MEMORIA

AR y BA AED II

Usuario utilizado: G2_82

Nombre Completo	Email	DNI
Juan Pedreño García	juan.pedrenog@um.es	
Patricia Cuenca Guardiola	patricia.cuencag@um.es	

Índice

1.INTRODUCCIÓN

La memoria de esta práctica describe el proceso de diseño, implementación y valoración de la solución para el problema E_AR y E_BA, resueltos usando el algoritmo de avance rápido y de backtracking respectivamente.

2.LISTA DE PROBLEMAS RESUELTOS

Avance Rápido:

El mejor envío es el 30:

#	Tiempo de Concurso	País	Equipo	Problema	Lenguaje	Resultado	Estado
30	312:30:41		G2 G2 82	E_AR	C++	3 Accepted	final
29	312:25:29		G2 G2 82	E_AR	C++	0 Wrong Answer	final
28	312:21:12		G2 G2 82	E_AR	C++	0 Compile Time Error	final
27	312:20:20		G2 G2 82	E_AR	C++	0 Compile Time Error	final
26	312:18:55		G2 G2 82	E_AR	C++	0 Compile Time Error	final
25	312:15:33		G2 G2 82	E_AR	C++	0 Compile Time Error	final
24	312:13:40		G2 G2 82	E_AR	C++	0 Compile Time Error	final
23	312:08:34		G2 G2 82	E_AR	C++	0 Compile Time Error	final
22	312:01:28		G2 G2 82	E_AR	C++	0 Compile Time Error	final
21	311:39:58		G2 G2 82	E_AR	C++	0 Wrong Answer	final
20	311:34:16		G2 G2 82	E_AR	C++	0 Wrong Answer	final

Al principio daba wrong answer porque no se tenía en cuenta el número de averías que podía reparar. Luego el compile time error se debe a que había una variable entera declarada y no inicializada correctamente.

Bactracking:

El mejor envío es el 611

#	Tiempo de Concurso	País	Equipo	Problema	Lenguaje	Resultado	Estado
612	1013:25:05		G2 G2 82	E_Ba	C++	0 Runtime Error	final
611	1013:16:59		G2 G2 82	E_Ba	C++	3 Accepted	final
608	1012:37:21		G2 G2 82	E_Ba	C++	3 Accepted	final
606	1012:12:26		G2 G2 82	E_Ba	C++	0 Time Limit Exceeded	final
605	1012:11:07		G2 G2 82	E_Ba	C++	0 Time Limit Exceeded	final
604	1012:08:10		G2 G2 82	E_Ba	C++	0 Time Limit Exceeded	final
603	1012:06:40		G2 G2 82	E_Ba	C++	0 Time Limit Exceeded	final
602	1012:03:50		G2 G2 82	E_Ba	C++	0 Time Limit Exceeded	final
601	1012:03:15		G2 G2 82	E_Ba	C++	0 Runtime Error	final
600	1012:00:05		G2 G2 82	E_Ba	C++	0 Runtime Error	final
598	1011:50:21		G2 G2 82	E_Ba	C++	0 Runtime Error	final
597	1011:46:52		G2 G2 82	E_Ba	C++	0 Runtime Error	final
595	1011:22:01		G2 G2 82	E_Ba	C++	0 Runtime Error	final
594	1011:20:22		G2 G2 82	E_Ba	C++	0 Time Limit Exceeded	final
593	1011:14:26		G2 G2 82	E_Ba	C++	3 Accepted	final

#	Tiempo de Concurso	País	Equipo	Problema	Lenguaje	Resultado	Estado
590	1011:06:11		G2 G2 82	E_Ba	C++	0 Compile Time Error	final
589	1011:05:37		G2 G2 82	E_Ba	C++	0 Runtime Error	final
588	1010:54:50		G2 G2 82	E_Ba	C++	0 Runtime Error	final
302	745:51:30		G2 G2 82	E_Ba	C++	3 Accepted	final
301	745:43:42		G2 G2 82	E_Ba	C++	0 Runtime Error	final
255	658:28:01		G2 G2 82	E_Ba	C++	3 Accepted	final

Al principio se diseñó una solución que empleaba matrices globales con un tamaño máximo (100). Sin embargo, decidimos que era mejor utilizar una matriz que dependiese del tamaño de entrada y por eso tenemos varios envíos con runtime error, ya que no conseguimos que el envío fuese aceptado, aunque los casos de prueba nos funcionasen. Por último decidimos implementar un algoritmo con poda que usase un algoritmo voraz como cota superior y usando vectores declarados en el main,. En el proceso de diseño de esta última solución daba time limit exceeded porque no estábamos calculando bien la aproximación.

3. RESOLUCIÓN DE PROBLEMAS

3.1 AVANCE RÁPIDO

3.1.1 DISEÑO SOLUCIÓN

PSEUDOCÓDIGO

```
var mecánicos, averías

leer (mecánicos), leer(averías)

var matriz[mecánicos][averías]

var averíasPorMecánico[averías] := {0, 0, 0, 0,...}

var averíasReparadas[averías] := {F, F, F, F,...}

var mecanicoPorAvería[averías] := {0, 0, 0, 0,...}

var mecanicosUsados[mecánicos] := {F, F, F, F,...}

//Montamos la matriz y el array de averías por mecánico

para i = 0...mecánicos hacer

        para j=0...averías hacer

                leer(matriz[i][j])
```

```
            si matriz[i][j] entonces averíasPorMecánico[i]+=1 finsi
        finpara
finpara
```

//Decisión voraz: Cada avería la reparará el mecánico que pueda reparar menos averías a parte de sobre la que se está iterando

```
para i=0…averías hacer
        var averiasMin = INFINITO
        var reparable = False
        var mecanico = 0
        para j=0…mecánicos hacer
            si matriz[j][i] AND NOT mecanicosUsados[j] entonces
                reparable := True
                si averiasPorMecanico[j] < averiasMin entonces
                    averiasMin := averiasPorMecanico[j]
                    mecanico = j
                finsi
            finsi
        finpara

        si reparable AND NOT mecanicosUsados[mecanico] entonces
            averiasReparadas[i] := True
            mecanicoPorAvería[i] := mecanico+1
            mecanicosUsados[mecanico] := True
            reparaciones++
```

finsi

finpara

//Sacamos el resultado

Escribir(reparaciones)

para i = 0…averías hacer

Escribir(mecanicoPorAvería[i])

finpara

EXPLICACIÓN DEL ALGORITMO

En el algoritmo empezamos montando la matriz de Mecánicos y Averías. Además de eso contamos el número de Averías que puede reparar cada mecánico y las metemos en el array AveriasPorMecanico.

