Programación II

Tema 10. Ramificación y Poda

10.1 Introducción

El método RyP (*Branch and Bound*) es una variante del de Vuelta Atrás y, por tanto, es aplicable a problemas:

- **cuya solución se expresa como una secuencia de decisiones**
- **en cada decisión se elige entre un conjunto finito de valores**

RyP da lugar a algoritmos de complejidad exponencial

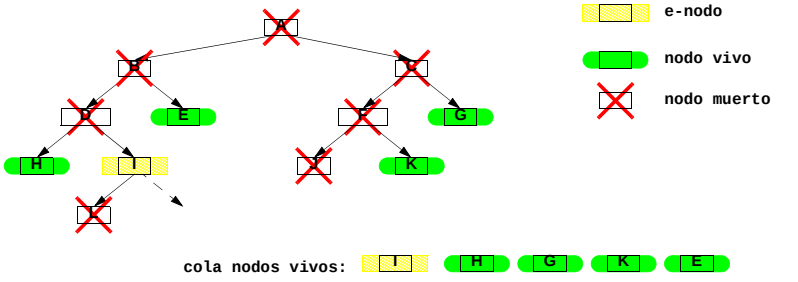
- **por lo que normalmente se utiliza en problemas complejos que no pueden resolverse en tiempo polinómico (NP-completos)**

Al igual que VA, *RyP realiza una exploración sistemática del árbol de búsqueda*, pero:

- **no utiliza recursividad para generar el árbol**
- **el árbol existe como una *cola de nodos vivos***
  - **ordenada en base a una *función de coste* que proporciona una estimación de lo prometedor que es cada nodo**

En RyP distinguiremos tres tipos de nodos:

- ***nodos vivos*: todavía no han sido *ramificados***
- ***nodo en expansión* (e-nodo): está siendo *ramificado* en este momento (sólo puede haber uno en cada etapa)**
- ***nodos muertos*: ya han sido ramificados o han sido descartados (*podados*)**



La clave de los algoritmos RyP está en la utilización de la función de coste

- ***coste de un nodo*: estimación (*nunca pesimista*) del valor de la mejor solución que podríamos encontrar ramificando ese nodo**

```
coste de un nodo = valor del nodo +
  estimación de lo que le falta para ser solución
```

La función de coste se utiliza para dos cosas:

1. **Ordenar la cola de nodos vivos de mejor a peor coste**
   - **con lo que se *ramifican* primero los nodos más prometedores**
2. ***Podar* las ramas que no puedan conducirnos a una solución que mejore la mejor solución encontrada hasta el momento**

RyP realiza de forma iterativa las siguientes etapas:

1. **selección: se extrae de la cola el nodo vivo más prometedor (el de mejor valor de su función de coste)**
   - **será el nodo en expansión o *e-nodo***
2. **ramificación y poda**
   - **se *ramifica* el e-nodo creando cada uno de sus nodos hijo**
   - **se *podan* (descartan) los nodos hijo desde los que no se puede mejorar la mejor solución alcanzada hasta el momento**
   - **los nodos no podados se insertan en la cola de nodos vivos**
3. **se repiten los pasos hasta que:**
   - **la cola se quede vacía**
   - **o el mejor nodo vivo no permita mejorar la mejor solución encontrada hasta el momento**

10.2 Esquema genérico de un algoritmo RyP

Estructuras utilizadas:

- **Clase `Nodo`:**
  - cada nodo (solución parcial) del árbol de búsqueda
  - tiene el métodos `coste` y `compareTo` (basado en `coste`)
- **Cola ordenada `colaNodosVivos`**
  - mantiene los nodos vivos ordenados en función de su coste
  - implementación eficiente con montículo (`PriorityQueue`)
    - inserciones y extracciones `O(log n)`

