
UNIVERSIDAD DE GRANADA

**Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones**

# Práctica 4: Exploración En Grafos

Doble Grado en Ingeniería Informática y Matemáticas

Algorítmica

Autores y correos:
Jaime Corzo Galdó: jaimecrz04@correo.ugr.es
Marta Zhao Ladrón de Guevara Cano: mzladron72@correo.ugr.es
Sara Martín Rodríguez: smarrod02@correo.ugr.es

27 de septiembre de 2024

# Índice

# 1. Autores

A continuación se va a presentar el trabajo realizado por cada uno de los componentes del grupo.

- Algoritmos y funciones de cota: han sido implementados entre los tres
- Análisis de eficiencia Backtracking: Marta
- Análisis de eficiencia Branch and Bound: Jaime
- Demostraciones de validez: Sara
- Comparación de las dos técnicas: Sara
- Memoria y presentación: entre los tres.

# 2. Objetivos

El objetivo de esta práctica es comprender y asimilar el funcionamiento de las técnicas de resolución de problemas basadas en exploración de grafos: Backtracking y Branch and Bound.

# 3. Definición del problema

El problema que nos fue asignado ha sido *Problema de la cena de gala*. El enunciado es el siguiente: se va a celebrar una cena de gala a la que asistirán $n$ invitados. Todos se van a sentar alrededor de una mesa circular, de forma que cada individuo tendrá sentados junto a él a dos comensales (uno a su izquierda y otro a su derecha). Las normas de protocolo establecen la conveniencia de que dos individuos, en base al cargo que ocupan, se sienten en lugares contiguos. Por simplicidad se ha asumido que el valor de conveniencia es un número entero positivo entre 0 y 1000. El nivel de conveniencia global se define como la suma de todos los niveles de conveniencia de cada invitado con cada uno de los dos comensales que tiene sentados a su lado.

El objetivo es sentar a los invitados de forma que el nivel de conveniencia global sea lo mayor posible; es decir, se trata de un problema de maximización.

A lo largo de la práctica se van a implementar dos técnicas de exploración de árboles para llegar a la solución correcta como ya se ha comentado previamente, el Backtracking y el Branch and Bound. La principal diferencia entre ambos es que el Backtracking se trata de una algoritmo recursivo mientras que el Branch and Bound es un algoritmo iterativo, y es por ello que los datos que recibe como entrada cada uno son distintos, los cuáles se detallarán más adelante . Sin embargo, ambos devuelven lo mismo, la mejor disposición de los comensales con la que se obtiene la mayor optimización de la conveniencia global.

Basándonos en el problema en concreto, sabemos que debemos colocar a todos los comensales alrededor de una mesa, luego la solución será un vector indicando la

disposición de éstos y para identificarlos, el vector almacenará un entero que será el identificador de cada comensal. Es decir, la solución se va a tratar de una n-upla $(x_1, x_2, x_3, ..., x_n)$ que maximice la conveniencia.

## 3.1. Casos usados en la evaluación de la eficiencia

Para realizar la evaluación de la eficiencia se van a ejecutar los algoritmos para distintos tamaños. Teniendo en cuenta que se tratan de algoritmos con una gran carga computacional, se va a ejecutar el algoritmo con problema de tamaño 20; es decir, suponiendo que tenemos que sentar 20 comensales a la mesa con saltos de dos en dos; es decir: 2, 4, 6, ..., 20

## 3.2. Entorno de análisis

A continuación, se detallan las características de los ordenadores de cada uno de los integrantes del grupo:

1. Jaime
    - Intel® Core™ i7-1065G7 CPU @ 1.30GHz × 8
    - Memoria RAM de 16GB.
    - Capacidad de disco de 512.1GB.
    - Sistema operativo Ubuntu 22.04.2 LTS 64 bits.
    - Compilador g++
2. Marta
    - Procesador Intel Core i5-10210U CPU @ 1.60Hz x 8.
    - Memoria RAM de 8GB.
    - Capacidad de disco de 512.1GB.
    - Sistema operativo Ubuntu 22.04.2 LTS 64 bits.
    - Compilador g++
3. Sara
    - Procesador 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80 GHz x 8.
    - Memoria RAM de 8GB.
    - Capacidad de disco de 512 GB.
    - Sistema operativo Ubuntu 22.04.2 LTS 64 bits.
    - Compilador g++

## 3.3. Descripción del método de medición de tiempos

Para realizar la medición de los tiempos se ha hecho uso de la biblioteca de la STL, chrono, que permite calcular el tiempo de forma precisa. Para ello se deben incluir en el código las siguientes sentencias:

```
high_resolution_clock::time_point t_antes,t_despues;
duration<double> transcurrido;

t_antes = high_resolution_clock::now();

Solucion sol= Backtracking(num:_invitados);

t_despues = high_resolution_clock::now();

transcurrido=duration_cast<duration<double>>(t_despues - t_antes);
```

# 4. Técnica Backtracking

## 4.1. Algoritmo diseñado

### 4.1.1. Descripción del algoritmo

El diseño total de este algoritmo consta de varios elementos.

1. En primer lugar, se ha implementado la clase **Solución** que consiste en lo siguiente:

   - Atributos privados (explicación en los comentarios del código)

   ```
   const int NULO = -1; //constante que indica si un asiento
       esta libre
   const int END =-2; // constante que indica si es el final
       de la mesa
   int mejor_optimizacion; // guarda la conveniencia total
       mejor hasta el momento
   int num_invitados; // numero de invitados a sentar
   int num_inv_sentados; // numero de invitados sentados
   vector<int> solucion; // guarda la solucion encontrada
       hasta el momento
   vector<int> mejor_solucion; // guarda la mejor solucion
       encontrada hasta el momento
   vector<vector<int>> datos; // matriz de conveniencias
       entre invitados
   vector<bool> estan_sentados; // guarda la disponibilidad
       de cada asiento
   vector<int> nodos; // vector con todas las conveniencias
   ```

   - Métodos privados:

     **Método ConsigueSuma**: calcula la conveniencia total que se obtiene con una disposición concreta de los comensales.

```
int ConsigueSuma(const vector<int> &v, int tam){

    int suma = 0;

    if(tam>1){

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

            int val = v.at(i);

            if (i==0){
                suma += datos.at(val).at(v.at(i+1));
                suma += datos.at(val).at(v.at(tam-1));

            }
            else{
                if (i<(tam - 1)){
                    suma += datos.at(val).at(v.at(i+1));
                    suma += datos.at(val).at(v.at(i-1));

                }
                else{
                    suma += datos.at(val).at(v.at(0));
                    suma += datos.at(val).at(v.at(i-1));
                }
            }
        }
    }

    return suma;
}
```

**Método MejorSolución**: devuelve si la solución obtenida es la mejor hasta el momento.

```
bool MejorSolucion(){
    int suma = ConsigueSuma(solucion, solucion.size());
    return (suma > mejor_optimizacion);
}
```

**Método CalculaCotaLocal**: calcula la cota local (la mejor solución que se podría obtener al expandir un nodo) asociada a un nodo usando la función de cota que se indica el entero pasado como parámetro.

```
int CalculaCotaLocal(int cota){

    int cotalocal = DecisionesTomadas();

    if(cota==1){
        cotalocal += Estimador1();
    }
    else{
        if(cota==2)
            cotalocal += Estimador2();
        else
            cotalocal += Estimador3();
    }

```

```
    return cotalocal;
}
```

**Método DecisionesTomadas**: calcula el valor total de conveniencia hasta el último invitado que ha sido sentado.

```
int DecisionesTomadas(){
    return ConsigueSuma(solucion, num_inv_sentados);
}
```

Además, entre los métodos privados también se incluyen 3 funciones de cota o de poda que se explicarán más adelante.

- Métodos públicos:

**Constructor de Solución**: se pasa como parámetro la matriz con los datos de las conveniencias y e inicializa todos los atributos adecuadamente. Además, dicha matriz de conveniencias se pasa a un vector de enteros que posteriormente se ordena de manera decreciente.

```
Solucion(const vector<vector<int>> &matriz) {

    datos = matriz;
    num_invitados=matriz.size();
    num_inv_sentados = 0;
    mejor_solucion = Greedy();

    // Inicializamos con conveniencia total de la mejor solucion (al principio es Greedy)
    mejor_optimizacion=ConsigueSuma(mejor_solucion, mejor_solucion.size());

    // Ponemos todos los invitados de la solucion a nulo (quiere decir que el sitio esta libre)
    for(int i=0; i<num_invitados; i++){
        solucion.push_back(NULO);
    }

    // Ponemos todos los sitios como libres
    for(int i=0; i<num_invitados; i++){
        estan_sentados.push_back(false);
    }

    IniciaComp(0);
    SigValComp(0);
    estan_sentados.at(0)=true;

    // Inicializamos el vector de nodos
    for(int i=0; i<num_invitados; i++){
        for(int j=0; j<num_invitados; j++){
            nodos.push_back(datos.at(i).at(j));
        }
    }

    // Ordenamos el vector de nodos por conveniencia entre invitados
```

```
33      sort(nodos.begin(), nodos.end(), orden_decreciente);
34  }
```

