Universidad Rey Juan Carlos

ETSII Escuela Técnica Superior de Ingeniería Informática

Algoritmos Avanzados

-Práctica 3b-

Técnicas de búsqueda
Ramificación y Poda

Curso 2022/2023

GII

Alberto Radlowski Nova
Pablo Rodea Piornedo

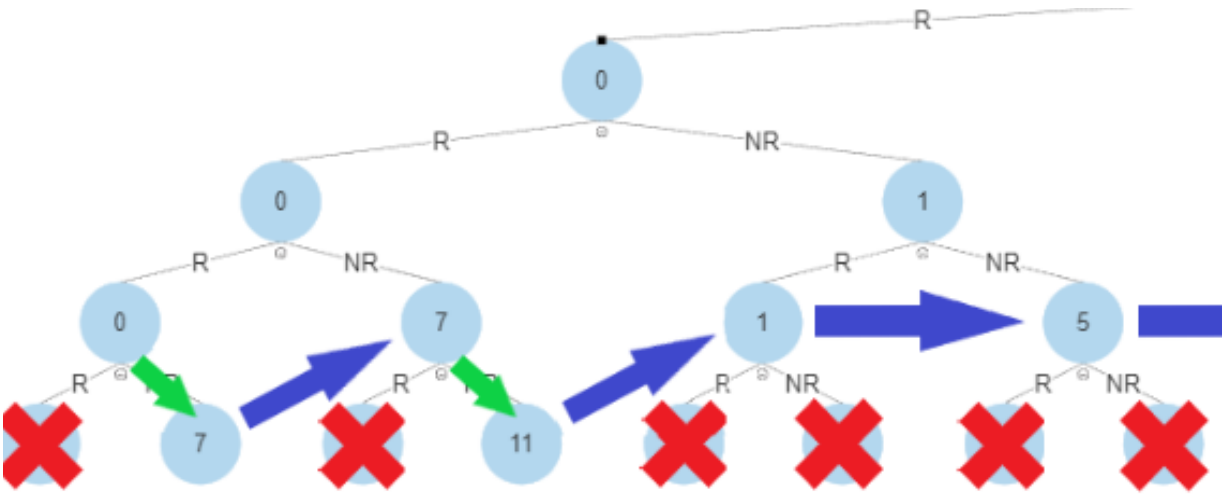
Técnica de ramificación y poda

Función de cota

- **Cota inicial:** suma de todos los valores de ds.

- **Actualización:** la cota se calcula en cada nodo sumando la solActual del nodo + el beneficio estimado, este beneficio viene dado por la suma de la siguiente posición de ds + cs actual -> (ds[etapa+1]+cs[índice]).

Esquema de poda de árbol nuevo (gyazo)
Árbol sin podar (por si se quiere comparar)



Explicación de la cota y poda atendiendo a la figura:

Tenemos 2 criterios de poda:

- El primero consiste en podar todos los nodos hoja que sean hijos izquierdos ya que conforman la opción de reiniciar en el último paso. Reiniciar en el último paso daría el mismo resultado que el nodo padre por lo que podamos este caso y reducimos el número de llamadas recursivas.

  Un ejemplo de aplicación sería con el nodo hoja = 0 más a la izquierda del árbol cuyo nodo padre también es un 0.
  Este nodo hoja representa la rama del reinicio en la ubicación ds[ds.length-1], como vemos esta rama nunca nos dará una mejor solución que el nodo padre por lo que procedemos a podarla directamente y saltamos a comprobar el hijo derecho en este caso el nodo hoja con valor = 7.

- El segundo criterio de poda consiste en la posibilidad de podar también los nodos hoja derechos, para ello se calcula si haciendo la suma del **valor del nodo padre + Math.min(ds[etapa+1], cs[indice])** es mayor que la cota actual.
  Etapa+1 es el valor que tendría el índice de **ds** en la siguiente iteración de la recursividad.

  En caso de que el valor calculado sea mayor a la cota actual, la recursividad continua sin podar el nodo hoja derecho y la cota se actualiza al valor calculado del nodo hoja derecho.