Después, hacemos lo que sería el algoritmo en sí y donde tomamos la decisión voraz. La decisión voraz es que para cada Avería, se asignará el mecánico que pueda reparar dicha Avería y pueda resolver menos Averías. Al ser un algoritmo voraz, una vez asignada la Avería a un Mecánico no se le puede cambiar. Si una Avería es reparable, en la array mecanicosUsados indicamos que ese mecánico ya tiene una tarea asignada. Y en la array mecanicoPorAveria apuntamos el Mecánico asignado.

En la parte final del algoritmo escribimos la solución.

3.1.2 IMPLEMENTACIÓN: PROGRAMACIÓN ALGORITMO

```
#include<iostream>

#include<vector>

#define MAX 2147483647

using namespace std;
```

```
int main(){

   int nCasos;

   cin >> nCasos;

   cout << nCasos << endl;

   for(int i = 0;i < nCasos;i++){

       int M, A;

       cin >> M >> A;

       int matriz[M][A];

        vector<int> averiasPorMecanico(M, 0);        //Avería que resuelve cada
mecanico

         vector<bool> averiasReparadas(A, false);      //Averias que han sido
reparadas

         vector<int> mecanicoPorAveria(A, 0);          //Mecanicos que reparan
cada avería concreta

         vector<bool> mecanicos(M, false);            //Mecanicos que han sido
usados o no

       int reparaciones = 0;

       //Montamos la matriz y el array de averías por cada mecánico

       for(int j = 0;j < M;j++){

           for(int k = 0;k < A;k++){

               cin >> matriz[j][k];

               if(matriz[j][k] == 1) averiasPorMecanico[j]++;

           }

       }
```

```
        //Decisión voraz: Cada avería la repararará el mecánico que pueda
reparar menos averías a parte de sobre la que se está iterando
      for(int j = 0;j < A;j++){
          int averiasMin = MAX;
          bool reparable = false;
          int mecanico = 0;
          for(int k = 0;k < M;k++){
              if(matriz[k][j] == 1 && !mecanicos[k]){
                  reparable = true;
                  if(averiasPorMecanico[k] < averiasMin){
                      averiasMin = averiasPorMecanico[k];
                      mecanico = k;
                  }
              }
          }
           //Si es reparable y el mecánico que se ha eledido está libre se le
asigna la tarea
          if(reparable && !mecanicos[mecanico]){
              averiasReparadas[j] = true;
              mecanicoPorAveria[j] = mecanico+1;
              mecanicos[mecanico] = true;
              reparaciones++;
          }
      }
```

```
        //Sacamos el resultados

        cout << reparaciones << endl;

        for(int j = 0;j < A;j++){

            cout << mecanicoPorAveria[j] << " ";

        }

        cout << endl;

    }

}
```

3.1.3 ESTUDIO TEÓRICO

Para hacer el estudio teórico de este algoritmo podemos simplemente ver el número de veces se van a ejecutar los distintos bucles que hay en el problema.

En el primer bucle nos encargamos de rellenar la matriz y el array de averíasPorMecánico depende de las averías y los mecánicos que haya así que su tiempo de ejecución sería M*A.

El siguiente bucle en el que se toma la decisión voraz. Se itera sobre cada avería y en cada repetición se itera sobre cada mecánico, ya que se le asignará la avería al mecánico que menos averías pueda reparar, el tiempo de ejecución de este bucle sería A*M = M*A.

El último bucle sería sacar la solución, que depende del número de averías.

En este análisis deducimos que el su tiempo de ejecución sería M*A + M*A + A que tiene un O(M*A).

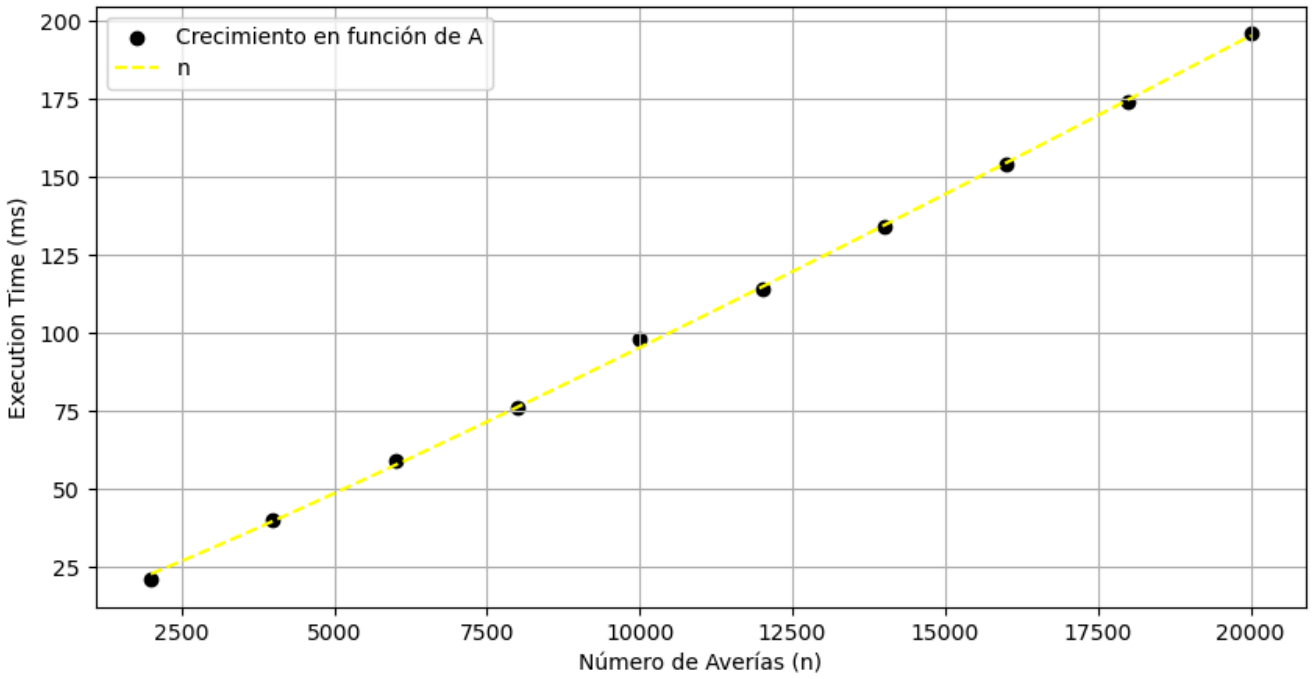
3.1.4 ESTUDIO EXPERIMENTAL

Vamos a hacer 3 estudios diferentes:

