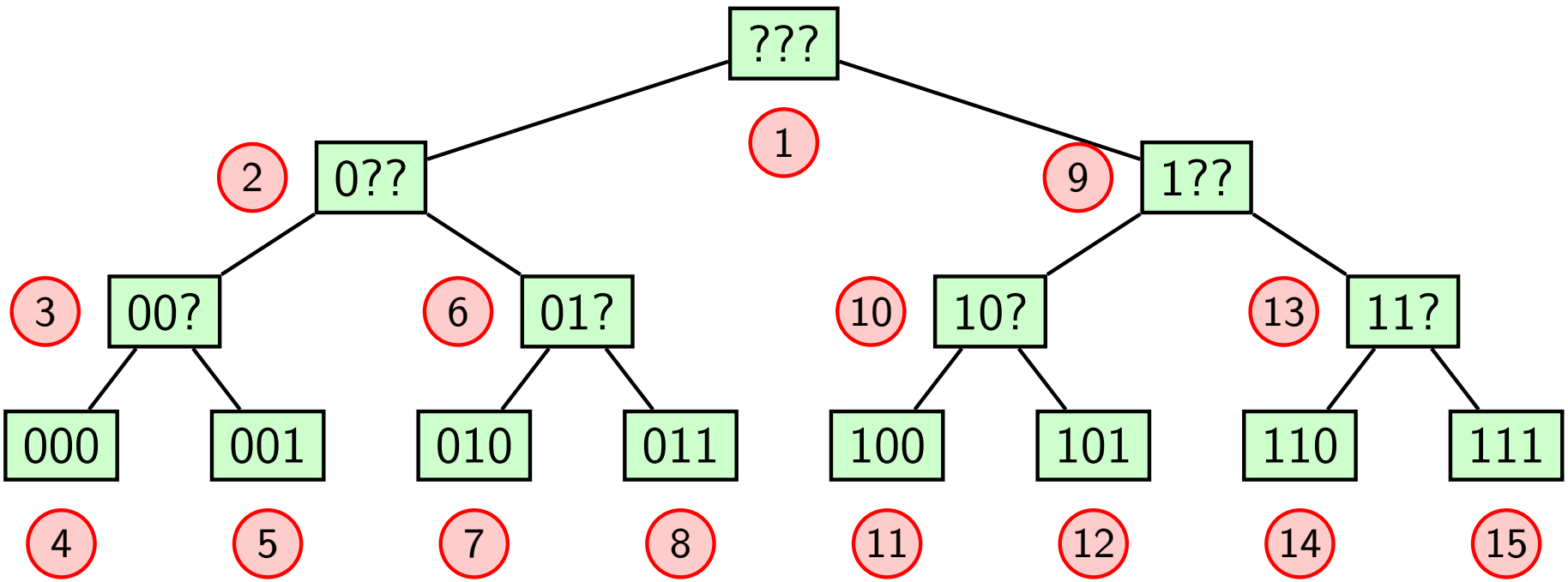
$$P = 16 \qquad p = (7, 8, 2) \qquad v = (49, 40, 20)$$

- Espacio de soluciones



Ejemplo Introductorio

- Combinaciones posibles:
  - Supongamos el ejemplo: $P = 16 \quad p = (7, 8, 2) \quad v = (49, 40, 20)$
  - Espacio de soluciones
    - Generación ordenada mediante vuelta atrás[1]
    - Nodos generados: 15
    - Nodos expandidos: 7