  En el caso de que la cota sea actualmente mayor que el beneficio calculado se poda la rama del nodo hoja hijo derecho y se sube por backtracking al nodo padre y continua la recursividad.

  Fijándonos en la figura, un **ejemplo de NO poda** sería en el **nodo 7** en la altura penúltima cuyo nodo hoja hijo derecho es el 11, no se poda el hijo derecho porque su beneficio calculado es:
  7 + Math.min(ds[3], cs[1]) = 7+4 = 11. El 11 es mayor a la cota anterior con valor 7 (la cota anterior viene determinada por el nodo hoja derecho del 0 más a la izquierda del árbol cuyo hijo derecho tiene un valor de 7)

Ahora un **ejemplo de Poda efectiva,** como se observa en la figura el nodo en la penúltima altura con **valor = 1** tiene los dos hijos podados. El nodo hoja izquierdo está podado según el criterio 1(valor 1 igual al nodo padre).
El nodo hoja derecho sigue el criterio 2 de poda, para ello se comprueba que **1 + Math.min(ds[etapa+1], cs[indice]) = 1+7 = 8**. Como vemos 8 < 11(cota anterior) por lo que se poda el nodo hoja derecho estableciendo el boolean podar a TRUE y evitando que entre la recursividad por la rama de `if((etapa == ds.length-1 && i == 1) && (!podar))` que es donde se calcula el valor del nodo hoja.

Código del algoritmo

```
public static int procesar4 (int[] ds, int[] cs){
   int [] combinaciones = new int[ds.length];
   int cota = 0;
   //Cota inicial = suma de todos los elementos de
ds(máximo posible irreal)
   for(int i = 0; i < ds.length; i++){
       cota += ds[i];
   }
   //Llama al método auxiliar
   return aux4(ds, cs, 0, combinaciones, 0, cota, false);
}
```

```
public static int[] comparar(int[] ds, int [] cs, int []
combinaciones){

   int [] resultado = {0,0};
   int indice = 0;
   //Se comprueba si la combinación del subArbol actual
es mejor que la solución óptima hallada hasta ahora
   for(int j=0; j<cs.length; j++){
       if(combinaciones[j] == 1 ){
```

```
            resultado[0] += Math.min(ds[j], cs[indice]);
            indice++;
        }else if(combinaciones[j] == 0){
            indice = 0;
        }
    }
    resultado[1] = indice;
    return resultado;
}
```

```
public static int aux4(int[] ds, int[] cs, int etapa, int []
combinaciones, int solSubArbol, int cota, boolean podar){
   //Arbol binario, en cada etapa se puede reiniciar=0 o seguir=1 el
array de cs
   for(int i=0; i<=1; i++){

       //Array que guarda de forma temporal la solución del subarbol
y se actualiza en cada iteración
       combinaciones[etapa] = i;

       if(etapa == ds.length-2){
           //resultado[0] = solucion del subarbol, resultado[1] =
indice de cs
           int [] resultado = comparar(ds, cs, combinaciones);

           //Si la sol de la penúltima etapa es <= a la (solActual +
beneficioEstimado) -> COTA, la rama se poda
           cota = resultado[0]+Math.min(ds[etapa+1],
cs[resultado[1]]);
           podar = false;
           //Se poda si la mejor solución posible por esa rama(cota)
es menor a solSubArbol
           if(cota <= solSubArbol){
              podar = true;
           }

       }
       //Si no se poda significa que la cota es mayor a la solución
óptima hallada hasta ahora
       //i==1 para podar todas las ramas de reinicio a los nodos
hoja(nunca va a ser mejor reiniciar en la última etapa)
       if((etapa == ds.length-1 && i == 1) && (!podar)){
```

```
            solSubArbol = cota;
        }
        //Continua el backtracking si no se ha podado la rama actual
y no ha llegado al final de ds
        else if (etapa < ds.length-1 && !podar){
            //Guarda en solSubArbol la solución de la hoja
            solSubArbol = aux4(ds, cs, etapa+1, combinaciones,
solSubArbol, cota, false);
        }
    }
    //Marca que la combinación ha sido evaluada
    combinaciones[etapa] = -1;

    //Devuelve el resultado del subarbol
    return solSubArbol;}
```

Comparación de optimalidad

Material del experimento

- **Algoritmo “Procesar”** es el algoritmo propuesto por el profesor el cual reinicia el servidor en el momento que cs[índice] < ds[i].

