

**PRÁCTICA 3: TÉCNICAS DE BÚSQUEDa**

**Algoritmos Avanzados. Grado en Ingeniería Informática**



20 de noviembre de 2017

JORGE ARANDA GARCÍA

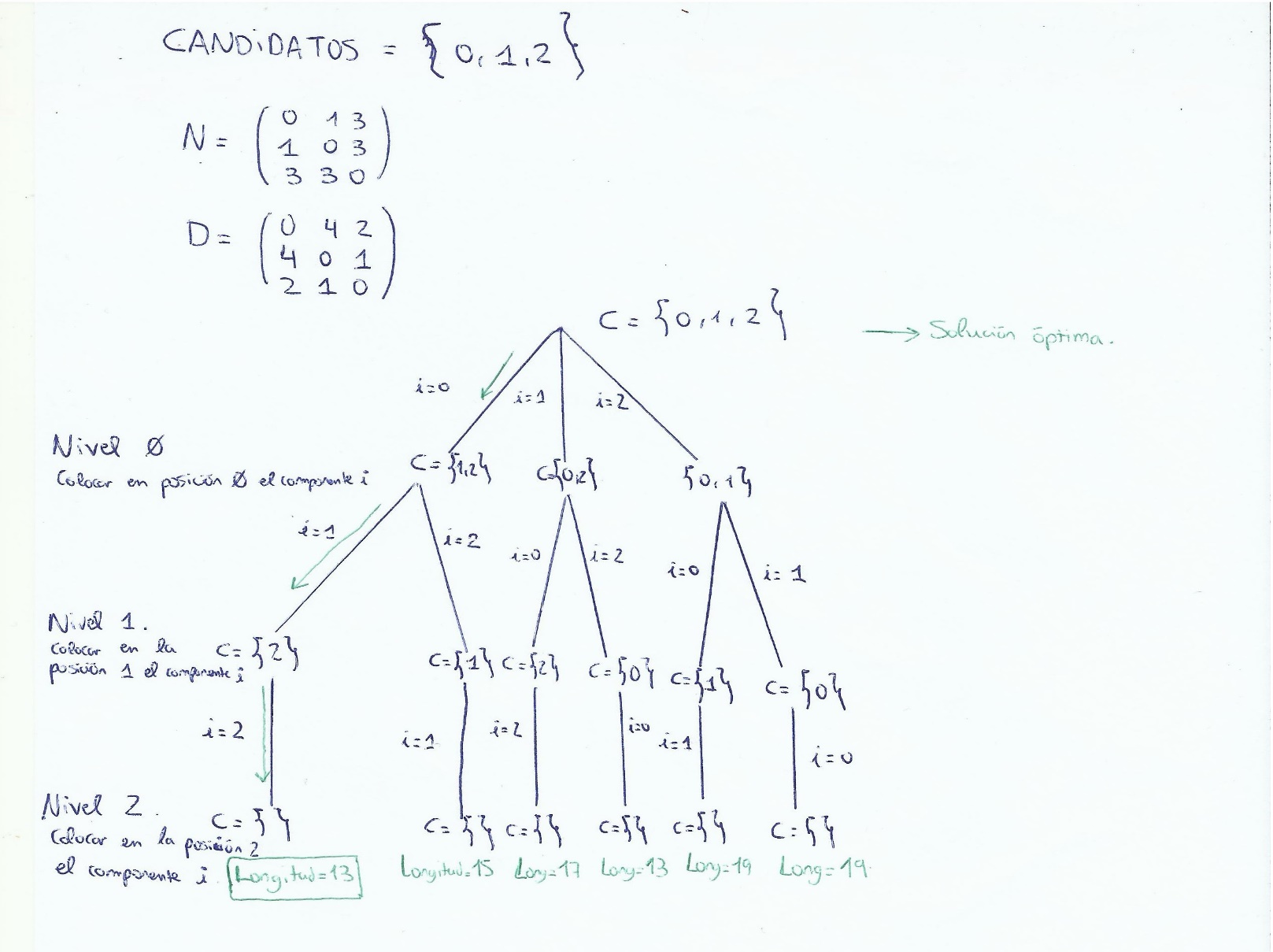
JOSE VICENTE BAÑULS GARCÍA

# **VUELTA ATRÁS**

## a) Diseño de un árbol de búsqueda adecuado para resolver el problema, mostrando el árbol en un gŕafico o, si es muy grande, parte de él. También debe explicarse explícitamente:

## 1) Niveles del árbol (número de niveles y qué papel cumple cada nivel)

## 2) Candidatos en cada nodo del árbol



Para cualquier árbol el número de niveles va a ser igual al número de posiciones que tiene la placa. Cada nivel significa, por tanto, la posición concreta de la tabla en la que se quiere colocar un componente.

El conjunto de candidatos de cada nodo el árbol esta expresado en el gráfico en forma de array, siendo en el nodo raíz un array con todos los componentes {1,2,3} y siendo un array vacío ({ }) en cada una de las hojas del árbol, puesto que al llegar a este caso, todos los componentes habrían sido colocados en alguna posición de la placa, correspondiente con el nivel. La solución óptima está indicada mediante flechas de color verde.

## b) Especificación y justificación de las comprobaciones incrementales a realizar en cada nodo del árbol de búsqueda

Estando en el nodo raíz no hace falta hacer ninguna comprobación, simplemente se empieza por colocar un componente en el nivel 0. (en nuestro caso, se va avanzando por los componentes en orden, es decir, primero el 0, luego el 1 y por último el 2.) En cada uno de los nodos intermedios hay que seleccionar alguno de los componentes restantes que no hayan sido seleccionados. Para ello, mantenemos un array de boolean, en el que cada vez que se escoge un componente se marca su posición en dicho array a true. Por lo tanto, en el siguiente nivel, los candidatos serán aquellos que estén a false. Cuando se llega a un nodo hoja del árbol, se calcula la longitud de la solución obtenida (se almacena en el array sol\_parcial, en el que cada posición del array se corresponde con la posición de la placa y donde se almacena el componente que se ha colocado en esa posición.) a partir de las comprobaciones realizadas y si es menor que la longitud calculada anteriormente se actualiza, excepto en la primera hoja en la que siempre se actualiza la longitud de cable, puesto que es la primera solución. Para volver atrás en el árbol hay que volver a marcar a false el componente y probar con el siguiente, para ver si se obtiene una solución mejor.

## c) Código del algoritmo, basado en el esquema de vuelta atrás, que incorpora el diseño anterior.

Se adjunta el fichero fuente que contiene el código del algoritmo, junto con la entrega de este informe.

El fichero que se corresponde es : PracticaBT

# **RAMIFICACIÓN Y PODA**

## a) Elegir y justificar un esquema (de coste mínimo u optimista-pesimista) para plantear el algoritmo de Ramificación y Poda.

En nuestro caso optamos por un esquema optimista-pesimista, porque creímos conveniente comparar la mejor solución obtenida hasta el momento con la mejor solución que se podría obtener a partir de un nodo cualquiera. Para calcular esta solución, se puede hacer de forma más real, respetando las restricciones o de forma más irreal saltándonos estas restricciones y buscado el mejor caso de todos, aunque sea infactible. Dependiendo de cómo de “buena” sea esta cota optimista, se podará más o menos, por lo que el algoritmo tendrá que explorar un mayor o menor tamaño del espacio de búsqueda.

## b) Definir un cota o cotas, según el esquema elegido, que pueda usarse en un algoritmo de ramificación y poda, indicando su valor inicial y cómo se actualiza.

