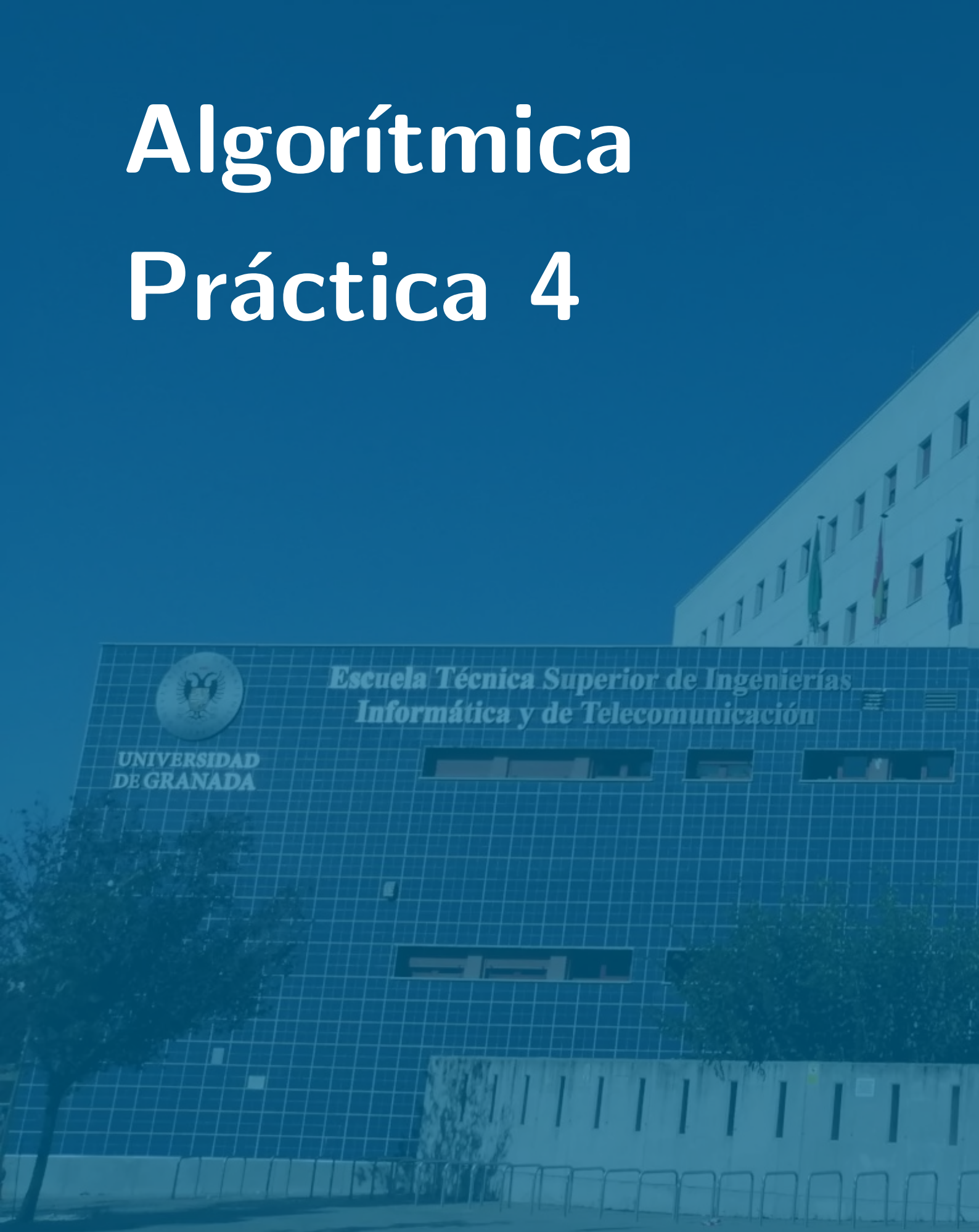
Algorítmica
Práctica 4



Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Algorítmica
Práctica 4

Los Del DGIIM, `losdeldgiim.github.io`

Laura Mandow Fuentes
Chengcheng Liu
Daniel Hidalgo Chica
Roberto González Lugo
Elías Monge Sánchez

Granada, 2023-2024

Índice

1. Participación

- Laura Mandow Fuentes. e.lauramandow@go.ugr.es 100 %
- Roberto González Lugo. e.roberlks222@go.ugr.es 100 %
- Daniel Hidalgo Chica. e.danielhc@go.ugr.es 100 %
- Chengcheng Liu. e.cliu04@go.ugr.es 100 %
- Elías Monge Sánchez. e.eliasmonge234@go.ugr.es 100 %

1.1. Participación específica

Aunque hayamos trabajado cada uno de forma global los contenidos de la práctica, a la hora de la redacción de la memoria, el trabajo se ha visto dividido en partes de carga de trabajo similar con el fin de aumentar la productividad.
En particular, las máquinas utilizadas para ejecutar los algoritmos son:

- Maquina 1
  - Máquina: Asus TUF fx505dt
  - Procesador: AMD Ryzen 7 3750h with Radeon Vega Mobile Gfx 2.3GHz
  - Tarjeta gráfica: Nvidia Geforce GTX 1650
  - Sistema Operativo: Arch Linux 64bits
- Maquina 2
  - Máquina: Acer Aspire A315-42
  - Procesador: Procesador: AMD Ryzen 5 3500U 2.10 GHz
  - Tarjeta Gráfica: Radeon Vega Mobile Gfx
  - Sistema Operativo: Ubuntu 22.04 64bits
- Máquina 3
  - Máquina: HP Laptop 15s-eq1xxx
  - Procesador:AMD Ryzen 5 4500U with Radeon Graphics
  - Sistema Operativo: Ubuntu 22.04 64 bits
- Máquina 4
  - Máquina: Surface Laptop 4
  - Procesador: Intel Core i7
  - Tarjeta Gráfica: Intel Corporation TigerLake-LP GT2 [Iris Xe Graphics] (rev 01)
  - Sistema Operativo: Ubuntu 22.04 64bits
- Máquina 5
  - Máquina: Acer Aspire A315-59
  - Procesador:AMD Ryzen 5 5500U
  - Tarjeta Gráfica: Radeon Graphics
  - Sistema Operativo: Ubuntu 22.04 64bits

2. Objetivos

El objetivo principal de esta práctica es que los estudiantes comprendan y dominen las técnicas de resolución de problemas basadas en la exploración de grafos, específicamente Backtracking y Branch and Bound. Estas técnicas se aplicarán al diseño y desarrollo de algoritmos para resolver problemas complejos, con un enfoque especial en el problema del viajante de comercio.

- **Comprensión Profunda de Técnicas Algorítmicas**:
  - **Backtracking**: Los estudiantes deberán diseñar e implementar algoritmos basados en Backtracking, incorporando diversas funciones de cota. Esto incluirá la validación de la corrección y eficiencia de las soluciones propuestas.
  - **Branch and Bound**: Similarmente, se diseñarán e implementarán algoritmos basados en la técnica Branch and Bound, evaluando diferentes funciones de cota y analizando su rendimiento.
- **Análisis de Eficiencia**:
  - Realizar un análisis detallado de la eficiencia de los algoritmos implementados, considerando tanto el número de nodos generados como el tiempo de ejecución dedicado a cada nodo. Este análisis comparativo permitirá evaluar la efectividad de cada técnica y sus funciones de cota en diferentes escenarios.
- **Desarrollo de Habilidades Técnicas y Blandas**:
  - **Colaboración en Equipo**: La práctica se realizará en grupos, fomentando el trabajo colaborativo, el intercambio de ideas y la división equitativa de tareas.
  - **Habilidades Transversales**: Además de la comprensión técnica, se busca desarrollar competencias como el pensamiento crítico, la comunicación efectiva y la gestión del trabajo en equipo.
- **Documentación y Comunicación**:
  - Redactar una memoria que documente exhaustivamente el trabajo realizado, incluyendo el diseño, implementación y análisis de los algoritmos. La memoria deberá seguir una estructura clara y detallada, facilitando la comprensión y evaluación del trabajo por parte de los profesores.

En resumen, los objetivos de esta práctica son profundizar en el conocimiento y aplicación de técnicas algorítmicas avanzadas, desarrollar habilidades para analizar y comparar la eficiencia de diferentes enfoques, y fomentar el trabajo en equipo y la comunicación efectiva. Esta experiencia integral prepara a los estudiantes para enfrentar desafíos complejos tanto en el ámbito académico como profesional dentro del campo de la ciencia de la computación.

3. Definición del problema del Viajante de Comercio

Tenemos un conjunto de n ciudades (puntos en un plano), cada una definida por las coordenadas en el mapa (x_i, y_i), con $i = 1, \ ... \ , n$. La distancia entre dos ciudades viene dada por la distancia euclídea entre sus coordenadas.

$$dist((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

El problema de viajante de comercio consiste en encontrar el orden en el que un viajante, partiendo de la ciudad de origen (por ejemplo (x_1, y_1)) pase por todas y cada una de las ciudades una única vez, para volver a la ciudad de partida, formando un ciclo.

El costo del ciclo será la suma de las distancias que hay entre todas las ciudades consecutivas.

El problema original del viajante de comercio consiste en encontrar el ciclo de costo mínimo entre todas las posibilidades existentes.

Aunque este problema es NP-Difícil y por tanto podemos esperar encontrar una solución óptima al mismo, aunque el tiempo no será viable para tamaño de problemas grandes.

3.1. Input/Output

Se ha tomado como datos de entrada un entero n indicando el número de ciudades a visitar, seguido de n líneas, cada una representando las coordenadas de la ciudad en cuestión siguiendo el siguiente formato: (x, y) donde x es un número en coma flotante que representa la posición de la ciudad en el eje X e y es lo mismo para el eje Y.

La salida del programa consiste en $n + 1$ líneas cada una representando las coordenadas de la ciudad en cuestión según el formato previamente especificado. La primera y última siempre contienen las coordenadas de la ciudad de origen. En general, representan un camino donde si se está en la ciudad de la línea i-ésima después se va a la ciudad de la línea (i+1)-ésima.

Ejemplo.

```
10
(0.61093,0.565811)
(0.179647,0.505768)
(0.183472,0.816686)
(0.422156,0.584653)
(0.31626,0.0253342)
(0.083659,0.0612762)
(0.978058,0.976791)
(0.0530768,0.873889)
(0.0923382,0.268988)
(0.562573,0.880963)
```

(a) Input data

```
(0.61093,0.565811)
(0.978058,0.976791)
(0.562573,0.880963)
(0.183472,0.816686)
(0.0530768,0.873889)
(0.179647,0.505768)
(0.0923382,0.268988)
(0.083659,0.0612762)
(0.31626,0.0253342)
(0.422156,0.584653)
(0.61093,0.565811)
```

(b) Output data

4. struct City

Para resolver el problema con mayor comodidad y facilitar la modularización y legibilidad del código, se ha hecho uso de una *struct City* para representar las ciudades del problema.

```
typedef long long ll;
typedef long double ld;
```

Se ha definido también una constante INF para representar un valor de distancia imposible mayor que cualquier otro de los que pueda haber dentro del problema.

```
// Infinity (biggest possible number)
const ld INF = 1e18;
```

La *struct City* representa las ciudades del problema mediante sus coordenadas (x, y).

```
struct City
{
    ld x,y;
```

Además implementa la función *dist* que calcula la distancia euclídea de una ciudad a otra y tiene sobrecargado el operador − para este mismo propósito.

```
    // Euclidean distance (symetrical)
    ld operator-(const City & other) const{
        return dist(other);
    }

    ld dist(const City & other) const{
        ld dx = x - other.x;
        ld dy = y - other.y;
        return sqrt(dx*dx+dy*dy);
    }
```

El operador $<$ también ha sido sobrecargado puesto que en nuestros algoritmos nos valimos de ordenar las ciudades respectod el eje x para dar con soluciones más óptimas y eficientes.

```
    // Sort by x axis
    friend bool operator<(const City & a, const City & b){
        if (a.x < b.x) {
            return true;
        }
        else if (a.x == b.x) {
            return a.y < b.y;
        }
        return false;
    }
```

Por último se han sobrecargado los operadores distinto ($!=$) y igual ($==$) por comodidad a la hora de trabajar con *City* y los operadores de entrada ($\gg$) y salida ($\ll$) para facilitar la lectura y escrituda de datos.

```
    friend bool operator==(const City & a, const City & b){
        return a.x == b.x && a.y == b.y;
    }

    friend bool operator!=(const City & a, const City & b){
        return !(a == b);
    }

    // I/O operators
    friend std::istream & operator>>(std::istream & is, City & p){
        char c;
        is >> c >> p.x >> c >> p.y >> c;
        return is;
    }
    friend std::ostream & operator<<(std::ostream & os, const City &p){
        os << "(" << p.x << "," << p.y << ")";
        return os;
    }
```

Para finalizar, se ha implementado la función *printCycle*() la cual recibe como parámetros el orden de los índices de las ciudades, la ciudad de origin y un array con las ciudades e imprime las ciudades empezando y acabando en la ciudad de origen en el orden indicado. Esto se ha hecho para facilitar la impresión de ciudades en el orden correcto teniendo en cuenta que el array de ciudades original ha sido ordenado.

```
1  void printCycle(const std::vector<int> & cycle, const City & origin, const std::
       vector<City> & v){
2      int ini = 0;
3      while(v[cycle[ini]] != origin) ++ini;
4      for(int i=ini; i<(int)cycle.size(); ++i){
5          std::cout << v[cycle[i]] << std::endl;
6      }
7      for(int i=0; i<ini; ++i){
8          std::cout << v[cycle[i]] << std::endl;
9      }
10     std::cout << origin << std::endl;
11 }
```

5. class Solution

Para resolver este problema utilizando las técnicas de backtracking y branch and bound, hemos creado una clase `TSP_solution` para modularizar y estructurar de una manera clara y limpia los algoritmos.

Struct Track
Antes de explicar la clase en sí, hemos utilizado una `struct` para almacenar información sobre el nodo en expansión para ambos algoritmos a la cual hemos llamado `Track`, la cual se muestra a continuación.

```
struct Track {
    // The track as an integer vector
    std::vector<int> track;
    // Information of visited cities
    std::vector<bool> visited;
    // Inferior bound for track
    ld aprox_cost;
    // Current cost of the track
    ld current_cost;

    /**
     * @brief Default constructor for Track
     * @param n Number of cities in the track
    */
    Track(int n = 0) : visited(n, false), aprox_cost(0), current_cost(0) {
        track.reserve(n);
    }
    /**
     * @brief Constructor of Track
     * @param track
     * @param visited
     * @param cost
     * @param current_cost
    */
    Track(std::vector<int>& track, std::vector<bool>& visited, ld cost, ld
    current_cost)
    : track(track), visited(visited), aprox_cost(cost), current_cost(current_cost)
    {}

    /**
     * @brief Greater operator acording to @p aprox_cost
    */
    bool operator>(Track other) const {
        return this->aprox_cost > other.aprox_cost;
    }
};
```

Esta struck es prescindible en el algoritmo backtracking, pero resulta especialmente útil en el algoritmo **BranchAndBound** para ordenar los nodos en la `priority_queue`.

Podemos comentar de ella los vectores `track` y `visited`, que almacenan la información de los nodos visitados en vectores, uno como vector de enteros (los índices en el vector `cities`) y otro como vector de booleanos, haciendo una correspondencia entre las posiciones en el vector `cities`.

A continuación, se muestra la propia clase `TSP_solution` utilizada para organizar la solución de los algoritmos Backtracking y BranchAndBound y definida en el fichero de cabecera **Solution.h**:

```
/**
 * @class TSP_solution
 * @brief Main class for the BK and B&B solution to TSP
*/
class TSP_solution {

protected:
    std::vector<City> cities;
    std::vector<int> best_ans;
    ld cost;

    int podas;
```

```
13      int generated;
14      int version;
15
16  public:
17      TSP_solution();
18      TSP_solution(const std::vector<City>& v, int version = 0);
19
20      std::vector<City> getCities() const;
21      std::vector<int> getSol() const;
22      ld getCost() const;
23      int getPodas() const;
24      int getGeneratedNodes() const;
25      ll getPossibleNodes() const;
26
27      void solve();
28      void printAns();
29
30  private:
31      virtual void algorithm(Track & e_node) = 0;
32
33      ld f_cota1(Track& e_node, int node);
34      ld f_cota2(Track& e_node, int node);
35      ld f_cota3(Track& e_node, int node);
36      ld f_cota4(Track& e_node, int node);
37      ld f_cota5(Track& e_node, int node);
38
39      ld sumMinEnter(const std::vector<bool>& visited, int node);
40      ld sumMinVisit(const std::vector<bool>& visited, int node);
41      ld enter_min_cost(const std::vector<bool>& visited, int node);
42      ld visit_min_cost(const std::vector<bool>& visited, int node);
43      std::pair<ld,ld> shortest_two_edges(const std::vector<bool>& visited, int node)
        ;
44      ld minimoCosteAristasRestantes(int nCiudadesRestantes);
45      void calcularMinimoCosteAristas();
46      ld min_edge();
47      ld min_edge(const Track& e_node);
48
49      void TSP_greedy();
50
51  protected:
52      ld f_cota(Track& e_node, int node);
53      std::pair<bool, ld> feasible(Track& e_node, int node);
54      void processSolution(const std::vector<int>& track);
55  };
56  #endif
```

Esta clase tiene como atributos el vector de ciudades `cities`, que es realmente el enunciado del problema, un vector `best_ans`, que durante la ejecución irá guardando la mejor solución hasta el momento en forma de vector de índices sobre `cities` y su coste `cost`, que cada vez que se actualiza `best_ans` se actualiza este dato también, conteniendo siempre el coste de la solución mínima hasta el momento, es decir, mientras se ejecutan los algoritmos backtracking y branch and bound, este campo actúa de **cota superior**.