[1] Recorrido fijo, ciego

Ejemplo Introductorio

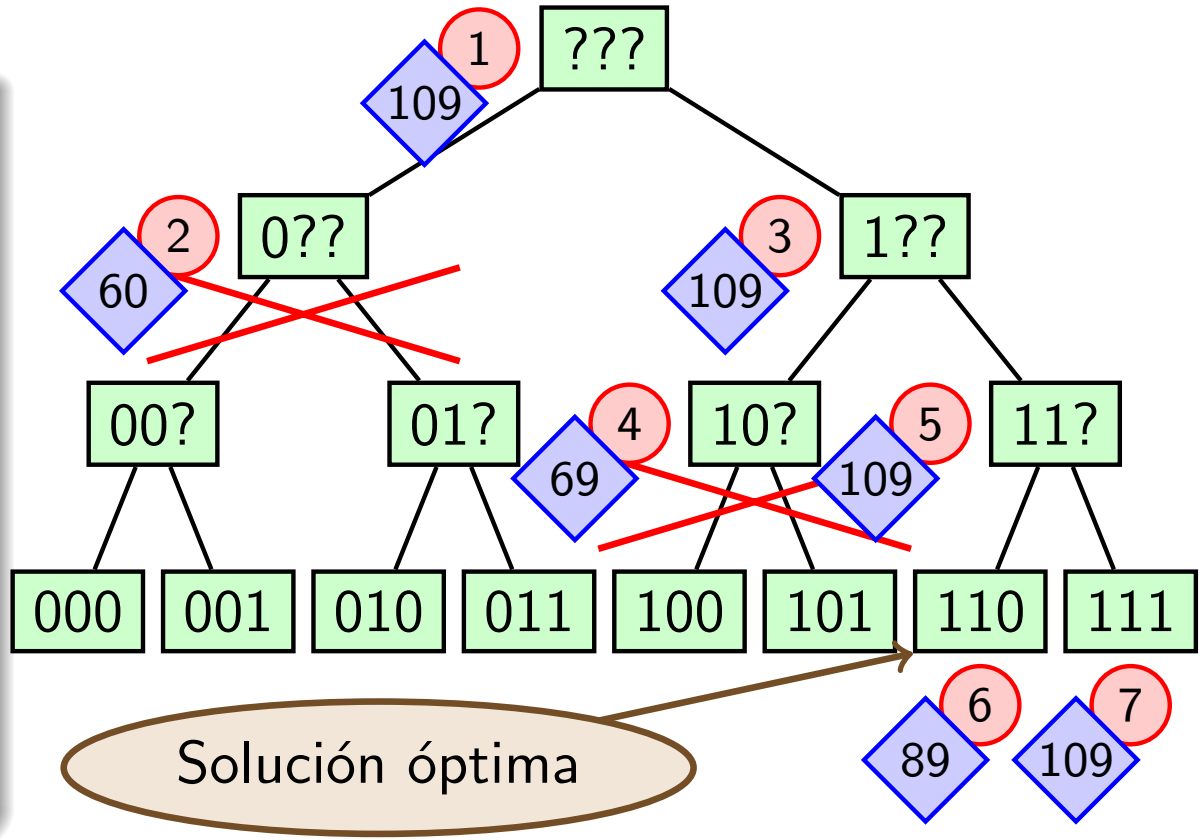
- ¿Podríamos llegar a la solución óptima con otro recorrido?
- ¿Permite esto reducir el número de nodos generados?
- ¿Cómo?
  - Usando una función cota optimista que permita acotar el valor máximo factible que se puede obtener a partir de la solución parcial que se está explorando.
  - No se explorarán aquellos nodos cuya cota no mejore el valor de la mejor solución obtenida hasta el momento
  - Se adecuará el orden de exploración del árbol de soluciones según nuestros intereses. Se priorizará para su exploración aquellos nodo mas prometedores (usualmente aquellos con cota mas elevadas)

Ejemplo Introductorio

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

Función de cota

Cada nodo toma cota 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.

