AED II - PRÁCTICA 2

Tema 3 - Avance Rápido
Tema 5 - Backtracking
2022/2023

Subgrupo 1.2
Profesor: Francisco José Montoya
Problemas: G) Avance rápido, A) backtracking
Sergio Ros Liarte
Jose Mª López Serrano

ÍNDICE

1. Introducción

En esta memoria se explica el algoritmo y su análisis de eficiencia planteado para resolver los problemas asignados en la práctica 2 de la asignatura AED II.En la resolución de los problema se deben usar las técnicas de Avance rápido y Backtraking. Nuestros problemas a resolver eran el G de Avance rápido y el A de backtracking.

G_AR) Tenemos un sistema monetario donde las conchas marinas hacen de monedas. Existen N tipos de conchas/monedas. Cada tipo i tiene un valor, v_i, y un peso, p_i.

El Banco Central Etrusco (BCE) sólo admite cambios por valor V. Es decir, los clientes pueden dar dinero por valor total V, y a cambio se les devuelve la misma cantidad en las monedas que el cliente desee. Nuestro propósito es dar las monedas con mayor peso total, y a cambio recibir las monedas con menor peso.

En definitiva, debes encontrar una combinación de monedas con valor total V, y donde el peso total sea el máximo. Y, por otro lado, también debes encontrar una combinación por valor V y donde el peso total sea el mínimo. Se supone que existe un número ilimitado de monedas de todos los tipos. Se admitirá cierto margen en la solución respecto del óptimo.

A_BT)*En una empresa se dispone de varios trabajadores para realizar una serie de trabajos. Cada trabajador tiene una cierta capacidad de realizar trabajos y puede que haya trabajos que no sabe resolver. Vuestro trabajo consiste en decidir los trabajos a realizar por cada trabajador teniendo en cuenta su capacidad máxima de trabajo y los trabajos que no sabe cómo resolver. Se pretende maximizar el beneficio en la realización de los trabajos asignados, pues la habilidad en la realización de los trabajos por los trabajadores varía.*

2. Pseudocódigos y explicación de los algoritmos

A_AR)

```
Tipo Solucion
     peso: entero
     conchas: array[entero]

operacion seleccionaMin (pesos, valores: array[entero]; N: entero,
elegidos:
                        array[booleano]): entero
     max, proporcion: float
     j: entero
     proporcion := 0.0
     j := 0
     max := -1.0
     PARA i := 1….N HACER
          proporcion := valores[i] / pesos[i]
          SI proporcion > max Y ! elegidos[i] ENTONCES
               j := i
               max := proporcion
          FINSI
     FINPARA
     devolver j
FIN

operacion seleccionaMax (pesos, valores: array[entero]; N: entero,
elegidos:
                        array[booleano]): entero
     max, proporcion: float
     j: entero
     proporcion := 0.0
     j := 0
     max := -1.0
     PARA i = 1….N HACER
          proporcion := pesos[i] / valores[i]
          SI proporcion > max Y ! elegidos[i] ENTONCES
               j := i
               max := proporcion
          FINSI
     FINPARA
     devolver j
FIN
```

```
operacion cambio (pesos, valores: array[entero]; V, N, type:
                        entero): Solucion
      //inicializar
      x: array[entero](1…N)
      elegidos: array[booleano](1…N)
      PARA i := 1…N HACER
            x[i] := 0
            elegidos[i] := FALSO
      FINPARA
      j, act: entero
      j := 0
      act := 0
      //distinción para caso máximo (type = 1) / mínimo (type = 2)
      SI type = 1 ENTONCES
            //mientras no solución
            MIENTRAS act != V HACER
                  //seleccionar
                  j := seleccionaMax(pesos, valores, N, elegidos)
                  //si factible añadirlo
                  SI valores[j] <= (V-act) ENTONCES
                        //insertar
                        x[j] := (V-act) / valores[j] //parte entera
                        act :+=  x[j] * valores[j]
                  FINSI
                  elegidos[j] := VERDADERO
            FINMIENTRAS
      SINO
            //mientras no solución
            MIENTRAS act != V HACER
                  //seleccionar
                  j := seleccionaMin(pesos, valores, N, elegidos)
                  //si factible añadirlo
                  SI valores[j] <= (V-act) ENTONCES
                        //insertar
                        x[j] := (V-act) / valores[j] //parte entera
                        act :+=  x[j] * valores[j]
                  FINSI
                  elegidos[j] := VERDADERO
            FINMIENTRAS
      FINSI

      sol: Solucion
      sol.conchas := x
      PARA i := 1….N HACER
            sol.peso :+= sol.conchas[i] * pesos[i]
      FINPARA

      devolver sol
```