Luego como atributos adicionales tenemos un contador de podas, que se incrementa cada vez que la función `feasible` devuelve `false`, otro de nodos generados, que se incrementa cada vez que se procesa un nodo, y un indicador de la version de la cota que se está utilizando, que se inicializa en el constructor con parámetros.

Los métodos están documentados en el fichero adjunto **Solution.h** en la carpeta Include e implementados en el fichero adjunto **Solution.cpp** en la carpeta Src.

Sí que podemos comentar algunos de los métodos más importantes como pueden ser el constructor con parámetros, que inicializa todos los atributos, en particular la mejor solución global y su coste que se inicializan mediante un algoritmo greedy (closest neighbour) que se comento en la práctica anterior, y se encuentra implementado en `TSP_greedy`.

También se puede destacar el método `feasible`, que calcula si una rama es factible o no según la cota que devuelva `f_cota`, método que se detalla en el siguiente apartado, pero grosso modo devuelve la cota inferior local de ir por una rama. Otro método interesante es el `processSolution`,

imprescindible en el algoritmo para ver si una solución local (al llegar a un nodo hoja) es mejor que la solución global y así actualizarla.
Finalmente, el método `solve` es el que se llama desde el `main` para resolver el problema, el cual inicializa el primer `e_node` (como `Track`), y llama al método abstracto `algorithm`. Este método es abstracto dado que también hemos realizado una clase especifica para backtracking `BK_solution` y otra para branch and bound `BB_solution` las cuales ambas **heredan** de `TSP_solution` y definen el método abstracto `algorithm`, siendo respectivamente el algoritmo de backtracking y el de branch and bound. De esta forma se generalizan las cotas y la estructura de clase para ambos algoritmos, al ser muy similares. Este método abstracto se detalla en la explicación de cada algoritmo.

El resto de métodos son *getters* y métodos de cálculo específicos de cada cota que se detallan en cada función de cota (ya sea en la memoria, en los códigos o en ambas).

6. Funciones de cota

Justificación del uso de funciones de cota
Antes de adentrarnos en las diversas funciones de cota que usaremos, sería conveniente entender por qué las usamos en primera instancia.

¿Qué son las funciones de cota?

Las funciones de cota son herramientas matemáticas que proporcionan estimaciones del valor mínimo o máximo posible de una función objetivo para un subconjunto específico del espacio de búsqueda. En nuestro contexto, el travelling salesman problem,las funciones de cota estiman la longitud mínima posible de un recorrido que visita todas las ciudades, y luego regresa al lugar de origen.

¿Por qué se utilizan las funciones de cota en BB y BK?

Las funciones de cota se utilizan en BB y BK por dos razones principales:

- **Reducción del espacio de búsqueda:**
  Las funciones de cota permiten descartar subconjuntos del espacio de búsqueda que no pueden contener la solución óptima debido a que, como mínimo, su coste se pasa del de la solución óptima. O bien como máximo, su coste se queda por debajo del mínimo necesario para dar con una solución válida. Al descartar estos subconjuntos, se reduce significativamente el número de nodos que el algoritmo debe explorar, lo que conduce a una mejora significativa en el tiempo de ejecución y la eficiencia computacional.
- **Priorización de la búsqueda:**
  Las funciones de cota también se pueden utilizar para guiar la búsqueda hacia las regiones más prometedoras del espacio de búsqueda. Esto se logra ordenando los nodos del árbol de búsqueda en función de sus valores de cota. Los nodos con cotas más prometedoras se exploran primero, lo que aumenta la probabilidad de encontrar rápidamente soluciones de buena calidad.

Ejemplo concreto: TSP

En el TSP, una función de cota común se basa en la distancia mínima recorrida para visitar un subconjunto de ciudades. Esta función estima la longitud mínima posible de un recorrido que visita las ciudades del subconjunto y regresa al punto de partida. Al descartar subconjuntos con cotas superiores a la mejor solución encontrada hasta el momento, los algoritmos BB y BK pueden reducir significativamente el espacio de búsqueda.

Conclusión

Las funciones de cota son herramientas esenciales para mejorar la eficiencia y la eficacia de los algoritmos BB y BK para resolver problemas de optimización combinatoria complejos como el TSP. Al proporcionar estimaciones del valor objetivo para subconjuntos del espacio de búsqueda, las funciones de cota permiten reducir el espacio de búsqueda que se debe explorar y guiar la búsqueda hacia las regiones más prometedoras, culminando así en mejores y más rápidas busquedas de soluciones.

Switch para elegir la función deseada
En nuestro caso, hemos utilizado un switch para poder elegir en cada momento que función de cota de las que hemos implementado queremos realizar:

```
ld TSP_solution::f_cota(Track& e_node, int node) {
    ld cota_inf = e_node.current_cost;
    cota_inf += (cities[node] - cities[e_node.track.back()]);
    switch (version) {
```

```
5           case 1:
6               cota_inf += f_cota1(e_node, node);
7               break;
8           case 2:
9               cota_inf += f_cota2(e_node, node);
10              break;
11          case 3:
12              cota_inf += f_cota3(e_node, node);
13              break;
14          case 4:
15              cota_inf += f_cota4(e_node,node);
16              break;
17          case 5:
18              cota_inf += f_cota5(e_node,node);
19              break;
```

El funcionamiento del mismo se entiende simplemente leyendo el código, en función de la variable *version* se llamará a una u otra función de cota.

> Cabe destacar que a pesar de que hemos desarrollado hasta 5 funciones de cota, en esta memoria nos centraremos solo en las 3 que hemos considerado más adecuadas para la explicación (5,2 y 3). Las demás pueden verse en el código.

Destacar también, a efectos de claridad y legibilidad nuestro enfoque sobre las cotas:

> En este caso, todas las funciones de cota se limitan a estimar las decisiones que quedan, es decir, el coste mínimo que supondría visitar el resto del camino actual, si como siguiente ciudad visitaramos la i-ésima.

6.1. Funcion de cota 1

6.1.1. Diseño y justificación de su validez

Nuestra primera función de cota es muy simple. Como ya hemos explicado antes, se preocupa tan solo de "predecir" **inferiormente** el coste que pueden suponer las decisiones que nos quedan por tomar (el orden de visita de las ciudades restantes). Es decir, calcular coste inferior a lo que nos supondrá tomar estas decisiones, pero que nos permita descartar (podar) algunos caminos.

Tenemos pues que si hemos visitado x ciudades de n, queremos ver cual es el coste mínimo que nos pueden suponer visitar todos los caminos escogiendo como siguiente ciudad la i-ésima. Una forma (muy básica) de hacerlo sería la siguiente. Sabemos que por cada ciudad no visitada, nos hace falta un arco para llegar hasta ella, por tanto si nos faltan x ciudades por visitar, necesitaremos $x+1$ arcos para recorrerlas todas (no nos olvidamos del arco necesario para completar el ciclo y volver a la ciudad de origen).

Sabemos que este enfoque será válido por lo ya mencionado, necesitamos $x+1$ arcos más **obligatoriamente**. Al coger los $x+1$ arcos mínimo de todos, estamos garantizando que siempre serán menores o iguales que la solución óptima (por ser los mínimos).

Por tanto, si tenemos las sumas precalculadas de los $x+1$ arcos mínimos $\forall x \in \mathbb{N}$ tal que $0 \leqslant x < n$, tenemos una estimación por debajo de lo que nos puede suponer completar nuestro camino cuando nos faltan x ciudades por visitar. La función de cota nos quedaría de la siguiente forma:

1. Precalculamos el coste todos los arcos del grafo (el coste de todos los caminos **directos** del grafo), es decir, el coste de ir de una ciudad a otra para todas las ciudades.
2. Ordenamos los arcos de menor a mayor (en función de su coste).
3. Precalculamos las sumas prefijas (*prefix sums*) del array ordenado.
4. Cada vez que se llame a la función de cota, en base al número de ciudades que queden por visitar (x), se devuelve como cota el coste de los $x+1$ arcos mínimos.

6.1.2. Detalles de implementación y análisis de eficiencia

Implementación

Examinemos con más detalle la implementación de esta función de cota. Al igual que todas, recibe como parámetros el camino que llevamos recorrido y el siguiente nodo (ciudad) a visitar, para comprobar que tan *factible* es tomar esta decisión (visitar como siguiente a esa ciudad).

```
ld TSP_solution::f_cota5(Track& e_node, int node) {
    int nVisitedCities = e_node.track.size();
    int nCities = this->cities.size();
    ld cota_inf = minimoCosteAristasRestantes(nCities-nVisitedCities);
    return cota_inf;
}
```

Tenemos pues como ya explicamos antes, devolvemos el `minimoCosteAristasRestantes`, es decir, la suma de las $x+1$ aristas mínimas donde llamamos x a el número de ciuades sin visitar, el cual obtenemos restandole al número de ciudades totales ($nCities$) el número de ciuades visitadas ($nVisitedCities$) que coincide con el tamaño de nuestro camino.

```
ld TSP_solution::minimoCosteAristasRestantes(int nCiudadesRestantes) {
    // Guardo en un array estatico el coste de todas las distintas aristas,
    // y solo lo calculo una vez, lo que controlo con otra variable
    // estatica
    static bool ya_calculado = false;
    static vector<ld> costesAristas, prefix_sum;

    if (!ya_calculado) {
        int n = cities.size();
        costesAristas.reserve((n*(n-1)/2));
        for(int i = 0; i < n; ++i){
            for(int j = i+1; j < n; ++j){
                ld dist = this->cities[i] - this->cities[j];
                costesAristas.push_back(dist);
            }
        }

        // Ordenamos el vector
        sort(costesAristas.begin(), costesAristas.end());

        // Acumulamos el coste
        prefix_sum.resize((n*(n-1)/2 + 1));
        prefix_sum[0] = 0;
        for (int i = 1; i < costesAristas.size(); i++) {
            prefix_sum[i] = prefix_sum[i-1] + costesAristas[i-1];
        }
        ya_calculado = true;
    }
```

Estudiemos con más detalle la implementación de `minimoCosteAristasRestantes` para ver como obtenemos la suma de las $x+1$ aristas mínimas. Para hacer nuestra cota más eficiente, como ya mencionamos el array cuya posición i-ésima contiene la suma de las $i+1$ aristas mínimas se **precalcula**. Como estamos trabajando con métodos con una cabecera predeterminada, emplearemos variables ***static*** (estáticas).

Hacemos uso de una variable *booleana estática* para saber si nuestro array ya ha sido calculado o no. En caso de haber sido calculado, simplemente devolvemos el resultado almacenado en la posición correspondiente al número de aristas restantes. En caso contrario, antes de eso debemos calcular el array.

El array se calcula de la forma explicada previamente. Primero almacenamos el coste de todas las aristas de nuestro grafo. Después las ordenamos de menor a mayor. Finalmente calculamos las sumas prefijas, de forma que la posición 0 vale 0 (si no nos quedan ciudades por visitar, no tiene sentido llamar a la función de cota, se iría al caso base, y como son *prefix_sum*, debe valer 0), en la posición 1 almacenamos la arista mínima, en la 2 la suma de las dos aristas mínimas, etc.

Notamos que nuestra función `minimoCosteAristasRestantes` si se llama con x ciudades sin visitar devuelve la suma de las x aristas mínimas, en vez de la suma de las $x+1$ aristas mínimas. Esto se debe a que la función `f_cota` ya suma la arista entre la última ciudad del camino y el nodo

escogido como siguiente (pero el nodo todavía no ha sido marcada como visitado y por tanto se contabiliza en los n).

Análisis de eficiencia teórica

Analicemos la eficiencia teórica de nuestra función de cota 1.

```
ld TSP_solution::f_cota5(Track& e_node, int node) {
    int nVisitedCities = e_node.track.size();
    int nCities = this->cities.size();
    ld cota_inf = minimoCosteAristasRestantes(nCities-nVisitedCities);
    return cota_inf;
}
```

Podemos ver que se trata de operaciones $O(1)$ y la función `minimoCosteAristasRestantes`. Estudiemos la eficiencia de está función en detalle:

```
ld TSP_solution::minimoCosteAristasRestantes(int nCiudadesRestantes) {
    // Guardo en un array estatico el coste de todas las distintas aristas,
    // y solo lo calculo una vez, lo que controlo con otra variable
    // estatica
    static bool ya_calculado = false;
    static vector<ld> costesAristas, prefix_sum;

    if (!ya_calculado) {
        int n = cities.size();
        costesAristas.reserve((n*(n-1)/2));
        for(int i = 0; i < n; ++i){
            for(int j = i+1; j < n; ++j){
                ld dist = this->cities[i] - this->cities[j];
                costesAristas.push_back(dist);
            }
        }

        // Ordenamos el vector
        sort(costesAristas.begin(), costesAristas.end());

        // Acumulamos el coste
        prefix_sum.resize((n*(n-1)/2 + 1));
        prefix_sum[0] = 0;
        for (int i = 1; i < costesAristas.size(); i++) {
            prefix_sum[i] = prefix_sum[i-1] + costesAristas[i-1];
        }
        ya_calculado = true;
    }
```

Como ya comentamos antes, esta función hace uso de variables *estáticas* para evitar recalcular constantemente cálculos ya realizado. De esta forma, tenemos que el cuerpo del `if` tan solo se **una única vez**. Consideraremos entonces su complejidad como un coste **aparte** del procesamiento de cada nodo (ya que **no** se hará por cada nodo).

El cuerpo del `if` está compuesto de las siguientes intrucciones en serie:

- 2 bucles `for` **anidados**. El primero realiza n iteraciones. El segundo realiza $\frac{n-1}{2}$ iteraciones y su cuerpo consiste en operaciones $O(1)$ (`push_back` es $O(1)$ al haber reservado memoria antes). Los `for` son entonces $O(n^2)$.
- La ordenación de un vector de tamaño $O(n^2)$. Esto tiene una complejidad de $O(n^2 log\, n^2)$.
- 1 bucle `for` que realiza n^2 (realmente realiza $\frac{n(n-1)}{2} + 1$ iteraciones) en las que realiza una suma que es $O(1)$ y una asignación que es $O(1)$. Su complejidad es $O(n^2)$.

Al estar en serie, tenemos que sus eficiencias se suman para dar con la complejidad total, que es $O(n^2 log\, n^2)$.

Pasamos ahora a la verdadera complejidad de la función de cota, es decir, el coste que supone llamarla por **cada nodo**. El contenido fuera del cuerpo de `if` (y las inicializaciones de variables estáticas) se limita a un mero `return` que simplemente accede a un vector, y por tanto es $O(1)$.

En conclusión:

$$\begin{cases} O(f\_cota1) = O(1) \\ O(calculos\_previos) = O(n^2 log\, n^2) \end{cases}$$

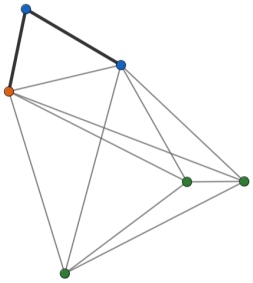
6.2. Funcion de cota 2

6.2.1. Diseño y justificación de su validez

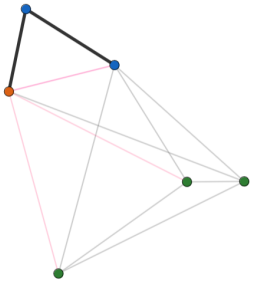
Procedamos a explicar nuestra segunda función de cota. La idea de esta es dada una serie de decisiones ya tomadas C (es decir, las ciudades a visitar primero y su orden), calcular en primer lugar el costo del camino que se genera con dichas decisiones, es decir, el coste que supondría pasar por todas las ciudades ya elegidas según el orden decidido (1). Seguido, para cada ciudad no visitada se aproximaría el coste que supondría pasar por ella con el coste mínimo para llegar a ella por alguna ciudad no seleccionada, o bien por la última seleccionada (2). Finalmente como es un ciclo, tenemos que tener en cuenta el camino de vuelta al origen, entonces también se aproximaría el coste de volver al origen con el mismo procedimiento anterior (3). Así, la suma de todos los costes anteriores sería una aproximación del camino a generar. Veamos ahora por qué es una aproximación por debajo que nos sirve como cota inferior para nuestro problema.

El valor obtenido en (1) es exactamente el coste que supondría pasar por las ciudades ya seleccionadas, luego es claro que es una aproximación exacta del coste para estas ciudades. Para el valor (2), tengamos en cuenta que para cada ciudad no visitada podemos llegar a ella unica y exclusivamente o bien por ciudades tampoco visitadas distintas de ella, o bien por la última ciudad visitada. Así, como el objetivo de TSP es minimizar los costes, en el mejor de los casos llegaríamos a ella por el camino más corto de los potenciales caminos que podemos tomar, es decir, alcanzarla desde ciudades de donde podemos partir (ciudades aun no están visitadas o por la última visitada). La misma justificación de (2) es claramente también válida para (3). Por tanto, la suma de (1), (2) y (3) será menor igual que el costo del camino completo generado a partir de las decisiones ya tomadas, C, y por ende esta cota es válida.