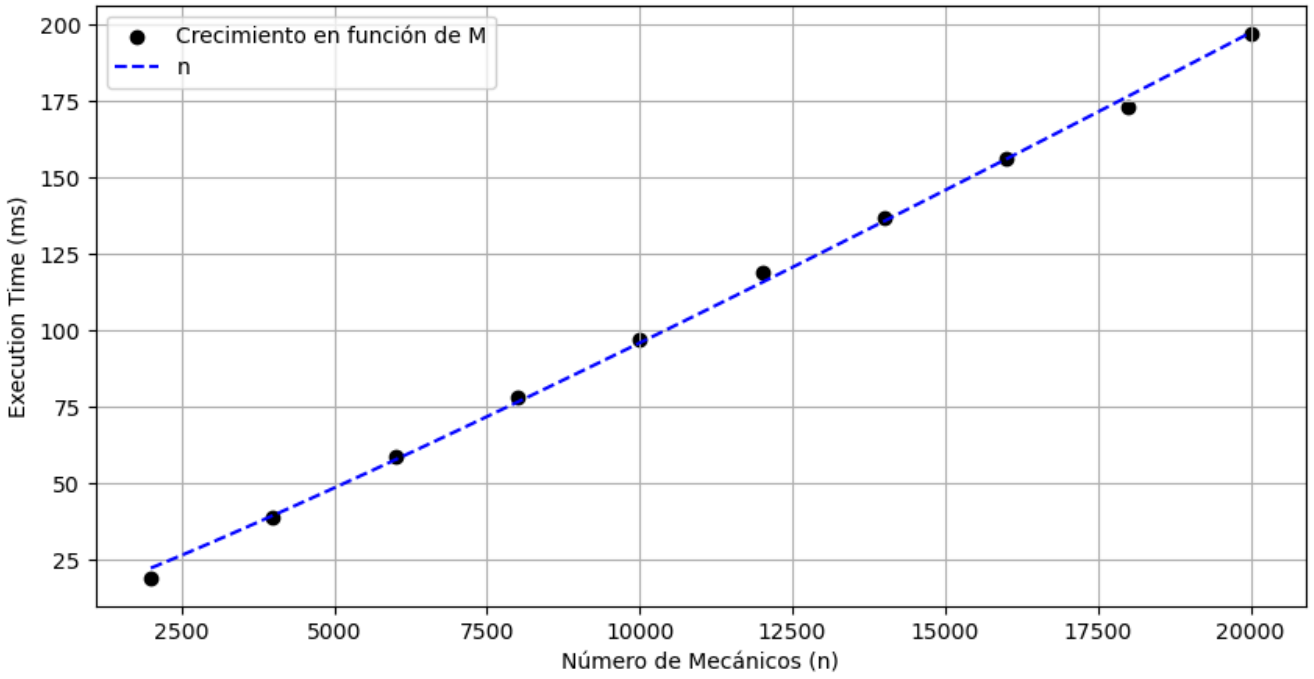
- Tiempo de ejecución en función de A y M constante (M = 100)
- Tiempo de ejecución en función de M y A constante (A = 100)
- Tiempo de ejecución en función de A y M

Si nuestro estudio teórico está bien, en los dos primeros deberíamos ver un crecimiento lineal del tiempo de ejecución y en el último un crecimiento cuadrático ya que vamos a aumentar A y M lo mismo.

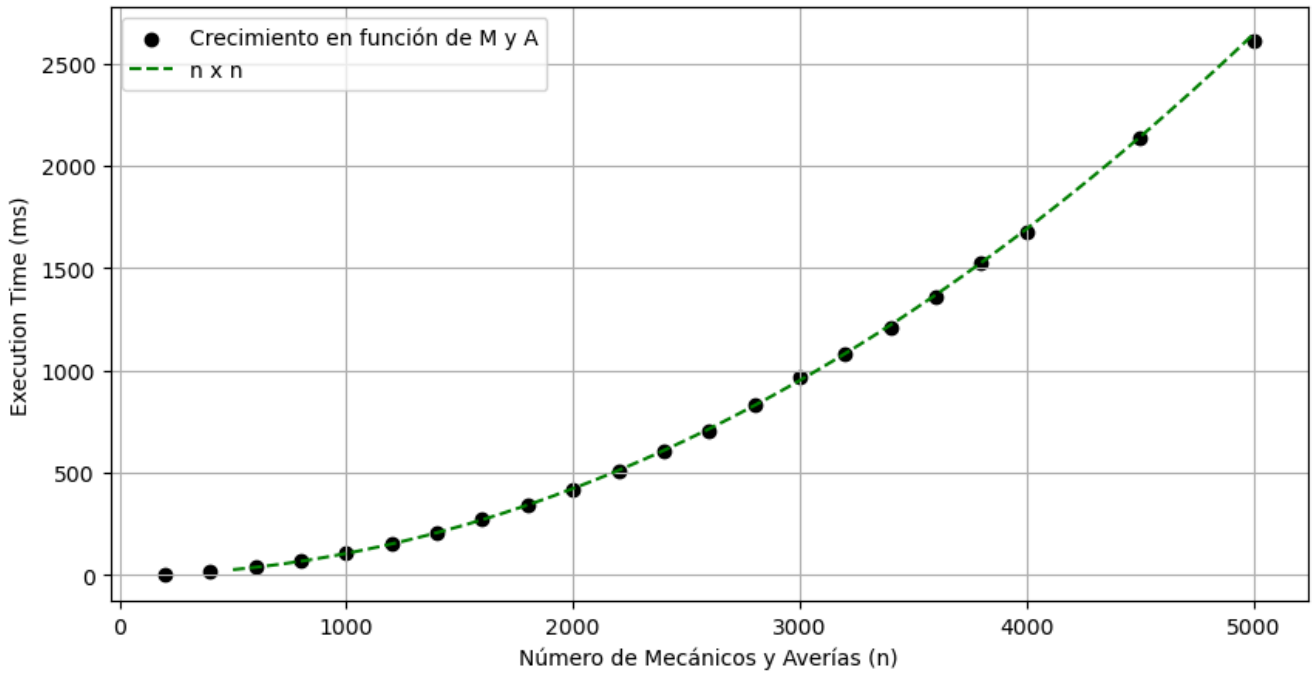
En función de A:



En función de M:



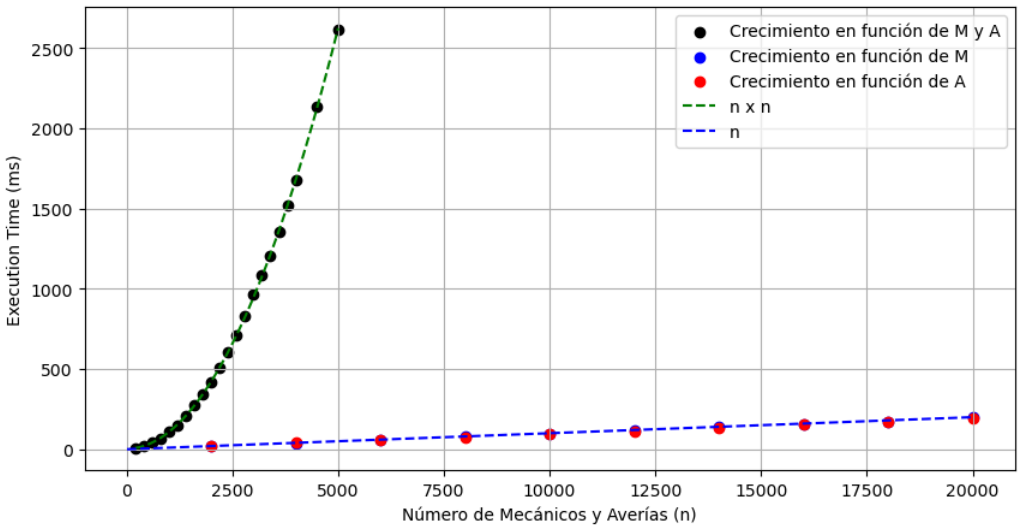
En función de A y M:



3.1.5 CONTRASTE ESTUDIO TEÓRICO Y EXPERIMENTAL

Como muestran los resultados que hemos obtenido, no hay discrepancias entre nuestro estudio teórico y el experimental.

En esta última gráfica mostramos una comparación entre los crecimientos teóricos y los obtenidos en el estudio experimental.



3.2 BACKTRACKING

3.2.1 DISEÑO SOLUCIÓN

PSEUDOCÓDIGO BACKTRACKING

```
var B[MAXN][MAXN];  // matriz de distancias
var voa = -INF;     // valor óptimo actual
var soa[MAXN];      // mejor solución encontrada
var m;              // tamaño del subconjunto
operacion valor(s: vector de enteros, m: entero, B:matriz de enteros):
        var total:=0
        para i:=0, ..., m hacer
                para j:=i+1, ..., m hacer
                        total += B[s[i]][s[j]] + B[s[j]][s[i]]
                finpara
        finpara
        devolver total
operacion generar(nivel: entero, s: vector de enteros)
        si s[nivel] == -1 entonces
                si nivel == 1 entonces
                        s[nivel] = 0
                sino
                        s[nivel] := s[nivel -1] +1
        sino
                s[nivel]++
```

```
operacion criterio( nivel, n: entero, s: array entero):
  devolver s[nivel] < n;
operacion solucion(nivel: entero) :
  devolver nivel == m + 1;
operacion masHermanos(nivel, n: entero, s: array entero)
  devolver s[nivel] < n - (m - nivel);
operacion retroceder(nivel:entero, s:vector de enteros):
        s[nivel] = -1
        nivel - -
operacion backtracking(n: entero):
  var s[-1, ..., -1];
  var nivel := 1;
  var voa := -INF;
  mientras nivel != 0 hacer
    generar(nivel, s);
    si solucion(nivel) entonces
      vat bact := valor(s);
      si bact > voa entonces
        voa := bact;
        soa := s
      finsi
    finsi
    si criterio(nivel, s, n) entonces
      nivel++;
```

```
        s[nivel] := -1;
      finsi
      sino
       mientras   NOT masHermanos(nivel, s, n) AND nivel > 1 hacer
          retroceder(nivel, s)
        finmientras
       si nivel == 1 AND NOT masHermanos(nivel, s, n) entonces
          nivel := 0;
       finsi
    finsino
  finmientras
 Escribir(voa)
```

PSEUDOCÓDIGO BACKTRACKING CON PODA

```
operacion voraz( n:entero, B: matriz de enteros):
        var voa:=0
        para i:=0, ..., n hacer
                para j:=i+1, ..., n hacer
                        voa:=max(voa, B[i][j] + B[j][i]
                finpara
        finpara
        devolver voa
operacion valor(s: vector de enteros, m: entero, B:matriz de enteros):
        var total:=0
        para i:=0, ..., m hacer
```

```
            para j:=i+1, ..., m hacer
                total += B[s[i]][s[j]] + B[s[j]][s[i]]
            finpara
        finpara
        devolver total
operacion generar(nivel: entero, s: vector de enteros)
        si s[nivel] == -1 entonces
            si nivel == 0 entonces
                s[nivel] = 0
            sino
                s[nivel] := s[nivel -1] +1
            finsino
        sino
            s[nivel]++
        finsino
operacion criterio(nivel, m, voa: entero, s: vector de enteros, B:matriz de enteros):
        var parcial  := valor(s, nivel+1, B)
        var faltan := m - (nivel +1)
        var estimado := parcial
        para i := 0, ..., faltan hacer:
            estimado += voa
        finpara
        devolver estimado > voa
operacion solucion(nivel, m:entero):
```

```
    devolver nivel == m -1
operacion masHermanos(nivel, n, m:entero, s:vector de enteros):
    devolver s[nivel] < n - (m -1 -nivel)
operacion retroceder(nivel:entero, s:vector de enteros):
    s[nivel] = -1
    nivel - -
operacion bactracking(n, m: entero, B: matriz de enteros):
    var soa =[-1, .., -1]
    var s =[-1, ..., -1]
    var nivel := 0
    var voa := voraz(n, B)
    var bact := 0
    mientras nivel >=0 hacer:
        generar(nivel,s)
        si NOT masHermanos(nivel, s, n, m) entonces:
            retroceder(nivel, s)
            continuar
        finsi
        si  solucion(nivel, m) entonces
            bact := valor(s, m, B)
            si bact > voa entonces
                voa := bact
                soa := s
            finsi
```

```
            finsi

            si criterio(nivel, s, m, voa, B) entonces

                  nivel ++

                  s[nivel] := -1

            finsi

            sino

                  mientras nivel > 1 AND NOT masHermanos(nivel, s, n, m)  hacer

                        retroceder(nivel, s)

                  finmientras

                  si nivel == 0 AND NOT masHermanos(nivel, s, n, m) entonces

                        nivel := -1

                  finsi

      finmientras

      Escribir(voa)
```

EXPLICACIÓN DEL ALGORITMO

Para resolver el problema hemos decidido utilizar una una matriz de enteros simétrica, B, que representa las distancias entre pares de elementos. Además, se utilizan dos enteros: n (número total de elementos) y m (tamaño del subconjunto deseado).