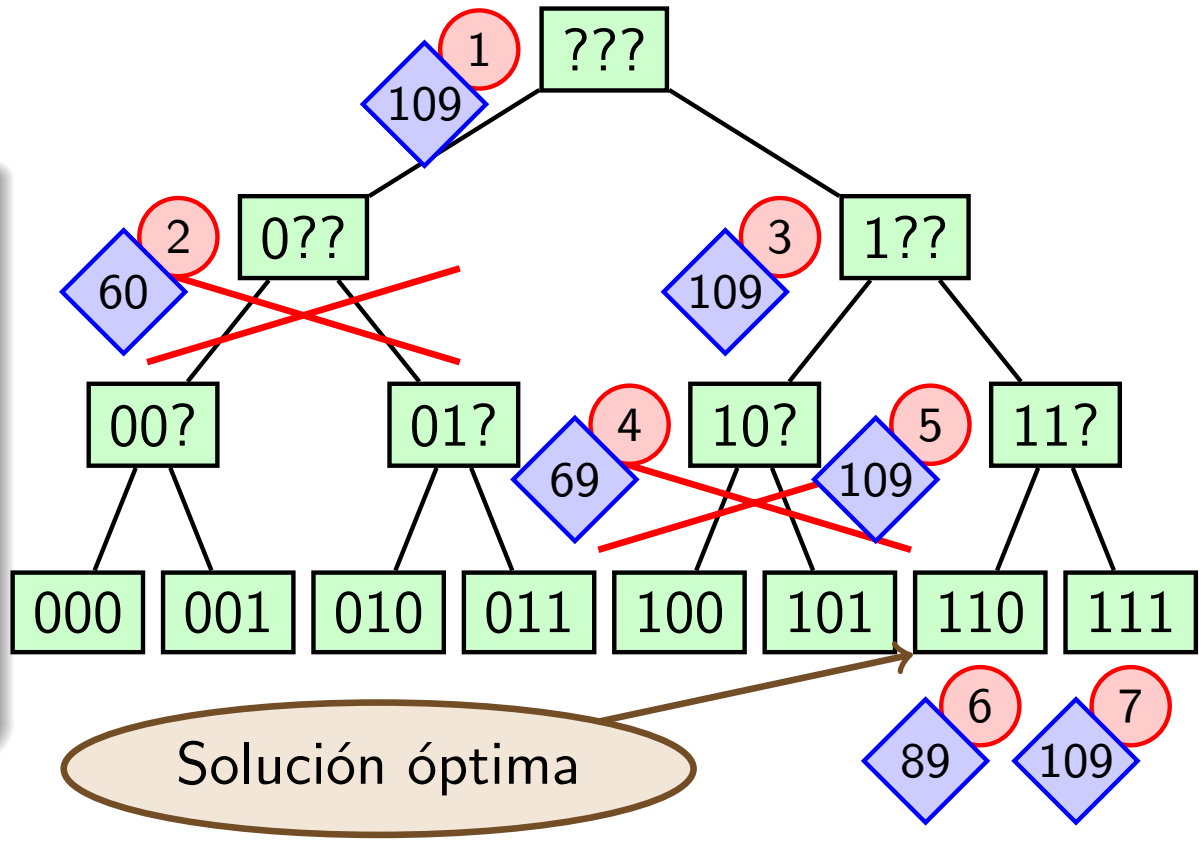


Ejemplo Introductorio

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

Espacio de soluciones

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



Definición y ámbito de aplicación

- Variante del diseño de Vuelta Atrás
- Realiza una enumeración parcial del espacio de soluciones mediante la generación de un árbol de expansión
- Permite la exploración de nodos empleando diferentes estrategias
  - Vuelta atrás: LIFO, búsqueda fija, ciega
  - FIFO, otra búsqueda fija, ciega
  - Ramificación y poda: búsqueda dirigida

Definición y ámbito de aplicación

- Uso de cotas para podar aquellas ramas del árbol de expansión que no conducen a la solución óptima
- Nodo vivo: aquel con posibilidades de ser ramificado (visitado pero no completamente expandido)
- Los nodos vivos se almacenan en estructuras que faciliten su recorrido y eficiencia de la búsqueda:

  - En anchura (estrategia FIFO) $\Rightarrow$ cola
  - En profundidad (estrategia LIFO) $\Rightarrow$ pila
  - Dirigida (estrategia mínimo coste LC) $\Rightarrow$ montículo (Heap)