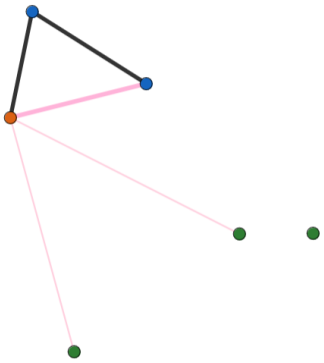
Veámoslo en un ejemplo:



(a) Estdo inicial.

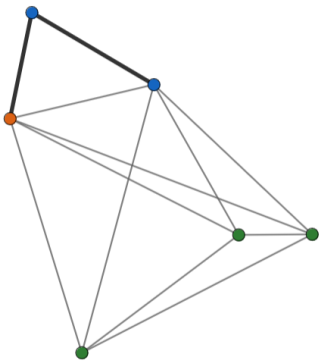


(b) Camninos que llegan al origen.

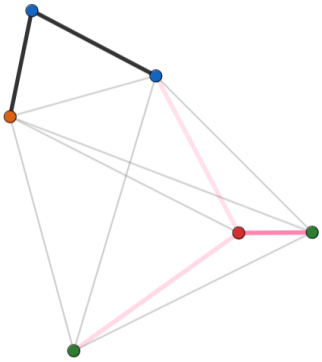


(c) Camino seleccionado.

En (a), (b), (c) estamos aproximando el coste para volver al origen según nuestra función cota.



(a) Estado inicial.



(b) Caminos candidato para el nodo rojo y camino seleccionado.

En (d) y (e) estamos haciendo lo mismo que lo anterior pero para una ciudad/nodo no visitado (el rojo).

6.2.2. Detalles de implementación y análisis de eficiencia

Teniendo en cuenta el diseño previamente explicado nos interesa:

1. Calcular (1)
2. Calcular para cada ciudad(nodo) el camino mismo para llegar hasta ella desde otra ciudad no visitada o desde la última seleccionada. La suma de todos ellos es (2).
3. Calcular el costo mínimo para volver al origen desde una ciudad no visitada o desde la última seleccionada.

Para lo primero solo necesitamos ir acumulando el coste de los caminos de las decisiones ya tomadas. Esto se realiza a medida que se toma las decisiones, luego lo explicaremos más adelante en la implementación de BB y BK, y aquí solo nos limitaremos a (b) y (c), tal y como hemos comentado anteriormete.

Por tanto, nuestra función cota se encarga esencialmente de (b) y (c). Hay un detalle que merece mencionar que es que vamos a considerar la última ciudad a visitar como aun no visitada, pero por visitar, para facilitar la implementación, de allí que la cabecera de nuestra función de cota tenga la siguiente forma:

```
1
2 ld TSP_solution::f_cota2(Track& e_node, int node) {
3     return sumMinEnter(e_node.visited,node);
4 }
```

En la función sumMinEnter se calcula (2)+(3). Para (2), llama para cada nodo no visitado a la función enter_min_cost para calcular (b), recordemos que la suma de todos los (b) es (2). En la última linea 9 se calcula (3) también llamando a enter_min_cost para el nodo origen.

```
1 ld TSP_solution::sumMinEnter(const vector<bool> & visited, int node) {
2     ld dist = 0;
3     for (int i = 0; i < cities.size(); ++i) {
4         if ((node != i) && !visited[i]) {
5             dist += enter_min_cost(visited, i);
6         }
7     }
8     dist += enter_min_cost(visited, 0);
9     return dist;
10 }
```

En enter_min_cost, dado las decisiones hasta el momento y un nodo no visitado, calcula (b) para dicho nodo. Elige el camino de mínimo coste entre todos los caminos que existen entre el propio nodo con el resto de nodos no visitados (el último nodo visitado en nuestro caso también se considera no visitado). Realiza todos estos cáculos delegando en el método shortest_two_edges.

```
1 ld TSP_solution::enter_min_cost(const vector<bool>& visited, int node) {
2     return shortest_two_edges(visited,node).first;
3 }
```

Como shortest_two_edges tiene eficiencia O(n), donde n es el número de ciudades, luego enter_min_cost también. Vemos que sumMinEnter está compuesto por un bucle que en el caso peor llama a enter_min_cost n veces, por tanto tiene eficiencia O(n^2).

Veamos el funcionamiento de `shortest_two_edges`:

```
1 pair<ld,ld> TSP_solution::shortest_two_edges(const vector<bool>& visited, int node)
      {
2     ld min_enter = INF, min_exit = INF;
3     for(int i=0; i < visited.size(); ++i)
4     {
5         if ((visited[i] || i == node) && i != 0) continue;
6         ld dist = cities[node] - cities[i];
7         if (dist < min_enter)
8         {
9             min_exit = min_enter;
10            min_enter = dist;
```

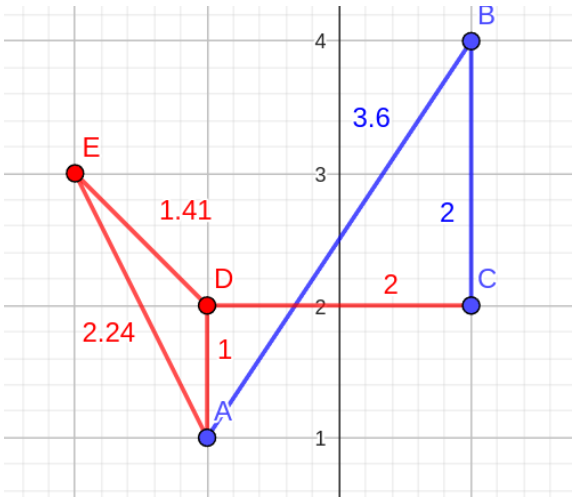
```
11          }
12          else if (dist < min_exit && dist != min_enter) {
13              min_exit = dist;
14          }
15      }
16      return {min_enter,min_exit};
17  }
```

La idea es calcular los dos mínimos de un vector, lo cual se hace exactamente igual que al calcular el mínimo de un vector, pero añadimos que cada vez que encontremos uno menor que el mínimo, el segundo mínimo pase a ser el mínimo anterior y el mínimo pase a ser el nuevo mínimo encontrado. El otro caso que tenemos que distinguir es que encontremos un elemento mayor que el primer mínimo (mínimo total) pero menor que el segundo mínimo, en cuyo caso se lo asignamos al segundo mínimo. Finalmente, se devuelven ambos en un pair.

6.3. Funcion de cota 3

6.3.1. Diseño y justificación de su validez

Para la tercera función de cota, la idea que hemos seguido es similar a la de la cota anterior, solo que en lugar de tomar la mínima distancia de entrar a cada nodo, hemos tomado la mínima distancia que supone visitar cada nodo, es decir, entrar y salir de él. Esto es, tomar las dos mínimas aristas de cada nodo y hacer la media. Como para cada nodo habrá que entrar y salir de él, este valor obtenido es cota inferior de pasar por cada nodo, y la suma de este valor para todos los nodos es cota inferior de la solución óptima de una cierta rama, probando con ello su validez. Veámoslo con un ejemplo:



En azul, se marcan los nodos (y los arcos) que ya se han escogido, por tanto esos arcos no se pueden quitar, y en rojo los nodos que quedan por visitar, así como los dos arcos mínimos de cada nodo. Hay dos formas de calcular la cota. La primera sería la forma en la que lo hemos explicado: tomar las dos aristas mínimas de cada nodo, hacer la media y sumar. De esta forma nos queda:

Nodos	minEnter	minExit	minVisit
A	1	3.6	$(1+3{,}6)/2 = 2{,}3$
B	3.6	2	$(3{,}6+2)/2 = 2{,}8$
C	2	2	$(2+2)/2 = 2$
D	1	1.41	$(1+1{,}41)/2 = 1{,}205$
E	1.41	2.24	$(1{,}14+2{,}24)/2 = 1{,}825$
Total	9.01	11.25	**10.13**

Destacar que la cota está por debajo de la solución óptima de la rama (11.25) y que el total de la columna de los minEnter no es ni más ni menos que la cota anterior, pues es la suma de las mínimas aristas de cada nodo, la cual observamos que es más baja que la suma de visitar

cada nodo. Esto va a ser siempre así, pues $minVisit(n) \geqslant minEnter(n)\ \forall n$ al ser $minVisit(n) = \frac{1}{2}(minEnter(n) + minExit(n))$ con $minEnter(n) \leqslant minExit(n)$. De esta forma se prueba que esta cota es más eficaz.

La otra forma sería tomar el coste del camino actual, tratar el camino como nodo y sumarle el coste de visitar el camino (promedio de entrar y salir del camino, es decir, mínimo coste de entrar al primer nodo más mínimo coste de salir del último) y por último, sumarle el mínimo coste de visitar cada nodo no visitado:

$$total = currentCost + visitTrackCost + sumMinVisit = 5{,}6 + \frac{1+2}{2} + (1{,}205 + 1{,}825) = \mathbf{10.13}$$

Observamos que ambos resultados coinciden.

6.3.2. Detalles de implementación y análisis de eficiencia

La segunda forma comentada de calcular la cota es más sencilla de implementar, por lo que será la forma en que se implemente esta cota:

```
ld TSP_solution::f_cota3(Track& e_node, int node) {

    ld cota_inf = (enter_min_cost(e_node.visited,e_node.track[0]) +
        enter_min_cost(e_node.visited,node)) / 2;

    cota_inf += sumMinVisit(e_node.visited,node);
    return cota_inf;
}
```

Donde se ha calculado primero el mínimo de pasar por el camino actual usando dos veces la función `enter_min_cost` ya comentada y luego se ha sumado el resultado de llamar a `sumMinVisit` que se describe a continuación:

```
ld TSP_solution::sumMinVisit(const vector<bool>& visited, int node) {
    ld dist = 0;

    for (int i = 0; i < cities.size(); ++i) {
        if ((node != i) && !visited[i]) {
            dist += visit_min_cost(visited,i);
        }
    }
    return dist;
}
```

Hasta ahora tiene la misma eficiencia que la función `sumMinVisit`.

Esta función llama para cada nodo no visitado a la función `visit_min_cost` y acumula los resultados en la variable `dist`, por tanto la eficiencia será la eficiencia de `visit_min_cost` por n. Veamos cómo funciona esta función:

```
ld TSP_solution::visit_min_cost(const vector<bool>& visited, int node) {
    pair<ld,ld> shortest_edges = shortest_two_edges(visited,node);
    return (shortest_edges.first + shortest_edges.second) / 2;
}
```

Observamos que esta función delega su trabajo en la función `shortest_two_edges`, ya comentada, que devuelve las dos aristas más cortas de cada nodo, y les hace la media. Como `shortest_two_edges` tiene eficiencia $O(n)$, la función de cota tiene eficiencia $O(n^2)$ con n el número de ciudades.

7. Algoritmo Backtracking

7.1. Diseño y detalles de implementación

Diseño

El algoritmo de **backtracking** (o *vuelta atrás*) aplicado a este problema consiste en una función recursiva mediante la cual se van explorando todos los nodos (ciudades) siguiendo el algoritmo de recorrido o búsqueda de grafos **dfs** (*depth first search* o *búsqueda en profundidad*). Al estar explorando todos los caminos posibles (recorriendo todas las ciudades en todos los órdenes posibles) tenemos garantizado que la solución encongtrada será la óptima.

No obstante, al ir actualizando siempre nuestra mejor solución hasta el momento e ir construyendo los posibles caminos poco a poco, podemos saber de antemano (gracias a una **función de cota**) cuando el camino que estamos recorriendo no nos va a dar una solución mejor a la encontrada hasta la fecha. De esta forma, podemos desechar ese camino sin necesidad de procesarlo entero.Tenemos pues que nuestro algoritmo backtracking es el siguiente:

- **Caso base:** El camino ya se ha completado (ya se han recorrido todos los nodos/ciudades). Procesamos la solución (si es mejor que la que ya teníamos la actualizamos)
- **Caso general**: Todavía no hemos acabado de procesar el camino (estamos visitando una ciudad/nodo intermedio). Procedemos a visitar todas las ciudades que no visitadas (en nuestro camino actual) **salvo** las que sepamos que no nos van a llevar a una solución mejor que la que ya tenemos (nuestra cota "poda" esa "rama" al **no ser factible**)

Implementación

Como hemos mencionado antes en las sección 5, nos hemos creado una clase *Solution* para un mejor diseño del código, que comprende los *structs*, funciones de **cota** y auxiliares necesarias para implementar tanto **backtracking** como *branch and bound* con diferentes cotas a elegir. De esta forma favorecemos también la reutilización de código y cotas entre ambas técnicas, al ser muy parecidas entre sí.

```
class BK_solution : public TSP_solution
{

public:

    BK_solution(const vector<City> & v) : TSP_solution(v) {};

private:

    void algorithm(Track& e_node) override {
```

En particular, dado que la única diferencia consiste en la implentación del recorrido de las ciudades, hemos creado una clase **BK_solution** para *backtracking* que hereda de la clase general Solution e implementa el método abstracto de `algorithm` acorde a backtracking. Examinemos este método más a fondo:

```
        generated++; // Counting number of e_nodes generates
        // Already visited all cities
        if (e_node.track.size() == cities.size())
        {
            // Keep the best solution
            processSolution(e_node.track);
            return;
        }
```

Para empezar contabilizamos todos los nodos generados (no es necesario para la técnica pero nos será útil en el estudio de eficiencia y comparativa de cotas).

Después tenemos el caso base ya explicado, en el cual si ya hemos completado nuestro camino procesamos la nueva solución generada (la comparamos con nuestra mejor solución hasta la fecha y en caso de ser mejor la actualizamos).

```
        // First (origin) city is ignored
        for (int i=1; i < cities.size(); ++i)
        {
            // Ignore visited cities
            if (e_node.visited[i]) continue;
            // Ignore paths we know are worse than our current best
            // (prune unfeasible branch)
            if (!feasible(e_node,i).first) {
                podas++; // Counting number of prunings
                continue;
            }
```

En caso de no cumplir los requisitos del caso base, pasamos al caso general. Procedemos pues a visitar todas las ciudades no visitadas. Ignoramos la primera ciudad por ser el origen y ya haberla visitado siempre ya que partimos de ahí (esto hará pasar la eficiencia de $O(n!)$ a $O((n-1)!)$). También, gracias a nuestra función de cota, la cual como ya vimos es llamada por `feasible`, comprobamos si visitar la ciudad i-ésima puede darnos una solución mejor a la que tenemos actualmente. En caso negativo la ignoramos también (podamos esta rama, lo que también contabilizamos por el mismo motivo que para los nodos generados).

```
            // Add the city i to the track
            e_node.current_cost += (cities[e_node.track.back()] - cities[i]);
            e_node.track.push_back(i);
            e_node.visited[i] = true;

            algorithm(e_node); // Visit the city i
```

Una vez tenemos claro que la ciudad no ha sido visitarla y que visitarla ahora nos puede proporcionar una solución mejor a nuestra mejor solución actual, la añadimos a nuestro *camino* (sumamos el coste de la arista al coste total del camino, añadimos la ciudad al camino y la marcamos como visitada). Llamamos a `algorithm` de nuevo para explorar el camino escogido.

```
            // Remove the city i
            e_node.track.pop_back();
            e_node.visited[i] = false;
            e_node.current_cost -= (cities[e_node.track.back()] - cities[i]);
```

Por último, habiendo ya explorado el camino de visitar como siguiente ciudada la i-ésima, eliminamos la ciudad del camino para poder considerar otras opciones, es decir, visitar otras ciudades como siguiente en vez de esa. Esto incluye restar el coste añadido por su arista (su "camino" desde la última ciudad hasta ella), eliminar la ciudad del camino y desmarcarla como visitada.

7.2. Analisis de eficiencia

Dado que el algoritmo *bractraking* lo que hace es explorar todos los caminos posibles formados por n ciudades, esto es, todas las permutaciones de $n-1$ (ya que la primera ciudade siempre es la ciudad origen, 0), tenemos que la eficiencia teórica en el caso pero es $O((n-1)!)$. Esto se debe a que si nuestra función de cota no consigue podar ninguna rama (descartar algún camino), se recorren todos.

Dado que cada conjunto de ciudades es diferente y no existe ninguna condición que verifiquen todos que nos sea de utilidad, no podemos garantizar que ninguna función de poda vaya a podar siempre x nodos como mínimo, dado que siempre puede existe un caso lo suficiente malo en el cual no se pode ningún nodo (ya que est problema es NP-Difícil).

Por tanto, para poder calcular la eficiencia del algoritmo empleando cada función de cota de forma que podamos compararlas entre sí (para saber cuál es mejor en **media**), haremos lo siguiente: estudiaremos el tiempo que tarda en procesar cada nodo (en base a la función de cota elegida) de forma teórica y el número de nodos que genera de forma empírica.

Veamos primero el tiempo que tarda en procesarse cada nodo. En el caso de *backtraking*, esto consiste en el tiempo que tarda en ejecutarse una llamada a la función `algorithm` (clase *BK_solution*) sin tener el cuenta el coste añadido por la recurrencia (ya que ese lo contabilizaremos en otros nodos).

```
    void algorithm(Track& e_node) override {
        generated++; // Counting number of e_nodes generates
```

```
        // Already visited all cities
        if (e_node.track.size() == cities.size())
        {
            // Keep the best solution
            processSolution(e_node.track);
            return;
        }
```

El caso base está compuesto de una suma y una comprobación `booleana` que son $O(1)$ y la eficiencia de `processSolution`. Estudiemos la eficiencia de está función en detalle:

```
void TSP_solution::processSolution(const vector<int>& track) {
    ld cost_aux = cycleDistance(track, cities);
    if (cost_aux < cost) {
        best_ans = track;
        cost = cost_aux;
    }
}
```

Su eficiencia consiste en operaciones $O(1)$ sumado a la eficiencia de `cycleDistance`. Estudiemos la eficiencia de está función en detalle:

```
ld cycleDistance(const vector<int>& cycle, const vector<City>& v) {
    if (cycle.size() < 2) return 0;
    ld total = 0;
    for (int i = 0; i < (int)cycle.size() - 1; i++) {
        total += (v[cycle[i + 1]] - v[cycle[i]]);
    }
    total += (v[cycle[0]] - v[cycle[cycle.size() - 1]]);
    return total;
}
```

Tenemos operaciones básicas de sumas y asignaciones que son $O(1)$ y un bucle `for` que se ejecuta n veces donde n es el número de ciudades de `cycle`, es decir de `track`.

