MEMORIA PRÁCTICA 2

CONVOCATORIA: ENERO

ALGORITMOS Y ESTRUCTURAS DE DATOS II

CHRISTIAN MATAS CONESA

christian.m.c@um.es

PROFESOR: JOAQUÍN CERVERA LÓPEZ

ENTREGA: 09/01/2024

CUENTA MOOSHAK: G2_59

Lista de problemas resueltos

He resuelto el problema C de avance rápido (Máxima diversidad) y el F de backtracking (Nos vamos de boda).

Envíos:
AR → 97 (fallo en tiempo de compilación), 202 (aceptado), 518 (aceptado y entrega definitiva).
Backtracking → 255 (tiempo límite excedido), 259 (tiempo límite excedido), 280 (aceptado), 595 (aceptado), 759 (tiempo límite excedido) y 760 (aceptado y entrega definitiva).

Resolución de problemas

AVANCE RÁPIDO

1. Pseudocódigo del algoritmo

```
Algoritmo leerMatriz(n, m):
   matriz = nueva Matriz de Enteros de n filas y m columnas

   Para i desde 0 hasta n-1 hacer:
      matriz[i] = nueva Fila de Enteros de m elementos
      Para j desde 0 hasta m-1 hacer:
         Leer matriz[i][j] desde la entrada estándar
      Fin Para
   Fin Para

   Devolver matriz
Fin Algoritmo
```

```
Algoritmo maxDiversidad(matriz, n, m):
  solucion = nueva Lista de Enteros de tamaño n

  Para i desde 0 hasta n-1 hacer:
    solucion[i] = 0
  Fin Para

  Si m <= 1 entonces:
    Imprimir 0
    Para i desde 0 hasta n-1 hacer:
      Imprimir solucion[i] seguido de un espacio y un salto de línea
    Fin Para
    Devolver
  Fin Si

  max = 0
  fila = -1
  columna = -1
  maximo = 0
  maximo1 = 0

  Para i desde 0 hasta n-1 hacer:
    Para j desde 0 hasta n-1 hacer:
      Si matriz[i][j] >= max entonces:
        max = matriz[i][j]
        fila = i
        columna = j
      Fin Si
    Fin Para
  Fin Para

  max = matriz[fila][columna] + matriz[columna][fila]
  solucion[fila] = 1
  solucion[columna] = 1
  m = m - 2

  Mientras m > 0 hacer:
    Para j desde 0 hasta n-1 hacer:
      Si solucion[j] == 0 y j != columna entonces:
        sumaBucle = 0
        Para i desde 0 hasta n-1 hacer:
          Si solucion[i] == 1 entonces:
            sumaBucle = sumaBucle + matriz[i][j] + matriz[j][i]
            maximo1 = sumaBucle
          Fin Si
        Fin Para
        Si maximo < maximo1 entonces:
          maximo = maximo1
```

```
                columna = j
            Fin Si
        Fin Si
    Fin Para
    max = max + maximo
    solucion[columna] = 1
    m = m - 1
  Fin Mientras

  Imprimir max
  Para i desde 0 hasta n-1 hacer:
    Imprimir solucion[i] seguido de un espacio
  Fin Para
  Imprimir un salto de línea
Fin Algoritmo
```

Algoritmo principal():

```
  T = Leer desde la entrada estándar
  Para i desde 0 hasta T-1 hacer:
    n, m = Leer desde la entrada estándar
    matriz = leerMatriz(n, n)
    maxDiversidad(matriz, n, m)
  Fin Para
Fin Algoritmo
```

En este algoritmo utilizo una función auxiliar para leer la matriz que se pasa por la entrada, luego viene la función principal "maxDiversidad" la cual recibe la matriz y dos enteros, el primero indica el tamaño de la matriz (por ejemplo, si n=4 significa que la matriz es de 4x4) y el segundo indica el número de elementos que podemos seleccionar.

Primero creamos un array con la solución final del problema (con todos los valores a 0), comprobamos que el número de elementos a escoger sea mayor que 0 ($m > 0$), buscamos por toda la matriz el valor de diversidad más grande (el número más grande de la matriz), guardamos este número al total de diversidad conseguido y también la fila y la columna en la que se encontraba. A partir de ahí busco el máximo de diversidad que se puede conseguir utilizando la estructura de avance rápido.