Para expresar la solución hemos creado una clase Solución, que consta de un entero *peso* y de una array de N enteros *conchas*. Esto lo hacemos porque resulta mejor a la hora de programar el algoritmo en c++ para no tener que calcular el peso total de las conchas en el método main, quedando el código más organizado. Peso representa el peso total de todas las conchas que hemos seleccionado en una solución. El array conchas representa, en cada posición, cuantas conchas del tipo i hemos escogido en la solución.

Las funciones seleccionaMin y seleccionMax son similares. Se encargan de escoger la siguiente concha que más conviene en la resolución del problema. Realmente son iguales solo que seleccionaMax prioriza coger las conchas que mejor relación peso/valor tienen (es decir, las más pesadas respecto a su valor) y seleccionaMin hace lo contrario, selecciona las más valiosas respecto a su peso. Primero inicializan max, para tener en cuenta la mejor proporción de las conchas, j, la posición de dicha concha en las arrays de pesos y valores, y la proporción, la división entre valor/peso o peso/valor de cada concha. Tras inicializar las variables, la función recorre con un for N para calcular la proporción de cada concha y compararla con la proporción más conveniente recorrida. No solo vale con que la proporción sea mejor, sino que también tiene que tener en cuenta que dicha concha no haya sido usada antes (sino cogería todo el rato las mismas conchas). Finalmente devuelve j. Esta operación se podría realizar de manera más eficiente si dispusieramos de unos arrays de proporciones ordenados, lo que nos permitiría solo tener que recorrer N una sola vez (coste lineal), pero al tener que imprimir por pantalla la solución en el mismo orden que se dió en la entrada no podemos preprocesar estos datos y ordenarlos, así que hay que recorrer el array completo cada vez que se selecciona un candidato.

En cuanto a la función principal (cambio), el comportamiento es sencillo, simplemente sigue el esquema general de avance rápido. Tanto en el propio pseudocódigo como en el código C++ están marcados como comentarios C // los trozos de código que hacen el trabajo de las funciones genéricas. Primero se inicializa el array de solución y el resto de variables auxiliares, y después, mientras que no se encuentre solución (no se haya llegado hasta el valor V) seleccionamos el mejor candidato, si este es factible añadirlo (contando con él no nos pasamos de V) lo añadimos a la solución (tantas veces como quepa), y si no lo descartamos. En ambos casos lo marcamos en elegidos para saber que ese ya ha sido usado. Lo único diferente en nuestro algoritmo es la condición que comprueba que tipo se le ha pasado como parámetro. Esto lo hemos hecho para no tener demasiado código repetido (podríamos haber propagado esta filosofía hasta selecciona y que se repitiera todavía menos, pero perdía legibilidad el código desde nuestro punto de vista). Si se le pasa como type un 1 este algoritmo obtendrá como siguientes candidatos siempre los objetos de mayor relación peso/valor, es decir, los más pesados, que son de los que nos queremos deshacer, por tanto si le decimos al

algoritmo type = 1 calculará las conchas que vamos a entregar al banco. Si por otro lado le ponemos type = 2 usará seleccionaMin, que escogerá siempre la concha con mejor valor/peso, las que nos interesa que nos devuelvan, y obtendrá la solución de peso minimizado. Así al hacer 2 llamadas en el método main, una con 1 y otra con 2 obtenemos las dos soluciones que necesitamos, la de máximo peso y la de mínimo peso, pudiendo imprimir la salida completa.