Definición y ámbito de aplicación

- Funcionamiento de un algoritmo de ramificación y poda
- Etapas
  - Partimos del nodo inicial del árbol
  - Se asigna una solución pesimista (subóptima, soluciones voraces)
  - Selección
    - Extracción del nodo a expandir del conjunto de nodos vivos
    - 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) aquellos que no conducen a una mejor solución
    - El resto de nodos se añaden al conjunto de nodos vivos
  - El algoritmo finaliza cuando se agota el conjuntos de nodos vivos

Definición y ámbito de aplicación

- Proceso de Poda
  - Necesitamos una función cota optimista
  - Para cada nodo del árbol, dicha función estima el mejor valor que podría alcanzarse al expandir el nodo.
  - Si la cota optimista de un nodo es peor que el valor de una solución ya encontrada, podemos podar ese nodo[2]
  - Consecuencias:
    - Para podar necesitamos encontrar una solución
    - Cuanto mejor sea la solución mayor nivel de poda se consigue
    - Cuanto mas ajustada sea la cota optimista mayores podas
    - Se suele emplear soluciones al problema con las restricciones relajadas (i.e. problema de la mochila continuo)

[2] Como en *vuelta atrás*

Esquema de Ramificación y Poda

```
1  Solution BB( Problem p ) {
2    Node init = initialNode(p);
3    Solution best = init.pessimistic_sol();
4    priority_queue<Node> q.push(init);
5    while( ! q.empty() ) {
6      Node n = q.top(); q.pop();
7      if( n.optimistic_b() > best.value() )
8        if( n.isTerminal() )
9          best = n.sol();
10       else
11         for( Node a : n.expand() )
12           if( a.isFeasible() )
13             q.push(a);
14   }
15   return best;
16 }
```

Esquema de Ramificación y Poda

- Funciones:
  - `initialNode(p)`: obtiene el nodo inicial para la expansión
  - `pessimistic_sol()`: devuelve una solución aproximada (factible pero no la óptima).[3]
  - `n.value()` devuelve el valor del nodo `n`
  - `n.optimistic_b()`: obtiene una cota superior al la mejor solución obtenible de la expansión de `n`. Si `n` es terminal devolverá `n.value()`
  - `n.isTerminal()`: mira si `n` es una posible solución
  - `n.sol()`: extrae una solución del nodo
  - `n.expand()`: devuelve la expansión de `n`
  - `n.isFeasible()`: comprueba si `n` cumple las restricciones

[3] Típicamente, una voraz

Características

- La estrategia puede proporcionar:
  - Todas las soluciones factibles
  - Una solución al problema
  - La solución óptima al problema
- 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 (mejor cota optimista)
  - Mayor requerimiento de memoria que los algoritmos de Vuelta Atrás
  - las complejidades en el peor caso suelen ser muy altas
- Ventajas
  - Son fáciles de implementar

Usando cotas pesimistas

```
1  Solution BB( Problem p ) {
2    Node best, init = initialNode(p);
3    Value pb = init.pessimistic_b();
4    priority_queue<Node> q.push(init);
5    while( ! q.empty() ) {
6      Node n = q.top(); q.pop();
7      if( n.optimistic_b() >= pb ){
8        pb = max( pb, n.pessimistic_b());
9        if( n.isTerminal() )
10         best = n.sol();
11       else
12         for( Node n : n.expand() )
13           if( n.isFeasible())
14             q.push(n);
15   }
16   return best;
17 }
```

Esquema de Ramificación y Poda

- Funciones:
  - `n.pessimistic_b()`: devuelve una cota inferior a la mejor obtenible de la expansión de `n`
- La ventaja de usar cotas pesimistas es que se puede podar antes de tener una solución factible
- La condición:

```
n.optimistic_b() >= pb
```

puede cambiarse por

```
n.optimistic_b() >= p * pb
```

(con $p > 1$) si se quieren hacer podas agresivas

Ejemplos de aplicación

- Ejemplos de aplicación:
  - El problema de la mochila
  - Secuenciación de tareas con penalización
- Ejercicio:
  - El puzzle

