Memoria Práctica 3: Algoritmos Greedy

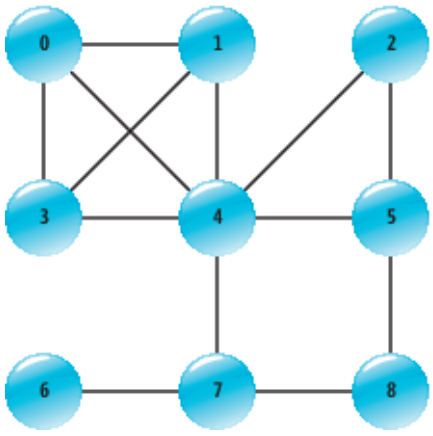
Laura Mandow Fuentes
Chengcheng Liu
Daniel Hidalgo Chica
Roberto González Lugo
Elías Monge Sánchez

Abril 2024

Titulación: Doble Grado de Matemáticas e Ingeniería Informática
Curso: 2º, 2º Cuatrimestre
Asignatura: Algorítmica

Correos:

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



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:

- P1
  - 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
- P2
  - 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 (Oracle VM VirtualBox, 2 cores)
- P3
  - Máquina: HP Laptop 15s-eq1xxx
  - Procesador:AMD Ryzen 5 4500U with Radeon Graphics
  - Sistema Operativo: Ubuntu 22.04 64 bits
- P4
  - V1
    - Máquina: HP Laptop 15s-eq1xxx
    - Procesador:AMD Ryzen 5 4500U with Radeon Graphics
    - Sistema Operativo: Ubuntu 22.04 64 bits
  - V2
    - 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
  - V3
    - Máquina: HP Laptop 15s-eq1xxx
    - Procesador:AMD Ryzen 5 4500U with Radeon Graphics
    - Sistema Operativo: Ubuntu 22.04 64 bits

2. Objetivos

Los objetivos de esta práctica se enfocan en la **comprensión y asimilación profunda** de la técnica de diseño de algoritmos voraces (*greedy algorithms*), aplicada a la resolución de problemas complejos mediante un enfoque sistemático y estratégico. Esta técnica se caracteriza por tomar decisiones secuenciales que parecen óptimas en el momento, con el fin de alcanzar una solución global eficiente. A través de esta metodología, los estudiantes aprenderán a descomponer los problemas y aplicar la estrategia voraz para lograr soluciones exactas o aproximadas.

Además, se promoverá el desarrollo de **heurísticas y algoritmos aproximativos** basados en la filosofía de los métodos voraces, especialmente útiles en situaciones donde no se conocen soluciones eficientes. Esta práctica, realizada en grupos, también fomentará habilidades de trabajo colaborativo y de comunicación efectiva, permitiendo el intercambio de ideas y la adaptación de las soluciones a diferentes contextos y tipos de problemas.

Por lo tanto, nuestros objetivos no solo incluyen el diseño e implementación de algoritmos siguiendo las directrices específicas, sino también una **profunda comprensión teórica** de los principios subyacentes a los algoritmos voraces. Buscamos desarrollar competencias técnicas en la solución de problemas y habilidades transversales que potencien la formación académica y profesional de los estudiantes en la ciencia de la computación.

En conclusión, los objetivos de nuestra práctica abarcan el **dominio de la técnica de algoritmos voraces** para el diseño y desarrollo de soluciones a problemas complejos, la capacidad para implementar soluciones efectivas y la habilidad para trabajar en equipo y comunicarse de manera efectiva, todo dentro de un marco que complementa y enriquece su formación técnica en la disciplina.

3. P1 Problema del hijo predilecto

3.1. Diseño del algoritmo

3.1.1. Descripción del problema

Un individuo desea repartir sus N bienes entre sus 2 hijos. Para cada bien se conoce el valor (positivo) del mismo. Previo al fallecimiento del individuo, el juzgado, de una forma ciega, determina un valor k, que nos indica el número de bienes que se le deben asignar a un hijo (al otro se le asignan N k). Conocido estos datos, el individuo desea distribuir los bienes entre sus hijos de forma que uno de ellos salga lo mas beneficiado posible. El beneficio obtenido por cada hijo se define como la suma de los beneficios de los bienes que se les lega.