2. Estudio teórico del tiempo de ejecución del algoritmo

- La función "leerMatriz" es del orden $O(n^2)$.
- Dentro de la función principal:
    - Se crea un array de tamaño "n" y se recorre para iniciar todos sus valores a 0, esto tiene un orden $O(n)$.
    - Se recorre la matriz para encontrar el número más grande de esta, esto tiene un orden $O(n^2)$.
    - Se usa un bucle while que itera m-2 veces y en cada iteración realiza un bucle for que tiene $O(n)$, por lo tanto, tiene un orden $O((m-2) \times n)$.
    - Se usa un bucle for dentro del bucle while, como se ha dicho en el apartado anterior, tiene orden $O(n)$.
    - Por último, se imprimen los resultados recorriendo el array solución, el cual tiene orden $O(n)$.

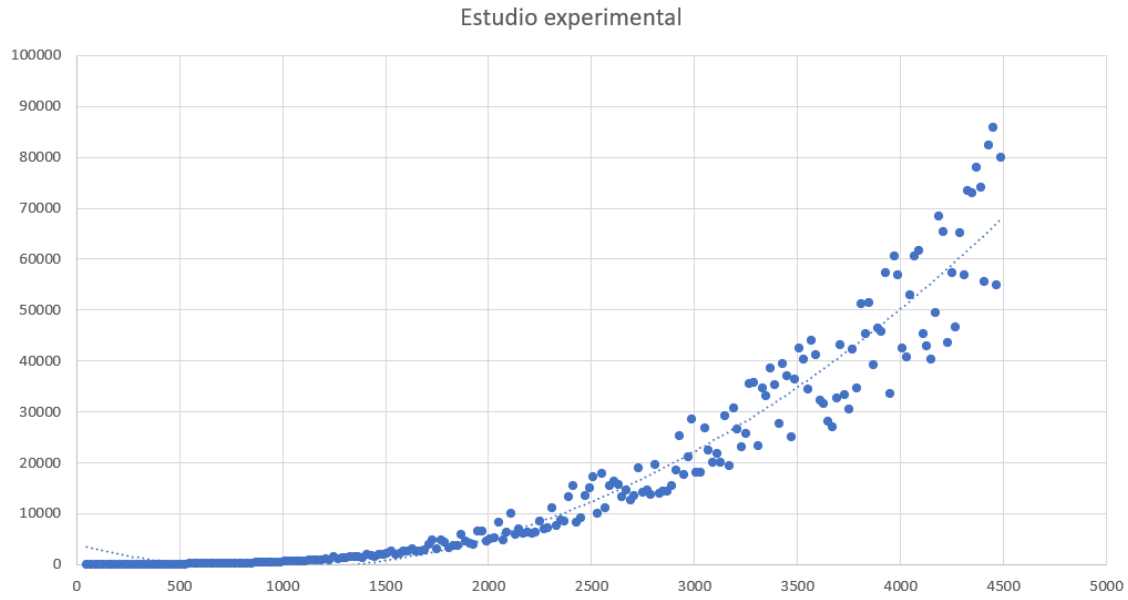
En general, el orden del algoritmo se ve dominado por el primer bucle que recorre la matriz en busca del mayor valor, el bucle while y el bucle que imprime los resultados. Por lo tanto, el orden del algoritmo es $O(n^2 + (m-2) \times n + n)$. Este orden es polinómico, en este caso el término dominante que afecta el crecimiento del algoritmo es n^2, ya que es el término cuadrático más grande. En conclusión, la complejidad asintótica del algoritmo es $O(n^2)$, que es polinómica.

3. Estudio experimental del tiempo de ejecución del algoritmo

Para estudiar el tiempo de ejecución del algoritmo, he creado el fichero "maxDiversidad-Eperimental.cpp" en el que está incluida una función que genera una matriz (que sustituye la función de leer la matriz) con valores aleatorios de 0-9, he tenido en cuenta esta función a la hora de tomar el tiempo que tarda el algoritmo, ya que esta función tiene la misma complejidad que la función leerMatriz() original. También he añadido las líneas de código correspondientes para tomar el tiempo que tarda cada iteración del bucle e imprima los resultados de forma que encajen en un fichero de formato .csv.

Con este fichero he generado la siguiente gráfica con 224 casos de prueba con valores de n desde 50 hasta 4490 y el valor de m en cada iteración era la mitad del valor de n.

En la gráfica, “x” son los valores de “n” e “y” es el tiempo en milisegundos.



4. Contraste estudio teórico y experimental

En esta gráfica he incluido una línea de tendencia polinómica que se corresponde con lo predicho en el estudio teórico.