En conclusión, la eficiencia del caso base es $O(n)$. Notamos que al haber un `return` en el `if`, el caso base y el caso general son excluyentes. Es decir, es como si tuvieramos una estructura `if-else`. Aplicando las reglas de cálculo de eficiencias, la eficiencia de la función será el **máximo** entre la del caso base ya calculado y el caso general.

```
        // First (origin) city is ignored
        for (int i=1; i < cities.size(); ++i)
        {
            // Ignore visited cities
            if (e_node.visited[i]) continue;
            // Ignore paths we know are worse than our current best
            // (prune unfeasible branch)
            if (!feasible(e_node,i).first) {
                podas++; // Counting number of prunings
                continue;
            }

            // Add the city i to the track
            e_node.current_cost += (cities[e_node.track.back()] - cities[i]);
            e_node.track.push_back(i);
            e_node.visited[i] = true;

            algorithm(e_node); // Visit the city i

            // Remove the city i
            e_node.track.pop_back();
            e_node.visited[i] = false;
            e_node.current_cost -= (cities[e_node.track.back()] - cities[i]);
        }
```

Pasemos al caso general. Tenemos un bucle `for` que se ejecuta $n-1$ veces (n número de ciudades), dentro del cual tenemos sumas, asignación y expresiones booleanas que son $O(1)$, la función `feasible` y la llamada recursiva que como mencionamos anteriormente ignoraremos en este caso. Estudiemos la eficiencia de la función `feasible` en detalle:

```
pair<bool,ld> TSP_solution::feasible(Track & e_node, int node)
{
    if(e_node.track.size() < 2) return {true,0};
```

```
4
5       ld cota_inf = f_cota(e_node, node);
6
7       return {(cota_inf < cost),cota_inf};
8   }
```

Consisten en expresiones booleanas que son $O(1)$ y la función `f_cota`. Estudiemos la eficiencia de está función en detalle:

```
1   ld TSP_solution::f_cota(Track& e_node, int node) {
2       ld cota_inf = e_node.current_cost;
3       cota_inf += (cities[node] - cities[e_node.track.back()]);
4       switch (version) {
5           case 1:
6               cota_inf += f_cota1(e_node, node);
7               break;
8           case 2:
9               cota_inf += f_cota2(e_node, node);
10              break;
11          case 3:
12              cota_inf += f_cota3(e_node, node);
13              break;
14          case 4:
15              cota_inf += f_cota4(e_node,node);
16              break;
17          case 5:
18              cota_inf += f_cota5(e_node,node);
19              break;
20          default:
21              cerr << "Invalid f_cota version (1, 2, 3, 4 or 5)" << endl;
22              exit(1);
23              break;
24      }
25      return cota_inf;
26  }
```

Esta función consiste una asignación y una suma (gracias a que vamos almacenando el coste del camino eficientemente en vez de calcularlo constantemente) que son $O(1)$ y una función f_cota_i, que representa la i-ésima función de cota implementada.

Por tanto, la eficiencia del cuerpo del `for` es $O(f\_cota_i)$, siendo entonces la complejidad del caso general $O(n \cdot f\_cota_i)$.

En conclusión, el tiempo de procesamiento de un nodo para un caso de entrada de n ciudades es, aplicando la regla del máximo:

$$max\{O(n), O(n \cdot f\_cota_i)\} = O(max\{n, n \cdot f\_cota_i\}) = O(n \cdot f\_cota_i)$$

ya que $O(f\_cota_i)$ será como mínimo $O(1)$.

De esto deducimos que la eficiencia de `algorithm` (clase BK_solution) en función del número de nodos generados y la cota aplicada es:

$$O(n \cdot f\_cota_i \cdot num\_nodos)$$

Notamos que para algunas cotas, se tiene la necesidad de realizar una serie de cálculos previos para evitar que se estén recalculando constantemente y que sea más eficiente. Estos cálculos se realizan aparte (previamente) y no dependen del número de nodos. Consideramos en esos casos reemplazar la fórmula mencionada arriba por esta otra:

$$O(n \cdot f\_cota_i \cdot num\_nodos) + O(calculos\_previos)$$

Por último, si queremos saber la eficiencia del algoritmo **backtracking**, examinamos primero el constructor de la clase `BK_solution` (heredado de la clase `Solution`).

```
1   TSP_solution::TSP_solution(const vector<City>& v) {
2       podas = generated = 0;
3       cities = v;
4       TSP_greedy();
5       cost = cycleDistance(best_ans, cities);
6   }
```

Ahí tenemos operaciones básicas $O(1)$, la función `TSP_greedy` que ya vimos en la práctica anterior que era $O(n^2)$ (se corresponde con la primera versión) y la función `cyleDistance` que ya hemos visto que es $O(n)$. La eficiencia del constructor es entonces $O(n^2)$.

Finalemente estudiemos la función `solve`, que es la que llama a la función `algorithm` con el algoritmo **backtracking** implementado que hemos visto antes.

```
void TSP_solution::solve() {
    if (cities.empty()) return;
```

```
    Track e_node(cities.size());
    e_node.visited[0] = true;
    e_node.track.push_back(0);
    algorithm(e_node);
}
```

Todas la operaciones son $O(1)$ salvo la función `algorithm`. En conclusión, la **eficiencia** del algorithmo **backtracking** es:

$$O(n \cdot f\_cota_i \cdot num\_nodos + n^2)$$

Procedamos ahora a **aproximar** *num_nodos* (número de nodos generados) mediante una función (preferiblemente un polinomio) $p(n)$ para cada función de cota. Juntando la expresión de esta función con la fórmula obtenida y la eficiencia teórica de cada cota daremos con una **aproximación** de la complejidad temporal para cada cota.

7.2.1. Funcion cota 1

Eficiencia teórica

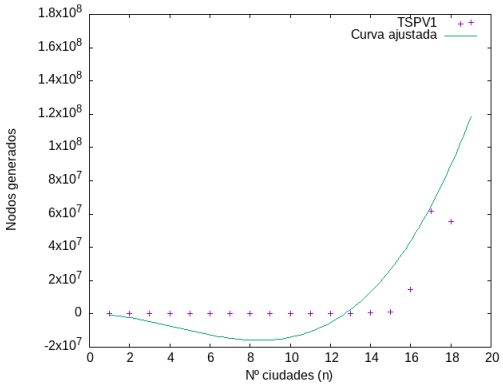
Recordamos que la eficiencia teórica de nuestra función de cota 1 vista en la sección 6.1.2, es $O(1)$, con un añadido de cálculos previos que constituyen $O(n^2 log\, n^2)$.

Sustituyendo, tenemos que la eficiencia de **backtracking** para la cota 1 en función del número de nodos generados es:

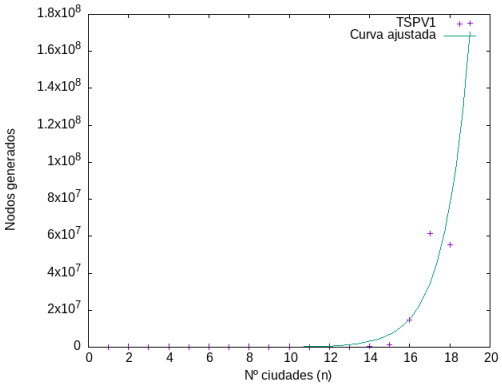
$$O(n{\cdot}f\_cota_i{\cdot}num\_nodos{+}n^2){+}O(calculos\_previos) = O(n{\cdot}num\_nodos{+}n^2){+}O(n^2) = O(n{\cdot}num\_nodos{+}n^2)$$

Aproximación número de nodos

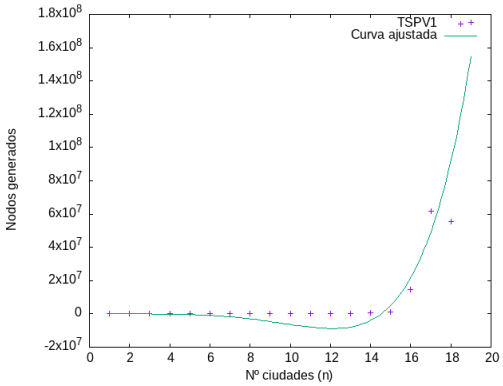
Estudiemos ahora el número de nodos generados (en función del tamaño de la entrada), tratando de aproximar esta función **híbridamente** mediante **regresiones**. Dado que no tenemos en principio ninguna expresión teórica de base que nos indique de que orden puede ser está función, hemos probado diversos ajustes híbridamente hasta dar con alguno que ajuste bien nuestra función.



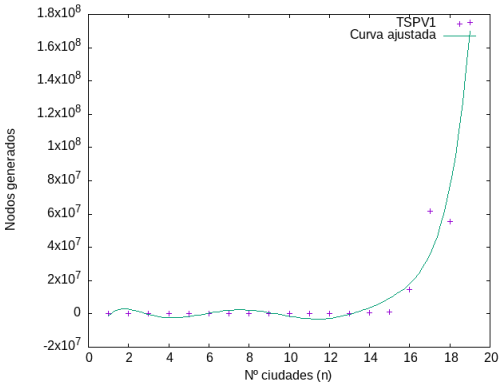
(a) Cúbica



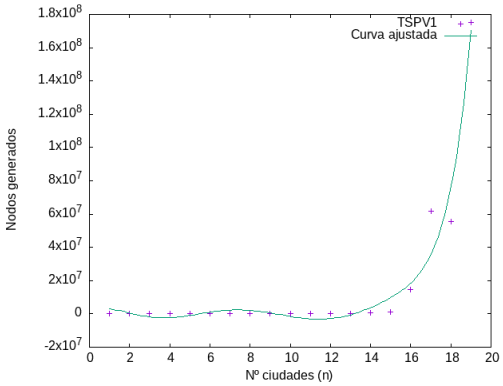
(b) Exponencial $O(e^n)$



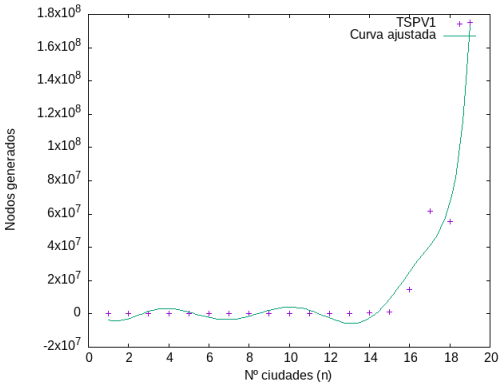
(a) Polinomio de grado 6



(b) Polinomio de grado 7



(c) Polinomio de grado 8



(d) Polinomio de grado 9

Observando las gráficas con los ajustes realizados, podemos ver a la hora de aproximar la función con un polinomio, empezamos a obtener resultados relativamente decentes a partir del grado 7, siendo muy bien ajustada para un polinomio de grado 9. Por otro lado, la función exponencial también ajusta bien a esta función. Veamos que tan buenas son las bondades de estos ajustes.

- Para la exponencial:

```
variance of residuals (reduced chisquare) = WSSR/ndf   : 7.38201e+13

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 6.42103          +/- 9.585        (149.3%)
b0              = 0.744808         +/- 0.07903      (10.61%)
```

- Para el polinomio de grado 9:

```
variance of residuals (reduced chisquare) = WSSR/ndf   : 9.39129e+13

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 1.62664          +/- 2.241        (137.7%)
b               = -127.662         +/- 201.9        (158.1%)
c               = 4118.48          +/- 7703         (187%)
d               = -70253.9         +/- 1.622e+05    (230.9%)
e               = 677934           +/- 2.056e+06    (303.3%)
f               = -3.65606e+06     +/- 1.608e+07    (439.8%)
g               = 1.00375e+07      +/- 7.632e+07    (760.4%)
h               = -1.05739e+07     +/- 2.074e+08    (1961%)
i               = 1.03157          +/- 2.838e+08    (2.751e+10%)
j               = 1.05309          +/- 1.424e+08    (1.352e+10%)
```

Comparando las **varianzas residuales**, tenemos que la función exponencial ajusta mejor a nuestra función que el polinomio de grado 9, no obstante nos quedaremos con este último dado que la diferencia no es muy significativa y los polinomios suelen ser más ventajosos. Notamos que a pesar de ver un buen ajuste en a gráfica, la varianza residual sigue siendo considerablemente **alta**.

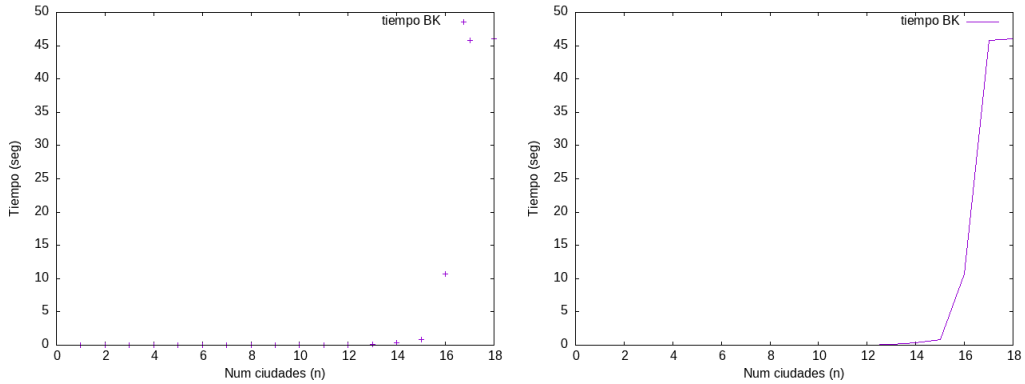
Esto es de esperarse, dado que el número de nodos generados depende mucho del caso en particular de entrada, y no solo del tamaño de la entrada (aunque este claramente también influye).

Tomaremos entonces como función de aproximación del número de nodos generados en base a n a una de orden $O(n^9)$. Sustituyendo *num_nodos* en la fórmula previamente deducida nos queda que la eficiencia teórica es:

$$O(n \cdot num\_nodos + n^2) = O(n \cdot n^9 + n2) = O(n^{10})$$

Eficiencia empírica

Veamos ahora la eficiencia empírica de nuestro algoritmo backtracking para la cota 1.



(a) Puntos (b) Líneas

Podemos ver que el algoritmo bracktracking crece **muy** rápidamente para la cota 1.

Eficiencia híbrida

Procedamos ahora a ajustar los datos obtenidos para la eficiencia empírica mediante la expresión obtenida en la eficiencia teórica, $O(n^{10})$.

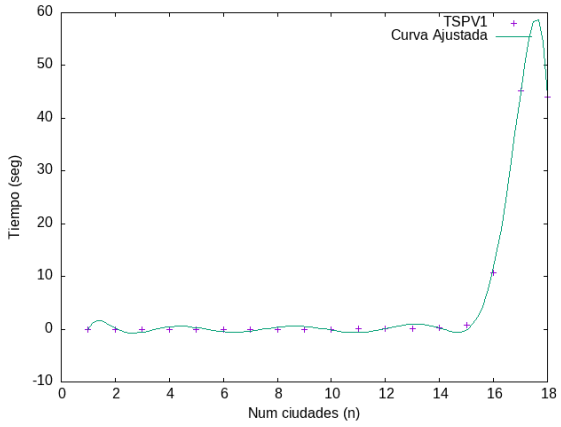


Figura 7: Tiempo ajustado por polinomio de grado 10

Veamos la varianza residual del ajuste:

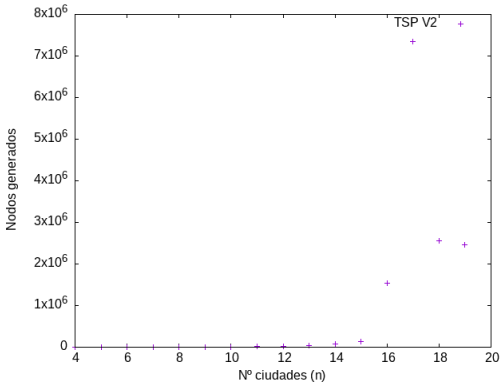
```
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.658663

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = -62.7335         +/- 26.75        (42.64%)
a1              = 154.11           +/- 61.05        (39.62%)
a2              = -145.847         +/- 53.06        (36.38%)
a3              = 71.9815          +/- 23.95        (33.28%)
```

```
a4              = -20.9532          +/- 6.38          (30.45%)
a5              = 3.82151           +/- 1.067         (27.92%)
a6              = -0.44809          +/- 0.1151        (25.68%)
a7              = 0.0337065         +/- 0.007988      (23.7%)
a8              = -0.00156995       +/- 0.0003446     (21.95%)
a9              = 4.11677e-05       +/- 8.399e-06     (20.4%)
a10             = -4.64101e-07      +/- 8.835e-08     (19.04%)
```

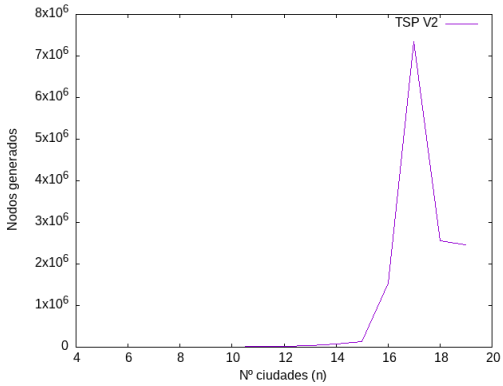
Como podemos ver, el ajuste está lejos de ser perfecto, como era de esperarse por no poder ajustar bien el número de nodos generados, pero aproxima bastante bien la complejidad temporal de backtracking para la cota 1, como se puede ver de la varianza residual.