3.1.2. Algoritmo Greedy

Proponemos una solución al problema que consiste en, una vez ordenados los bienes según su valor (de menor a mayor), se elije si el hijo predilecto tendrá k o $N-k$ bienes ¿($k \leq \frac{N}{2}$)?, y en función de eso se rellenan los vectores del hijo predilecto y del otro, primero el del otro hijo, y después el del predilecto. Esto es debido a que los bienes están ordenados de menor a mayor (en un set)

3.1.3. Componentes y demostración de validez

Las componentes del algoritmo greedy son:

- **Previo:** De aquí en adelante, si $k \leq \frac{N}{2}$, el hijo predilecto recibirá $N-k$ bienes, sino, recibirá k bienes. A este tamaño, lo llamaremos tam
- **Conjunto candidatos:** Los diferentes valores de los bienes del problema
- **Conjunto seleccionados:** Los tam bienes del hijo predilecto
- **Función selección:** En cada paso, se añade el bien con mayor valor

A continuación, demostraremos la validez del algoritmo Greedy por inducción:

- **Caso base:** (Red abs) Consideramos que tenemos un conjunto solución B que no incluye al bien con valor máximo (Sea B_m). El valor total de los bienes será $V(B)$ Introducimos B_m en la solución, y quitamos cualquier otro elemento b_n, entonces, el valor de este nuevo conjunto es $V(B)+V(B_m)-V(b_n)$, pero como $V(B_m) > V(b_n) \implies V(B)+V(B_m)-V(b_n) > V(B)$ CONTRADICCIÓN
- **Paso de inducción:** Consideramos un conjunto solución B' que incluye a $\{b_1, ...b_{m-1}\}$ y no incluye a b_m (Siendo $V(b_m) > V(b_n)$ para algún $n \in \{1, ...(m-1)\}$, entonces, introduciendo b_m en el conjunto y eliminando b_n, tenemos $V(B')+V(b_m)-v(b_n)$, y como $V(b_m) > V(b_n) \implies V(B')+V(b_m)-v(b_n) > V(B')$ CONTRADICCIÓN.

Queda así demostrada la validez de nuestro algoritmo.

- **NOTA:** Nuestro algoritmo rellena primero el hijo NO predilecto, debido a que hemos elegido ordenar el set por defecto (de menor a mayor), esto no afecta a nuestra demostración, pues es equivalente a ordenarlo de mayor a menor y rellenar primero el hijo predilecto, es una mera cuestión de estilo en el código.

3.2. Implementación y estudio de eficiencias

3.2.1. Datos de entrada

El formato de los datos de entrada para este problema será simple. En primer lugar, recibe la cantidad de bienes con los que se va a trabajar, es decir, nuestra N. En segundo lugar (en la siguiente linea), recibe el valor de la k, es decir, la cantidad de bienes que tendrá un hijo. Por último (en la siguiente linea), recibe, separados por espacios, el valor de todos los N bienes con los que trabajará.

Aquí un ejemplo de entrada para este algoritmo:

```
10
5
61  24  21  81  32  39  24  55  89  30
```

3.2.2. Implementación del algoritmo

A continuación mostramos la implementación del algoritmo desarrollado, es bastante directa de comprender debido a que se trata de un problema relativamente sencillo.

- Primeramente, se comprueban los argumentos y se abren los archivos, se prepara todo para poder leer los datos directamente en orden (añaadiendolos a un set)

```
int main (int argc, char *argv[]) {
    #include <fstream> // Include the necessary header file for std::ifstream
    #include <ios> // Include the necessary header file for std::ios::in

    if (argc < 2) {
        std::cerr << "Error: faltan parametros\n";
        std::cerr << "<nombre_fichero_entrada>" << std::endl;
        return -1;
    }

    std::ifstream fin(argv[1]);
    if (!fin) {
        std::cerr << "Error: no se pudo abrir el archivo " << argv[1] << std::endl
    ;
        return -1;
    }

    clock_t tStart = clock();
```

- A continuación, la lectura de datos y el algoritmo en sí

```
    int N, k;
    fin >> N >> k;
    std::set <int> values; // Set to store the values of the goods
    for (int i = 0; i < N; ++i) {
        int n;
        fin >> n;
        values.insert(n);
    }
    //Ya los tenemos ordenados de menor a mayor, a continuación, elegimos el
    vector que será del hijo predilecto (k<=N/2?)

    std::vector <int> notPreSon;
    std::vector <int> preSon;

    if (k<=N/2){ //Si k es menor, el hijo predilecto tiene n-k elementos
        for (int i = 0; i < k; ++i){
            notPreSon.push_back(*values.begin());
            values.erase(values.begin());
        }
        while (!values.empty()){
            preSon.push_back(*values.begin());
            values.erase(values.begin());
        }
    }
    else{
        for (int i = 0; i < N-k; ++i){
            notPreSon.push_back(*values.begin());
            values.erase(values.begin());
        }
        while (!values.empty()){
            preSon.push_back(*values.begin());
            values.erase(values.begin());
        }
    }
```

- Teniendo esto, el problema queda resuelto, el resto del código es simplemente contar el tiempo que se ha tardado e imprimir los resultados

```
    clock_t tEnd = clock();
    double elapsed = double(tEnd - tStart) / CLOCKS_PER_SEC;


    //Imprimimos los valores
    if (DEBUG_PRINT){
        std::cout << N << " " << k << std::endl;
        int sumaNPS = 0, sumaPS = 0;
        for (int i = 0; i < notPreSon.size(); ++i){
            std::cout << notPreSon[i] << " ";
            sumaNPS += notPreSon[i];
        }
        cout << endl << "Suma no predilecto: " << sumaNPS << endl;
        std::cout << std::endl << "PRESON" << std::endl;
        for (int i = 0; i < preSon.size(); ++i){
            std::cout << preSon[i] << " ";
            sumaPS += preSon[i];
        }
        std::cout << std::endl << "Suma predilecto: " << sumaPS << std::endl;
    }
    else{
        std::cout << N << " " <<  elapsed << std::endl;
    }
```