A_BT)

```
var
     nt,nw :entero

operacion solucion (s, capacidades: array[entero]; nivel: entero)
:
          booleano
     devolver nivel = nt Y capacidades[s[nivel]] != -1

operacion masHermanos (s: array[entero]; nivel: entero) : booleano
     devolver s[nivel] < nw

operacion criterio (s, capacidades, cotas: array[entero];
beneficios: array[entero][entero]; nivel, b, BOA : entero) :
booleano

     SI nivel = nt O capacidades[s[nivel]] < 0
               O  beneficios[s[nivel]][nivel] = -1 ENTONCES
          devolver FALSO
     SINO
          p: entero
          p := poda(b, nivel+1, cotas)
          SI p > BOA ENTONCES
               devolver VERDADERO
          SINO devolver FALSO
          FINSI
     FINSI

operacion poda(cotas: array [entero]; nivel, b : entero) :
booleano
     max: entero
     max:=0
     PARA i := nivel…nt HACER
          max:+= cotas[i]
     FINPARA
     devolver max+b
```

```
operacion generar (s,capacidades : array [entero]; beneficios :
array [entero][entero]; nivel,b : entro)

     s[nivel]++
     SI (s[nivel] = 0) ENTONCES
          b:+= beneficios[s[nivel][nivel]
          capacidades[s[nivel]]--
     SINO
          b := b + beneficios[s[nivel]][nivel]
                    - beneficios[s[nivel]-1][nivel]
          capacidades[s[nivel]]--
          capacidades[s[nivel]-1]++;
     FINSI

operacion retroceder (s,capacidades : array [entero]; beneficios :
array [entero][entero]; nivel,b : entro)

     b:-= beneficios[s[nivel]][nivel]
     capacidades[s[nivel]]++
     s[nivel] := -1
     nivel—-

operacion asigna (capacidades, cotas : array [entero]; beneficio :
array [entero][entero]) : entero

     s : int [entero]
     BOA : int
     PARA i: i,...,nt HACER
          s[i] := -1
     FINPARA
     b:=0
     BOA := 0
     nivel:=1
     generar(s,nivel,b,beneficios,capacidades)
     HACER

          SI solucion(s, nivel, capacidades) Y b > BOA ENTONCES
               BOA := b
          FINSI

          SI (criterio(s, nivel, b, BOA, beneficios,
                    capacidades))HACER
               nivel++
          SINO
               MIENTRAS nivel > 0 Y NO masHermanos(s, nivel)HACER
                    retroceder(s,nivel,b,beneficios,capacidades)
               FINMIENTRAS
          FINSI
```

```
MIENTRAS (nivel > 0)
devolver BOA
```

A la hora de plantear el problema nos dimos cuenta que era una variación del problema de la asignación y que por tanto se generaría un árbol permutacional, en el que la profundidad es en número de tareas y el número de hermanos por nivel es el número de trabajadores.

Para guardar la resolución parcial del problema hace falta una array de enteros S en la que se plasma que trabajadores han sido asignados a cada trabajo (siendo el número de trabajos el tamaño de la array) y un entero que representa el beneficio. Esta es la tupla solución. Este beneficio se tiene que maximizar, por lo que también hace falta un entero BOA que se va actualizando si se encuentran soluciones más óptimas mientras se va resolviendo el problema. El árbol usa esta misma variable y un array cotas en criterio para podar una rama. Cotas es un array de nt elementos que contiene para cada trabajo el beneficio del trabajador que más beneficio obtiene en ese trabajo en concreto. Para cada nodo que se genera, antes de estudiar a todos sus hijos y descendientes, se comprueba que el beneficio máximo que se podría alcanzar con las tareas que quedan y el beneficio que ya llevamos sea superior a nuestro valor óptimo actual, y si no es así poda esa rama, ya que no se va a encontrar ninguna solución mejor por ahí. Además de esa condición de poda también está la poda por factibilidad que nos evita obtener soluciones que no sean posibles. En criterio si se asigna un trabajo a un trabajador que no sabe realizarlo esa rama se poda. Aparte de las podas no hay nada especial en la resolución de este problema, simplemente sigue el esquema general de backtracking para problemas de optimización. La operación solución comprueba si el nodo actual forma una solución. Esto ocurre si es posible que ese trabajador haga ese trabajo por capacidad y si se está en el último nivel, puesto que entonces se han asignado todos los trabajos. La función criterio determina si se puede seguir bajando en el árbol, aumentando el nivel. Esto es posible cuando no se ha llegado al último nodo y cuando el trabajador al que le estamos asignando el trabajo actual puede hacerlo tanto por capacidad como porque sabe hacerlo. masHermanos comprueba que un nodo tiene más hermanos, esto es cuando en el array solución en la posición del trabajo actual no se ha alcanzado el último valor de trabajador (nw). La función genera crea un nuevo nodo. Si no existe un nodo en el nivel actual simplemente lo genera, y si ya había otro elimina el beneficio que aportaba el trabajador anterior y pone el suyo, y le libera 1 de capacidad al anterior trabajador. Por último, la función retrocede vuelve al nivel anterior, eliminando de la solución el nodo actual, poniendo la posición del nivel en el que estamos antes de subir de vuelta a -1 y eliminando el beneficio que aporta el trabajador que hay asignado.