**Método getNumInvitados()**: tal y como indica el nombre devuelve el número de invitados que se han de sentar.

```
1  int getNumInvitados() const{
2      return num_invitados;
3  }
```

**Método getSolución()**: devuelve el vector con la solución que se ha encontrado hasta el momento.

```
1  vector<int> getSolucion(){
2      return solucion;
3  }
```

**Método getMejorSolución()**: devuelve el vector la mejor solución que se ha encontrada hasta el momento.

```
1  vector<int> getMejorSolucion(){
2      return mejor_solucion;
3  }
```

**Método IniciaComp()**: asigna el valor nulo a la posición k-ésima del vector solución.

```
1  void IniciaComp(int k){
2      solucion.at(k)=NULO;
3  }
```

**Método SigValComp()**: asigna el siguiente valor válido de la posición k-ésima.

```
1  void SigValComp(int k) {
2
3      int val = solucion.at(k);
4
5      if(val<num_invitados-1){
6          if(val==NULO){
7              num_inv_sentados++;
8          }
9
10         val++;
11         solucion.at(k)=val;
12     }
13     else{
14         solucion.at(k)=END;
15     }
16 }
```

**Método TodosGenerados()**: comprueba la posición k es el final de la solución.

```
1  bool TodosGenerados(int k){
2      return (solucion.at(k)==END);
3  }
```

**Método ProcesaSolucion()**: comprueba si la solución obtenida es la mejor posible.

```
void ProcesaSolucion(){

    if (MejorSolucion()){
        mejor_solucion=solucion;
        mejor_optimizacion=ConsigueSuma(mejor_solucion,
   mejor_solucion.size());
    }
}
```

**Método Factible()**: devuelve "true"si la solución actual almacenada cumple con las restricciones y false en caso contrario.

```
void Factible(){

   bool es_factible = false;

    if (!estan_sentados.at(solucion.at(k))){
        int suma_local = CalculaCotaLocal(cota);

        if (suma_local>mejor_optimizacion) {
            es_factible = true;
            estan_sentados.at(solucion.at(k))=true;
        }
    }

    return es_factible;
}
```

**Método erase_invitado()**: quita el invitado de la posición $k$ de la mesa .

```
void erase_invitado(int k){
    int val = solucion.at(k);
    estan_sentados.at(val)=false;
    num_inv_sentados--;
}
```

**Método Greedy()**: obtiene una solución posible (se lanza para la primera solución).

```
vector<int> Greedy() {
    vector<int> sol;
    int suma = 0;
    vector<bool> usados;
    int num_invitados = datos.size();

    for (int i=0; i<num_invitados; i++){
       usados.push_back(false);
    }

    int invitado_actual=0;
    sol.push_back(0);
    usados.at(0)=true;

    while (sol.size()<num_invitados){

       int max=-1;
       int pos_max=invitado_actual;

```

```
20          for (int i=0; i<num_invitados; ++i){
21              int c = datos.at(invitado_actual).at(i);
22              if (c>max && c>0 && !usados.at(i)) {
23                  max=c;
24                  pos_max=i;
25              }
26          }
27
28          usados.at(pos_max)=true;
29          sol.push_back(pos_max);
30          invitado_actual=pos_max;
31      }
32
33      suma = ConsigueSuma(sol, sol.size());
34
35      return sol;
36  }
```

2. Por otro lado, se ha implementado la función **back_recursivo()**:

   - La función tiene como parámetros un objeto del tipo *Solución* por referencia donde se va construyendo la solución y se va guardando la mejor de ellas, la posición $k$ del nodo vivo a desarrollar y el número de cota.
   - El funcionamiento de la función es el siguiente. Primero, se comprueba si la posición del nodo vivo a desarrollar que se ha pasado como parámetro es igual al número de invitados. En ese caso se llama a la función **ProcesaSolucion** que guarda la solución en caso de que sea mejor que otra obtenida anteriormente. En el caso contrario se inicializa el nodo vivo k. Si este es factible, se crean todos sus nodos hijos hasta desarrollar toda la hoja. Cuando se acaba, se realiza una vuelta atrás eliminando todos los hijos hasta llegar de nuevo al nodo k, que se sustituye por el siguiente nodo vivo.

```
1  void back_recursivo(Solucion &s, int k, int cota){
2
3      if(k==s.getNumInvitados()){
4          s.ProcesaSolucion();
5      }else{
6          s.IniciaComp(k);
7          s.SigValComp(k);
8          while(!s.TodosGenerados(k)){
9              if(s.Factible(k,cota)){
10                 back_recursivo(s, k+1,cota);
11                 s.erase_invitado(k);
12             }
13
14             s.SigValComp(k);
15         }
16     }
17 }
```

### 4.1.2. Demostración de validez

Partimos del hecho de que un algoritmo Backtracking realiza un recorrido en profundidad del árbol de estados, de manera que si no se hace uso de ninguna cota, el algoritmo calcularía todas las posibles soluciones y tomaría la mejor, lo cual nos garantiza que se llega a la solución óptima. Esto tiene como inconveniente, que se trata de un algoritmo con una eficiencia exponencial. El uso de cotas y restricciones, ya sean implícitas o explícitas, tienen como objetivo mejorar esta eficiencia mediante funciones de cota que eliminarán aquellas ramas que no llevan a solución y explorando a aquellas que son candidatas de devolver una.

Supongamos que $S$ es la solución óptima de algoritmo y que se trata de una solución con $n$ invitados, esto quiere decir que:

$\sum_{i=1}^{n} conveniencia(i) \geq \sum_{i=1}^{k} conveniencia(i)\ con\ k \in \mathbb{N}\,, 1 \leq k < n$
tal que $conveniencia(i)$ es la conveniencia del invitado i-ésimo con los que tiene sentados al lado.
Si vamos por la componente k-ésima el algoritmo inicializara la componente k+1 y por el método de selección seguirá explorando ese nodo hasta llegar a la solución $S$, pues las funciones de poda tienen ese objetivo, que es testeara si la tupla que se está formando tiene posibilidad de éxito, y en este caso explorarán del árbol hasta llegar a la solución. Pues en caso de no seguir explorando se estaría ante una implementación errónea del algoritmo.

En este tipo de implementaciones, por ejemplo la ejecución de un algoritmo Greedy de eficiencia cuadrática en el constructor del objeto Solución son prácticamente insignificantes, pues la carga computacional de la recursividad es muy alta.

### 4.1.3. Funciones de cota

Se han diseñado 3 funciones de cota o poda para poder mejorar el tiempo de ejecución del algoritmo de Backtracking. Estas se han incluido entre los métodos privados de la clase Solución. Se utilizan según la cota que elija el usuario.

- **Método Estimador1()**: se toma la mayor de todas las conveniencias del vector de nodos (donde previamente se han guardado todos los datos y se han ordenado de mayor a menor) siempre que haya algún invitado sentado. Para estimar la conveniencia máxima que se puede obtener se multiplica el resultado por el número de invitados sin sentar y por dos pues la conveniencia de un invitado depende de los sentados a la izquierda y a la derecha.

```
int Estimador1(){

    int invitados_sin_sentar= num_invitados - num_inv_sentados;
    int estimador=0;

    if(num_inv_sentados>0){
        estimador = nodos.front();
    }

    return 2*estimador*invitados_sin_sentar;
}
```

- **Método Estimador2()**: siempre que haya invitados sentados para estimar la conveniencia máxima que se puede obtener, acumulamos la conveniencia máxima se toma para cada uno (cada fila de la matriz *datos*) en valor máximo de conveniencia y se multiplica por *dos* pues su conveniencia depende del invitado de ambos lados.

```
int Estimador2(){
    int estimador2 = 0;
    if(num_inv_sentados > 0) {
        int tam = getNumInvitados();

        for (int i = 0; i < tam; i++) {
            if (!estan_sentados.at(i) && i != solucion.at(
    num_inv_sentados - 1)) {
                vector<int>::iterator it = max_element(datos.at
    (i).begin(), datos.at(i).end());
                estimador2 += 2 * (*it);
            }
        }
    }
    return estimador2;
}
```

- **Método Estimador3()**: siempre que haya invitados sentados para estimar la conveniencia máxima que se puede obtener, sumamos para cada invitado las dos máximas conveniencias con otros invitados, es decir para cada fila de la matriz de datos tomamos los dos enteros más grandes y lo acumulamos en el estimador.

```
int Estimador3(){
        int estimador3 = 0;
        int max = 0;
        if(num_inv_sentados>0) {
            int tam = getNumInvitados();
            for (int i = 0; i < tam; i++) {
                max = 0;
                if (!estan_sentados.at(i) && i != solucion.at(
    num_inv_sentados - 1)) {
                    // Buscamos la conveniencia maxima de cada
    fila
                    int indice_max;
                    for (int j = 0; j < datos.at(i).size(); j
    ++) {

                        if (datos.at(i).at(j) > max){
                            max = datos.at(i).at(j);
                            indice_max = j;
                        }

                    }

                    // Buscamos el segundo maximo de esa misma
    fila
                    int otro_max = 0;
                    for (int j = 0; j < datos.at(i).size(); j
    ++) {
```

```
23
24                          if (datos.at(i).at(j) > otro_max &&
    indice_max != j)
25                              otro_max = datos.at(i).at(j);
26                      }
27
28                      // La maxima conveniencia de cada inivitado
     estara compuesta por la suma de los dos maximos de su fila
29                      max += otro_max;
30                  }
31
32                  estimador3 += max;
33              }
34          }
35
36          return estimador3;
37      }
```

## 4.2. Análisis de eficiencia

### 4.2.1. Análisis teórico

Realizar un análisis teórico o híbrido es una tarea difícil pues no se trata de un algoritmo que simplemente genere todas las posibles soluciones y luego tome la óptima, sino que realiza operaciones de cota para reducir el número de nodos a expandir y por tanto, el tiempo de ejecución. El objetivo es podar las ramas del árbol virtual lo antes posible, pero todas estas operaciones hacen más complejo o casi imposible encontrar una función que se ajuste, pues que al final ya depende del problema en concreto el momento en que se realicen las cotas.
Es por ello, que la eficiencia del backtracking va a depender del número de llamadas a cada una de las funciones, del tiempo necesario para generar la siguiente componente del número de soluciones generadas que satisfagan las restricciones implícitas, del tiempo en determinar la solución de factibilidad o del número de nodos generados.