- **Algoritmo “Procesar2”** en este caso es un algoritmo más optimizado ya que calcula las medias de cs y ds y solo reinicia cuando se queda por debajo de estos valores.

- **Algoritmo “Procesar3”** aquí hemos utilizado backtracking para resolver un árbol binario de búsqueda, para ello, se comprueban todas las combinaciones posibles hasta llegar a un nodo hoja y va arrastrando el valor hacia arriba de la llamada recursiva actualizándose en caso de encontrar una combinación mejor.

  Estas combinaciones están definidas en el array “combinaciones” cuyo índice “i” se actualiza a 1 si se está probando esa combinación (sin reinicio), a 0 si ha reiniciado y a -1 si ya la ha probado y está volviendo a subir por la rama del árbol.

- **Algoritmo “Procesar4”** , este algoritmo hace uso de dos métodos auxiliares, **“aux4”** y **“comparar”, “comparar”** se utiliza para calcular el valor de los nodos hoja según las combinaciones de 1’s y 0’s que haya en el array combinaciones.
  Cabe destacar que **el valor de la solución no se calcula hasta el nodo hoja**, simplemente se lleva la cuenta de la combinación de la rama escogida, es decir en el nodo hoja con combinación [1,0,1,0] significa que en el primer paso NO se ha reiniciado, en el segundo SI, el tercero NO y el último SI

  “Comparar” atendiendo al array combinaciones y a los valores correspondiente a estos índices de ds y cs calcula el valor del nodo aplicando los reinicios en caso de combinaciones[j] == 0 o sumando el valor actual + Math.min(ds[etapa] + cs[indice]) en caso combinaciones[j] == 1.

  Este método también se usa para comprobar si se poda el nodo hoja derecho, cuyo valor constituye el valor del nodo padre que es comparado con valor del nodo hijo derecho en etapa+1 y la actualización de la cota si corresponde.
  Esto se realiza en el método **“aux4”** en la rama del if(etapa==ds.length-2).

  Respecto al método **“aux4”** es el método que hace el backtracking en sí apoyándose en el método anterior **“comparar”.**
  El for de 0 a 1 sigue la estructura de un árbol binario.

  El `else if (etapa < ds.length-1 && !podar)` sirve para bajar de forma recursiva hasta la **altura = ds.length-2** en la que en esa iteración entra por la rama del if primero.
  En esta rama del if, **“**`if(etapa == ds.length-2)`**”** se comprueba si está en la penúltima altura y en caso afirmativo se encarga de podar los nodos hoja derechos según el criterio expuesto anteriormente.
  El boolean “podar” se establecerá a TRUE en caso de poda y por tanto saldrá de esa rama de la recursividad sin llegar a la altura del nodo hoja.

El otro if, “`if((etapa == ds.length-1 && i == 1) && (!podar)`”
el **i == 1** sirve para entrar por la recursividad en la última altura solo en el caso de los nodos hoja derechos, ya que según el criterio 1 de poda anteriormente expuesto, se podan todas las ramas de reinicio en los nodos hoja (hijos izquierdos con valor i = 0).
Si además de estar en un nodo hoja derecho (NO se reinicia en el último paso), podar es FALSE, es decir **“comparar”** en el if anterior ha determinado que el resultado de ese nodo va ser mejor que la cota actual, entonces se establece que la solución del subArbol mejor hasta la fecha es = a la cota calculada en el if anterior(ha actualizado la cota a la suma de etapa+1), se iguala directamente a la cota para no tener que llamar de nuevo a **“comparar”** y hacer 2 veces la suma.
Por último sube por recursividad con
`else if (etapa < ds.length-1 && !podar)` al nodo padre actualizando la etapa y el array de combinaciones.
En combinaciones la posición “i” se actualiza a 1 si se está probando esa combinación (sin reinicio), a 0 si ha reiniciado y a -1 si ya la ha probado y está volviendo a subir por la rama del árbol.