Algoritmo genérico de un algoritmo RyP:

```
método algoritmoRyP() retorna Nodo

  // mejor solución inicial (con el peor valor posible)
  mejorSol = nuevo Nodo(peores valores posibles)

  // crea el nodo inicial y le pone como primer nodo en expansión
  eNodo = nuevo Nodo(valores iniciales)
```

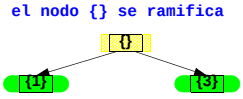


```
  // lazo mientras queden nodos
  hacer
    // ramifica el e-nodo y poda o encola sus hijos
    para cada posibilidad de ramificación hacer
      nHijo = un hijo del eNodo
      // ¿poda?
      si nHijo.coste mejor que mejorSol.coste entonces
         // el nodo NO se poda
         si nHijo es solución entonces
           // actualiza la mejor solución
           mejorSol = nHijo
         sino
           // el nodo no es solución: se encola en base a su coste
           colaNodosVivos.añade(nHijo)
         fsi
      fsi // si (no podar)
    fhacer // ramificación
    // obtiene el siguiente nodo en expansión
    eNodo = colaNodosVivos.extraePrimero()
  mientras eNodo != null y eNodo.coste mejor que mejorSol.coste
  retorna mejorSol
fmétodo
```


10.3 Ejemplo de ramificación y poda

Problema del cambio: monedas={v_1=1,v_2=3} cCambiar=5

el nodo {} se ramifica

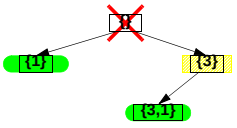


cola nodos vivos: {3} {1}

los nodos se ordenan de menor a mayor "coste"

coste= número de monedas en el nodo + monedas de valor máximo que faltarían para alcanzar cCambiar

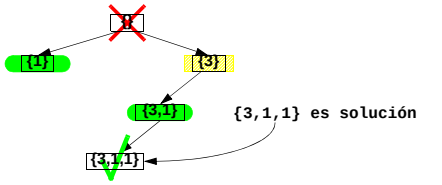
el nodo {3} se extrae de la cola y se realizan sus ramificaciones válidas



cola nodos vivos: {3,1} {1}

Ejemplo de ramificación y poda (cont.)

el nodo {3,1} se extrae de la cola y se realizan sus ramificaciones válidas



cola nodos vivos: {1} 3

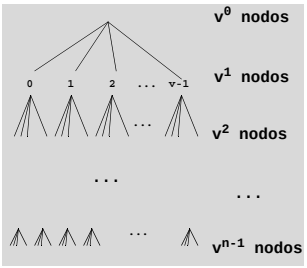
mejor solución encontrada: {3,1,1} 3

el algoritmo ha finalizado, puesto que el primer elemento de la cola de nodos vivos {1} no tiene mejor coste que la mejor solución encontrada (no es posible mejorar la mejor solución encontrada ramificando {1})

10.4 Eficiencia de los algoritmos RyP

Eficiencia temporal de peor caso:

- igual que para VA: $O(v^n)$



Si la función de coste es buena

- el algoritmo se dirige rápidamente a la solución podando muchas ramas
- $O(v^n)$ es un valor muy pesimista
- para obtener un valor promedio más próximo a la realidad es necesario hacer medidas

Eficiencia espacial:

- depende de máxima longitud de la cola de nodos vivos
- difícil de valorar a priori
  - también es necesario hacer medidas

10.5 Laberinto

Se trata de encontrar el camino más corto entre la entrada y la salida de un laberinto

- representado mediante una matriz filas×columnas

El problema era abordable utilizando Vuelta Atrás, por lo que también lo será con RyP

Es necesario definir la función coste de un nodo:

- se calculará como la suma del número de pasos dados hasta él
- más la distancia "Manhattan" desde él hasta la salida
  - es el valor del teórico camino más corto que pasaría por ese nodo
  - quizá no exista tal camino

