RAMIFICACIÓN Y PODA (Branch and Bound)

- Técnica de **exploración de grafos** dirigidos implícitos
- El grafo será acíclico, un **árbol**
- Está asociada a **problemas de optimización**
- Es una **variante** del diseño "***vuelta atrás***" (VA)
- Al igual que VA realiza una enumeración parcial del espacio de soluciones basándose en la generación de un árbol de expansión

RAMIFICACIÓN Y PODA

Diferencia con VA:

- posibilidad de generar nodos siguiendo diferentes estrategias
- VA: genera descendientes mediante un recorrido en profundidad

Generación de nodos en Ramificación y Poda (RP):

- Siguiendo un recorrido en **anchura**
- Siguiendo un recorrido en **profundidad**
- Utilizando el cálculo de **funciones de coste** para seleccionar el nodo que en principio parece más prometedor

RAMIFICACIÓN Y PODA

- **Cada arista** de grafo (árbol) tendrá asociado un **coste**
- RP utiliza **cotas** para podar aquellas ramas del árbol que no conducen a la solución óptima
- En cada nodo se calcula una cota del valor de las soluciones que se encontraran generando sucesores
- Si la cota indica que las soluciones a encontrar son peores que la mejor que tenemos no se sigue explorando esa rama (proceso de **poda**)
- La cota se utiliza para seleccionar el camino más prometedor

RAMIFICACIÓN Y PODA

***Nodo vivo*: un nodo con posibilidades de ser ramificado, no ha sido podado**

- Para determinar qué nodo va a ser expandido necesitaremos almacenar todos los ***nodos vivos*** en una **estructura de datos**
- La estructura variará según la estrategia de búsqueda o generación de nodos

RAMIFICACIÓN Y PODA

Recorrido en anchura o amplitud

- Los nodos se exploran en el mismo orden en que se van creando
- Una estructura de datos *cola* (FIFO) almacena los nodos creados y aún no explorados

Recorrido en profundidad

- Los nodos se exploran en el orden inverso al de su creación
- Una estructura de datos *pila* (LIFO) almacena los nodos creados y aún no explorados

RAMIFICACIÓN Y PODA

Recorrido en función del coste

- La función de coste permite decidir cuál de nodos vivos es más prometedor
- Una estructura de datos ***lista de prioridad*** almacena los nodos vivos ordenados por su coste

Implementación de una lista dinámica de prioridad

- Un montículo es la estructura de datos más adecuada
- Como trabajamos con **costes** deberá ser un **montículo de mínimos**, en la raíz estará el nodo cuya cota sea inferior

RAMIFICACIÓN Y PODA

Etapas de un algoritmo de RP

1.- Selección

- extrae un nodo del conjunto de nodos vivos
- depende de la estrategia de búsqueda empleada

2.- Ramificación

- se construyen los posibles nodos hijos del nodo seleccionado en el paso anterior

3.- Poda

- se eliminan algunos de los nodos creados en la etapa anterior

RAMIFICACIÓN Y PODA

Etapas de un algoritmo de RP

- La poda contribuye a reducir en lo posible el espacio de búsqueda, atenuando la complejidad
- Los nodos no podados pasan a formar parte del conjunto de *nodos vivos*
- Se repiten las 3 etapas hasta que finaliza el algoritmo
- El algoritmo para cuando:
  - encuentra la solución
  - se agota el conjunto de *nodos vivos*

RAMIFICACIÓN Y PODA

- Dado un nodo, una función de coste debe estimar el valor de la solución si siguiéramos por ese camino
- Si una cota del coste de una solución (que será mejor que la solución real) es peor que una solución ya obtenida por otra rama, podemos podar la rama de la cota ya que no interesa seguir por ella
- No se debe podar hasta haber encontrado una solución
- Las funciones de coste deben ser crecientes con respecto a la profundidad del árbol

  si h es una función de coste, $h(n) \leq h(n')$, para todo n' descendiente de n

RAMIFICACIÓN Y PODA

Dificultad habitual: encontrar una buena función de coste
- que garantice la poda
- su cálculo no sea muy costoso

- Antes de podar, debemos encontrar una solución
- Si no se quiere esperar a encontrar una solución se puede utilizar un algoritmo voraz que aunque no encuentre una solución óptima, puede encontrar una cercana a la óptima

Cada nodo debe contener toda la información necesaria para la ramificación, la poda y para reconstruir la solución encontrada hasta ese momento

RAMIFICACIÓN Y PODA

Ventajas de RP:

- Posibilidad de disponer distintas estrategias de exploración del árbol y de acotar la búsqueda de la solución (eficiencia)
- Posibilidad de ejecución en paralelo (un proceso para cada nodo vivo)

RAMIFICACIÓN Y PODA

Desventaja de RP:

- Requiere más memoria que los algoritmos de VA (un nodo debe contener información completa)

PROBLEMA DE LA ASIGNACIÓN

Problema: Se desean asignar *n* tareas a *n* agentes de forma que cada uno realice una tarea y se minimice el coste total de ejecutar las *n* tareas

Si al agente *i* con $1 \leq i \leq n$ se le asigna la tarea *j* con $1 \leq j \leq n$, el coste será : $c_{i,j}$

Los costes se representarán mediante una matriz de costes

PROBLEMA DE LA ASIGNACIÓN

Ejemplo: Tenemos 3 agentes (a,b,c) y 3 tareas (1,2,3)

	1	2	3
a	4	7	3
b	2	6	1
c	3	9	4

¿ Por qué no se emplea el esquema voraz?