Rangos de valores: Tamaño **“cs”** y **“ds”** = 15, **ds** con valores repetidos sin orden, y **cs** con valores repetidos ordenados de forma descendente, número de ejemplos = 1000 y rango de valores de 0 a 30.

Conclusión

- **“Procesar”** y **“Procesar2”** son algoritmos heurísticos, por tanto son inexactos ya que no realizan una búsqueda exhaustiva de todas las combinaciones posibles si no que intentan acercarse a la solución óptima mediante una aproximación, en el caso de “Procesar2” mediante la media.

- **“Procesar3”** en cambio sí que es un algoritmo exacto ya que mediante el uso de backtracking prueba todas las combinaciones posibles y termina hallando la solución óptima en el 100% de los casos.

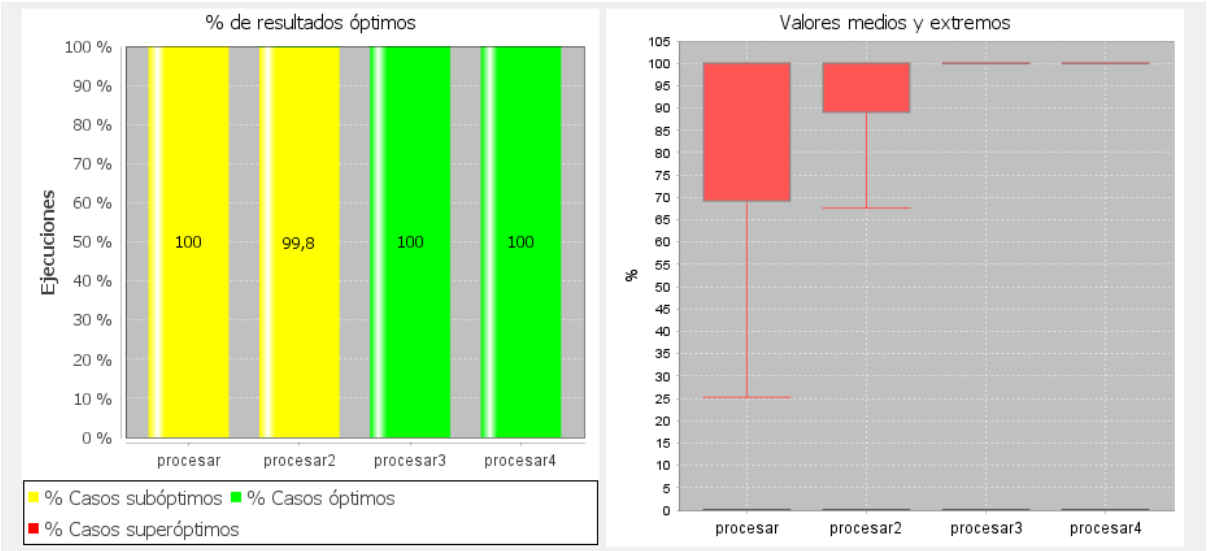
- **“Procesar4”** también es un algoritmo exacto ya que es una optimización de “Procesar3” mediante poda pero sigue usando backtracking por lo que hallará la solución óptima en el 100% de los casos igualmente.

Evidencias

Medidas	procesar	procesar2	procesar3	procesar4
Núm. ejecuciones	1000	1000	1000	1000
Núm. ejec. válidas del método	1000	1000	1000	1000
Núm. ejec. válidas en total	1000	1000	1000	1000
% Soluciones subóptimas	100,00 %	99,80 %	0,00 %	0,00 %
% Soluciones óptimas	0,00 %	0,20 %	100,00 %	100,00 %
% Soluciones sobreóptimas	0,00 %	0,00 %	0,00 %	0,00 %
% Diferencia media subóptima	69,21 %	89,04 %	0.00 %	0.00 %
% Diferencia máxima subóptima	25,24 %	67,61 %	0,00 %	0,00 %
% Diferencia media sobreóptima	0,00 %	0,00 %	0,00 %	0,00 %
% Diferencia máxima sobreóptima	0,00 %	0,00 %	0,00 %	0,00 %

En la tabla podemos ver como **“procesar3” y “Procesar4”**, los algoritmo que usan backtracking son óptimos el 100% de las veces en cambio los otros 2 no.