El algoritmo busca un subconjunto s de tamaño m tal que la suma B[s[i]][s[j]] + B[s[j]][s[i]] (para cada par distinto del subconjunto) sea máxima.

En cuanto al diseño de la solución, la función voraz(n, B), calcula una cota superior simple, para poder llevar a cabo la poda. Para cada par (i, j), se evalúa B[i][j] + B[j][i] y se guarda el valor máximo observado. Este valor (voa) representa el mejor beneficio posible por par, y será usado para estimar cotas superiores en criterio.
Seguidamente, la. función valor(s, m, B) suma los valores reales de los pares del subconjunto s de tamaño m. El objetivo es calcular el valor actual de una solución completa (o parcial si m se reemplaza por nivel+1). Se emplea para evaluar soluciones completas y para construir estimaciones en la poda.

La función generar(nivel, s) asigna el siguiente valor válido en el nivel actual del vector s: si está vacío (-1), asigna el primer valor posible y si ya tenía un valor, lo incrementa para generar al "hermano siguiente". Genera combinaciones de elementos sin repetición y en orden creciente.

La función criterio(nivel, m, voa, s, B) estima el valor máximo posible de una solución a partir de un estado parcial. Suma el valor actual parcial con una estimación basada en cuántos elementos faltan por asignar y la cota voa por cada nuevo par. Si el valor estimado no supera el mejor valor (voa), se descarta esta rama del árbol (poda).

Además, tenemos las funciones: solucion(nivel, m) , que verifica si ya se han seleccionado m elementos; masHermanos(nivel, n, m, s), comprueba si aún se pueden generar más combinaciones en este nivel y retroceder(nivel, s), que borra el valor actual y retrocede un nivel.

Por último, la función backtracking backtracking(n, m, B). Esta función implementa el algoritmo de búsqueda y poda. El árbol de nuestro problema es combinatorio, ya que cada nodo representa una combinación parcial ordenada de elementos, y se explora en profundidad con poda basada en una cota superior. Usa como estructuras de datos: un vector s que representa una combinación parcial, utiliza voa como mejor valor conocido, al llegar a una solución completa, se evalúa y se actualiza el óptimo si corresponde y usa criterio para decidir si continuar expandiendo o podar. Si ya hemos probado todas las posibles opciones en este nivel (nivel), retrocedemos (bajamos al nivel anterior) y continuamos con la siguiente iteración. Si hemos alcanzado el último nivel (nivel == m - 1), entonces tenemos una combinación completa. Calculamos su valor con valor(s, m, B) y, si mejora la mejor que teníamos (voa), la guardamos. Si la solución parcial que llevamos podría mejorar el valor óptimo (voa), avanzamos al siguiente nivel (nivel++) y preparamos ese nuevo nivel con s[nivel] := -1. Es decir, solo seguimos si tiene potencial de ser mejor que lo que ya tenemos. Si no vale la pena seguir con esta rama, empezamos a retroceder hasta encontrar un nivel donde aún queden opciones por explorar.. Si estamos en la raíz (nivel = 0) y ya no hay más opciones, terminamos (nivel := -1). Al final del algoritmo, se muestra el valor óptimo (voa) que corresponde a la mejor combinación encontrada.

3.2.2 IMPLEMENTACIÓN: PROGRAMACIÓN DEL ALGORITMO

Sin poda:

```
#include <cstring>

using namespace std;



#define MAXN 100

#define INF 1000000000
```

```
int B[MAXN][MAXN];  // matriz de distancias
int voa = -INF;     // valor óptimo actual
int soa[MAXN];      // mejor solución encontrada
int m;              // tamaño del subconjunto

//Calculamos la suma de distancias
int valor(int s[]) {
   int total = 0;
   for (int i = 1; i <= m; ++i)
       for (int j = i + 1; j <= m; ++j)
           total += B[s[i]][s[j]] + B[s[j]][s[i]];
   return total;
}

//Generar nivel
void generar(int nivel, int s[]) {
   if (s[nivel] == -1) {
       if(nivel == 1){
           s[nivel] = 0;
       }else{
           s[nivel] = s[nivel -1] +1;
       }
   } else {
```

```
        s[nivel]++;
    }
}

//Comprueba si el valor actual está dentro del rango de valores
bool criterio(int nivel, int s[], int n) {
    return s[nivel] < n;
}

// Verificar si la solucion es la definitiva
bool solucion(int nivel) {
    return nivel == m + 1;
}

// Comprobar si hay más hermanos
bool masHermanos(int nivel, int s[], int n) {
    return s[nivel] < n - (m - nivel);
}

// Volver a una rama anterior
void retroceder(int &nivel, int s[]) {
    s[nivel] = -1;
    nivel--;
}
```

```
void backtracking(int n) {
   int s[MAXN];     // Solución parcial
   memset(s, -1, sizeof(s));  //Inicialización a -1
   int nivel = 1;             // Nivel inicial
   voa = -INF;                // Inicializamos el valor
   while (nivel != 0) {
       // Generar siguiente hermano
       generar(nivel, s);
       //Si completamos la solución
       if (solucion(nivel)) {
           int bact = valor(s);
           if (bact > voa) {
               voa = bact;
               memcpy(soa, s, sizeof(int) * (m + 1));
           }
       }
       //Verificar si merece la pena seguir
       if (criterio(nivel, s, n)) {
           nivel++;
           s[nivel] = -1;
       } else {
           // Retroceder si no hay más hermanos por explorar
           while (!masHermanos(nivel, s, n) && nivel > 1)
               retroceder(nivel, s);
```

```
            if (nivel == 1 && !masHermanos(nivel, s, n))
                nivel = 0;  // Terminar el backtracking
        }
    }
    cout << voa << endl;
}
```