7.2.2. Funcion cota 2

Eficiencia teórica-híbrida
Teniendo en cuenta que esta función de cota tiene eficiencia $O(n^2)$, veamos si podemos aproximar el número de nodos que se generan por el Backtraking se puede aproximar por alguna función. Ejecuntado el algoritmo con diferentes tamaños de entrada, entre 4 y 19, obtenemos las siguientes gráficas:



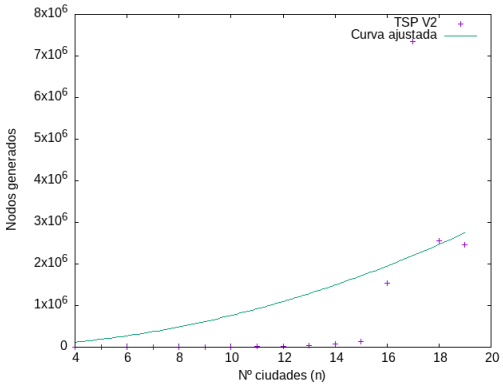
(a) Puntos.



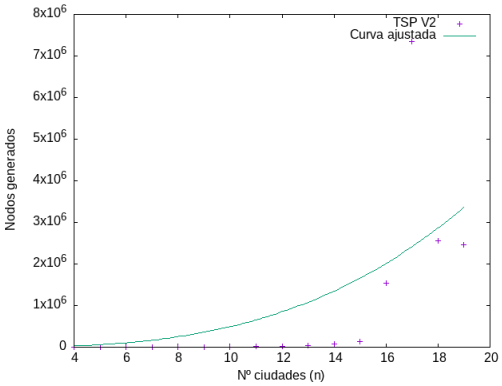
(b) Lineas.

Podemos observar que el número de nodos generados no es estable y fluctúa mucho, esto es debido a que el algoritmo lista las distintas posibilidad en un orden sistemático y nuestra función de poda no puede influir en esto, solo puede ayudar a que se generen menos nodos, posibilitando así que la solución óptima puede estar entre uno de los últimos nodos generados.

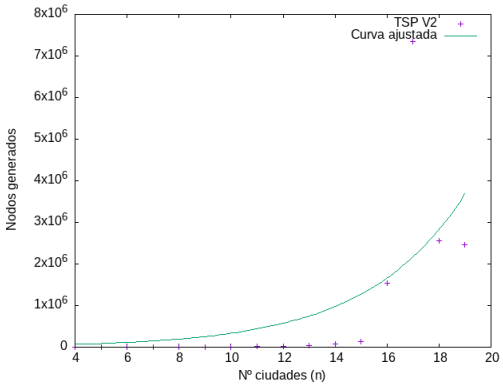
Aproximando el comportamiento por diferentes tipos de funciones, desde polinómicas y exponenciales hasta factoriales, obtenemos:



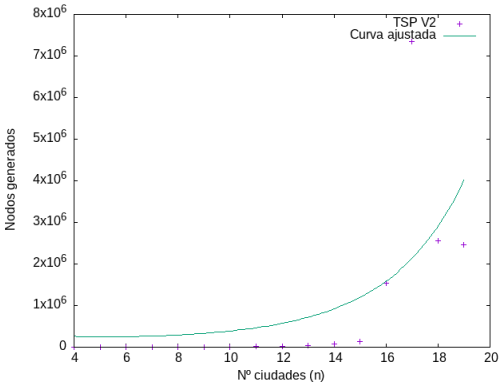
(a) Cuadratica con $f(x) = 7625{,}1x^2$



(b) Cubica con $f(x) = 491{,}71x^3$



(c) Exponencial con $f(x) = 23374e^{0,27x}$



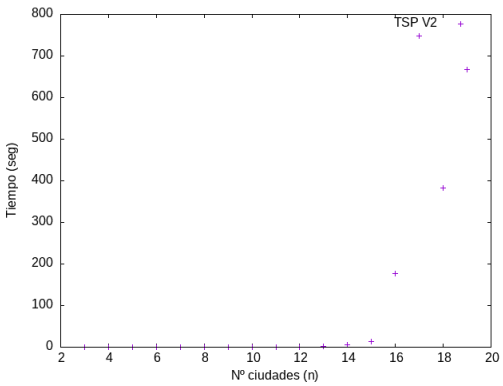
(d) Factorial con $f(x) = 282197\Gamma(0{,}23(x+1))$

Visualmente podemos ver que ninguna de ellas aproximan decentemente a los resultados obtenidos y analizando los errores de la regresión confirman lo anterior, todos ellos tienen una varianza residual del orden de 10^{12} (más detalles en el fichero BK/fit.log).

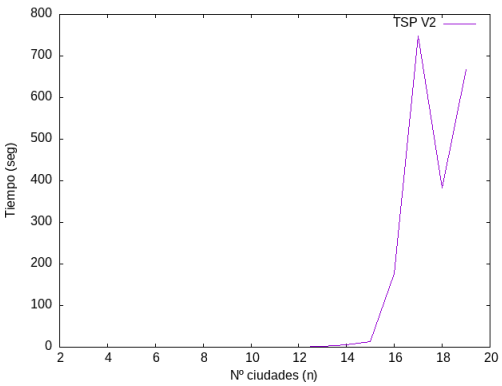
Tomaremos como función de aproximación de nodos la función exponencial ya que de entre todas las mediocres regresiones es la mejor. Por tanto, combinando todo tenemos que si T(n) es la eficiencia del BK, entonces $T(n) \in O(n^3e^n + n^2) = O(n^3e^n)$.

Eficiencia empírica e híbrida

Recogiendo diferentes tiempos de ejecución con datos de entrada entre 4-19 ciudades obtenemos las siguientes gráficas:



(a) Puntos.

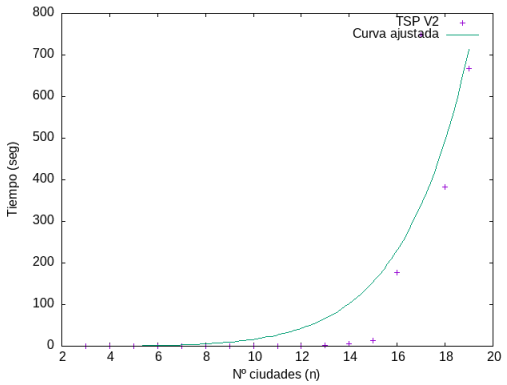


(b) Lineas

Podemos ver que el comportamiento es muy irregular e inestable, ya que tiene un factor del

azar en su comportamiento, pero refleja a la perfección el crecimiento en tiempo de la ejecución a medida que aumentan los tamaños de entrada.

Ahora ajustamos con una curva del tipo $f(x) = ax^3e^{bx}$, tipo de función obtenida teorica-empíricamente, y obtenemos el siguiente resultado:



$$f(x) = 0{,}00213x^3e^{0{,}205x}$$

Analicemos la calidad de este ajuste:

```
degrees of freedom    (FIT_NDF)                        : 15
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 121.438
variance of residuals (reduced chisquare) = WSSR/ndf   : 14747.3

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                 = 0.00213164     +/- 0.003907     (183.3%)
b                 = 0.204579       +/- 0.1007       (49.2%)
```

Podemos ver que tanto la varianza residual como el porcentaje de error asintótico estándar toman valores muy altos, esto quiere decir que nuestro ajuste no es preciso, tal y como se puede apreciar en la gráfica. Podemos atribuir dicho fracaso de la regresión al comportamiento relativamente impredictible de nuestro algoritmo, ya que depende de según dónde se encuentre la solución óptima, tendremos que recorrer más o menos nodos.

7.2.3. Funcion cota 3

Como ya se comentó en la sección 6.3.2, la función de cota 3 tiene eficiencia cuadrática en el número de ciudades, luego la eficiencia del algoritmo backtracking será de:

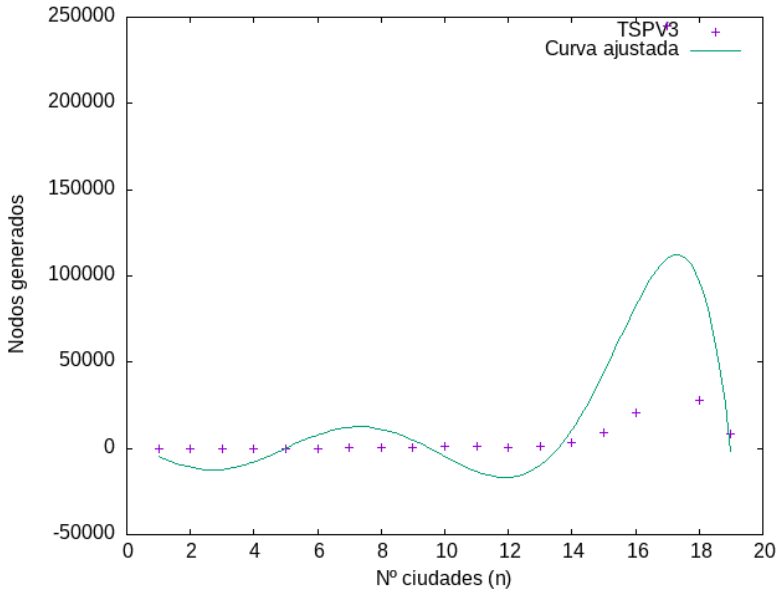
$$O(n^3 \cdot num\_nodos)$$

Pues ya será mayor que n^2. Luego nuestro objetivo es estudiar el número de nodos que se generan. Lo ideal sería encontrar una función devuelva el número de nodos generados en función del número de ciudades, no obstante eso no es posible, pues depende mucho del número de casos, como se muestra en la tabla:

Lo máximo que se puede hacer es buscar una regresión que aproxime lo máximo posible a un muestreo, que en nuestro caso sera $1 \leqslant n \leqslant 19$. Se ha probado con n^p para $1 \leqslant p \leqslant 9$ y con una regresión exponencial, los mejores resultados se han obtenido para n^6 y para n^9, como se muestra a continuación:

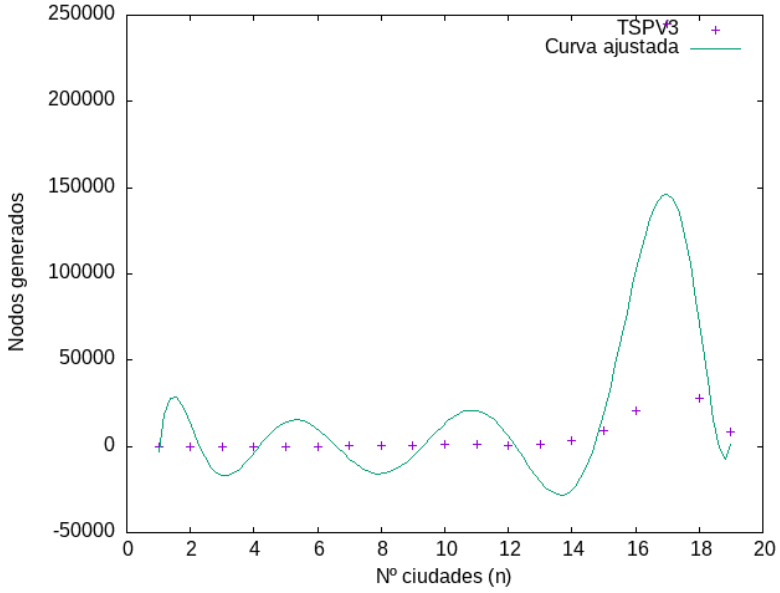
Ciudades	Nodos generados
1	1
2	2
3	5
4	12
5	25
6	40
7	169
8	482
9	600
10	1253
11	1297
12	426
13	1209
14	3015
15	8809
16	20653
17	245151
18	27792
19	8702

Tabla 1: Nodos generados para la cota 3 en el algoritmo backtracking



$$f(x) = -0{,}622198 \cdot x^6 + 29{,}2992 \cdot x^5 - 484{,}335 \cdot x^4 + 3294{,}8 \cdot x^3 - 7637{,}95 \cdot x^2 + 1{,}25944 \cdot x + 1{,}02235$$

```
degrees of freedom    (FIT_NDF)                        : 12
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 49724.6
variance of residuals (reduced chisquare) = WSSR/ndf   : 2.47253e+09

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = -0.622198        +/- 0.9206       (148%)
b               = 29.2992          +/- 55.39        (189.1%)
c               = -484.335         +/- 1288         (265.9%)
d               = 3294.8           +/- 1.452e+04    (440.8%)
e               = -7637.95         +/- 8.145e+04    (1066%)
f               = 1.25944          +/- 2.062e+05    (1.637e+07%)
g               = 1.02235          +/- 1.726e+05    (1.689e+07%)
```



$f(x) = 0{,}0170416 \cdot x^9 - 1{,}49802 \cdot x^8 + 55{,}3892 \cdot x^7 - 1121{,}19 \cdot x^6 + 13550{,}3 \cdot x^5 - 100071 \cdot x^4 + 443999 \cdot x^3 - 1116220 \cdot x^2 + 1401350 \cdot x - 644323$

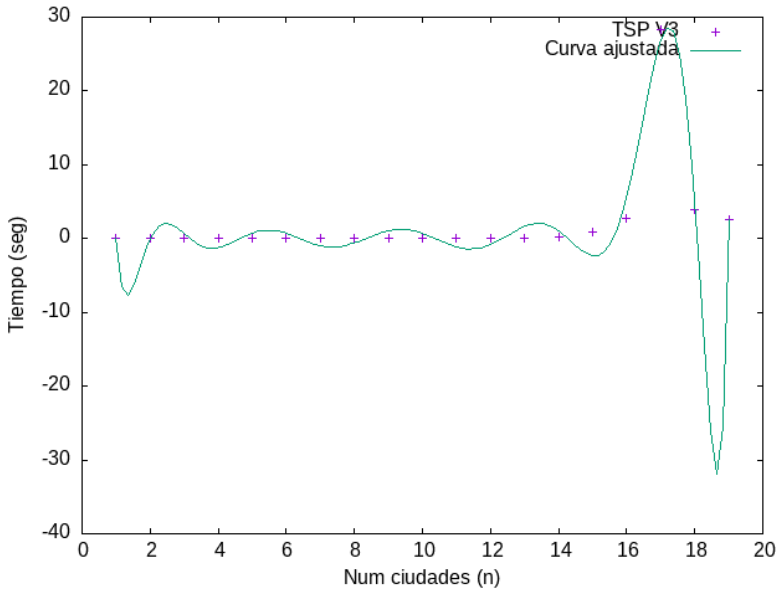
```
degrees of freedom    (FIT_NDF)                        : 9
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 48737.8
variance of residuals (reduced chisquare) = WSSR/ndf   : 2.37538e+09

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                = 0.0170416       +/- 0.01127      (66.12%)
b                = -1.49802        +/- 1.015        (67.77%)
c                = 55.3892         +/- 38.74        (69.95%)
d                = -1121.19        +/- 815.7        (72.75%)
e                = 13550.3         +/- 1.034e+04    (76.32%)
f                = -100071         +/- 8.087e+04    (80.82%)
g                = 443999          +/- 3.839e+05    (86.45%)
h                = -1.11622e+06    +/- 1.043e+06    (93.44%)
i                = 1.40135e+06     +/- 1.427e+06    (101.8%)
j                = -644323         +/- 7.16e+05     (111.1%)
```

Observamos que las gráficas no aproximan ni por asomo a los puntos, lo cual lo refleja el alto valor de la varianza residual, aunque de todas las regresiones que se han hecho son las dos con varianza residual más baja. Es por eso que nos quedaremos con estas regresiones, en particular con la mejor de ellas: n^9. Siguiendo este modelo podríamos decir que nuestro algoritmo backtracking con la cota 3 tiene una eficiencia de $O(n^{12})$, aunque esto no tenga ningún sentido.