Como se ve en la implementación recursiva del backtracking, para cada posición del vector solución se realiza una llamada recursiva a la propia función, luego $T(n) = n * T(n-1)$, lo cual supone una eficiencia exponencial.

### 4.2.2. Análisis empírico

| Núm. inivitados | Tiempo (seg) |
|---|---|
| 1 | 2.26e-07 |
| 2 | 6.06e-07 |
| 3 | 2.123e-06 |
| 4 | 4.784e-06 |
| 5 | 5.061e-06 |
| 6 | 2.5246e-05 |
| 7 | 1.2864e-05 |
| 8 | 6.1335e-05 |
| 9 | 0.000140356 |
| 10 | 0.000295105 |
| 11 | 0.0123971 |
| 12 | 9.2463e-05 |
| 13 | 0.000178548 |
| 14 | 0.00902211 |
| 15 | 0.176633 |
| 16 | 0.0173022 |
| 17 | 0.0610204 |
| 18 | 0.229655 |
| 19 | 0.120442 |
| 20 | 0.0357941 |

Cuadro 1: Tabla Backtracking (uso de cota 1)

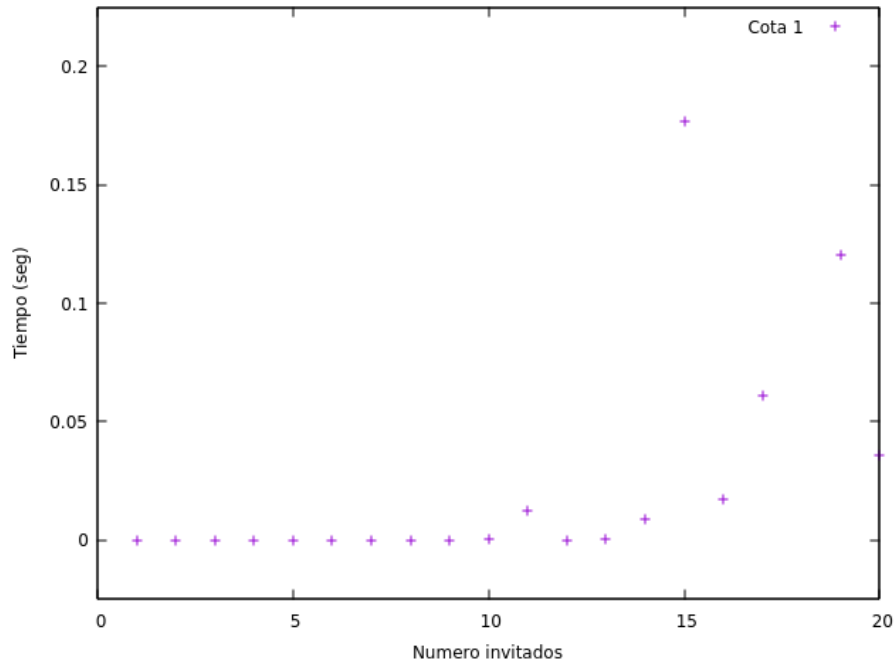


Figura 1: Gráfica Backtracking para la Cota 1

| Núm. inivitados | Tiempo (seg) |
|---|---|
| 1 | 3.17e-07 |
| 2 | 1.118e-06 |
| 3 | 2.813e-06 |
| 4 | 3.504e-06 |
| 5 | 6.188e-06 |
| 6 | 4.8838e-05 |
| 7 | 2.0314e-05 |
| 8 | 7.3337e-05 |
| 9 | 6.7853e-05 |
| 10 | 0.000657306 |
| 11 | 0.0201522 |
| 12 | 0.00334337 |
| 13 | 0.0115399 |
| 14 | 0.0269769 |
| 15 | 0.302986 |
| 16 | 0.422951 |
| 17 | 7.46373 |
| 18 | 35.1566 |
| 19 | 13.4768 |
| 20 | 514.225 |

Cuadro 2: Tabla backtracking (uso de cota 2)

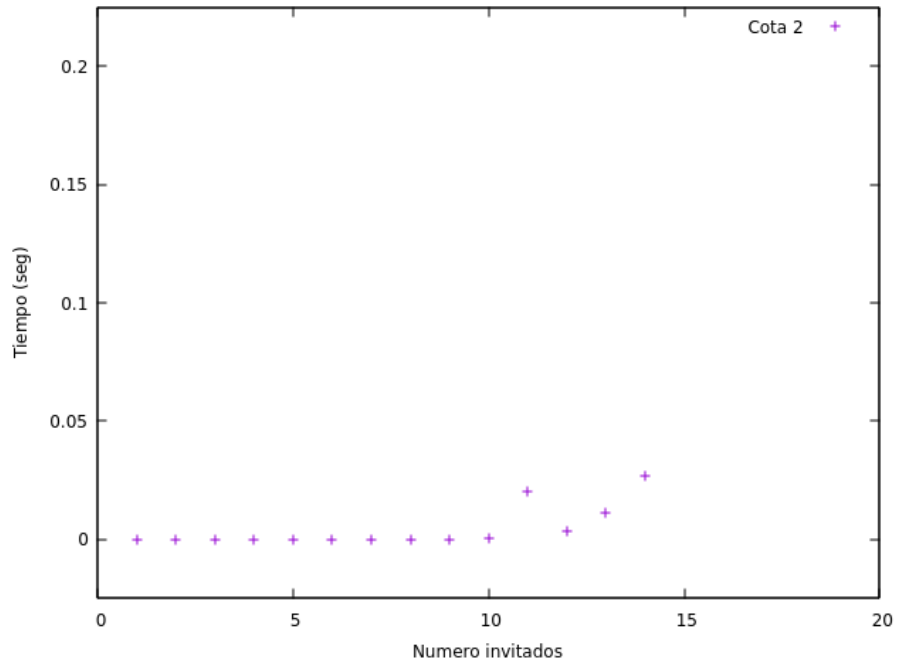


Figura 2: Gráfica Backtracking para la Cota 2

| Núm. inivitados | Tiempo (seg) |
|---|---|
| 1 | 6.23e-07 |
| 2 | 6.83e-07 |
| 3 | 3.333e-06 |
| 4 | 4.93e-06 |
| 5 | 5.714e-06 |
| 6 | 4.7246e-05 |
| 7 | 6.8042e-05 |
| 8 | 1.3649e-05 |
| 9 | 0.000285906 |
| 10 | 0.000479275 |
| 11 | 0.00646103 |
| 12 | 0.00894766 |
| 13 | 0.00662544 |
| 14 | 0.0333424 |
| 15 | 0.225136 |
| 16 | 0.146996 |
| 17 | 7.78845 |
| 18 | 30.9359 |
| 19 | 7.16152 |
| 20 | 183.969 |

Cuadro 3: Tabla backtracking (uso de cota 3)

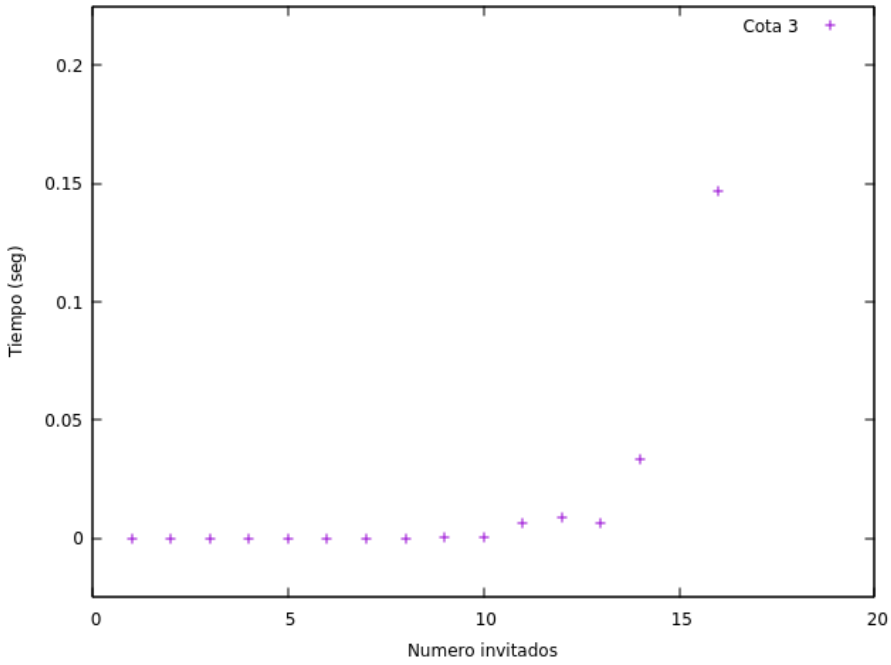


Figura 3: Gráfica Backtracking para la Cota 3

### 4.2.3. Análisis híbrido

Como vimos en el análisis teórico, la ejecución de este tipo de algoritmos que implementan funciones de cota dependen de muchos aspectos, además del tamaño del problema. Es por ello, que no existe una función que se ajuste a los datos obtenidos.