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-2	-1	0	0	-1
-1	2	-1	0	-1	-1	0	-1	-1	0	-1
-1	3	-1	7	-1	-1	0	0	0	0	-1
-1	4	5	6	0	-1	0	-1	0	-1	-1
-1	-1	0	-1	0	0	0	-1	0	-1	-1
-1	-1	0	-1	0	-1	-1	-1	0	-1	-1
-1	0	0	-1	0	0	0	0	0	0	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

coste=7+2+3=12

Suponemos que en principio no se conoce la situación de la salida

- **todos los nodos tienen el mismo coste:**
  - **búsqueda "a ciegas" (similar a utilizar Vuelta Atrás)**
- **una vez encontrada la salida se puede comenzar a podar y a seleccionar los nodos en base a su coste**

Cada nodo debe contener la información suficiente para saber como se llegó a él y como se puede continuar avanzando:

- **copia del laberinto (con los pasos dados hasta llegar a él)**

El algoritmo va apuntando la mejor solución encontrada hasta el momento (`mejorSol`)

- **se inicializa con un coste infinito**

Pseudocódigo

```
método caminoMasCortoLaberinto(entero xIni,
                entero yIni, entero[][] laberinto)
  mejorSol.coste = ∞ // mejor solución inicial
  // crea el nodo inicial (posición de salida) y le pone
  // como primer e-nodo
  eNodo = nuevo Nodo(laberinto,xIni,yIni,pasos=1)
  // lazo mientras queden nodos
  hacer
    // ramifica el e-nodo (una rama para cada posible
    // movimiento) y poda o encola sus hijos
    para mov en (izq.,arriba,abajo,derecha) hacer
      (xNew, yNew) = movimiento(mov,eNodo.x,eNodo.y)

      si eNodo.laberinto[yNew][xNew] es camino entonces
        // crea el hijo con su propia copia del lab.
        nHijo = nuevo Nodo(eNodo.laberinto,
                           xNew,yNew,eNodo.pasos+1)
```

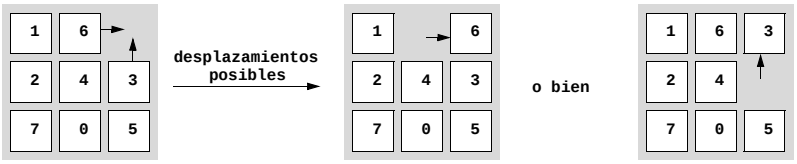


```
        si nHijo.coste < mejorSol.coste entonces
          // el nodo NO se poda
          si nHijo es solución entonces
            si es la primera solución encontrada entonces
              reordena colaNodosVivos en base al coste
            fsi
            // actualiza la mejor solución
            mejorSol = nHijo
          sino // el nodo no es solución
            colaNodosVivos.añade(nHijo)
          fsi
        fsi // si (no podar)
      fhacer // bucle movimientos
      // obtiene el siguiente nodo en expansión
      eNodo = colaNodosVivos.extraePrimero()
    mientras eNodo != null y eNodo.coste > mejorSol.coste
  retorna mejorSol
fmétodo
```

10.6 Puzzle de desplazamiento

Se trata de resolver un puzzle formado por una cuadrícula de `F` filas y `C` columnas que contiene `F×C-1` piezas y un hueco

- **las piezas se pueden desplazar para ocupar el hueco**



Podemos intentar aplicar RyP ya que:

- **la solución se expresa como una secuencia de decisiones**
  - **cada uno de los desplazamientos**
- **para cada decisión se elige entre un conjunto finito de valores**
  - **cada una de las fichas que pueden ocupar el hueco**

El puzzle se representa mediante una matriz de `F×C` que tiene un número del `0` al `F×C-2` en cada posición o un `-1` en el hueco

- **estará ordenado cuando toda casilla `(f,c)` (`0≤f<F` y `0≤c<C`) contenga el valor `f·C+c`**
- **salvo la casilla `(F-1,C-1)` que contendrá el valor `-1`**

```
{ 5,12,-1,14},
{ 1, 9,10, 2},
{ 8, 2,11, 3},
{13, 0, 6, 7}
```
puzzle desordenado

```
{ 0, 1, 2, 3},
{ 4, 5, 6, 7},
{ 8, 9,10,11},
{12,13,14,-1}
```
puzzle ordenado