Con poda:

```
// Solucion aproximada para cota superior
int voraz(int n, vector<vector<int>>& B) {
    int voa = 0;
    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            voa = max(voa, B[i][j] + B[j][i]);
        }
    }
    return voa;
}
// Calculamos la suma de distancias
int valor(vector<int>& s, int m, vector<vector<int>>& B) {
    int total = 0;
    for (int i = 0; i < m; ++i) {
        for (int j = i + 1; j < m; ++j) {
            total += B[s[i]][s[j]] + B[s[j]][s[i]];
```

```
        }
    }
    return total;
}
// Generar nivel
void generar(int nivel, vector<int>& s) {
    if (s[nivel] == -1) {
        if (nivel == 0) {
            s[nivel] = 0;
        } else {
            s[nivel] = s[nivel - 1] + 1;
        }
    } else {
        s[nivel]++;
    }
}
// Verifica si merece la pena seguir explorando
bool criterio(int nivel, vector<int>& s, int m, int voa, vector<vector<int>>&
B) {
    int parcial = valor(s, nivel + 1, B);
    int faltan = m - (nivel + 1);
    int estimado = parcial;
    for (int i = 0; i < faltan; ++i) {
      estimado += voa;//Añadir el valor máximo posible para los pares faltantes
```

```
    }
    return estimado > voa;
}
// Verificar si la solucion es la definitiva
bool solucion(int nivel, int m) {
    return nivel == m - 1;
}
// Comprobar si hay más hermanos
bool masHermanos(int nivel, vector<int>& s, int n, int m) {
    return s[nivel] < n - (m - 1 - nivel);
}
// Volver a una rama anterior
void retroceder(int& nivel, vector<int>& s) {
    s[nivel] = -1;
    nivel--;
}
void backtracking(int n, int m, vector<vector<int>>& B) {
    vector<int> s(m, -1);        // Solución parcial
    vector<int> soa(m, -1);      // Mejor solución encontrada
    int nivel = 0;               // Nivel actual
    int voa = voraz(n, B);       // Usamos voraz como cota inicial
    int bact = 0;
    while (nivel >= 0) {
        // Generar siguiente hermano
```

```
    generar(nivel, s);
    // Verificar si podemos continuar con la rama
    if (!masHermanos(nivel, s, n, m)) {
        retroceder(nivel, s);
        continue;
    }
    // Si completamos la solución
    if (solucion(nivel, m)) {
        bact = valor(s, m, B);
        if (bact > voa) {
            voa = bact;
            soa = s;
        }
    }
    // Verificar si merece la pena continuar con la rama
    if (criterio(nivel, s, m, voa, B)) {
        nivel++;    // Siguiente nivel
        s[nivel] = -1;
    } else {
        // Retroceder si no hay más hermanos por explorar
        while (nivel > 1 && !masHermanos(nivel, s, n, m)) {
            retroceder(nivel, s);
        }
        if (nivel == 0 && !masHermanos(nivel, s, n, m)) {
```

```
                nivel = -1;  // Terminar el backtracking
            }
        }
    }
    cout << voa << endl;
}
int main() {
    int T;
    cin >> T;
    for (int nCasos = 0; nCasos < T; nCasos++) {
        int n, m;
        cin >> n >> m;
        vector<vector<int>> B(n, vector<int>(n));
        for (int i = 0; i < n; ++i){
            for (int j = 0; j < n; ++j){
                cin >> B[i][j];
            }
        }
        backtracking(n, m, B);
    }
     return 0;
}
```

3.2.3 ESTUDIO TEÓRICO

Sin poda:

Si no se aplicara ningún criterio de poda, el algoritmo tendría complejidad exponencial, ya que cada solución se evalúa con la función valor(s, m, B), que es cuadrática en m. La cantidad de combinaciones posibles es:

$$\binom{n}{m} = \frac{n!}{m! \cdot (n-m)!}$$

Este es el número máximo de nodos hoja del árbol (es decir, soluciones completas). Además, hay nodos intermedios (soluciones parciales), que está acotado por $O(n^m)$ si m es pequeño comparado con n. Por tanto, el número de nodos por el tiempo por nodo sería:

$$\Theta(n^m \cdot m^2)$$

Con poda (criterio):

En este caso, depende de la función que genera la cota superior, por tanto, tendría un orden de O(C(n, m) * m^2). En general, e reduce el número de nodos explorados, pero la cota sigue siendo exponencial en el peor caso, es decir, sería $O(n^m * m^2)$.

3.2.4 ESTUDIO EXPERIMENTAL

El algoritmo empleado para generar casos de prueba en backtracking:

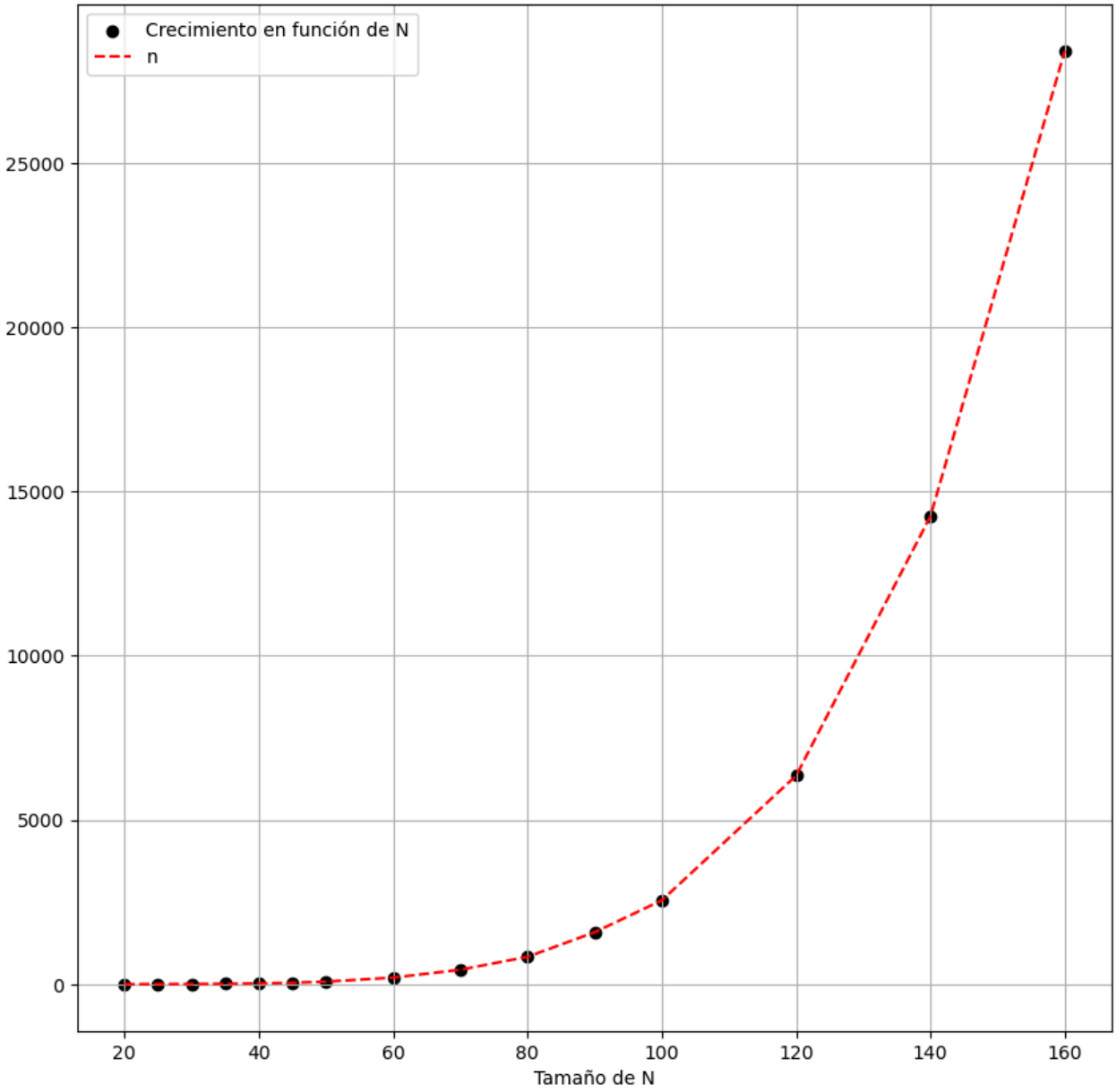
```
#include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>

using namespace std;

// Función para generar un número aleatorio entre min y max

int generarAleatorio(int min, int max) {

   return rand() % (max - min + 1) + min;
```

```
}

// Función para generar los casos de prueba

void generarCasosDePrueba(int T, int max_n = 100, int max_m = 5, int
max_distancia = 100000000) {

   cout << T << endl;  // Imprime el número de casos de prueba

   for (int t = 0; t < T; t++) {

       // Generar valores n y m aleatorios

       int n = generarAleatorio(2, max_n); //cambiar para generar casos

       int m = generarAleatorio(1, min(n, max_m));

       // Crear la matriz de distancias

       vector<vector<int>> distancias(n, vector<int>(n));


       for (int i = 0; i < n; i++) {

           for (int j = 0; j < n; j++) {

               if (i != j) {

                   distancias[i][j] = generarAleatorio(9990000, max_distancia);

               } else {

                   distancias[i][j] = 0; // La distancia consigo mismo es 0

               }

           }

       }

       cout << n << " " << m << endl;

       // matriz de distancias

       for (int i = 0; i < n; i++) {
```

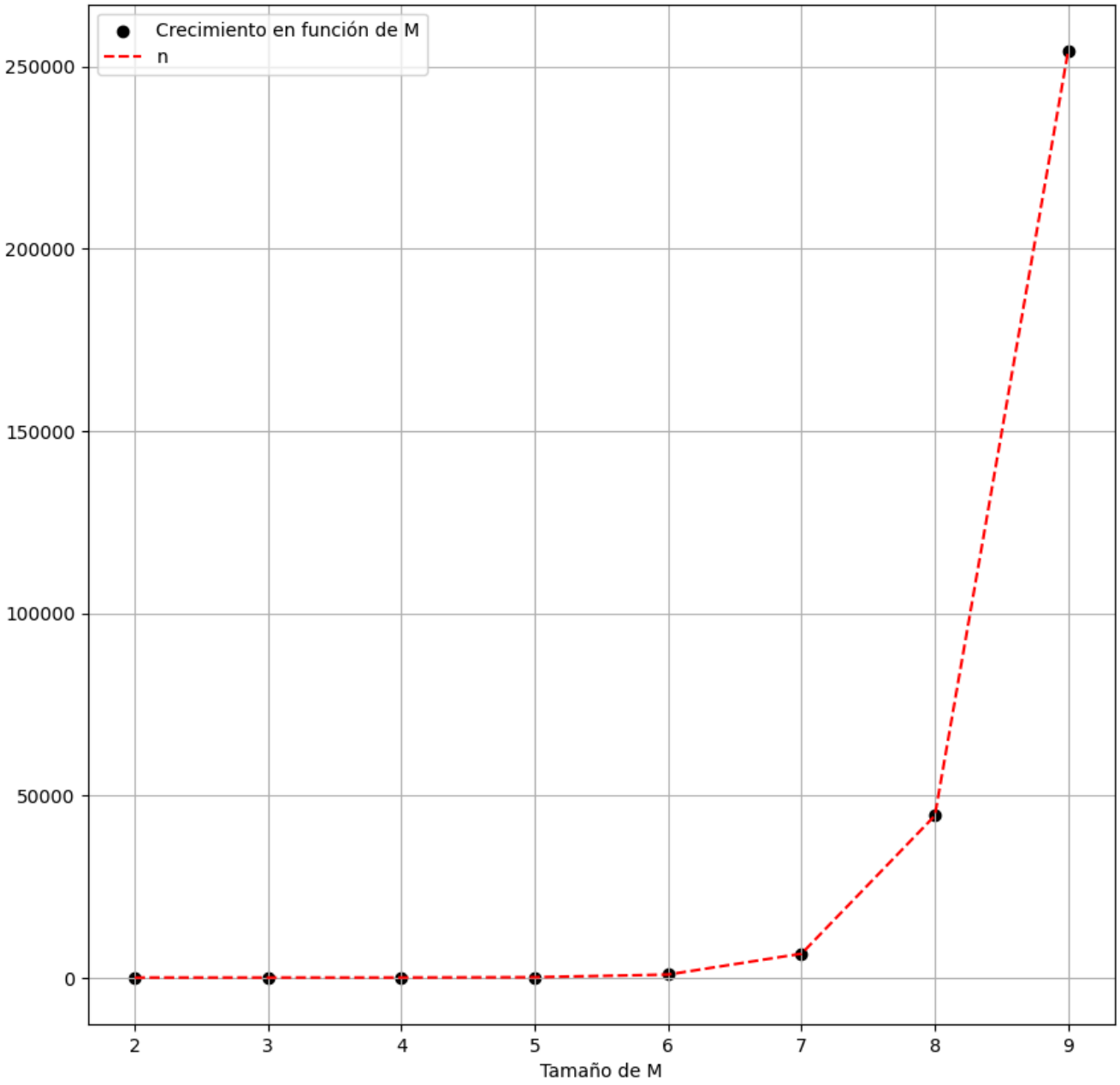
```
            for (int j = 0; j < n; j++) {
                cout << distancias[i][j] << " ";
            }
            cout << endl;
        }
    }
}
int main() {
    srand(time(0));
    int T = 10;  // Número de casos de prueba
    generarCasosDePrueba(T);
    return 0;}
```