## 4.3. Análisis comparativo de las cotas

Para hacer un análisis comparativo de las tres funciones de poda se va proceder a realizar un análisis de eficiencia:

- **Estimador 1**: tiene eficiencia $O(1)$ pues todas las operaciones son constantes ya que en la variable *estimador* se toma directamente el primer elemento del vector de nodos previamente ordenado en orden decreciente asegurándonos que tomamos el máximo.
- **Estimador 2**: tiene eficiencia $O(n^2)$, debido a que en el peor caso para todos los invitados se comprueba si están sentados (operación constante) y se obtiene su conveniencia máxima con el uso del algoritmo *max_element* de la biblioteca STL que tiene eficiencia lineal y se añade al estimador multiplicado por dos pues depende de los invitados sentados a ambos lados.
- **Estimador 3**: tiene eficiencia $O(n^2)$, ya que dentro del bucle principal que recorre todos los invitados encontramos dos bucles consecutivos que buscan los dos máximos con operaciones constantes.

Estas tres funciones nos dan una estimaciones adecuadas pero la más rápida sin duda es la primera y se refleja en los tiempos obtenidos en la ejecución del Backtracking para las 3 cotas. Además, se puede ver en el siguiente gráfico que compara los tiempos de las tres ejecuciones:

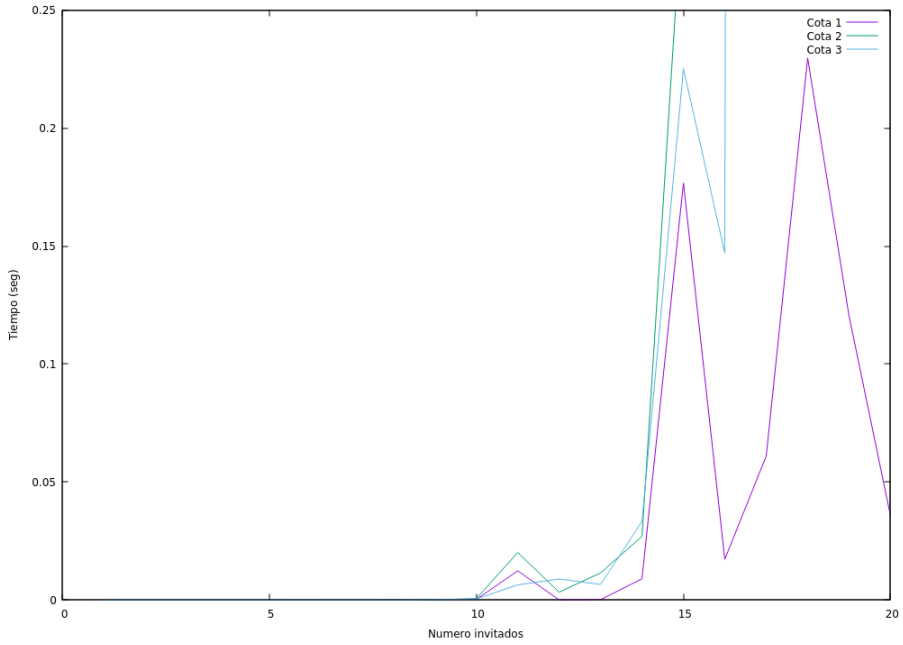


Figura 4: Comparación de Backtracking para las 3 cotas

# 5. Técnica Branch and Bound

## 5.1. Algoritmo diseñado

### 5.1.1. Descripción del algoritmo

El diseño total de este algoritmo consta de varios elementos.

1. En primer lugar, se ha implementado la clase **Solución** que consiste en lo siguiente:

   - Atributos privados

```
const int NULO = -1; //constante que indica si un asiento esta libre
const int END =-2; // constante que indica si es el final de la mesa
vector<int> solucion; // Almacenamos la solucion
int pos_e;  // Posicion de la ultima decision en solucion
int conveniencia_acumulada; // Conveniencia acumulada
int invitados_sentados; // Numero de invitados sentados
int estimador_; // Valor del estimador para el nodo X
int num_invitados;  //Tamanio del problema
vector<vector<int>> datos;   //Conveniencias entre los distintos invitados
vector<bool> estan_sentados;  //Indica si un invitado ha sido sentado
vector<nodo> nodos;  //Distintas conveniencias para poder ordenarlas
int num_cota;  //Indica la cota a emplear

```

- Metodos públicos

  **Constructor de Solución**: se pasa como parámetro la matriz con los datos de las conveniencias y la cota que se va a usar en la ejecución. Se inicializa todos los atributos adecuadamente. Además, dicha matriz de conveniencias se pasa a un vector de enteros que posteriormente se ordena de manera decreciente. Por último, se incializa la posicion 0 del vector solución a 0 para evitar soluciones repetidas.

```
Solucion( const vector<vector<int>> & m, int cota){
        num_cota = cota;
        num_invitados = m.size();
        datos = m;
        for(int i=0; i<num_invitados; i++){
            solucion.push_back(NULO);
        }

        conveniencia_acumulada=0;
        invitados_sentados=0;
        pos_e=-1;
        estimador_=0;

        for(int i = 0; i < num_invitados; i++){
            estan_sentados.push_back(false);
        }

        for(int i = 0; i <num_invitados; i++){
            for(int j = 0; j < num_invitados; j++){
                if(i != j){ //Los de la diagonal no nos
    interesan

                    nodo nuevo={i,j, datos.at(i).at(j)};
                    nodos.push_back(nuevo);

                }

            }
        }

        sort(nodos.begin(), nodos.end());
        PrimerValorComp(0);
}
```

  **Método Factible**(Comprueba que para las posiciones anteriores a pos_e se satisfagan las condiciones implícitas, es decir, que cada invitado solo se puede sentar una vez)

```
    bool Factible() const {
        bool es_factible = true;
        int i = 0;

        while (i <pos_e && es_factible){
            if(solucion.at(i) == solucion.at(pos_e))
    es_factible = false;
            else i++;
        }

        return es_factible;
```

```
    }

```

**Método Cota Local**(En función de la cota que se le pase como parámetro, devuelve el resultado de llamar a un Estimador u otro)

```
    int CotaLocal(int cota) const{
        int cota_local =0;
        if(cota==1){
            cota_local += Estimador1();
        }else if(cota==2){
            cota_local += Estimador2();
        }else{
            cota_local += Estimador3();
        }

        return cota_local;
    }

```

**Evalua**(Llama al método Evalua2 para obtener la suma de todas las conveniencias de los invitados sentados

```
    int Evalua() {
        int suma = Evalua2(getSolucion(), pos_e+1);
        return (suma);
    }


```

**Evalua2**

```
    int Evalua2(const vector<int> &v, int tam) const{
        int suma = 0;
        if(tam>1) {
            for (int i =0; i<tam; i++) {

                int val = v.at(i);

                if (i==0) {

                    suma += datos.at(val).at(v.at(i+1));
                    suma += datos.at(val).at(v[tam-1]);

                } else if (i<(tam-1)) {
                    suma += datos.at(val).at(v.at(i+1));
                    suma += datos.at(val).at(v.at(i-1));
                } else {
                    suma += datos.at(val).at(v[0]);
                    suma += datos.at(val).at(v.at(i-1));
                }
            }
        }

        return suma;
    }


```

**Método getSolución**: devuelve el vector con la solución que se ha encontrado hasta el momento.

```
vector<int> getSolucion(){
    return solucion;
}
```

**Método HayMasValores**: Comprueba si el valor de la pos k-ésima del vector solución se puede seguir aumentando

```
bool HayMasValores (int k) const{
        return ((solucion.at(k) <= (num_invitados-1)&&
   solucion.at(k)>=-1));
}
```

**Método PrimerValorComp**: asigna el valor cero a la posición k-ésima del vector solución. Indica que el invitado 0 está sentado y aumenta pos_e

```
void PrimerValorComp(int k){
        solucion.at(k) = 0;
        estan_sentados.at(solucion.at(k)) = true;
        pos_e++; // Cada vez que se inicializa un valor se
    aumenta la posicion, ya que pos_e es la posicion de la
    ultima
                        // decision en solucion, es decir,
    del ultimo elemento que hemos aniadido a solucion
    }
```