Tambíen se puede observar que a pesar de que **“Procesar”** y **“Procesar2”** no son exactos, **“Procesar2”** es un mejor algoritmo heurístico ya que la media de su solución con respecto al backtracking es de un 89% de la solución óptima en cambio **“Procesar”** se queda en el 69% de media.



En el gráfico podemos observar como **“Procesar3”** y **“Procesar4”** son siempre óptimos.

Además de nuevo se puede observar claramente como **“Procesar2”** da mejores resultados que **“Procesar”** ya que según los bigotes del gráfico **“Procesar”** llega a dar valores incluso de solo el 25% de la solución óptima.

En cambio el algoritmo **“Procesar2”** da resultados menos dispares acercándose bastante generalmente a la solución óptima (83%) y cuyos peores resultados se alejan a solo el 67% de la solución óptima de **“Procesar3**” y **“Procesar4”**.

También se puede observar como **“Procesar”** ha obtenido la solución óptima en el 0% de los casos y **“Procesar2”** en el 0,2% de los 1000 casos probados.

Comparación de eficiencia en tiempo

Conclusión

En la eficiencia en tiempo hay una gran diferencia entre los algoritmos inexactos y los exactos.

Los inexactos se ejecutan mucho más rápido que los inexactos, esto es debido a que los inexactos aplican la técnica del backtracking por lo que al volver atrás para probar más combinaciones tienen que rehacer muchos más cálculos.

En cambio los inexactos se ejecutan de manera lineal y realizan los cálculos una sola vez dando como resultado una mayor velocidad de ejecución.

Evidencias

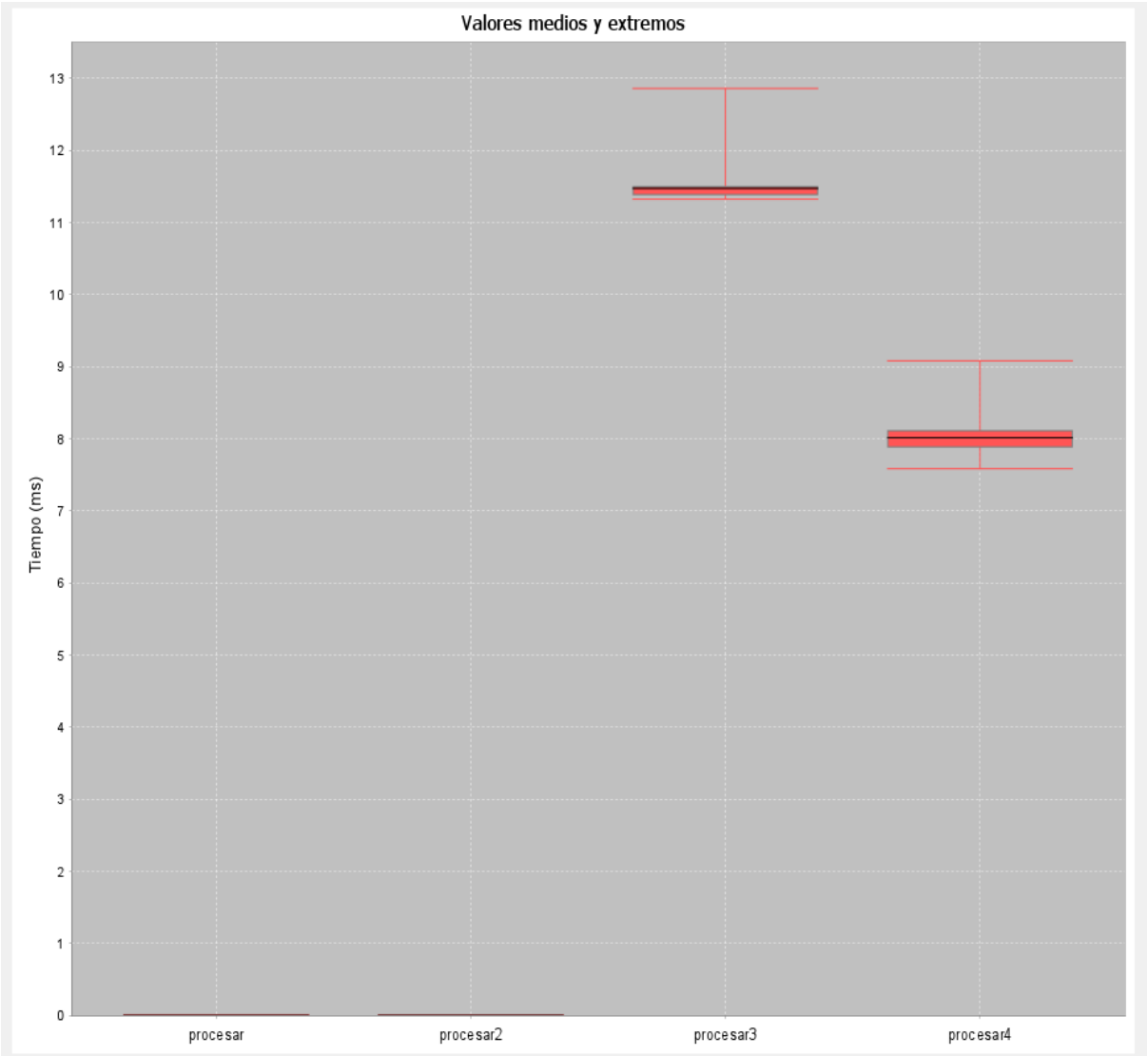
Medidas	procesar	procesar2	procesar3	procesar4
Núm. ejecuciones	1000	1000	1000	1000
Núm. ejec. válidas del método	1000	1000	1000	1000
Núm. ejec. válidas en total	1000	1000	1000	1000
Tiempo máximo	0.011 ms	0.006 ms	12.858 ms	9.080 ms
Tiempo medio	0.000 ms	0.000 ms	11.471 ms	8.015 ms
Tiempo mínimo	0.000 ms	0.000 ms	11.323 ms	7.584 ms