Trataremos de encontrar una secuencia de desplazamientos que conduzca a la solución

- **no importa que no sea la secuencia más corta posible**

Algoritmo

Cada nodo debe contener la información suficiente para determinar su estado y como se llegó a él:

- **copia del puzzle en la situación correspondiente al nodo**
- **la `historia` de los pasos dados hasta llegar a él**

Los nodos vivos se encolan en la `colaNodosVivos`:

- **ordenados de menor a mayor coste**

La ramificación del e-nodo la realiza el método de la clase `Nodo`:

```
Nodo hijo(Movimiento mov)
```

- **retorna null cuando el movimiento se sale del puzzle**
- **si el movimiento es válido el nodo hijo retornado tiene:**
  - **una copia de la historia del padre a la que se ha añadido él mismo**
  - **una copia del puzzle en la situación correspondiente al nodo**

En `colaNodosRecorridos` se van almacenando todas las posiciones alcanzadas

- **para no repetir ninguna**
- **evitando, de esta forma, entrar en un bucle en el que se realice constantemente la misma secuencia de decisiones**

El algoritmo finaliza al encontrar el primer nodo que contiene todas las piezas en su lugar

- **la solución es la secuencia de movimientos hasta ese nodo (almacenada en su `historia`)**

No se realiza poda de nudos:

- **se podría realizar si modificáramos el algoritmo para que buscara la solución óptima**

La clave de que el algoritmo encuentre la solución en un tiempo aceptable estará en la elección de la función `coste`.

Pseudocódigo del algoritmo

```
// método que resuelve el puzzle de desplazamiento
// utilizando la estrategia de RyP
// recibe el puzzle en su configuración original
// retorna una secuencia de movimientos para
// resolverle o una lista vacía cuando no hay
// solución
método resuelvePuzzle(entero[][] puzzle)
                              retorna ListaMovimientos
  // crea el nodo inicial y le encola
  nodoInicial = nuevo Nodo(puzzle)
  colaNodosVivos.añadeEnOrden(nodoInicial)
  colaNodosRecorridos.añade(nodoInicial)

  // mientras que queden nodos vivos
  mientras colaNodosVivos no está vacía hacer
    // extrae el nodo vivo más prometedor
    eNodo=colaNodosVivos.extraePrimero()
```



```
    // ramifica el e-nodo y encola sus hijos
    para mov en (izq.,arriba,abajo,derecha) hacer
      nHijo=eNodo.hijo(mov) // un hijo de eNodo
      si nHijo != null y
            nHijo no en colaNodosRecorridos entonces
        si nHijo.coste()==0 entonces
          retorna nHijo.historia// nHijo es solución
        sino // no es solución
          // encola nHijo
          colaNodosVivos.añadeEnOrden(nHijo)
          colaNodosRecorridos.añade(nHijo)
        fsi
      fsi
    fhacer // movimientos
  fhacer // mientras colaNodosVivos no está vacía
  retorna cola vacía // no hay solución
fmétodo
```

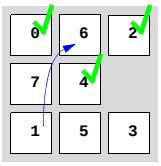
Elección de la función `coste`

La elección de una función `coste` apropiada es la clave para que el algoritmo encuentre la solución en un tiempo aceptable

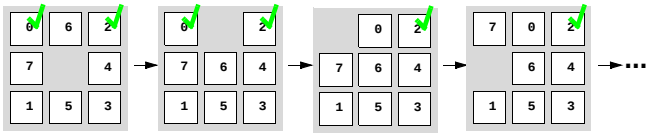
Opciones sencillas para utilizar como función coste:

- **número de fichas que faltan por colocar**
- **distancia manhattan de todas las fichas hasta su posición final**
- **ambas son inapropiadas, porque para resolver algunas situaciones hay que pasar por configuraciones de coste mayor:**

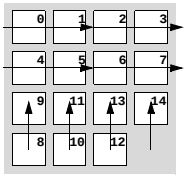
`para resolver:`



`hay que pasar por:`

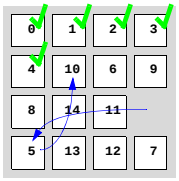


Trataremos de buscar una función `coste` que provoque que el puzzle se vaya completando en el orden adecuado



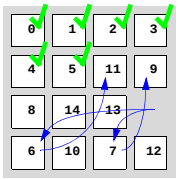
Función `coste`:

- **para todas las filas menos las dos últimas:**



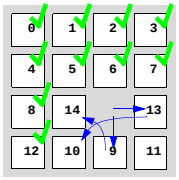
```
coste=11                (casillas que faltan por colocar)
      +(2+1)/100        (distancia Manhattan de la ficha 5 a su posición final)
      +(1+3)/100        (distancia Manhattan del hueco a la ficha 5)
```


- **para las dos últimas columnas de cada fila:**



```
coste=10                (casillas que faltan por colocar)
      +(2+2)/100        (distancia Manhattan de la ficha 6 a su posición final)
      +(2+1)/100        (distancia Manhattan de la ficha 7 a su posición final)
      +(1+3 + 1+1)/100  (distancia del hueco a las fichas 6 y 7)
```

- **las dos últimas filas se rellenan columna a columna:**



```
coste=6                 (casillas que faltan por colocar)
      +(1+1)/100        (distancia Manhattan de la ficha 9 a su posición final)
      +(1+2)/100        (distancia Manhattan de la ficha 13 a su posición final)
      +(1+1 + 1+1)/100  (distancia del hueco a las fichas 9 y 13)
```

10.7 El problema de la asignación

Un tipo de problemas clásicos que se resuelven por VA o por RyP son los *problemas de asignación*

- **se trata de asignar de forma óptima los elementos de un conjunto con los de otro de forma que se maximice o minimice una función**

Como ejemplo resolveremos el problema de la *asignación óptima de compañeros*

- **ya le hemos resuelto con un heurístico y por VA**

Asignación óptima de compañeros

Una empresa de reparación de averías telefónicas debe dividir sus operarios por parejas para atender a los trabajos que le encargan

La dirección de la empresa considera fundamental que los componentes de una pareja se lleven lo mejor posible

- **por lo que solicita a cada operario que indique su afinidad con cada uno de sus compañeros**
- **la afinidad se indica de forma numérica, siendo 100 la máxima afinidad y 0 la mínima**

Con los resultados obtenidos la dirección de la empresa construye la matriz de afinidades

- **Ejemplo de matriz de `afinidades` para una empresa con 4 operarios:**

	0	**1**	**2**	**3**
0	0	85	88	47
1	13	0	54	4
2	34	6	0	78
3	48	69	73	0

- **`afinidades[i][j]` indica la afinidad indicada por el operario `i` con respecto al operario `j`**
- **la afinidad de la pareja `(i,j)` será la suma de las afinidades indicadas por ambos operarios: `afinidades[i][j]+afinidades[j][i]`**

Se trata de conseguir una asignación de parejas que maximice la afinidad total

- **en el caso anterior esta asignación óptima es la formada por las parejas `(0,1)` y `(2,3)`**

```
afinidad de (0,1) = afinidades[0][1]+afinidades[1][0]
                  = 85+13 = 98
afinidad de (2,3) = afinidades[2][3]+afinidades[3][2]
                  = 78+73 = 151
afinidad total = 98 + 151 = 249
```


Solución con un algoritmo de RyP

Utilizaremos la clase `SolParcialParejas`

- **Cada una de las soluciones parciales (nodo del espacio de búsqueda del problema)**
- **Con los métodos: `coste()`, `compareTo()` e `hijo()`**