**Método SigValComp**: asigna el siguiente valor válido de la posición k-ésima.

```
void SigValComp(int k){
        int val = solucion.at(k);

        if (val<num_invitados -1) {
            if(val==NULO){
                invitados_sentados++;
            }

            val++;

            solucion.at(k)=val;
            estimador_ =DecisionesTomadas()+CotaLocal(
   num_cota);
        }
        else {
            solucion.at(k) = END;
        }

    }
```

**Método CompActual**: Devuelve la posición del último elemento insertado en solución

```
int CompActual() const{
    return pos_e;
}
```

**Método size**: Devuelve el numero de comensales

```
int size() const{
    return num_invitados;
}
```

**Greedy**(Usamos el algoritmo Greedy para obtener una aproximación de la mejor solución, obtenemos la mejor solución fijando un punto)

```
    int Greedy() {
        vector<int> sol;
        int suma = 0;
        vector<bool> usados;
        int tam = size();

        for (int i=0; i<tam; i++) {
            usados.push_back(false);
        }

        int invitado_actual = 0;
        sol.push_back(0);
        usados.at(0) = true;
        while (sol.size()<tam) {

            int max = -1;
            int pos_max = invitado_actual;

            for (int i=0; i<tam; ++i) {
                int c = datos.at(invitado_actual).at(i);
                if (c > max && c>0 && !usados.at(i)) {
                    max = c;
                    pos_max = i;
                }
            }

            usados.at(pos_max) = true;
            sol.push_back(pos_max);
            invitado_actual = pos_max;
        }

        suma = Evalua2(sol, sol.size());

        solucion = sol;
        pos_e = num_invitados-1;
        conveniencia_acumulada = suma;
        invitados_sentados = num_invitados;
        estimador_ = suma;
        estan_sentados = usados;

        return suma;
    }

```

**EsSolucion**(Comprueba si se trata de una solución, es decir, que todos los invitados estén sentados y que no haya ninguno repetido)

```
bool EsSolucion() const{
    return(pos_e==num_invitados-1)&& this->Factible();
}
```

2. Función **Branch_and_Bound**:
   Usando la clase anteriormente implementada, realiza el algorítmo branch and bound para obtener la mejor solución al problema.
   Esta recibe como parámetros la matriz con las conveniencias entres los invitados y la cota que queremos usar para mejorar la eficiencia del algoritmo.
   El algoritmo va a ir almacenando los los nodos vivos en una cola con prioridad, de tal manera que el primer elemento es el nodo con una mayor conveniencia estimada, es decir, el primer nodo que se va a explorar de cara a obtener la solución óptima. Esto lo hacemos recorriendo los nodos vivos cuya CotaLocal supere a la cotaglobal(inicializada con el método Greedy). En caso de que no lo supere este nodo se poda. Una vez escogido el e-nodo, se generan todos sus hijos y se actualiza la mejor_solucion y la cota global en caso de que la conveniencia del e_nodo supere a la cota global. De esta manera vamos explorando todas las ramas del árbol dándole prioridad a las que tengan una mayor Cota Local y podando las que tengan una CotaLocal menor que la global.

```
Solucion Branch_and_Bound( const vector<vector<int>> &datos,
    int num_cota){
     priority_queue<Solucion> Q;
     Solucion n_e(datos, num_cota), mejor_solucion(datos,
    num_cota);
     int k;
     int cota_global=mejor_solucion.Greedy();
     int conveniencia_actual;

     Q.push(n_e);

     while(!Q.empty() && ( Q.top().CotaLocal(num_cota) >
    cota_global )){
          n_e=Q.top();
          Q.pop();
          k= n_e.CompActual();
          for(n_e.PrimerValorComp(k+1); n_e.HayMasValores(k+1);
    n_e.SigValComp(k+1)){
               if(n_e.EsSolucion()){
                    conveniencia_actual=n_e.Evalua();
                    if(conveniencia_actual > cota_global){
                         cota_global=conveniencia_actual;
                         mejor_solucion=n_e;
                    }
               }else{
                    int cota = n_e.CotaLocal(num_cota);
                    if(cota>cota_global && n_e.Factible() ){
                         Q.push(n_e);
                    }
               }

          }
     }

     return mejor_solucion;
}


```

### 5.1.2. Demostración de validez

B&B al igual que Backtracking acabaría haciendo un recorrido completo del árbol de estados en caso de que nos se utilizasen funciones de cota, con la diferencia de que éste sabe a priori que una zona tiene más posibilidades de tener la solución óptima debido al estructura de datos en que se almacenan las n-uplas de las soluciones que se van construyendo. El B&B una vez llega a un e-nodo genera todos sus descendientes y toma el mejor de ellos paras seguir expandiendo. Supongamos que $S$ es la solución óptima de algoritmo y que se trata de una solución con $n$ invitados, esto quiere decir que:

$\sum_{i=1}^{n} conveniencia(i) \geq \sum_{i=1}^{k} conveniencia(i)\ con\ k \in \mathbb{N}\,, 1 \leq k < n$
tal que $conveniencia(i)$ es la conveniencia del invitado i-ésimo con los que tiene sentados al lado.
Si vamos por la componente k-ésima el algoritmo inicilizará las componentes k+1-esimas de todos los descendientes del nodo k-esimo y por el método de selección seguirá explorando la mejor n-upla de todas las anteriores hasta llegar a la solución $S$, pues las funciones de poda tienen ese objetivo, que es testear si la nupla que se está formando tiene posibilidad de éxito, y en este caso explorarán el árbol hasta llegar a la solución. Pues en caso de no seguir explorando se estaría ante una implementación errónea del algoritmo.

### 5.1.3. Funciones de cota

Se han diseñado 3 funciones de cota o poda para poder mejorar el tiempo de ejecución del algoritmo de Branch and Bound. Se utilizan según la cota que elija el usuario.

- **Método Estimador1()**: se toma la mayor de todas las conveniencias del vector de nodos (donde previamente se han guardado todos los datos y se han ordenado de mayor a menor). Para estimar la conveniencia máxima que se puede obtener se multiplica el resultado por el número de invitados sin sentar y por dos pues la conveniencia de un invitado depende de los sentados a la izquierda y a la derecha. A esto se le suma las decisiones tomadas previamente al nodo actual, esto lo hacemos mediante el metodo Evalua2 previamente explicado.

```
1  int Estimador1() const{
2
3          int invitados_sin_sentar= num_invitados - (pos_e+1);
4          int estimador=0;
5
6          //Cogemos el que tenga mayor conveniencia con el(es
       decir, de su fila en la matriz datos)
7          bool seguir=true;
8          int conveniencia_=0;
9          int i=0;
10
11         conveniencia_ = nodos.front().conveniencia;
12
13         estimador = conveniencia_;
14
```

```
15          return  Evalua2(this->getSolucion(), pos_e+1)+(2*
    estimador*invitados_sin_sentar);
16
17      }
```

- **Método Estimador2()**: siempre que haya invitados sentados para estimar la conveniencia máxima que se puede obtener, acumulamos la conveniencia máxima se toma para cada uno (cada fila de la matriz *datos*) en valor máximo de conveniencia y se multiplica por *dos* pues su conveniencia depende del invitado de ambos lados. A esto se le suma las decisiones tomadas previamente al nodo actual, esto lo hacemos mediante el metodo Evalua2 previamente explicado.

```
1  int Estimador2() const{
2          int estimador2 = 0;
3
4          for(int i=0; i<datos.size(); i++){
5
6              if(estan_sentados.at(i)==false) {
7                  vector<int>::const_iterator it = max_element(
    datos.at(i).begin(), datos.at(i).end());
8                  estimador2+=2*(*it);
9              }
10         }
11
12         return  Evalua2(this->getSolucion(), pos_e+1)+
    estimador2;
13     }
```

- **Método Estimador3()**: siempre que haya invitados sentados para estimar la conveniencia máxima que se puede obtener, sumamos para cada invitado las dos máximas conveniencias con otros invitados, es decir para cada fila de la matriz de datos tomamos los dos enteros más grandes y lo acumulamos en el estimador. A esto se le suma las decisiones tomadas previamente al nodo actual, esto lo hacemos mediante el metodo Evalua2 previamente explicado.

```
1  int Estimador3() const{
2          int estimador3 = 0;
3          int max = 0;
4
5          for(int i=0; i<datos.size(); i++){
6              max= 0;
7              if(estan_sentados.at(i)==false) {
8                  int indice_max;
9                  for(int j = 0; j < datos.at(i).size(); j++){
10
11                     if(datos.at(i).at(j) > max )
12                         max = datos.at(i).at(j);
13                     indice_max=j;
14                 }
15
16                 //BUSCAR EL OTRO MAXIMO
17                 int otro_max = 0;
18                 for(int j = 0; j < datos.at(i).size(); j++){
19
```

```
20                      if(datos.at(i).at(j) > otro_max &&
    indice_max!=j)
21                          otro_max = datos.at(i).at(j);
22                  }
23
24                  max +=otro_max;
25              }
26
27              estimador3+=max;
28          }
29
30          return  Evalua2(this->getSolucion(), pos_e+1)+
    estimador3;
31      }
```