Aquí podemos ver claramente cómo **“Procesar”** y **“Procesar2”** son algoritmos extremadamente rápidos, su tiempo medio es de menos de 0.000 ms y su máximo de 0.011ms, aún así se puede comprobar como **“Procesar2”** tiene un tiempo máximo del 50% del de **“Procesar”**.

En cuanto a los algoritmos exactos son mucho más pesados, tardan en ejecutarse en torno a 10.000 veces más.

También podemos observar que la poda ha hecho efecto, **“Procesar3”** que no tiene poda, es un 30 % más lento de media que **“Procesar4”**.
Además **“Procesar4”** tiene un tiempo máximo de ejecución menor que y un tiempo mínimo de ejecución menor que **“Procesar3**”.

Se considera por tanto que **“Procesar4”** es más eficiente que **“Procesar3”**



En el gráfico se observa como los tiempos de ejecución de **“Procesar”** y **“Procesar2”** son prácticamente inapreciables.

Aquí se muestra más claramente la abismal diferencia de tiempo de ejecución entre los algoritmos exactos y los inexactos.

En relación a los inexactos, **“Procesar4”** es más eficiente que **“Procesar3”**, los tiempos medios de ejecución son más bajos y los bigotes del gráfico también se quedan por debajo de los de **“Procesar3”** lo que significa que sus valores mínimos y máximos están por debajo, es decir, es más eficiente **“Procesar4”** en el 100% de los 1000 casos probados.

Conclusiones

Como conclusión principal de la realización de esta práctica nos gustaría destacar los problemas encontrados a la hora de seleccionar un criterio de poda fiable, así como destacar la importancia de que para que una poda sea útil esta debe mejorar la eficacia del algoritmo en contraste con el algoritmo sin poda.

Es posible que el intento de poda en caso de no ser viable estrese más al programa que lo que se ha ahorrado podando y aumente la ineficiencia con respecto a no haber comprobado nada.