Pseudocódigo algoritmo RyP:

```
método asignaParejasRyP(int[][] afinidades) retorna SolParcialParejas

   // Crea la cola de nodos vivos: cola de prioridad en base al coste
   colaNodosVivos = nueva Cola de prioridad de SolParcialParejas

   // mejor solución inicial (ninguna pareja formada)
   mejorSol = nuevo SolParcialParejas()

   // el primer e-nodo es una solución sin ninguna pareja formada
   eNodo = nuevo SolParcialParejas()
```



```
   // lazo mientras queden nodos
   hacer
     // crea cada uno de los posibles hijos: crea una pareja
     // para cada una de las personas que aún están libres
     para cada persona p hacer
       nHijo = eNodo.hijo(p)
       // ¿poda?
       si nHijo.coste mejor que mejorSol.coste entonces
          // el nodo NO se poda
          si en nHijo todas las personas tienen pareja entonces
            // nHijo es solución así que pasa a ser la mejor solución
            mejorSol=nHijo.copia()
          sino
            // el nodo no es solución: se encola en base a su coste
            colaNodosVivos.añade(nHijo)
          fsi
       fsi // si (no podar)
     fhacer // ramificación
     // obtiene el siguiente nodo en expansión
     eNodo = colaNodosVivos.extraePrimero()
   mientras eNodo != null y eNodo.coste mejor que mejorSol.coste
   retorna mejorSol
fmétodo
```

Distintas alternativas para la función de coste

Se trata de buscar una cota superior a la afinidad máxima que se puede lograr desde una solución parcial

- **tenemos que estimar la afinidad que se puede lograr con las parejas que faltan por hacer**

- **Función sencilla: "Parejas Perfectas"**
  - **Se supone que todas las parejas que faltan por hacer van a ser "perfectas" (afinidad máxima: `100*2 = 200`)**

```
Coste = afinidad de las parejas ya hechas
        + parejas que faltan por hacer * 200
```


- **Una aproximación mejor: "Mejores Parejas"**
  - **Se precalcula la afinidad de todas las posibles parejas y se ordenan de mejor a peor**
    - **En el ejemplo anterior sería:**

```
(2,3)->151, (0,2)->122, (0,1)->98, (0,3)->95,
(1,3)->73, (1,2)->60
```

  - **Se crea una tabla con la afinidad acumulada de la mejor pareja, las dos mejores, las tres mejores, ...**

```
costeMejores = [151, 273, 371, 466, 539, 599]
```

  - **El coste se calcula como:**

```
Coste = afinidad de las parejas ya hechas
     + costeMejores[parejas que faltan por hacer]
```


Uso de un heurístico en el algoritmo RyP

Podemos utilizar el algoritmo heurístico como primera solución

- **de esta forma desde el principio podaremos los nodos “muy malos”**

```
método asignaParejasRyP(int[][] afinidades) retorna SolParcialParejas

  // Crea la cola de nodos vivos: cola de prioridad en base al coste
  colaNodosVivos = nueva Cola de prioridad de SolParcialParejas

  // mejor solución inicial (ninguna pareja formada)
  mejorSol = asignaParejasHeurístico()

  // el primer e-nodo es una solución sin ninguna pareja formada
  eNodo = nuevo SolParcialParejas()

  // lazo mientras queden nodos
  hacer
    ...
    ...
```

10.8 Bibliografía

[1] **Brassard G., Bratley P., *Fundamentos de algoritmia*. Prentice Hall, 1997.**

[2] **Aho A.V., Hopcroft J.E., Ullman J.D., *Estructuras de datos y algoritmos*. Addison-Wesley, 1988.**

[3] **Sartaj Sahni, *Data Structures, Algoriths, and Applications in Java*. Mc Graw Hill, 2000**
El capítulo sobre Ramificación y Poda está en la web:
http://www.cise.ufl.edu/~sahni/dsaaj/