## 5.2. Análisis de eficiencia

### 5.2.1. Análisis teórico

La eficiencia del algoritmo Greedy viene determinada por $max(\mathcal{O}(forinicial), \mathcal{O}(While), , \mathcal{O}(forfinal))$. Vemos claramente que el máximo es el while que tiene eficiencia de $\mathcal{O}(n^2)$, ya que recorre el num_invitados y dentro tiene un for que los vuelve a recorrer, en cambio los dos for tienen eficiencia de $\mathcal{O}(n)$). Luego la eficiencia del algoritmo Greedy es $\mathcal{O}(n^2)$.
Sin embargo, esta eficiencia va a ser insignificante pues estamos ante un algoritmo de eficiencia exponencial

Realizar un análisis teórico de este tipo de algoritmos que implementan funciones de cota no es tarea fácil pues no dependen no sólo del tamaño del problema, sino también de aspectos tales como la eficiencia de las funciones de cota, que como hemos visto varía entre cada una de las 3 cotas. Por otro lado, el número de llamadas, lo pronto o tarde que se pode una determinada rama influyen significativamente en la eficiencia del algoritmo. En adición, la eficiencia está en función del tiempo necesario para generar la siguiente solución, el número de las soluciones que se están creando a partir de la psoición k que cumplen con las restricciones explícitas, el tiempo de determinar la función de factibilidad y el número de soluciones que la satisfacen. Y no se puede despreciar el hecho de cada inserción en la cola con prioridad tiene un coste logarítmico, pues supone reordenar el APO. Es por ello, que nos existe una función que se ajuste. Por tanto, los análisis que se van a plantear a continuación se hacen sobre casos particulares.

En el caso de que no tuviésemos funciones de cota que implementar, el algoritmo tendría una eficiencia factorial que sería de la de generar todas las posibles soluciones. Al tratarse de permutaciones sin repetición, la eficiencia es $n!$ siendo $n$ el tamaño del problema; es decir, el número de comensales que hay que sentar a la mesa.

### 5.2.2. Análisis empírico

Para estudiar la eficiencia empírica del algoritmo, se ha medido el tiempo de ejecución para distintos tamaños de entrada, en este caso, al tener varias cotas, lo hemos hecho para las tres. Ya que el Estimador1 es del orden $\mathcal{O}(pos\_e + 1) \in$

$\mathcal{O}(n)$(debido a la llamada a Evalua2) mientras que el Estimador2 y el Estimador3 son del orden $\mathcal{O}(datos.size()) \in \mathcal{O}(\mathrm{n}^2)$. Algo que hemos podido comprobar al realizar el análisis. Los datos han sido mostrados son hasta un numero de invitados de 18 aunque para el estimador 1 podíamos ejecutarlo sin problemas para un número mayo. Los resultados han sido los siguientes:

| **Tamaño** | **Tiempo** |
|---|---|
| 2 | 1.6788e-05 |
| 3 | 2.8708e-05 |
| 4 | 5.079e-05 |
| 5 | 0.000158454 |
| 6 | 0.000569584 |
| 7 | 0.000231088 |
| 8 | 0.000251884 |
| 9 | 0.000973574 |
| 10 | 0.000833155 |
| 11 | 0.00222258 |
| 12 | 0.00130448 |
| 13 | 0.00135914 |
| 14 | 0.00148958 |
| 15 | 0.00116127 |
| 16 | 0.00708797 |
| 17 | 0.0364431 |
| 18 | 0.00142621 |

Cuadro 4: Tabla BranchandBound para la Cota1

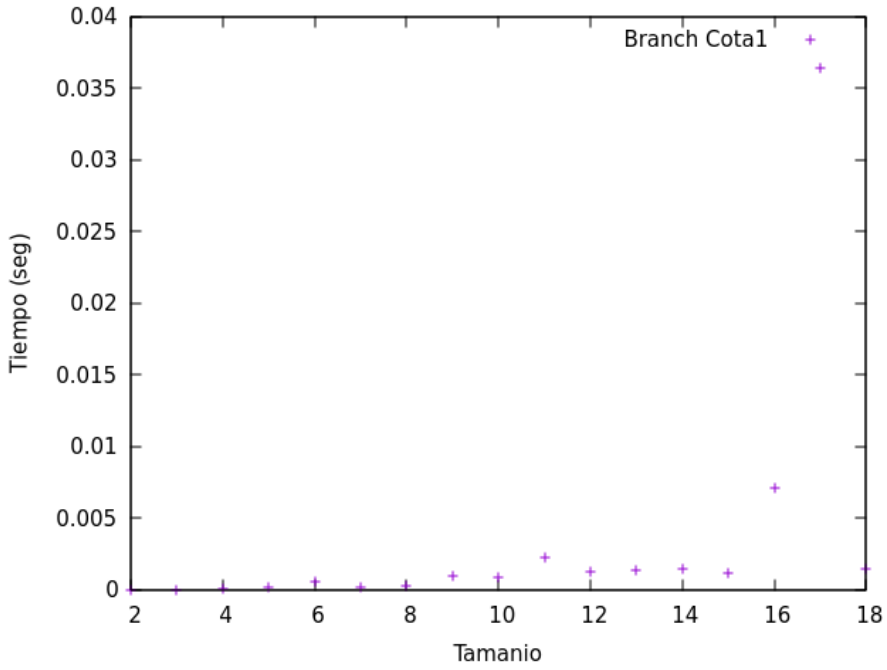


Figura 5: Gráfica BranchandBound para la Cota1

| **Tamaño** | **Tiempo** |
|---|---|
| 2 | 1.5447e-05 |
| 3 | 3.1181e-05 |
| 4 | 7.1425e-05 |
| 5 | 0.000214941 |
| 6 | 0.00114082 |
| 7 | 0.00724701 |
| 8 | 0.0247757 |
| 9 | 0.197604 |
| 10 | 3.04685 |
| 11 | 15.427 |
| 12 | 26.00334337 |
| 14 | 35.0269769 |
| 16 | 49.422951 |
| 18 | 57.1566 |

Cuadro 5: Tabla BranchandBound para la Cota2

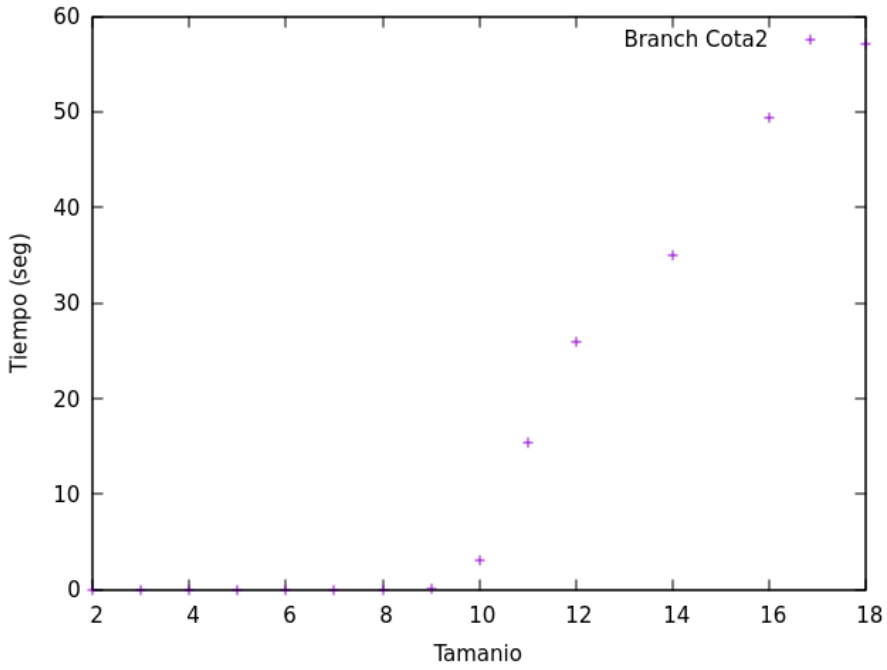


Figura 6: Gráfica BranchandBound para la Cota2

| **Tamaño** | **Tiempo** |
|---|---|
| 2 | 1.4386e-05 |
| 3 | 1.5011e-05 |
| 4 | 1.6281e-05 |
| 5 | 2.0912e-05 |
| 6 | 0.00129995 |
| 7 | 2.3753e-05 |
| 8 | 0.0119919 |
| 9 | 0.224378 |
| 10 | 2.3241 |
| 11 | 3.56266 |
| 12 | 20.54672 |
| 14 | 53.8821 |
| 16 | 64.1771 |
| 18 | 71.262073 |