Aún así vamos a ajustar la gráfica de los tiempos por un polinomio de grado 12 y ver qué ocurre:

```
degrees of freedom    (FIT_NDF)                        : 6
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 2.29668
variance of residuals (reduced chisquare) = WSSR/ndf   : 5.27472

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0               = 448.07          +/- 259.9        (58%)
a1               = -1219.1         +/- 678.1        (55.62%)
a2               = 1331.53         +/- 704          (52.87%)
a3               = -789.242        +/- 394.9        (50.04%)
a4               = 287.395         +/- 135.9        (47.28%)
a5               = -68.6027        +/- 30.66        (44.69%)
a6               = 11.1218         +/- 4.704        (42.29%)
a7               = -1.24438        +/- 0.499        (40.1%)
a8               = 0.0960471       +/- 0.03661      (38.11%)
```

```
a9                      = -0.00501725           +/- 0.001822        (36.31%)
a10                     = 0.000169223           +/- 5.868e-05       (34.67%)
a11                     = -3.32313e-06          +/- 1.103e-06       (33.2%)
a12                     = 2.88407e-08           +/- 9.189e-09       (31.86%)
```



$$f(x) = 2{,}88407 \cdot 10^{-8} \cdot x^{12} - 3{,}32313 \cdot 10^{-6} \cdot x^{11} + 0{,}000169223 \cdot x^{10} - 0{,}00501725 \cdot x^{9} + 0{,}0960471 \cdot x^{8} - 1{,}24438 \cdot x^{7} + 11{,}1218 \cdot x^{6} - 68{,}6027 \cdot x^{5} + 287{,}395 \cdot x^{4} - 789{,}242 \cdot x^{3} + 1331{,}53 \cdot x^{2} - 1219{,}1 \cdot x + 448{,}07$$

Aunque la varianza residual nos haya salido considerablemente menor, insistimos en que esta forma de modelizar el tiempo en función del número de ciudades no tiene mucho sentido al no seguir el número de nodos una monotonía respecto al número de ciudades.

8. Algoritmo Branch and Bound

8.1. Diseño y detalles de implementación

Introducción

El algoritmo Branch and Bound (BB) es una técnica de búsqueda heurística ampliamente utilizada para resolver problemas de optimización combinatoria, como el problema del vendedor ambulante (TSP). Al igual que el algoritmo Backtracking (BK), el objetivo de BB es encontrar la ruta óptima que visita todas las ciudades y regresa al punto de partida. Sin embargo, BB utiliza una estrategia diferente para explorar el espacio de soluciones y, en general, es más eficiente que BK para problemas de gran tamaño con cotas lo suficiente buenas.

Funcionamiento del algoritmo

El algoritmo BB se basa en la idea de dividir el problema en subproblemas más pequeños y resolverlos, al menos de forma natural, recursivamente. [1] Cada subproblema representa una parte del espacio de búsqueda, y la solución óptima del problema original se encuentra al combinar las soluciones óptimas de sus subproblemas.

Para implementar el algoritmo BB, se utilizan dos técnicas principales:

- Ramificación: Se divide el problema en subproblemas más pequeños seleccionando una variable de decisión y asignándole diferentes valores.
- Acotaciones: Se calculan cotas superior e inferior para el valor óptimo de cada subproblema. Si la cota superior de un subproblema es menor que la cota inferior de otro subproblema, el primer subproblema se puede descartar, ya que no puede contener la solución óptima.

Ventajas del algoritmo BB

- Eficiencia: El algoritmo BB puede ser más eficiente que el algoritmo BK para problemas de gran tamaño con cotas eficaces, ya que al implementarse con una `priority_queue` tenemos que procesamos antes las soluciones que serán posiblemente mejores, provocando esto que se realicen (probablemente) más podas que en backtracking (al tener una cota superior menor y por tanto mejor).
- Flexibilidad: BB se puede adaptar a diferentes problemas de optimización combinatoria con modificaciones menores.

Desventajas del algoritmo BB

- Complejidad: BB puede ser más complejo de implementar que BK.
- Memoria: BB puede requerir más memoria que BK, ya que almacena información sobre las ramas del árbol de búsqueda que se han explorado. (Y más en este caso concreto, almacenando los nodos en una priority_queue)

Conclusión

El algoritmo BB es una técnica de búsqueda heurística poderosa que se puede utilizar para resolver problemas de optimización combinatoria complejos, como el TSP. Su eficiencia y flexibilidad lo convierten en una herramienta valiosa para una amplia gama de aplicaciones.

[1] Si bien en este caso lo hemos implementado de forma iterativa con una cola con prioridad

Comparación con el algoritmo Backtracking (BK)

El algoritmo BB es similar al algoritmo Backtracking (BK) en el sentido de que ambos exploran sistemáticamente el espacio de búsqueda para encontrar la solución óptima. Sin embargo, existen algunas diferencias clave entre los dos algoritmos:

- Recorrido del árbol de búsqueda: El algoritmo BB utiliza un recorrido de la estructura de árbol de búsqueda similar al **bfs**. Se diferencia en que en este caso usamos una `priority_queue` en vez de una `queue` normal. Esto constrasto con el recorrido realizado en backtracking que era justamente un *dfs*.
- Eficiencia: En general, el algoritmo BB se considera más eficiente que el algoritmo BK para problemas de gran tamaño con una buena cota.

En resumen, el algoritmo BB es una herramienta más versátil y eficiente que el algoritmo BK para resolver problemas de optimización combinatoria complejos.

Implementación

Como hemos mencionado antes en la sección 5, nos hemos creado una clase *Solution* para un mejor diseño del código, que comprende los *structs*, funciones de **cota** y auxiliares necesarias para implementar tanto *backtracking* como **branch and bound** con diferentes cotas a elegir. De esta forma favorecemos también la reutilización de código y cotas entre ambas técnicas, al ser muy parecidas entre sí.

```
class BB_solution : public TSP_solution
{

public:

    BB_solution(const vector<City> & v) : TSP_solution(v) {};
```

En particular, dado que la única diferencia consiste en la implementación del recorrido de las ciudades, hemos creado una clase **BB_solution** para *branch and bound* que hereda de la clase general Solution e implementa el método abstracto de `algorithm` acorde a branch and bound. Examinemos este método más a fondo:

```
    void algorithm(Track& first_node) override {
        priority_queue<Track, vector<Track>, greater<Track>> nodos_vivos;
        nodos_vivos.push(first_node);

        while (!nodos_vivos.empty()) {
            generated++;
            Track e_node = nodos_vivos.top();
            nodos_vivos.pop();
```

Para empezar, creamos la priority_queue en la que almacenaremos los nodos según su cota inferior, añadimos el primer nodo y comenzamos a pocesar: Almacenamos en e_node el nodo que toca según la cola (en la primera iteración el primero) y lo sacamos de la cola.

```
            if (e_node.aprox_cost >= cost) continue;

            if (e_node.track.size() == cities.size()) {
                processSolution(e_node.track);
                continue;
            }
```

Después, si hemos superado el costo de nuestra mejor solución, podemos olvidarnos de este camino y seguir con el siguiente.
Suponiendo que no es así, tenemos el caso base ya explicado, en el cual si ya hemos completado nuestro camino procesamos la nueva solución generada (la comparamos con nuestra mejor solución hasta la fecha y en caso de ser mejor la actualizamos).

```
            for (int i = 1; i < cities.size(); ++i) {
                if (e_node.visited[i]) continue;
                pair<bool, double> viable = feasible(e_node, i);
                if (!viable.first) {
```

```
                    podas++;
                    continue;
                }
```

En caso de no cumplir los requisitos del caso base, pasamos al caso general. Procedemos pues a visitar todas las ciudades no visitadas. Ignoramos la primera ciudad por ser el origen y ya haberla visitado siempre ya que partimos de ahí (esto hará pasar la eficiencia de $O(n!)$ a $O((n-1)!)$). También, gracias a nuestra función de cota, la cual como ya vimos es llamada por `feasible`, comprobamos si visitar la ciudad i-ésima puede darnos una solución mejor a la que tenemos actualmente. En caso negativo la ignoramos también (podamos esta rama, lo que también contabilizamos por el mismo motivo que para los nodos generados).

```
                Track aux = e_node;
                aux.current_cost += (cities[aux.track.back()] - cities[i]);
                aux.track.push_back(i);
                aux.aprox_cost = viable.second;
                aux.visited[i] = true;

                nodos_vivos.push(aux);
```

Una vez tenemos claro que la ciudad no ha sido visitada y que visitarla ahora nos puede proporcionar una solución mejor a nuestra mejor solución actual, la añadimos a nuestro *camino* (sumamos el coste de la arista al coste total del camino, añadimos la ciudad al camino y la marcamos como visitada). Además la añadimos a nuestra priority_queue para que en la siguiente iteración se procese (y así seguir recorriendo este camino)

8.2. Analisis de eficiencia

Ahora vamos a estudiar la eficiencia de este algoritmo utilizando distintas funciones de cota. En primer lugar, vamos a estudiar la cantidad de nodos generados en relación al número de ciudades. Teóricamente en el caso peor este debería ser (n-1)! si n fuese el número de ciudades, pero dependiendo de lo bueno que sea la función de cota utilizada esta cantidad puede verse reducida. Por tanto, el objetivo es encontrar una función p(n) (preferiblemente un polinomio) que aproxime dicha cantidad de nodos.

Una vez determinado lo anterior, notemos que el algoritmo esta compuesto en esencia de un bucle while que itera p(n) veces, ya que no para hasta que la cola que guarda todos los nodos vivos esté vacía. Su cuerpo está formado por, a parte de sentencias simples de orden O(1), por otro bucle for que itera n-1 veces y llama a la función de cota en cada iteración. También hay una inserción en la cola de prioridad en el bucle, pero como no siempre ocurre, ya que hay un 'continue' que le precede, concluimos que esta inserción ocurre p(n) veces (ya que lo que se insertan son nodos factibles), independientemente de las iteraciones del bucles, y como cada inserción es del orden logarítmico, las inserciones en total tiene coste $p(n)\log p(n)$. Por tanto, si llamamos a la eficiencia de la función cota q(n) y T(n) a la eficiencia del algoritmo de Branch And Bound, aplicando la regla del máximo y de la suma tenemos:

$$T(n) = p(n)(n-1)q(n) + p(n)\log p(n) = p(n)((n-1)q(n) + \log p(n)) = p(n)(n\,q(n) + \log p(n))$$

8.2.1. Funcion cota 1

Eficiencia teórica

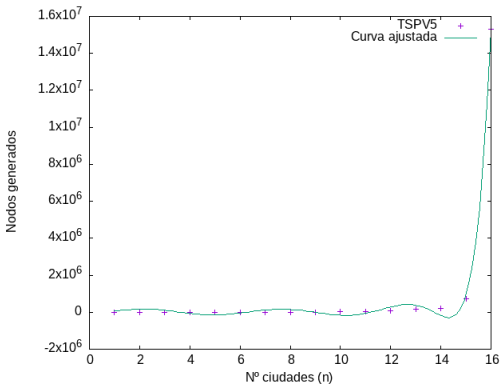
Recordamos que la eficiencia teórica de nuestra función de cota 1 vista en la sección 6.1.2, es $O(1)$, con un añadido de cálculos previos que constituyen $O(n^2 log\, n^2)$.

Sustituyendo la eficiencia de nuestra cota ($q(n) = 1$), tenemos que la eficiencia de **branch and bound** para la cota 1 en función del número de nodos generados es:

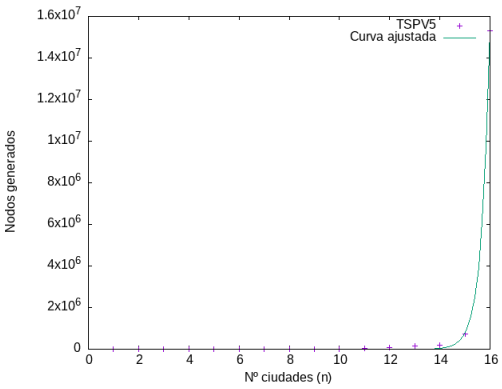
$$O(p(n)(n\,q(n) + \log p(n))) = O(p(n)(n \cdot 1 + \log p(n))) + O(n^2) = O(p(n)(n + \log p(n)) + n^2)$$

Tratatemos de dar ahora con una expresión **relativamente aproximada** del número de nodos generados en función del tamaño de la entrada ($p(n)$).

Aproximación número de nodos
Estudiemos ahora el número de nodos generados (en función del tamaño de la entrada), tratando de aproximar esta función **híbridamente** mediante **regresiones**. Dado que no tenemos en principio ninguna expresión teórica de base que nos indique de que orden puede ser está función, hemos probado diversos ajustes híbridamente hasta dar con alguno que ajuste bien nuestra función.



(a) Polinomio de grado 9



(b) Exponencial $O(e^n)$

Podemos observar que ambos ajustes son muy malos. Probemos aumentando el grado del polinomio.

Comparemos sus **varianzas residuales** para ver cual ajusta mejor a la función:

- Exponencial:

```
After 5554 iterations the fit converged.
final sum of squares of residuals : 6.31257e+10
rel. change during last iteration : -9.99537e-06

degrees of freedom    (FIT_NDF)                        : 14
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 67149
variance of residuals (reduced chisquare) = WSSR/ndf   : 4.50898e+09

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 1.70587e-14      +/- 2.38e-14     (139.5%)
b0              = 3.01533          +/- 0.08514      (2.823%)

correlation matrix of the fit parameters:
                a0     b0
a0              1.000
b0             -1.000  1.000
```

- Polinomio de grado 9:

```
degrees of freedom    (FIT_NDF)                        : 6
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 268393
variance of residuals (reduced chisquare) = WSSR/ndf   : 7.20349e+10

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 0.312568         +/- 0.3875       (124%)
b               = -18.2986         +/- 29.67        (162.2%)
c               = 431.87           +/- 964.4        (223.3%)
d               = -5232.41         +/- 1.734e+04    (331.4%)
e               = 34100.2          +/- 1.884e+05    (552.6%)
f               = -112389          +/- 1.27e+06     (1130%)
g               = 144711           +/- 5.228e+06    (3612%)
h               = 0.577974         +/- 1.245e+07    (2.153e+09%)
i               = 0.471788         +/- 1.514e+07    (3.21e+09%)
j               = 0.577249         +/- 6.907e+06    (1.197e+09%)
```

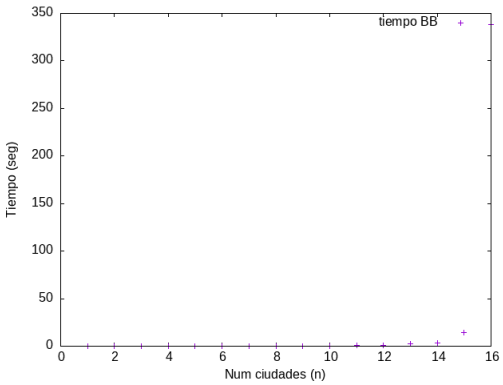
Podemos ver que la función exponencial ajusta mejor al número de nodos generados en base al tamaño de entrada. En este caso, si nos fijamos en la varianza residual, podemos ver que la de la función exponencial es mucho menor (un orden menos) que la del polinomio. Si tomamos como $p(n) = O(e^n)$ nos queda:

$$O(p(n)(n + \log p(n)) + n^2) = O(e^n(n + \log e^n) + n^2) = O(n\, e^n + e^n \log n^9) = O(n\, e^n)$$

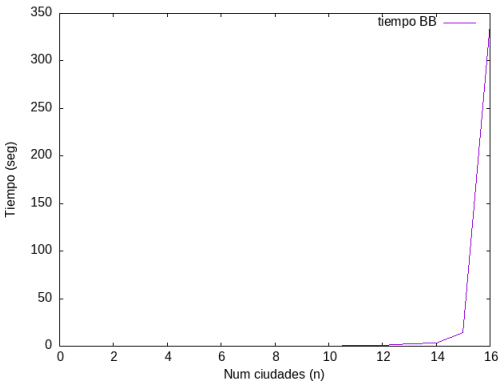
Por otro lado si escogiésemos el polinomio de grado 9, nos quedaría al final un polinomio de grado 10.

Eficiencia empírica

Veamos ahora la eficiencia empírica de nuestro algoritmo branch and bound para la cota 1.



(a) Puntos

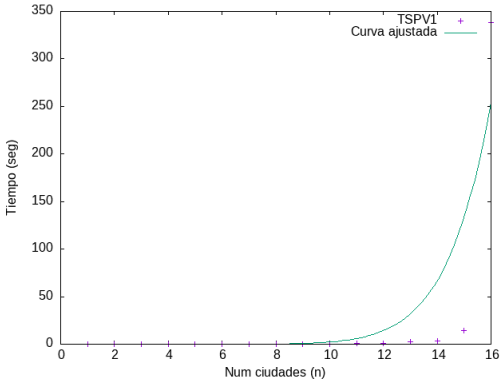


(b) Líneas

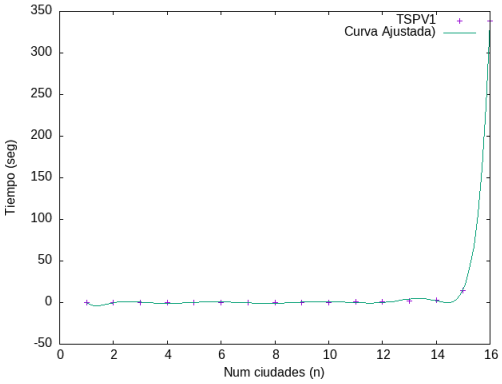
Podemos ver que la complejidad crece muy rápidamente pero con picos notables carácteristicos de branch and bound. Esto se debe a lo que hablamos antes, que el número de nodos generados no depende exclusivamente del tamaño de entrada, sino que cada caso particular es propenso a tener más o menos podas.

Eficiencia híbrida

Procedamos ahora a ajustar los datos obtenidos para la eficiencia empírica mediante la expresión obtenida en la eficiencia teórica. En este caso probaremos tanto con el ajuste exponencial $O(n\, e^n)$ como con el ajuste del polinomio de grado 9 (que nos queda un polinomio de grado 10).