El estudio experimental lo hemos realizado con un algoritmo sin poda, cambiando el tamaño máximo de la matriz, ya que en el caso de backtracking con poda depende de la estimación que se haga y cómo generamos valores aleatorios, el tiempo de ejecución del algoritmo es incoherente.

La primera gráfica depende del tamaño de n, es decir, del tamaño de la matriz, manteniendo el valor de m constante (M = 5).

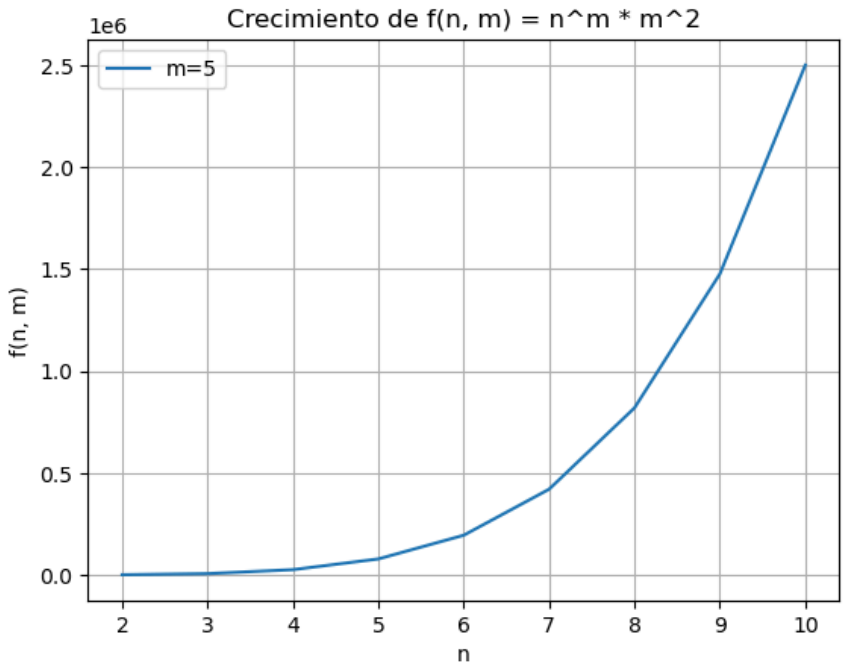


La segunda gráfica muestra el tiempo en función de M, manteniendo el valor de n constante (N = 50)

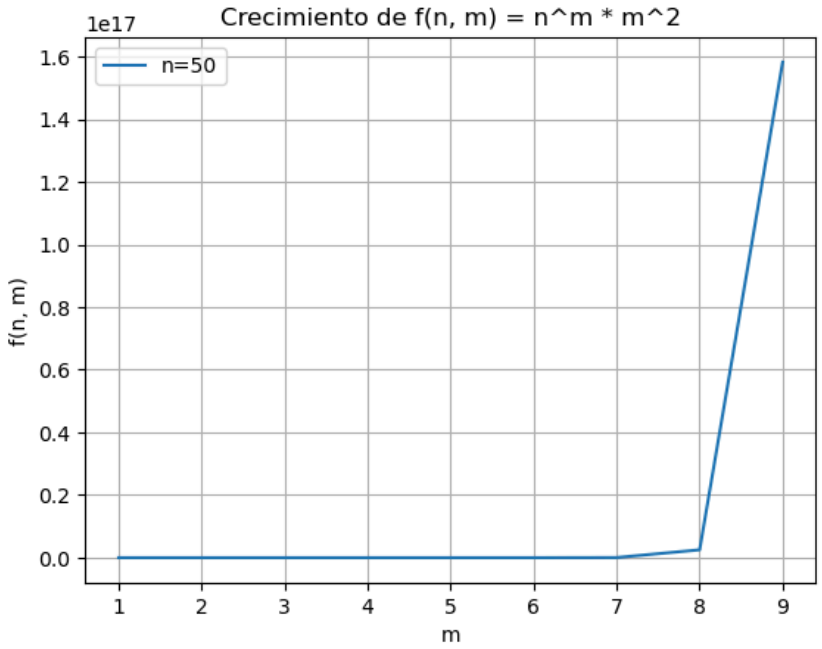

Crecimiento en función de M
n
250000
200000
150000
100000
50000
0
2
3
4
5
6
7
8
9
Tamaño de M

3.2.5 CONTRASTE ESTUDIO TEÓRICO Y EXPERIMENTAL

Como podemos observar, si tomamos como constante el valor de m la gráfica teórica se asemeja mucho a la obtenida en el estudio experimental.



De la misma forma, si tomamos como constante el valor de n, la gráfica teórica también es muy parecida a la experimental.



4. CONCLUSIONES

Esta práctica nos ha parecido muy útil para entender tanto el algoritmo de avance rápido como el de backtracking, pues nos hemos tenido que enfrentar a un problema y aplicar de forma práctica lo aprendido en teoría. Para realizar el algoritmo nos ayudamos de las diapositivas de teoría, así como de los scripts de código que hemos encontrado en el repositorio.

Las mayores dificultades que hemos afrontado son el análisis tanto teórico como experimental, ya que en el caso de backtracking nos costó determinar el tipo de árbol así como cuanto tiempo tarda en ejecutar cada nodo.

Hemos calculado que aproximadamente le hemos dedicado unas 30 horas al proyecto cada miembro, contando el tiempo invertido planteando el algoritmo, refinando el código, haciendo los respectivos análisis, etc.

En definitiva, nos ha resultado un trabajo muy importante para lograr comprender las dos técnicas, aunque nos haya llevado su tiempo terminarlo.