Cuadro 6: Tabla BranchandBound para la Cota3

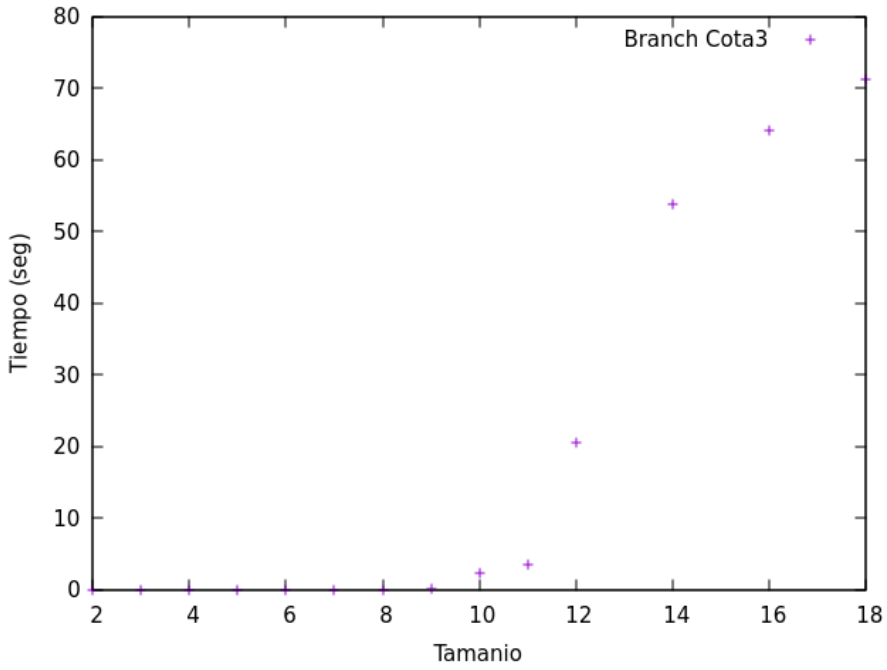


Figura 7: Gráfica BranchandBound para la Cota3

### 5.2.3. Análisis híbrido

Como vimos en el análisis teórico, la ejecución de este tipo de algoritmos que implementan funciones de cota dependen de muchos aspectos, además del tamaño del problema. Es por ello, que no existe una función que se ajuste a los datos obtenidos.

## 5.3. Análisis comparativo de las cotas

Como ya comentamos a la hora de hacer el análisis empírico, el Estimador1 es del orden $\mathcal{O}(pos\_e + 1) \in \mathcal{O}(n)$, mientras que el Estimador2 y el Estimador3 son del orden $\mathcal{O}(datos.size()) \in \mathcal{O}(\mathrm{n}^2)$. Luego podemos ver como el tiempo de ejecución para la cota1 es mucho menor que para las otras dos.

# 6. Análisis comparativo de las dos técnicas

En esta sección, se va a llevara a cabo una comparación entre las dos técnicas implementadas para resolver el problema propuesto. La principal diferencia a nivel de implementación que se puede ver entre el *Backtracking* y el *Branch&Bound* es que la primera de ellas se trata de un algoritmo recursivo, mientras que la otra de un algoritmo iterativo. No obstante, si se analizasen los algoritmo sin las funciones de cota; es decir, implementando algoritmos de fuerza bruta, la eficiencia de ambas sería exponencial, sendas crearían todas las posibles soluciones y tomarían la mejor, con el inconveniente de que se generarían muchas soluciones que no cumplirían con las restricciones explícitas, lo cuál se traduciría en una pérdida de tiempo y gran coste computacional.

Una de las características del *Backtracking* y es en lo que se basa su esencia es

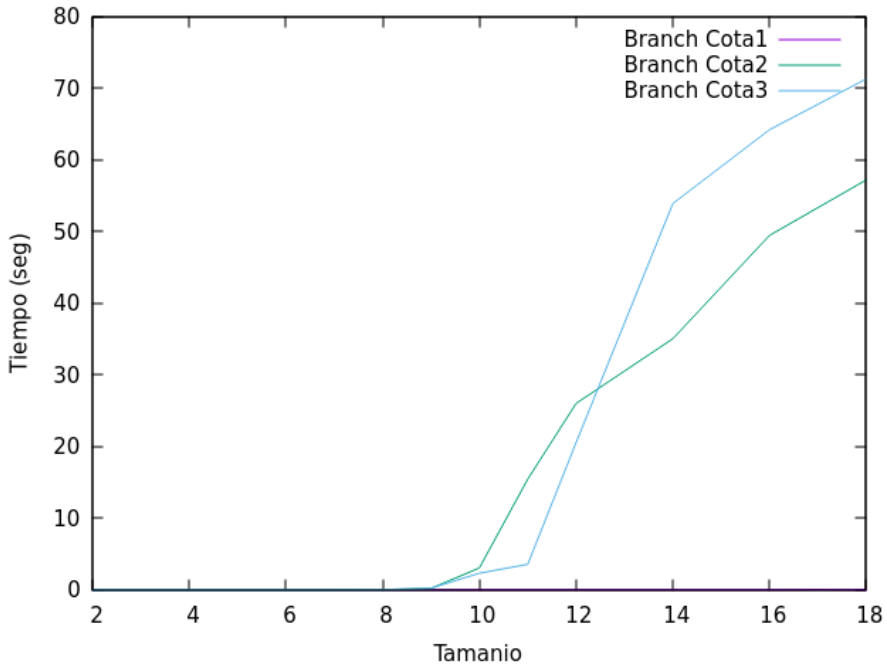


Figura 8: Gráfica comparación de las tres cotas BranchandBound

que implementa la vuelta atrás; es decir, si está crenado una solución y al insertar el elemento k-ésimo éste no cumple las restricciones, vuelve al paso anterior y no sigue buscando esa rama. Mientras que el algoritmo *Branch&Bound* cuando llega a la posición k-ésima, crea todas las posibles combinaciones que hay en esa posición y mete esas combinaciones en la cola con prioridad y mete solo en la cola con prioridad aquella que van cumpliendo las restricciones.

## 6.1. Comparación en términos de tiempo de ejecución

En la siguiente sección, se van a comparar las dos técnicas usadas entre ellas, para ver cuál de las dos ofrece los tiempos de ejecución y en general ofrecería una mejor eficiencia, aunque ya hemos explicado previamente que es complejo dar una eficiencia o un tiempo de ejecución exacto.

### 6.1.1. Comparación del Estimador 1

Como ya se han mostrado en las implementaciones de los métodos anteriores, el estimador 1 toma la mayor de las conveniencias del vector de nodos. En el siguiente gráfico se muestran los tiempos de ejecución de Backtracking y Branch and Bound haciendo uso de esta cota.

Se puede observar como en la mayoría de los casos el Brancha and Bound se encuentra por encima del Backtracking lo cual indica que este último ofrece tiempos de ejecución menores en general.

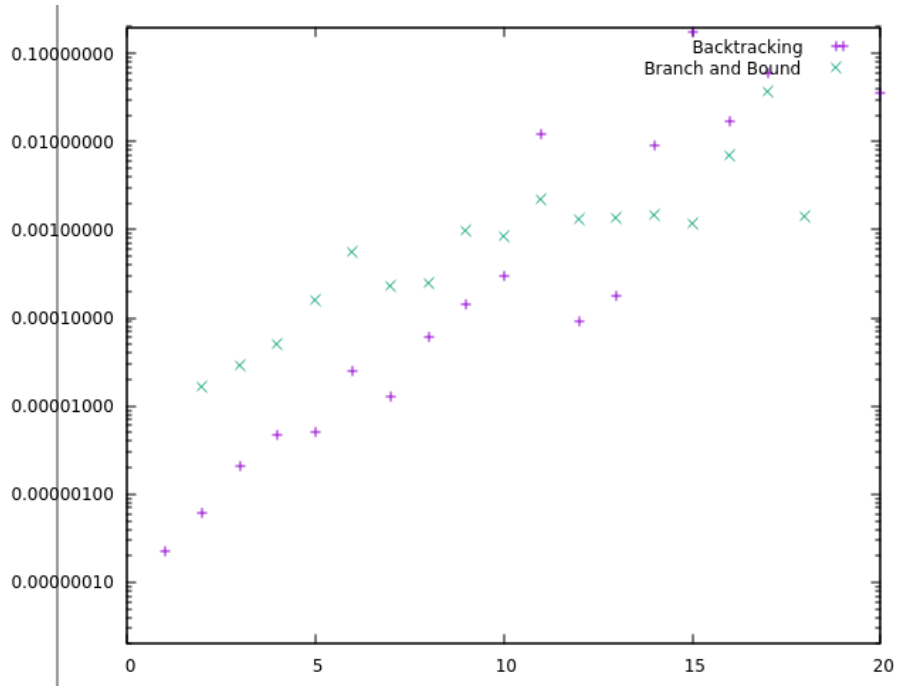


Figura 9: Gráfica comparación del Estimador 1

### 6.1.2. Comparación del Estimador 2

Este estiamdor era algo más preciso que el anterior pues en vez de tomar la mayor de todas las conveniencias, tomaba la mayor para cada uno de los nodos que quedaban con sentar y la multiplicaba por dos, con el inconveniente de que empeora la eficiencia del método.