(a) Tiempo ajustado por $O(n\, e^n)$



(b) Tiempo ajustado por polinomio de grado 10

Veamos la varianza residual del ajuste de la figura 13a:

```
variance of residuals (reduced chisquare) = WSSR/ndf   : 698.618

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 2.1293e-06       +/- 1.745e-07    (8.196%)
```

Veamos la varianza residual del ajuste de la figura 13b:

```
variance of residuals (reduced chisquare) = WSSR/ndf   : 2.19275

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 179.028          +/- 80.89        (45.18%)
a1              = -460.203         +/- 195.5        (42.48%)
a2              = 463.496          +/- 182.8        (39.43%)
a3              = -246.808         +/- 89.74        (36.36%)
a4              = 78.3653          +/- 26.2         (33.43%)
a5              = -15.7263         +/- 4.831        (30.72%)
a6              = 2.04334          +/- 0.577        (28.24%)
a7              = -0.171328        +/- 0.0445       (25.98%)
a8              = 0.00894024       +/- 0.002138     (23.92%)
a9              = -0.000263844     +/- 5.817e-05    (22.05%)
a10             = 3.36193e-06      +/- 6.839e-07    (20.34%)
```

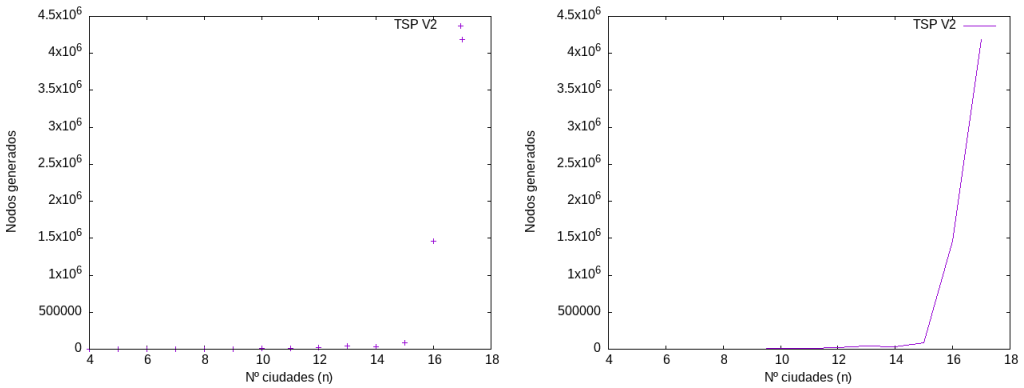
Aquí observamos que aunque en principio la función exponencial aproximaba mejor a la función de número de nodos, podemos ver en el caso de eficiencia, la varianza de la expresión obtenida utilizando el polinomio de grado 9 es menor que el de la exponencial.

Como podemos ver, el ajuste está lejos de ser perfecto, como era de esperarse por no poder ajustar bien el número de nodos generados, pero aproxima medianamente la complejidad temporal de branch and bound para la cota 1, como se puede ver de la varianza residual.

8.2.2. Funcion cota 2

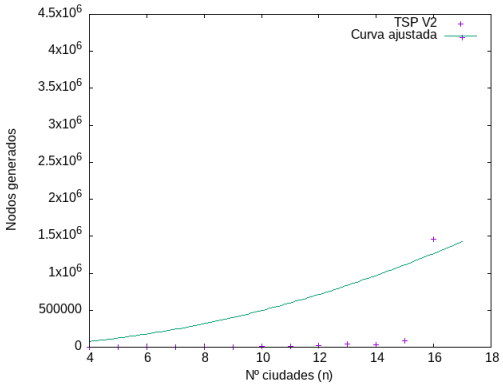
Eficiencia teórica-híbrida

En primer lugar, la eficiencia de esta función de cota, como ya hemos comentado anteriormente es del orden $O(n^2)$ en el caso peor, es decir, $O(q(n)) = O(n^2)$. Por otra parte, calculando el número de nodos generados obtenemos los siguientes datos (hemos trabajado entre 4-17 ciudades debido a las limitaciones HW):

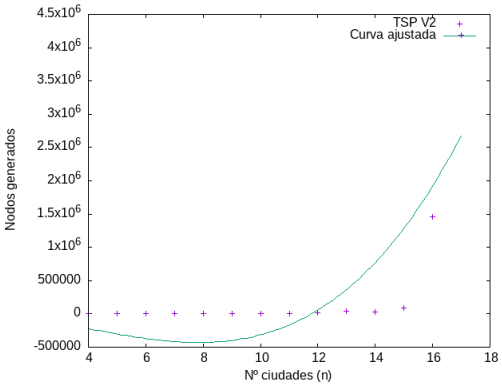


(a) Puntos. (b) Lineas.

Podemos ver que la velocidad de crecimiento sigue siendo factorial o exponencial. En un intento de aproximarlos por polinomios obtenemos un total fracaso:

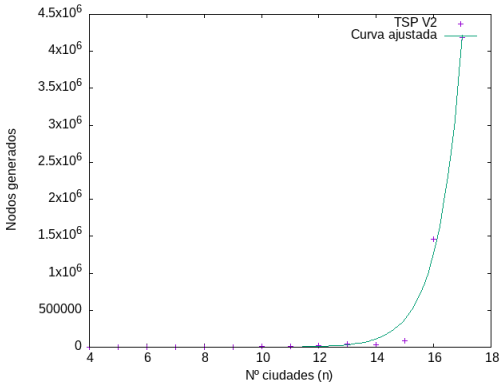


(a) Cuadratica.

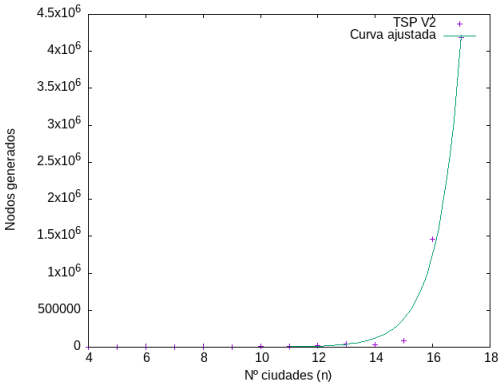


(b) Cubica.

Aproximando mediante funciones exponenciales y factoriales obtenemos mejores resultados:



(a) Exponencial con $f(x) = 0{,}0054 * e^{1{,}2x}$



(b) Exponencial con $f(x) = 14{,}59\Gamma(0{,}55(x+1))$

Graficamente son practicamente idénticas, pero analizando los errores más a fondo tenemos:

- Análisis de la regresión facorial:

```
degrees of freedom    (FIT_NDF)                        : 12
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 106858
variance of residuals (reduced chisquare) = WSSR/ndf   : 1.14186e+10

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 14.5886          +/- 13.01        (89.2%)
b               = 0.549952         +/- 0.02203      (4.006%)
```

- Análisis de la regresión exponencial:

```
degrees of freedom    (FIT_NDF)                        : 12
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 103078
variance of residuals (reduced chisquare) = WSSR/ndf   : 1.0625e+10

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 0.00543297       +/- 0.006434     (118.4%)
b               = 1.20413          +/- 0.06935      (5.759%)
```

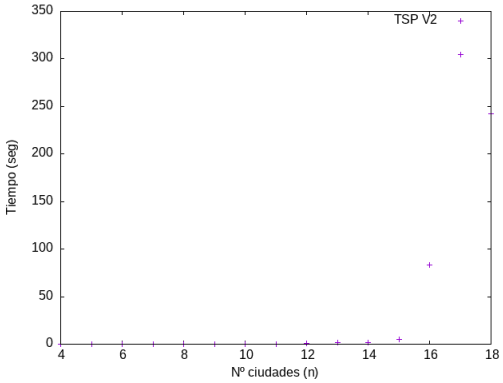
A la vista de los datos obtenidos, notamos que la diferencia entre ellos con estos datos es también casi irrelevante, así, tomaremos el exponencial por ser levemente menor el error, $O(p(n)) = O(e^x)$.

A la vista de $O(p(n)) = O(e^x)$ y $O(q(n)) = O(n^2)$ tenemos que la eficiencia de BB con esta función de cota queda de la siguiente manera:

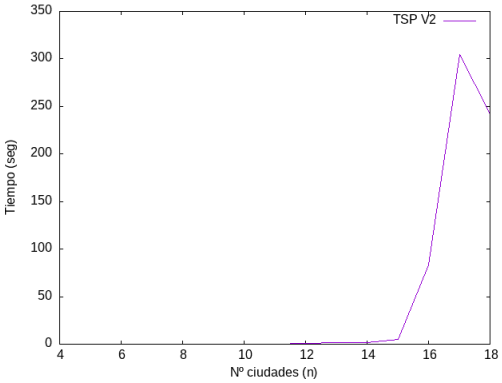
$$T(n) = p(n)(n \cdot q(n) + \log p(n)) \in O(e^n \cdot (n^3 + \log e^n)) = O(n^3 e^n)$$

Eficiencia empírica e híbrida:

Ejecutando el programa con diferentes tamaños de entrada (3-17) obtenemos los siguientes datos:



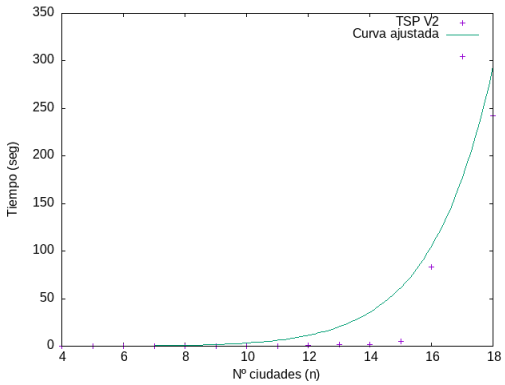
(a) Puntos



(b) Lineas

Como era de esperar los tiempos crecen exponencialmente según los datos y además son muy inestables, esto es debido, como ya hemos comentado anteriormente, a que la poda ayuda a reducir el tiempo según el caso pero no determina el comportamiento del algoritmo.

Ajustándolo con una curva del tipo $f(x) = ax^3e^{bx}$ obtenemos el siguiente resultado:



$$f(x) = 0{,}0001x^3e^{0{,}339717x}$$

Veamos la calidad de este ajuste:

```
1
2 degrees of freedom    (FIT_NDF)                        : 13
3 rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 43.3845
4 variance of residuals (reduced chisquare) = WSSR/ndf   : 1882.21
5
6 Final set of parameters            Asymptotic Standard Error
7 =======================            ==========================
8 a               = 0.000111446      +/- 0.0002569    (230.5%)
9 b               = 0.339717         +/- 0.1314       (38.69%)
```

Podemos ver esta regresión tiene una varianza residual del orden 10^3 y un error asintótico del 230.5 %, lo cual quiere decir que nuestro ajuste no es del todo preciso. En su defensa, la función poda depende mucho del caso a tratar, mejora la eficiencia del algoritmo pero no de manera estable, y por tanto hace incluso más difícil predecir el comportamiento del algoritmo ya que tiene un factor del azar intrínseco.

En conclusión, tenemos una curva que predice el comportamiento del algoritmo de manera no muy precisa pero ofrece una aproximación de este.

8.2.3. Funcion cota 3

A continuación, pasamos a estudiar la eficiencia del algoritmo BranchAndBound al utilizar la función de cota tercera. Nuevamente, deberemos estimar el número de nodos generados por una función y como ya sabemos que nuestra cota es cuadrática en el número de ciudades del problema, la eficiencia será de:

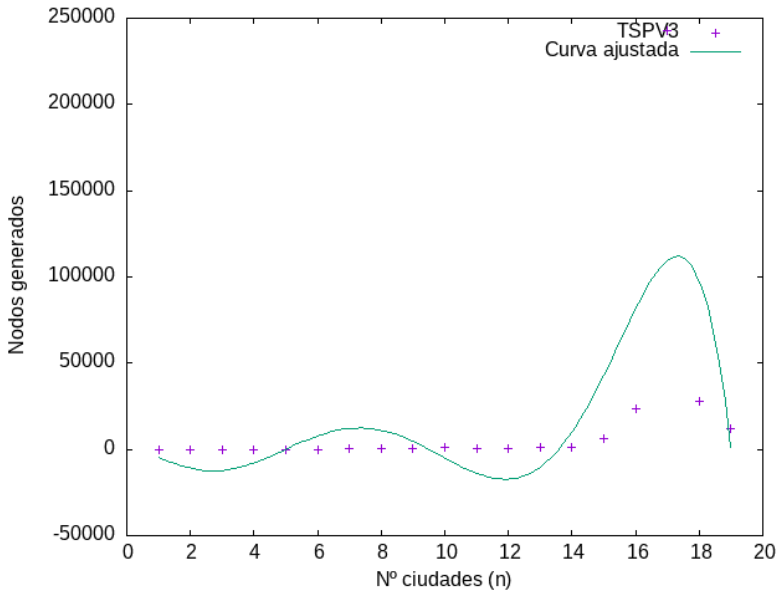
$$O(n^3 \cdot num\_nodos)$$

Al calcular el número de nodos, obtenemos:

Ciudades	Nodos generados
1	1
2	2
3	5
4	12
5	25
6	55
7	166
8	379
9	570
10	977
11	570
12	428
13	1290
14	1170
15	6398
16	23467
17	243090
18	27657
19	12016

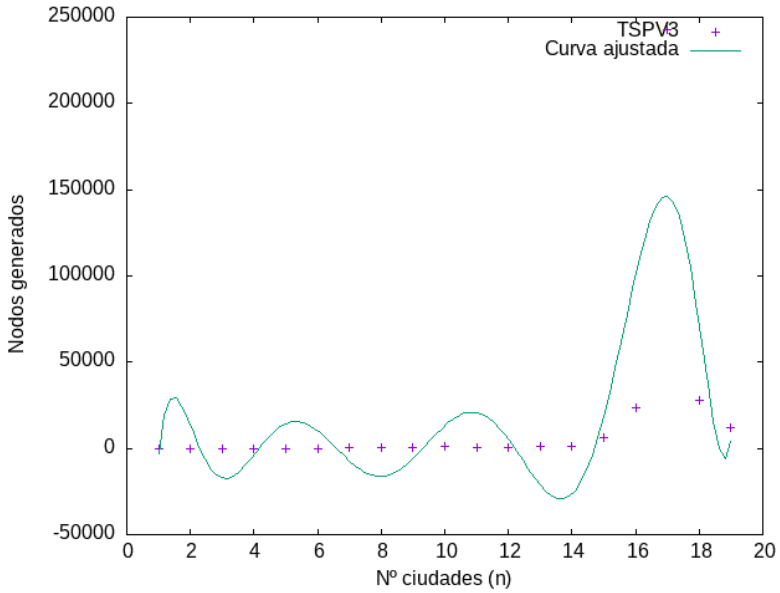
Tabla 2: Nodos generados para la cota 3 por el algoritmo branch & bound

Utilizaremos las mismas regresiones que en la sección 7.2.3, para ver, cuál de ellas tiene menor varianza residual:



$$f(x) = -0{,}616606 \cdot x^6 + 29{,}0489 \cdot x^5 - 480{,}284 \cdot x^4 + 3266{,}63 \cdot x^3 - 7569{,}32 \cdot x^2 + 1{,}26687 \cdot x + 1{,}02253$$

```
degrees of freedom    (FIT_NDF)                        : 12
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 49201.4
variance of residuals (reduced chisquare) = WSSR/ndf   : 2.42077e+09

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                  = -0.616606         +/- 0.9109       (147.7%)
b                  = 29.0489           +/- 54.81        (188.7%)
c                  = -480.284          +/- 1274         (265.3%)
d                  = 3266.63           +/- 1.437e+04    (439.9%)
e                  = -7569.32          +/- 8.059e+04    (1065%)
f                  = 1.26687           +/- 2.04e+05     (1.611e+07%)
g                  = 1.02253           +/- 1.708e+05    (1.671e+07%)
```



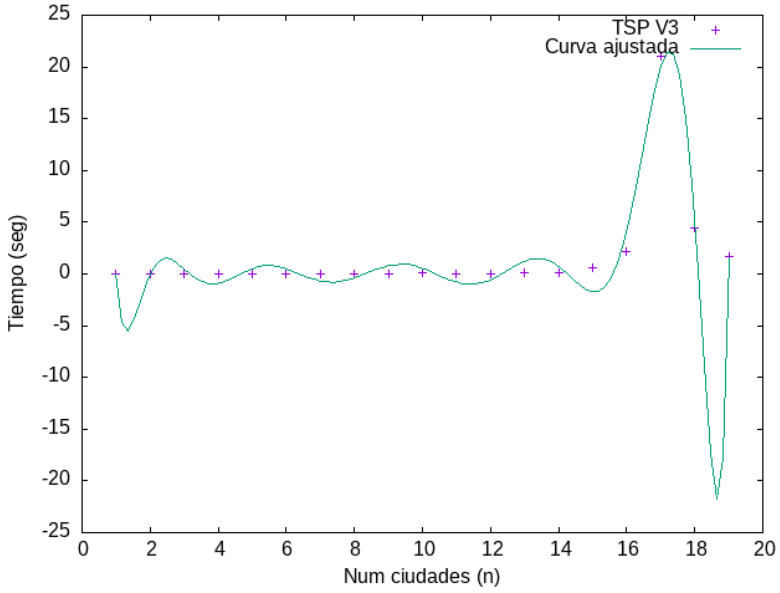
$f(x) = 0{,}01728 \cdot x^9 - 1{,}51841 \cdot x^8 + 56{,}1232 \cdot x^7 - 1135{,}67 \cdot x^6 + 13720{,}9 \cdot x^5 - 101300 \cdot x^4 + 449324 \cdot x^3 - 112930 \cdot x^2 + 1417440 \cdot x - 651607$

```
degrees of freedom    (FIT_NDF)                        : 9
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 47820.5
variance of residuals (reduced chisquare) = WSSR/ndf   : 2.2868e+09

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                  = 0.01728           +/- 0.01106      (63.98%)
b                  = -1.51841          +/- 0.9962       (65.61%)
c                  = 56.1232           +/- 38.01        (67.73%)
d                  = -1135.67          +/- 800.3        (70.47%)
e                  = 13720.9           +/- 1.015e+04    (73.95%)
f                  = -101300           +/- 7.935e+04    (78.33%)
g                  = 449324            +/- 3.766e+05    (83.82%)
h                  = -1.1293e+06       +/- 1.023e+06    (90.62%)
i                  = 1.41744e+06       +/- 1.4e+06      (98.79%)
j                  = -651607           +/- 7.025e+05    (107.8%)
```

Nuevamente, las gráficas no aproximan ni por asomo a los puntos, lo cual lo refleja el alto valor de la varianza residual, aunque de todas las regresiones que se han hecho son las dos con varianza residual más baja. Es por eso que nos quedaremos con estas regresiones, en particular con la mejor de ellas: n^9. Siguiendo este modelo podríamos decir que nuestro algoritmo branch & bound con la cota 3 tiene una eficiencia de $O(n^{12})$, aunque esto no tenga ningún sentido.