3. Programación de los algoritmos

Para no hacer demasiado extensa esta memoria el código fuente de ambos problemas en C++ no se incluye. Este está contenido en su totalidad en los ficheros “G_AR.cpp” para el de avance rápido y “A_BT.cpp” para el de backtracking, que se incluye en la entrega de esta práctica. El código está comentado explicando qué es cada función o trozo de código y que es lo que hace. En cuanto a cómo se ejecutan los programas, primero hay que compilarlos con g++ x_xx.cpp -o <<*nombre*>> (aunque se incluye un fichero precompilado y listo para ejecutar de ambos problemas). Para hacer una ejecución hay que darle una entrada al programa, bien por la terminal, con teclado, o con un fichero con el formato que especifica en el juez online.

4. Estudio teórico del algoritmo

G_AR)

Para obtener el tiempo de ejecución y el orden de magnitud del algoritmo voraz primero hay que obtener el tiempo de ejecución de selecciona. Solo calculamos el de 1 de las 2 porque son idénticas en cuanto a cálculos. Ambas son de orden n. Teniendo esto se puede calcular el orden del algoritmo voraz. El orden de complejidad de este depende de cuántos tipos distintos de monedas m formen la solución, ya que no es lo mismo que solo se ejecute el bucle act != V 1 sola vez, con lo que solo recorrería N buscando candidatos 1 vez y el orden sería n, que si lo tiene que recorrer N veces, con lo cual asciende a n^2. En general en el caso promedio se supone que tendrá que recorrerlo m veces.

$$t_{selecciona}(n) = 10 + \sum_{i=1}^{n}(2 + 2 * 1/4) = 3 + n(2 + 2 * 1/4) \Rightarrow$$

$$t_{selecciona} \in O(n)$$

$$t_{p:\,cambio}(n,m) = 9 + 2n + \sum_{i=m}^{n}(4 + n) =$$

$$= 9 + 2n + (n - m + 1) * (4 + n) = 13 + 7n + n^2 - 4m$$

$$- m * n \qquad t_{p:\,cambio} \in O(n^2)$$

$$t_{M:\,cambio}(n) = 9 + 2n + \sum_{i=1}^{n}(4 + n) =$$

$$= 9 + 2n + (n - m + 1) * (4 + n) = 9 + 4n + n^2$$

$$t_{M:cambio} \in O(n^2)$$

$$t_{m:\,cambio}(n) = 9 + 2n + 4 + n =$$

$$= 13 + 3n$$

$$t_{m:cambio} \in O(n)$$

A_BT)

El tiempo de ejecución teórico de un algoritmo de backtracking puede ser difícil de estudiar. Al incorporar nuestro algoritmo funciones de poda, esto lo hace prácticamente impredecible, pues no se sabe en qué nodos va a podar, o siquiera si va a podar alguna vez. Por ello decidimos solo estudiar el peor caso, en el que el árbol nunca haga una poda. Para ello hay que obtener el número de nodos que se van a recorrer, estos serán el número de trabajos, que son los niveles del árbol, por el número de trabajadores, que es el número de nodos que hay en cada nivel. Con esto y teniendo en cuenta que cada nodo tendrá normalmente una ejecución del bucle principal del programa, obtenemos el tiempo peor de ejecución según el tamaño del árbol. Primero hay que obtener el tiempo de ejecución de las operaciones genéricas.

$$t_{solucion} = 2 \Rightarrow t_{solucion} \in O(1)$$

$$t_{masHermanos} = 1 \Rightarrow t_{masHermanos} \in O(1)$$

$$t_{generar} = 4 \Rightarrow t_{generar} \in O(1)$$

$$t_{retroceder} = 4 \Rightarrow t_{retroceder} \in O(1)$$

$$t_{M:criterio}(nivel, nt) = 7 + t_{poda}(n) = 7 + 3 + \sum_{i=nivel}^{nt}(1)$$

$$t_{M:criterio} \in O(n)$$

$$t_{m:criterio} = 3 \Rightarrow t_{m:criterio} \in O(1)$$

$$t_{M:asigna}(nw, nt, nivel) = 6 + \sum_{nivel=1}^{nt} 4^{nivel} * (t_{solucion} + t_{masHermanos} + t_{generar} + t_{criterio(nt,nivel)} + t_{retroceder})$$