En el estudio experimental hemos obtenido unos resultados que están estrechamente relacionados con los del estudio teórico, ya que el tiempo de ejecución crece polinómicamente.

Backtracking

1. Pseudocódigo del algoritmo

```
Algoritmo generar
   Si s[nivel] no es 0 entonces
      pact = pact - precios[nivel][s[nivel]]
      s[nivel] = s[nivel] + 1
      pact = pact + precios[nivel][s[nivel]]
   Sino
      s[nivel] = s[nivel] + 1
      pact = pact + precios[nivel][s[nivel]]
   Fin Si
Fin Algoritmo


Algoritmo solucion
   Devolver (nivel == C) y (pact <= M)
Fin Algoritmo


Algoritmo criterio(voa)
   Si (nivel == C) o (pact >= M) o (pact + precios[nivel + 1][0] <= voa) entonces
      Devolver falso
   Fin Si

   Si pact + precios[0][nivel + 1] > M entonces
      Devolver falso
   Fin Si

   Si voa == M entonces
      Devolver falso
   Fin Si

   Devolver verdadero
Fin Algoritmo


Algoritmo mas_hermanos
   Devolver s[nivel] < modelos[nivel]
Fin Algoritmo


Algoritmo retroceder
   pact = pact - precios[nivel][s[nivel]]
   s[nivel] = 0
   nivel = nivel - 1
Fin Algoritmo
```

Algoritmo backtracking

```
   nivel = 1
   pact = 0
   voa = -1
   Repetir
      generar()
      Si (solucion()) y (pact > voa) entonces
         voa = pact
      Fin Si
      Si criterio(voa) entonces
         nivel = nivel + 1
      Sino
         Mientras (no mas_hermanos() o voa == M) y (nivel > 0) hacer
            retroceder()
         Fin Mientras
      Fin Si
   Hasta que nivel <= 0
   Si voa == -1 entonces
      Escribir "no solution"
   Sino
      Escribir voa
   Fin Si
Fin Algoritmo
```

Algoritmo principal

```
   ncasos = Leer
   Para i desde 1 hasta ncasos hacer
      M, C = Leer
      Para j desde 0 hasta MAX_PRENDAS + 1 hacer
         s[j] = 0
         modelos[j] = 0
         Para h desde 0 hasta MAX_MODELOS + 1 hacer
            precios[j][h] = -1
         Fin Para
      Fin Para

      Para j desde 1 hasta C hacer
         nmodelos = Leer
         maximo = 0
         minimo = 999999
         modelos[j] = nmodelos

         Para h desde 1 hasta nmodelos hacer
            precios[j][h] = Leer
            Si precios[j][h] > maximo entonces
               maximo = precios[j][h]
            Fin Si
            Si precios[j][h] < minimo entonces
               minimo = precios[j][h]
            Fin Si
```

```
            Fin Para
            precios[j][0] = maximo
            precios[0][j] = minimo
        Fin Para

        Para j desde C-1 hasta 1 con paso -1 hacer
            precios[j][0] = precios[j][0] + precios[j+1][0]
            precios[0][j] = precios[0][j] + precios[0][j+1]
        Fin Para

        backtracking()
    Fin Para
Fin Algoritmo
```

2. Estudio teórico del tiempo de ejecución del algoritmo

El tiempo de ejecución en un algoritmo de backtracking depende del número de nodos y el tiempo que se necesite para cada nodo, siempre que el tiempo en cada nodo sea del mismo orden. Además, está condicionado al coste de sus funciones.
En cuanto al número de nodos, cada tipo de árbol tendrá un número total de nodos diferente (binario, combinatorio, k-ario, permutacional). En nuestro caso, al tratarse de un árbol combinatorio con n niveles y k elementos, esperaríamos un orden factorial, tal que $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.
Para poder mejorar un poco el tiempo he incluido una condición más aquí:

```
while ((!Mas_hermanos() || voa == M) && (nivel > 0)){
      Retroceder();
}
```