El problema de la mochila

- Veamos cómo instanciar el esquema general de RyP para implementar la resolución del problema
- Tipificación del problema:
  - Solución $= X = (x_1, x_2, \ldots, x_n) \quad x_i \in \mathbb{R}$
  - Restricciones:
    - Implícitas:

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

    - Explicitas:

$$\sum_{i=1}^{n} x_i p_p \leqslant P$$

  - Función objetivo: máx $\sum_{i=1}^{n} x_i v_i$
- Valor de cota de un nodo:
  - Valor que alcanzaría el nodo si incluimos los objetos que quedan pendientes de analizar

Estructuras de datos

```
1 class Knapsack {
2 private:
3     double P;
4     vector<double> v;
5     vector<double> w; ...
```

[Se asume que v y w están ordenados por valor específico]

```
1 class Node {
2 private:
3     static Knapsack p;
4     double pw;
5     double pv;
6     vector<bool> s;
7 public:
8     Node(): pw(0.0), pv(0.0) {}; ...
```

Funciones

```
bool isTerminal() const {
    return s.size() == p.v.size();
}
```

```
bool isFeasible() const {
    return pw <= p.P;
}
```

```
int value() const {
    return pv;
}
```

```
static Node initialNode( const Knapsack& _p ) {
  p = _p;
  return Node();
}
```

Funciones

Mochila Voraz

```
1  Node pessimistic_sol() {
2    Node n;
3    double n.pv = pv;
4    double n.pw = pw;
5    for( unsigned i = s.size(); i < p.v.size(); i++ ){
6      if( n.pw + p.w[i] <= p.P ) {
7        n.pv += p.v[i];
8        n.pw += p.w[i];
9        n.s.push_back(true);
10     } else {
11       n.s.push_back(false);
12     }
13   }
14   return n;
15 }
```

Funciones

Mochila Continua

```
1  double optimistic_b() const {
2    double value = pv;
3    double weight = pw;
4    for( unsigned i = s.size(); i < p.v.size(); i++){
5      if( weight + p.w[i] <= p.P ) {
6        value += p.v[i];
7        weight += p.w[i];
8      } else {
9        value += p.v[i] * (p.P - weight ) / p.w[i];
10       break;
11     }
12   }
13   return value;
14 }
```

Funciones

Mochila Voraz

```
1  double pessimistic_b() const {
2    double value = pv;
3    double weight = pw;
4    for( unsigned i = s.size(); i < p.v.size(); i++){
5      if( weight + p.w[i] <= p.P ) {
6        value += p.v[i];
7        weight += p.w[i];
8      }
9    }
10
11   return value;
12 }
```

Funciones

```
1  vector<Node> expand() const {
2      Node n1(*this);
3      n1.s.push_back(false);
4
5      Node n2(*this);
6      n2.pw += p.w[s.size()];
7      n2.pv += p.v[s.size()];
8      n2.s.push_back(true);
9
10     return vector<Node>({n1,n2});
11 }
```

Uso de una cola de prioridad

```
1  class mycomp {
2  public:
3      bool operator()(
4              const Node& n1,
5              const Node& n2
6      ) const {
7          return n1.optimistic_b() < n2.optimistic_b();
8      }
9  };
10
11 typedef
12     priority_queue<Node, vector<Node>, mycomp>
13     myqueue;
```

Secuenciación de tareas con penalización

- Enunciado:
  - Se dispone de N tareas pendientes de realizar. Cada tarea i dispone de un tiempo de ejecución t_i, un plazo límite de finalización f_i y una penalización p_i (indemnización) que ha de soportar en el caso de que la tarea no se ejecute a tiempo
  - Diseñar un algoritmo mediante RyP que determine la fecha de comienzo de cada tarea de forma que la indemnización que haya que pagar en el caso de que no puedan realizarse todas las tareas sea mínima
  - Ejemplo:

Tarea	**1**	**2**	**3**	**4**
Tiempo de ejecución (días)	2	1	2	3
Plazo límite (días)	3	4	4	3
Indemnización (miles de euros)	5	15	13	10