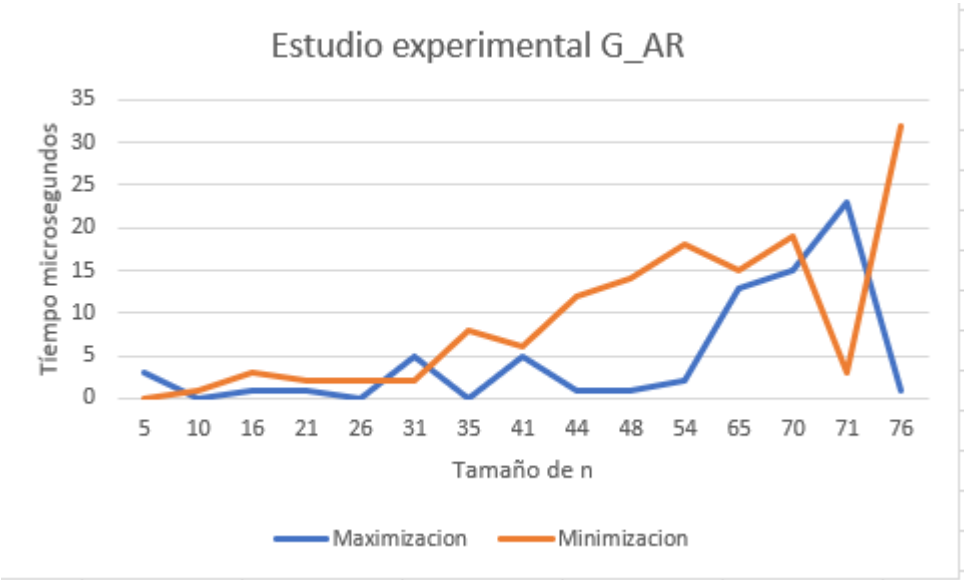
$$t_{CombinarSoluciones} \in O(4^{nw} * nt^2)$$

5. Estudio experimental

G_AR)

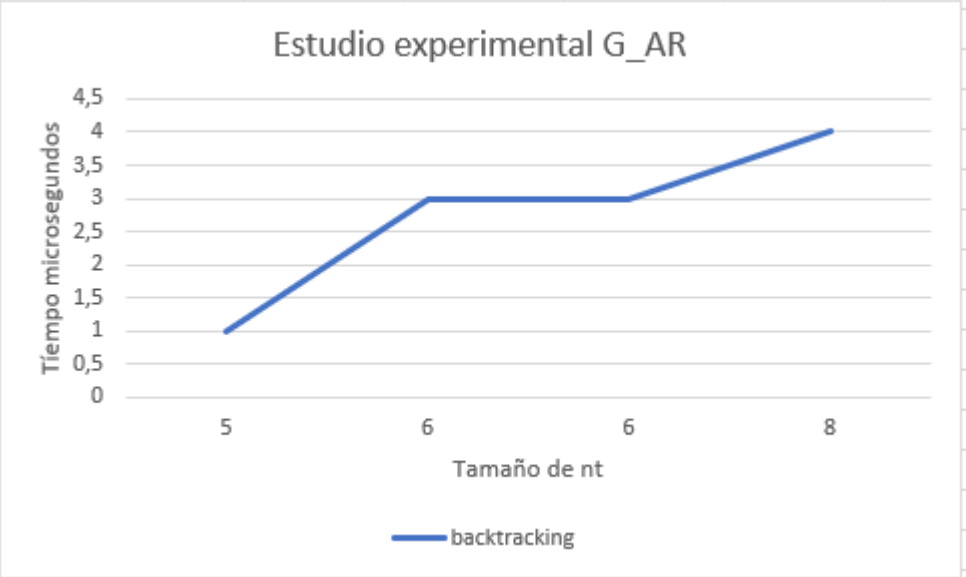
Para realizar un estudio experimental del tiempo de ejecución de nuestro algoritmo usamos los métodos de la librería sys/time.h para medir el tiempo del sistema entre ejecuciones. El código que usamos está comentado en el fichero de código fuente para que quede constancia de cómo lo utilizamos. Usamos un fichero de entrada de prueba con 15 casos con tamaños de n distintos. El fichero de entrada se encuentra en la entrega de esta práctica.