No hay una función de selección que garantice obtener la solución óptima sin tener que cambiar decisiones

PROBLEMA DE LA ASIGNACIÓN

	1	2	3
a	4	7	3
b	2	6	1
c	3	9	4

Func. Sele.: mínimo coste para cada agente: 3+2+9=14

Func. Sele.: " " por tarea : 2+7+4=14

No son soluciones óptimas hay otra de menor coste:

7+1+3= 11 (a-2, b-3, c-1)

Las posibles asignaciones son *n*!

PROBLEMA DE LA ASIGNACIÓN

Sea la siguiente una matriz de costes para el problema de la asignación:

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

Estructuras de datos necesarias:

- Una matriz de costes
- Un tipo registro para implementar los nodos (asignaciones realizadas, coste …)
- Un montículo de mínimos con los nodos vivos
- Una lista o pila para almacenar los nodos sucesores del que se está explorando (compleciones del nodo)

PROBLEMA DE LA ASIGNACIÓN

Primero se necesita obtener una **cota superior del coste** de la solución = **primera solución**

- Se eligen arbitrariamente las asignaciones
  Por ejemplo:se seleccionan las que corresponden a la diagonal principal

  a →1, b →2, c →3, d →4

  11 + 15 + 19 + 28= 73

  solución óptima ≤ cota calculada

- Se va creando el árbol:

  raíz: no se ha realizado ninguna asignación

  nivel *k*: se han asignado a *k* agentes las tareas

- En cada nivel se realiza la asignación de un agente

PROBLEMA DE LA ASIGNACIÓN

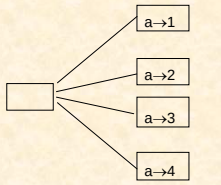
- Para cada nodo se calcula una cota del coste de las soluciones partiendo de la asignación parcial realizada

Cota → da idea del coste de la solución a partir de la asignación parcial

- Dichas cotas dirigirán la exploración del árbol y pueden determinar la eliminación de caminos (poda)
- En nuestro ejemplo para calcular una cota en el nivel *k* vamos a tener en cuenta las tareas con menor coste de las no asignadas en los niveles *k*-1, *k*-2, ..1

PROBLEMA DE LA ASIGNACIÓN

Recordamos que nuestra cota superior es 73



60 (**11**+14+13+22)

58 (11+**12**+13+22)

65 (11+14+**18**+22)

78 (11+14+13+**40**) *

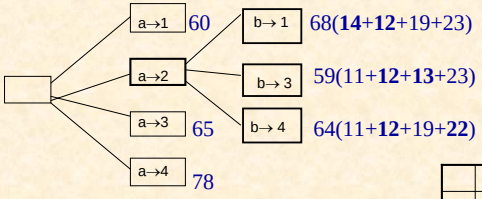
	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

Se obtienen cotas mínimas del coste de la solución

* 78 > cota superior ⇒ no se explorará, no será óptima

PROBLEMA DE LA ASIGNACIÓN

A continuación se explora el nodo más prometedor: cota 58

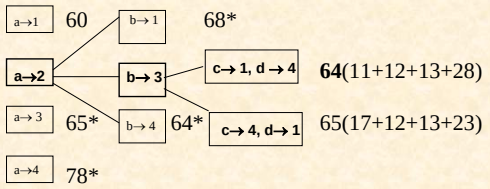


El nodo más prometedor es el de cota 59

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

PROBLEMA DE LA ASIGNACIÓN

A continuación se explora el nodo más prometedor: cota 59

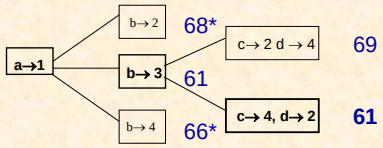


La solución encontrada será nuestra nueva cota superior: 64

Descartamos los nodos a→3 y a→4 > cota superior y el resto con "*"

PROBLEMA DE LA ASIGNACIÓN

a→1 tiene una cota más prometedora que la obtenida y se sigue la exploración a partir de él



- Se encuentra la solución óptima
- Se ha reducido el coste que supondría una exploración completa

PROBLEMA DE LA ASIGNACIÓN

- El coste real de esta técnica depende de la calidad de la primera cota seleccionada
- Si la cota es buena:
  - se examinan menos nodos
  - la solución se encontrará en menos pasos
- En el caso peor, una cota buena puede que no nos permita podar muchas ramas del árbol
- Los cálculos asociados a la obtención de las cotas tienen un coste
- En general suele ser rentable obtener una buena cota