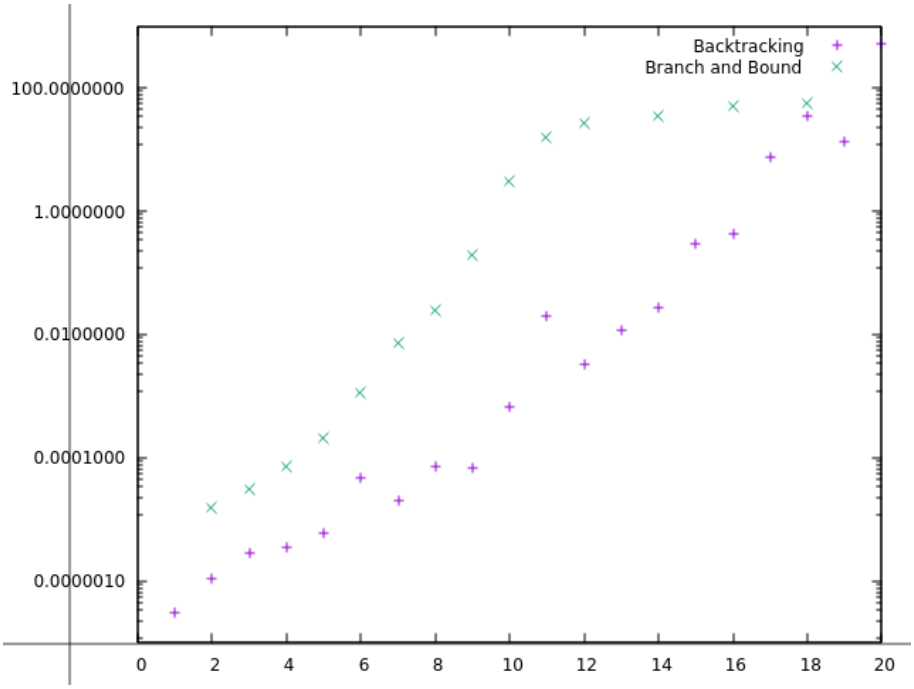


Figura 10: Gráfica comparación del Estimador 2

En este caso, se ve mucho más claro como el Branch and Bound obtiene tiempos de ejecución mayores que el Backtracking.

### 6.1.3. Comparación del Estimador 3

La diferencia de este último estimador con respecto al anterior es que en vez de multiplicar por dos la conveniencia máxima para cada nodo, toma las dos conveniencias máximas, luego la cota es aun más precisa.

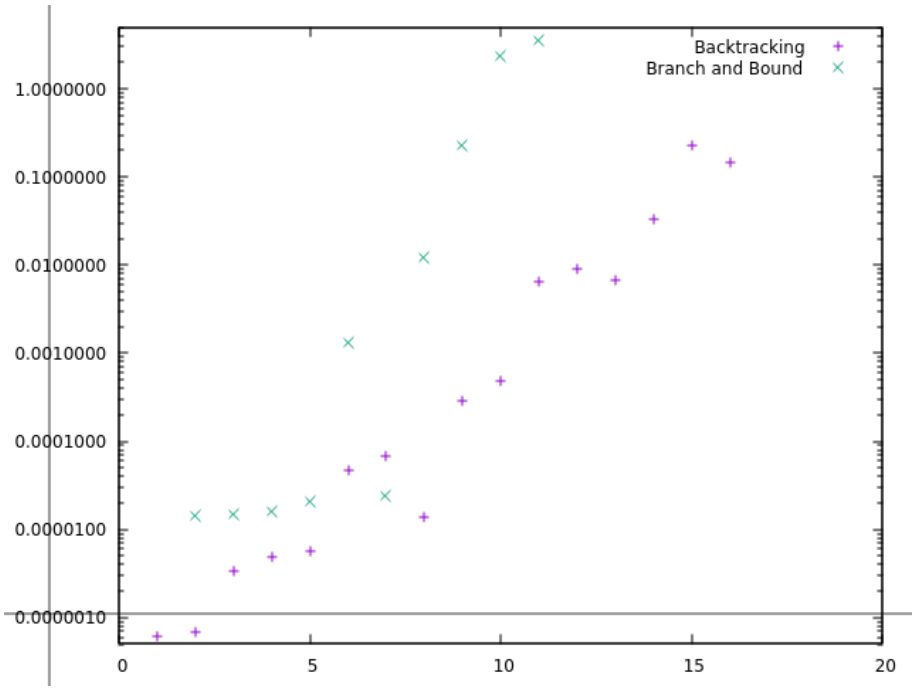


Figura 11: Gráfica comparación del Estimador 3

En esta gráfica se vuelve a corroborar lo que ya se venía observando en los dos casos anteriores y es que el Branch and Bound ofrece peores tiempos de ejecución.

### 6.1.4. Comparación de las dos técnicas y todas las cotas

A continuación, se van a mostrar en una mismo gráfico las dos técnicas utilizadas cada una con sus tres estimadores para estudiar cuál es el mejor y el peor en tiempo de ejecución.

A simpl vista no es fácil discernir cuál ofrecen valores mayores o menores pues hay tramos dónde los resultados son similares. Sin embargo, se claramente como hay dos de las cotas que se encuentran por encima del resto y ambas son resultados de ejecución del Branch and Bound, en concreto el Estimador 2 y 3. Por otro lado, si intentamos encontrar quiénes ofrecen los menores tiempo de ejecución se observa con aquellas que han sido ejecutadas con la cota 1 tanto Backtarcking como Branch and Bound.

La conclusión que se puede obtener de todas estas comparaciones realizadas es que en particular para cada cota el Backtracking ofrece tiempos de ejcución menores que el Brancha and Bound y en cómputo global que la cota uno es la qu menores tiempos de ejecución ofrece a pesar de ser una cota más negativa en el sentido de

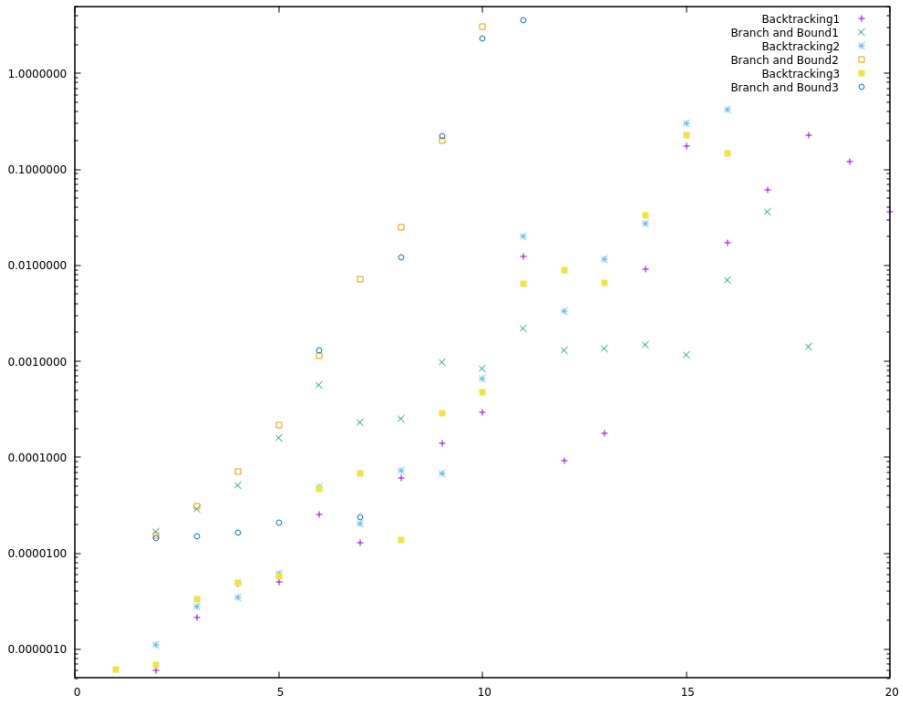


Figura 12: Gráfica comparación

que es la menos precisa de las 3, luego a pesar de realizar una aproximación peor que las otras dos ofrece la solución de manera más rápida. Con otras palabras, se ha probado como la en este ejercicio el hecho de realizar una función de cota más precisa no se ha traducido en una mejora en los tiempos de ejecución, luego valdría la pena usar siempre el Estimador 1 y si puede ser con Backtracking pues a pesar de la gran carga computacional que suponen los algoritmos recursivos es mejor que el Branch and Bound que se trata de uno iterativo pero que encad inserción se tiene que actualizar.

# 7. Conclusiones

El desarrollo de esta práctica nos ha permitido ver dos algoritmos que, de cara a obtener la mejor solución, exploran todas los soluciones posibles, "volviendo atrás."[en] el caso del backtracking si detecta que por el nodo actual no se alcanza la solución correcta, o recorriendo todos los nodos hijos del nodo actual, el branchandbound, sea como fuere, en ambos casos nos encontramos antes dos algoritmos sumamente ineficientes, que pueden llegar a ser del orden exponencial.

Para mejorar esta situación, hemos visto como la implementación de funciones de poda, puede llegar a reducir considerablemente el número de nodos a explorar, aunque estas incluyen un sobrecosto, por lo que el objetivo es encontrar el equilibrio ideal entre el número de nodos recorridos(funciones de poda) y el tiempo gastado en cada nodo(tiempo de hacer las estimaciones).

A la vista de los resultados obtenidos en las comparaciones de ambas técnicas se

llega a la conclusión de que en este caso el algoritmo *Backtracking* es el que ofrece los mejores tiempos de ejecución que el *Branch and Bound*.