n = 5, 10, 16, 21, 26, 31, 35, 41, 44, 48, 54, 65, 70, 71, 76



A_BT)

El estudio experimental que realizamos para este problema no es muy representativo por varios motivos: disponíamos de pocos casos de prueba en mooshak, además no tenemos un estudio teórico completo porque es imposible saber cómo va a realizar la poda el árbol en según qué casos, y por último tampoco podíamos inventarnos casos con nt y nw muy grandes porque con el alto coste de ejecución que tiene el algoritmo nuestros ordenadores necesitarían demasiado tiempo para ejecutar un problema relativamente grande, lo suficiente como para que se puedan obtener conclusiones significativas. Aún así, hemos ejecutado el programa midiendo el tiempo que tardan los 4 casos de prueba de los que disponíamos en mooshak.



6. Contraste entre los resultados del estudio experimental y el teórico

G_AR)

Como se aprecia en la gráfica, el tiempo de ejecución real no se asemeja a lo que nos dice el estudio teórico si lo tomamos respecto a n. Si es cierto que en algunos casos da valores que podrían ser cercanos pero en muchos da como resultado anomalías bastante dispares. Esto se debe a que el factor principal del tiempo de ejecución está más relacionado con el parámetro m que con el n. Esto es porque la mayor parte del trabajo de nuestro algoritmo se pierde

buscando al siguiente candidato. Recordemos que m indica cuántos tipos de monedas forman la solución, pero se pueden haber recorrido muchos más que no se han tomado por no ser factibles, con lo cual no podemos medir realmente con casos aleatorios cual es la probabilidad de que se recorran x veces los N tipos de conchas. Esto se podría mejorar para hacer un algoritmo más eficiente, por ejemplo modificando cómo tratamos los datos, reordenando el array de pesos y valores y convirtiéndolo en un array de conchas ordenado por su proporción valor/peso, pero ante nuestra falta de tiempo para explorar otras opciones hemos decidido dejar así el algoritmo porque aunque no es muy eficiente (es bastante ineficiente, en realidad), obtiene unas soluciones correctas para los problemas.

A_BT)

Si ser demasiado representativo por los motivos que hemos explicado en el punto anterior, sí que, como es lógico, se puede ver que la gráfica crece por nt, que es el parámetro que más rápido hace crecer la función ya que es el que mayor término tiene. Aunque no hemos podido hacer un estudio representativo del tiempo de ejecución del algoritmo, creemos que podemos afirmar que el estudio teórico es representativo del problema real y que el algoritmo de backtracking es tan lento como se espera que sea, y es normal teniendo en cuenta la cantidad de nodos que tiene que recorrer. Por ejemplo, para un nw=4, solo con 4 trabajos ya estaríamos ante más de 320 nodos. Además, a cada nodo hay que aplicarle las respectivas funciones genéricas de solución, criterio, masHermanos y retroceder.

7. Conclusiones

Con esta práctica creemos haber comprendido los fundamentos de la programación de algoritmos voraces y de backtracking. No obstante, en cuanto al análisis de la eficiencia del algoritmo nos quedamos insatisfechos. Nos hubiera gustado hacer unas pruebas más exhaustivas, quizás con unos cuantos cientos de casos, pero por tiempo (de cálculo, especialmente en backtracking, de generación y de análisis) es algo que no hemos podido realizar pues nos hubiera supuesto un trabajo enorme. Además tuvimos problemas con el manejo de memoria en ambos problemas que nos hicieron perder mucho tiempo y nos costó acabar las implementaciones a tiempo, antes de que cerrara el juez online, lo que nos quitó tiempo para realizar el resto de apartados. Aún así estamos satisfechos con la implementación de los 2 algoritmos puesto que funcionan como se espera.

La práctica nos ha supuesto a los 2 compañeros en total aproximadamente cerca de 30 horas entre la planificación, diseño, programación, comprobación y estudio de eficiencia del algoritmo así como a la redacción de la memoria.