Por todo lo explicado en el apartado anterior, nuestra cota pesimista es la mejor solución obtenida hasta el momento porque como mínimo se va obtener esa longitud de cable y nuestra cota optimista es aquella que calcula una solución buena, a pesar de ser irreal, a partir de un nodo, más la solución real obtenida hasta el mismo. Es irreal porque no se tienen en cuenta el número de conexiones Para ello, se han sumado las distancias mínimas al siguiente hueco. Estas dos cotas definen las condiciones de poda, porque si la cota pesimista es mejor, es decir, menor que la cota optimista, la rama no es explorada (se poda), puesto que la solución que va a devolver va a ser siempre peor que la que ya tenemos, y se continúa con la siguiente rama. La cota optimista se recalcula en cada nodo, para comprobar si no hay que podar, mientras que la cota pesimista solo se podría actualizar en los nodos hojas si, la solución que se consigue a través de esa rama es menor que la anterior.

## c) Código del algoritmo, basado en el esquema de ramificación y poda, que incorpora el diseño anterior.

Cabe destacar que la única variación, pero a su vez de gran importancia, con respecto al código del esquema de vuelta atrás es la inclusión de la cota optimista y de su cálculo, porque de alguna forma, la cota pesimista ya se tenía, ya que se calculaba la longitud de cada solución.

Se adjunta el fichero fuente que contiene el código del algoritmo, junto con la entrega de este informe.

El fichero correspondiente es PracticaRyP.java

Además tanto para el algoritmo anterior como para éste, se incluye un fichero Main.java en el que se realizan las ejecuciones de ambos y donde también se ofrecen algunos casos de prueba válidos.

# **COMPARACIÓN DE ALGORITMOS**

## a) ¿Para qué tipo de problemas utilizarías los algoritmos basados en Técnicas de Búsqueda?

Los tipos de problemas que para los que se usan este tipo de algoritmos son aquellos problemas en los que se necesita realizar una búsqueda exhaustiva porque otras técnicas como, por ejemplo, los algoritmos voraces fallan. Para ello se pueden encontrar todas las soluciones o solo la óptima.

En nuestro caso, el esquema de vuelta atrás se puede usar tanto para problemas combinatorios como de optimización, mientras que el esquema de ramificación y poda únicamente se puede usar para resolver problemas de optimización.

## b) ¿Qué tienen en común y en qué se diferencian los algoritmos de Vuelta Atrás y de Ramificación y Poda?

Ambos algoritmos garantizan encontrar una solución óptima del problema. Básicamente los algoritmos de Ramificación y Poda son una mejora de los algoritmos de Vuelta atrás.

Como ya se ha mencionado anteriormente, los algoritmos de Vuelta Atrás se pueden utilizar para resolver problemas combinatorios o de optimización, mientras que los de Ramificación y Poda sólo pueden resolver problemas de optimización. En segundo lugar, los algoritmos de Vuelta Atrás expanden sus nodos de forma similar a un recorrido en profundidad, mientras que los algoritmos de Ramificación y Poda, pueden hacerlo en anchura o en profundidad (en nuestro caso, ambos algoritmos expanden en profundidad). Por último, y probablemente la diferencia principal entre estos dos tipos de algoritmos, es el espacio de búsqueda. Mientras que los algoritmos de Vuelta Atrás necesitan explorar todo el espacio para comprobar cuál es la solución óptima, los algoritmos de Ramificación y Poda se basan en unas estimaciones (cotas) para evitar explorar aquellas ramas que no van a mejorar la solución, por lo que por lo general utilizan menos recursos de tiempo y memoria, puesto que en el peor de los casos ambos algoritmos van a ser igual de complejos y van a explorar el mismo espacio.

## c) ¿Qué ventajas y desventajas tienen respecto del resto de algoritmos vistos en clase?

Mientras que los algoritmos voraces, y en mucha menor medida aún, los algoritmos aproximados, no garantizan la solución óptima, los algoritmos basados en técnicas de búsqueda sí. Además, no requieren de una función de selección, en el caso de los algoritmos voraces, o de una de función de aproximación en el caso de los algoritmos aproximados.

El principal problema que tienen estos algoritmos es la gran cantidad de recursos que requieren, puesto que suelen abarcar problemas grandes, que consumen gran cantidad de tiempo y memoria, mientras que los algoritmos voraces y aproximados, son por lo general rápidos y consumen menos recursos de memoria.