Aún así vamos a ajustar la gráfica de los tiempos por un polinomio de grado 12 y ver qué ocurre:

```
degrees of freedom    (FIT_NDF)                        : 6
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 1.65641
variance of residuals (reduced chisquare) = WSSR/ndf   : 2.74371

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 321.485          +/- 187.4        (58.3%)
a1              = -874.755         +/- 489          (55.91%)
a2              = 955.524          +/- 507.7        (53.14%)
a3              = -566.434         +/- 284.8        (50.28%)
a4              = 206.287          +/- 98           (47.51%)
a5              = -49.2475         +/- 22.11        (44.9%)
a6              = 7.9848           +/- 3.392        (42.49%)
a7              = -0.893475        +/- 0.3599       (40.28%)
a8              = 0.0689667        +/- 0.0264       (38.28%)
a9              = -0.00360276      +/- 0.001314     (36.47%)
a10             = 0.000121514      +/- 4.232e-05    (34.83%)
a11             = -2.38616e-06     +/- 7.956e-07    (33.34%)
a12             = 2.07075e-08      +/- 6.627e-09    (32%)
```



$f(x) = 2{,}07075 \cdot 10^{-8} \cdot x^{12} - 2{,}38616 \cdot 10^{-6} \cdot x^{11} + 0{,}000121514 \cdot x^{10} - 0{,}00360276 \cdot x^9 + 0{,}0689667 \cdot x^8 - 0{,}893475 \cdot x^7 + 7{,}9848 \cdot x^6 - 49{,}2475 \cdot x^5 + 206{,}287 \cdot x^4 - 566{,}434 \cdot x^3 + 955{,}524 \cdot x^2 - 874{,}755 \cdot x + 321{,}485$

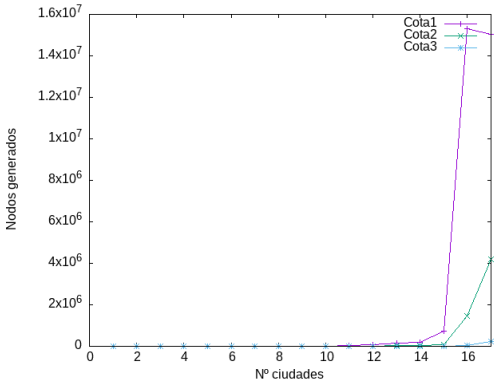
Aunque la varianza residual nos haya salido considerablemente menor, insistimos en que esta forma de modelizar el tiempo en función del número de ciudades no tiene mucho sentido al no seguir el número de nodos una monotonía respecto al número de ciudades.

9. Análisis comparativo del rendimiento

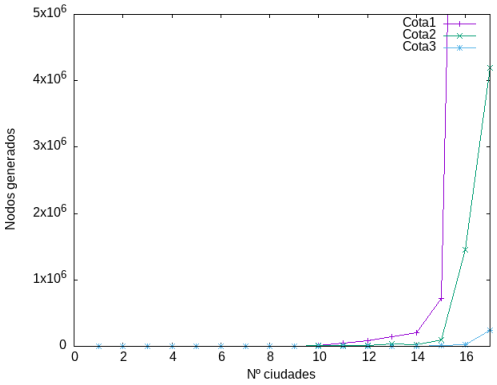
Vamos a comparar el rendimiento entre las tres funciones de cotas para distintos tamaños de entrada, entre 3 a 15-17-19, según la cota, debido a limitaciones HW. Para ello comenzaremos con la implementación de BB donde compararemos las tres funciones de cotas tanto por el número de nodos generados como por el tiempo de ejecución. Seguido, haremos lo mismo con backtraking. Y finalmente comparemos ambas implementaciones.

9.1. Análisis comparativo con Branch and Bound

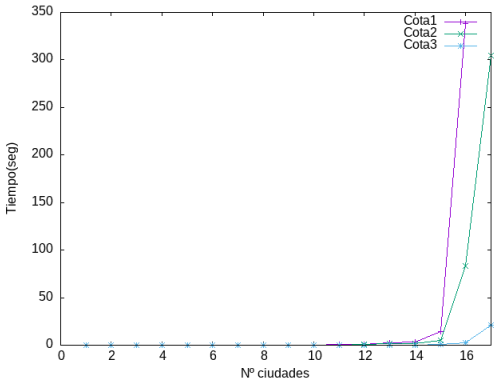
Ejecutando Branch and Bound utilizando las tres funciones de cotas obtenemos los siguientes datos:



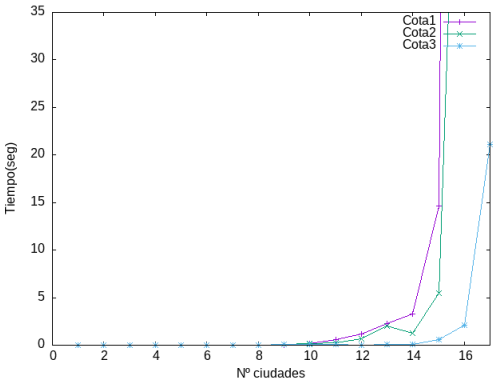
(a) Comparativa nodos generados.



(b) Nodos generados (escala aumentada).



(a) Tiempos de ejecución

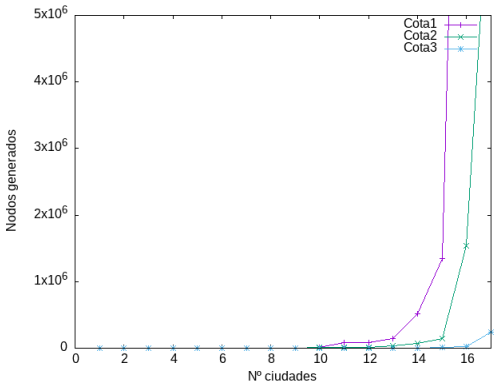


(b) Tiempo de ejución (escala aumentada).

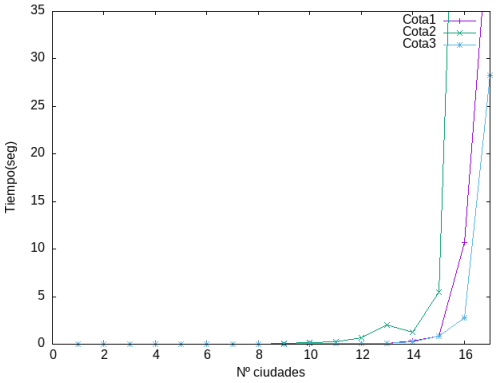
De los datos podemos ver que la función de cota3 es la mejor tanto en número de nodos generados como en tiempo de ejecución, seguido de la cota2. Por tanto, la mejor cota con la implementación de Branch and Bound es la cota 3.

9.2. Análisis comparativo con Backtraking

Ejecutando backtraking utilizando las tres funciones de cotas obtenemos los siguientes datos:



(a) Comparativa nodos generados.



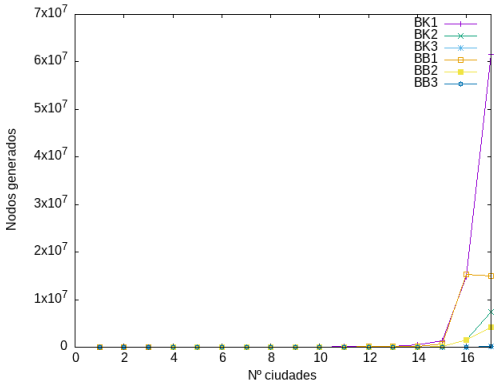
(b) Tiempo de ejecución.

Otra vez, vemos que la tercera cota es la más eficiente, tanto en número de nodos generados como en tiempo de ejecución, seguida de la primera cota. Podemos ver que esta vez la cota1 es mejor que la cota2 en tiempo de ejecución, a pesar de no serlo en número de nodos generados. Esto puede ser debido a que la cota2 tenga eficiencia cuadrádica mientras que la cota 1 tiene eficiencia constante. Notemos que el hecho de que la cota2 sea mejor que la cota1 en BB tiene sentido puesto que en este se puede elegir los caminos a seguir según la cota y aquí en BK está limitado a seguir el camino sistemático, es decir la cota solo puede influir en la poda pero no en la elección.

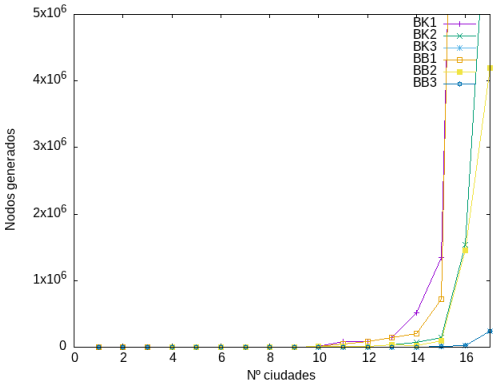
Por tanto, la mejor cota con la implementación de backtraking es la cota 3.

Comparación entre Branch and Bound y Backtraking

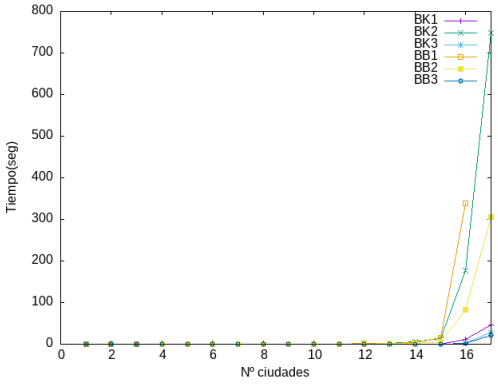
Aquí vamos a comparar todos los resultados:



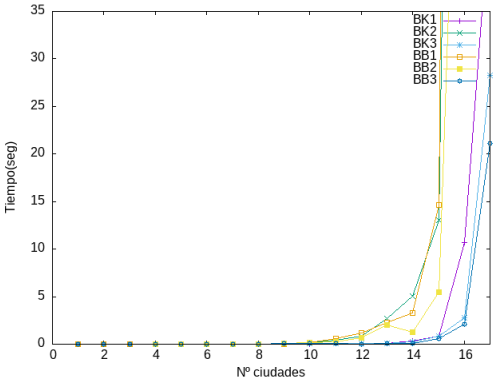
(a) Nodos generados



(b) Nodos generados (escala aumentada)

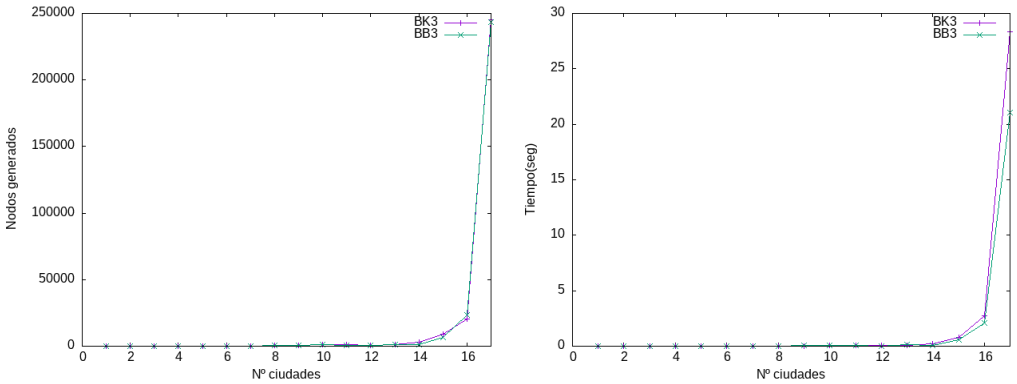


(a) Tiempo de ejecución.



(b) Tiempo de ejecución (escala aumentada).

Vemos claramente que las versiones de BB son generalmente más eficientes en tiempo de ejecución que las de BK, excepto en la cota1 que concuerda con lo que explicamos anteriormente. Por otra parte, es claro que la cota 3, ya sea en versión BB o en versión BK, genera menos nodos y tarda menos que cualquier otra función de cota. Veamos ahora más a fondo la función de cota 3:



(a) Nodos generados

(b) Tiempo de ejecución.

Vemos que el número de nodos generados en ambos casos es parecido y BB en tiempo de ejecución es ligeramente más rápido, en especial cuanto mas aumenta el tamaño (tal y como se explica en la definición del mismo) con lo cual podemos concluir que BB aunque tenga más exigencias de memoria y en principio su eficiencia en el caso peor sea peor que la de branch and bound (debido a la `priority_queue`) encuentra la solución óptima antes generalmente.

Conclusión de la comparativa

Vemos claramente que en todas las comparativas se mantiene en la mayoría de los casos una correlación entre al número de nodos generados y el tiempo de ejecución, la cota que menos nodos genera suele resultar también en ser la más eficiente. Notemos que aunque se realicen más cómputo por nodo, empeorando así la eficiencia teórica, para podar más ramas, puede resultar rentable (en comparación con un número inicial de nodos del orden factorial). Esto se refleja en el hecho de que la cota1 aun siendo del orden $O(1)$ y la cota 3 del orden $O(n^2)$, esta última, que es la que menos nodos genera, es la mejor y la primera, que la que más nodos genera, es la peor. El caso de la cota1 y cota2 es un caso particular.

Cabe mencionar también el comportamiento irregular que obtenemos en el número de nodos generados y en el tiempo de ejecución según la instancia, esto, como mencionamos anteriormente, es porque hay un factor del azar intrínseco en nuestro algoritmo. Podemos perfectamente encontrar la solución buscada en la primera rama que recorramos como encontrarla en la última rama. Así, no nos debe extrañar los picos de las gráficas, eso sí, sí que podemos ver que todos siguen un comportamiento esencialmente creciente.

10. Conclusiones

En el transcurso de esta práctica, hemos logrado una comprensión profunda y una implementación eficaz de técnicas algorítmicas avanzadas, específicamente Backtracking y Branch and Bound, aplicadas a la resolución del problema del viajante de comercio. Esta experiencia ha complementado y extendido nuestros conocimientos teóricos, permitiéndonos enfrentar y superar retos complejos mediante la exploración sistemática de grafos.

La práctica nos ha permitido no solo diseñar e implementar algoritmos utilizando estas técnicas, sino también realizar un análisis detallado de su eficiencia. Hemos evaluado la eficacia de diversas funciones de cota, observando cómo impactan en el rendimiento de los algoritmos en términos de tiempo de ejecución y número de nodos generados. Este análisis comparativo ha sido crucial para entender las fortalezas y debilidades de cada enfoque en diferentes escenarios, reforzando la importancia de una correcta selección y validación de las funciones de cota.

Durante el desarrollo de la práctica, se han manifestado conceptos teóricos clave en la implementación y optimización de los algoritmos, tales como:

- La importancia de diseñar funciones de cota efectivas que mejoren significativamente la eficiencia de los algoritmos de Backtracking y Branch and Bound.
- La observación de que, aunque las diferencias en hardware y software pueden afectar los tiempos de ejecución absolutos, la eficiencia relativa entre diferentes algoritmos y enfoques permanece consistente, resaltando la robustez de nuestras comparaciones.

Además de los aspectos técnicos, la práctica ha fomentado el desarrollo de habilidades blandas esenciales. El trabajo en equipo ha sido fundamental, promoviendo la colaboración, el intercambio de ideas y la distribución equitativa de tareas. Estas competencias transversales, junto con el pensamiento crítico y la comunicación efectiva, han enriquecido nuestra experiencia, preparándonos mejor para futuros desafíos tanto en el ámbito académico como profesional.

Finalmente, la documentación exhaustiva de nuestro trabajo, que incluye el diseño, implementación y análisis de los algoritmos, ha sido un ejercicio valioso en comunicación técnica. Esta memoria no solo facilita la evaluación por parte de los profesores, sino que también sirve como un recurso de referencia para futuros proyectos.

En resumen, esta práctica ha sido una oportunidad invaluable para aplicar y profundizar en técnicas algorítmicas avanzadas, desarrollar habilidades analíticas y colaborativas, y reforzar nuestra preparación para enfrentar problemas complejos en el campo de la ciencia de la computación. Consideramos que esta experiencia ha sido altamente enriquecedora, proporcionando herramientas y perspectivas cruciales para nuestro desarrollo académico y profesional.