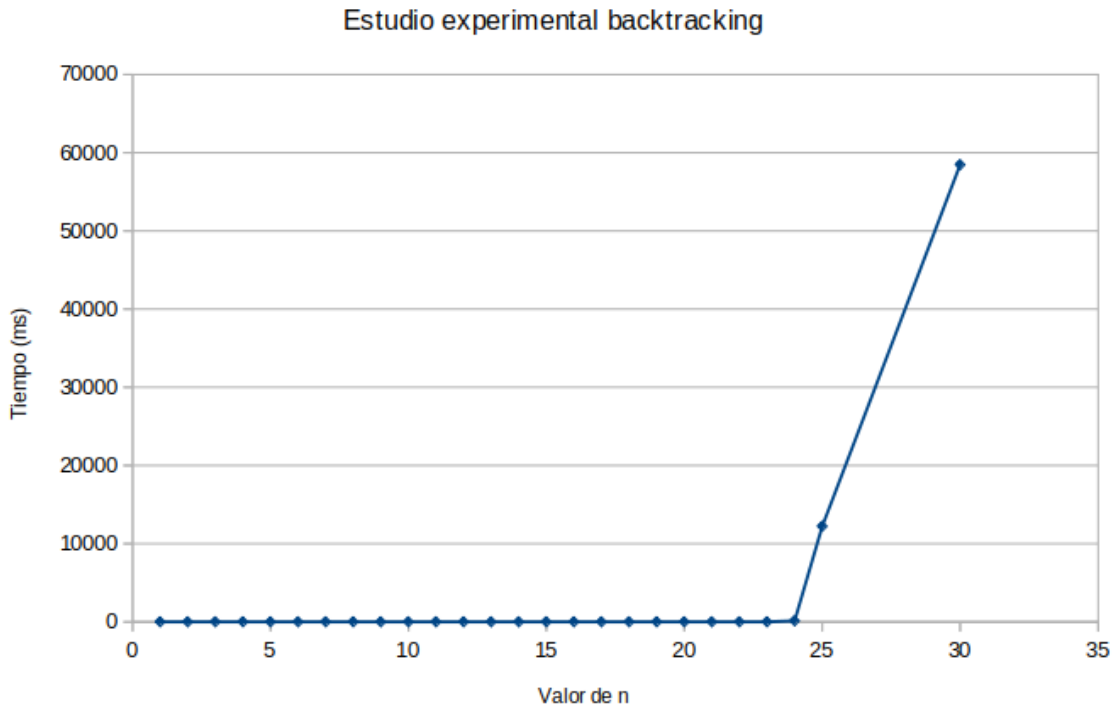
Si la solución “voa” es la máxima cantidad de dinero entonces no hace falta revisar los hermanos (poda).

3. Estudio experimental del tiempo de ejecución

Para el estudio experimental del tiempo de ejecución he incluido un nuevo fichero llamado “generarCasos.cpp”, el cual se encarga de generar los casos de prueba que va a analizar el fichero “Backtracking-experimental.cpp”. Lo que hago para analizarlo es:

1. ./generarCasos > entrada.in
2. ./Backtracking-experimental < entrada.in > estudio.csv

De esta forma he obtenido el siguiente gráfico:



En este gráfico se puede apreciar lo dicho en el estudio teórico y es que en cuanto aumenta un poco el tamaño de n, los tiempos se disparan, ya que tiene que analizar todas las combinaciones posibles al ser un árbol combinatorio.
No he podido analizar casos más grandes porque tardaba demasiado en ejecutarse el algoritmo, quizás se podría hacer alguna poda para reducir los tiempos.

Tanto los ficheros usados, como la imagen del gráfico, como el fichero .csv y la hoja de cálculo se encuentran adjuntos dentro del .zip.

4. Contraste del estudio teórico y el experimental

Si contrastamos los datos del estudio teórico con los del estudio experimental, podemos comprobar que ambos llegan a la conclusión de que el tiempo de ejecución aumenta factorial, al igual que lo hace el número de nodos. Por lo tanto, la predicción hecha en el análisis teórico es correcta, ya que la curva del gráfico del estudio experimental es claramente factorial.

Conclusión

En conclusión, las técnicas de Avance Rápido y Backtracking son fundamentales en el estudio de algoritmos, ya que permiten abordar problemas complejos en un plazo de tiempo corto (AR) y mediante la exploración exhaustiva de todas las soluciones posibles (BT). Estas técnicas encuentran soluciones aproximadas (como el caso de AR), o en cambio explorar todo el árbol en busca de la solución óptima (como hace BT), aunque su rendimiento puede depender del tamaño y la estructura del problema en cuestión. En el caso de Backtracking, en algún problema puede decidirse en vez de usar esta técnica, usar Avance Rápido. La práctica 2 de la asignatura es una buena forma de entender de forma práctica las técnicas de Avance rápido y Backtracking. En el estudio teórico y en el experimental es donde realmente podemos observar el comportamiento de cada técnica.