PROBLEMA DE LA ASIGNACIÓN

```
Funcion ramificacionYPoda () dev nodo;
  m ← monticuloVacio();
  cotaSuperior ← cotaSuperiorInicial;
  solucion ← primeraSolucion;
  nodoInicial(m, nodo); {raíz del árbol} {puede ser generar el nivel 1}
  Mientras ¬ vacio(m) hacer
  n ← extraerRaiz(m);
  Si valido(n) entonces {están completas las asignaciones }
    Si costeAsig(n) < cotaSuperior entonces
        solucion ← n;
        cotaSuperior ← costeAsig(n)
    FSi
  Sino
    Si cotaInferior(n) ≥ cotaSuperior entonces dev solucion
    Sino
        Para cada hijo en compleciones(n) hacer {para todos los sucesores}
            Si cotaInferior(hijo) < cotaSuperior
                añadirNodo(m, hijo)
            FSi
        FPara
    FSi
  FSi FMientras
FFuncion
```

PROBLEMA DE LA ASIGNACIÓN

- En la descripción del esquema no se ha tenido en cuenta la estructura de registro de *nodo* a la hora de hacer referencia a sus campos
- Habría que especificar las funciones:
  - compleciones()
  - cotaInferior()
  - valido()
- En otros problemas habría que añadir una función *condicionesPoda*() que permitiría determinar si un nodo es prometedor o no antes de añadirlo al montículo

PROBLEMA DE LA ASIGNACIÓN

Estructuras de datos necesarias:

1.- Una **matriz de costes**

2.- Un **tipo registro** para implementar los **nodos** (asignaciones realizadas, coste …)

3.- Un **montículo de mínimos** con los **nodos**

4.-Una lista o **pila** para almacenar los nodos sucesores a partir del que se está explorando (**compleciones** del **nodo**)

PROBLEMA DE LA ASIGNACIÓN

2.- La implementación de los nodos podría ser:

nodo = **registro**
  asignaciones: **vector**[1..4] de **natural**
  costeAsig: **natural**
  tareasSinAsignar: **lista/pila de natural**
  ultimoAgenteAsignado: **natural**

donde *asignaciones*[i]: la tarea;
i: agente

PROBLEMA DE LA ASIGNACIÓN

3.- Un montículo de mínimos con los nodos que serían de tipo *nodo*

```
Procedimiento hundir(T[1..n], i)  {hunde el nodo i}
k ← i
Repetir
   j ← k
   Si 2j ≤ n  y  T[2j] > T[k] entonces
         k ← 2j
   FSi
   Si 2j < n  y  T[2j + 1] > T[k] entonces
         k ← 2j + 1
    FSi
   intercambiar T[j]  y T[k]
Hasta   j=k
```

PROBLEMA DE LA ASIGNACIÓN

4.- Una lista o pila para almacenar las **compleciones** del nodo

Compleciones: los nodos que se generan (sucesores) a partir de uno dado, consistirá en:

- realizar una asignación más a partir de una asignación parcial con el fin de obtener una cota inferior

- para ello se utiliza la lista *tareasSinAsignar*

PROBLEMA DE LA ASIGNACIÓN

```
Funcion compleciones(e:nodo) dev lista de nodo
  listaCompleciones ← listaVacía()
  Para cada tarea en e.tareasSinAsignar hacer
     complecion ← e
     agente ← e.ultimoAgenteAsignado +1
     complecion.asignaciones[agente] ← tarea
     complecion.costeAsig ← complecion.costeAsig +
                            matrizCostes[tarea, agente]
     complecion.tareasSinAsignar ←
        eliminar(tarea, complecion.tareasSinAsignar)
     complecion.ultimoAgenteAsignado ← agente
     complecion.cotaInf ← cotaInferior(complecion)
     listaCompleciones←añadir(complecion, listaCompleciones)
  FPara
dev lista_compleciones
FFuncion
```


PROBLEMA DE LA ASIGNACIÓN

Cálculo de una cota inferior:

```
Funcion cotaInferior (e:nodo) dev natural
  vectorMinimos ← [11, 12, 13, 22]
  Para cada tarea en e.tareasSinAsignar hacer
    e.coste_asig ←  e.coste_asig +
                    vectorMinimos[tarea]
  Fpara
  dev e.coste_asig
FFuncion
```

Función que determina si un nodo es válido

```
Fun valido(e:nodo) dev booleano
   dev e.ultimoAgenteAsignado = 4
FFuncion
```

PROBLEMA DE LA ASIGNACIÓN

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

c	M	L	cota-sup	nodo
Inicio	**58,60, 65,78**	**Ø**	**73**	
1	**59,60,64,65,68,78,**	**68, 59, 64**		**58**
2	**60,64,64,65,65,68,78**	**64,65**		**59**
3	**61,64,64,65,65,,66,68 ,68,78**	**68,61,66,**		**60**
4	**61,64,64,65,65,,66,68 ,68, 69,78**	**69,61**		**61**
5	**64,64,65,65,66,68,68, 69,78**		**61**	**61**
6				**64**

La solución sería el nodo con coste total 61