Secuenciación de tareas con penalización

- Tipificación del problema:
  - Número de tareas: N
  - Representación de la solución: $X = (x_1, x_2, \ldots, x_N)$
    - x_i representa el día de comienzo de la tarea i
    - Cada x_i puede tomar los valores: $x_i \in \{0, \ldots, (f_i - t_i + 1)\}$
    - El valor cero indica que la tarea no se realiza

Secuenciación de tareas con penalización

- Tipificación del problema:
  - Representación de la solución: $X = (x_1, x_2, \ldots, x_N)$
  - Cada x_i puede tomar los valores: $x_i \in \{0, \ldots, (f_i - t_i + 1)\}$
- Restricciones:
  - Dos tareas no pueden estar ejecutándose al mismo tiempo (Para cada tarea i ya asignada, la nueva tarea k empieza después de finalizar la tarea i o bien, la tarea k termina antes de empezar la tarea i)

$$(x_k > x_i + t_i - 1) \vee (x_k + t_k - 1 < x_i) \quad \forall i \in [1, k]$$

- Función objetivo:
  - Minimizar el valor de la indemnización total asociada a X

$$\text{mín} \sum_{i=1}^{N} \text{ind}(x_i, p_i) \qquad \text{ind}(x_i, p_i) = \begin{cases} p_i & x_i = 0 \\ 0 & x_i \neq 0 \end{cases}$$

Estructuras de datos

```
1 class Seq {
2 private:
3     vector<int> len;
4     vector<int> due;
5     vector<int> penalty;
6 ...
```

```
1 class Node {
2 private:
3     static Seq p;
4     vector<int> start;
5     int penalty;
6 public:
7     Node(): penalty(0) {};
8 ...
```

Funciones

```
bool isTerminal() const {
    return start.size() == p.len.size();
}
```

```
bool isFeasible() const {
    return true;
}
```

```
Node pessimistic_sol() {
  Node n;
  n = *this;
  for( unsigned i=start.size(); i<p.len.size();i++){
    n.start.push_back(0);
    n.penalty += p.penalty[i];
  }
  return n;
}
```

Funciones

```
1  list<Node> expand() const {
2    list<Node> vn;
3    int pt = start.size();
4    Node n(*this);
5    n.penalty += p.penalty[pt];
6    n.start.push_back(0);
7    vn.push_back(n);
8    for( int s = 1; s <= p.due[pt] - p.len[pt]; s++ ){
9      if( fit( pt, s) ) {
10       Node n(*this);
11       n.start.push_back(s);
12       vn.push_back(n);
13     }
14   }
15   return vn;
16 }
```

Funciones

```
1 int optimistic_b() const {
2   return penalty;
3 }
```

```
1 int pessimistic_b() const {
2   int pb = penalty;
3   for( unsigned i=start.size(); i<p.len.size();i++){
4     pb += p.penalty[i];
5   }
6   return pb;
7 }
```

Funciones

```
1 bool fit( int task, int s ) const {
2   for( int prev_task=0; prev_task<task; prev_task++)
3     if( start[prev_task] > 0 &&
4       start[prev_task] < s + p.len[task] &&
5       start[prev_task] + p.len[prev_task] > s ) {
6       return false;
7     }
8   return true;
9 }
```

Uso de una cola de prioridad

```
1  class mycomp {
2  public:
3      bool operator()(
4              const Node& n1,
5              const Node& n2 ) const {
6          return n1.optimistic_b() > n2.optimistic_b();
7      }
8  };
9
10 typedef
11     priority_queue<Node, vector<Node>, mycomp>
12     myqueue;
```

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*.[4]
- 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

[4] Guerequeta y Vallecillo, p. 162 y ss.

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

- Es posible resolver el problema mediante Ramificación y Poda utilizando dos funciones de coste diferentes:
  - La primera calcula el número de piezas que están en una posición distinta de la que les corresponde en la disposición final
  - La segunda se basa en 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:

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