3.2.3. Estudio de eficiencia

Eficiencia teórica

Estudiemos la eficiencia teórica. En primer lugar, el insertado en un set (que incluye la ordenación) es de orden $O(logn)$. Como esto se realiza n veces, nos queda, por ahora, $O(nlogn)$.
A continuación entramos en el primer if, la condición es $O(1)$ así que podemos ignorarla. Distingamos casos:

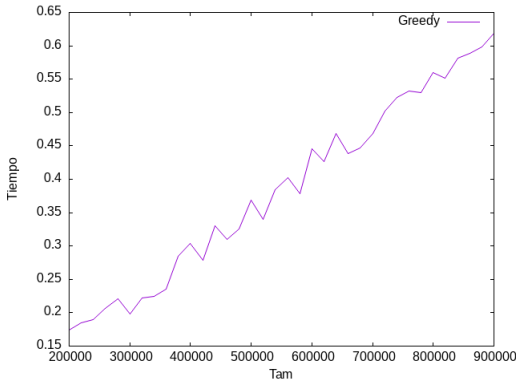
- Si entra al if: El for se ejecutará k veces, haciendo una inserción en un vector, y un borrado del set, lo cual es $O(1)$ y $O(logn)$ respectivamente. Por ahora, dentro de este if, tendremos un $O(klogn)$ A continuación entra en el $while$, donde ejecutará $N-k$ veces, y por cada iteración le cuesta lo mismo que al if, así que en total tenemos $O(klogn) + O((N-k)logn) = O(nlogn)$

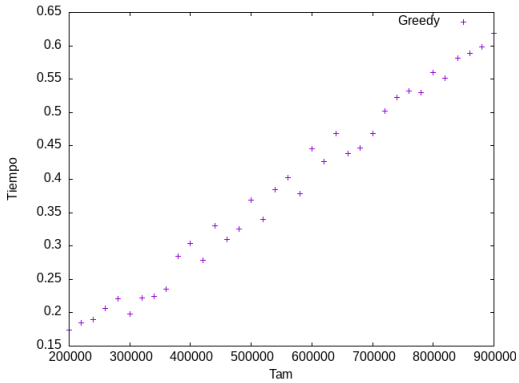
- Si no entra al if: realmente pasará lo mismo que si hubiera entrado, ya que el código es el mismo pero en el orden contrario, quedando así $O((N-k)logn) + O(klogn) = O(nlogn)$

Si juntamos esto con el $O(nlogn)$ que teníamos ya, nos queda: $O(nlogn + nlogn) = O(2nlogn) = O(nlogn)$

Eficiencia empírica e híbrida

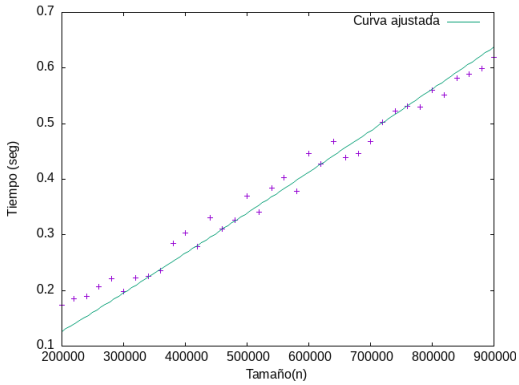
En cuanto a la eficiencia empírica, ejecutando casos desde 200000 y hasta 900000 con saltos de 20000, nos queda la siguiente gráfica de tiempos:





Como podemos ver, aunque haya leves fluctuaciones podemos ver que tiene un crecimiento casi lineal, y si lo ajustamos con una curva de regresión del tipo $a_0 n \log n$ obtenemos los siguientes resultados:

```
iter      chisq       delta/lim  lambda   a
   0 2.2362560812e+15   0.00e+00  7.88e+06    1.000000e+00
   1 1.6334960418e+12  -1.37e+08  7.88e+05    2.702708e-02
   2 1.2597138746e+05  -1.30e+12  7.88e+04    7.557128e-06
   3 2.0112930936e-02  -6.26e+11  7.88e+03    5.172693e-08
   4 2.0111958940e-02  -4.83e+00  7.88e+02    5.170608e-08
   5 2.0111958940e-02  -1.73e-11  7.88e+01    5.170608e-08
iter      chisq       delta/lim  lambda   a

After 5 iterations the fit converged.
final sum of squares of residuals : 0.020112
rel. change during last iteration : -1.72507e-16

degrees of freedom    (FIT_NDF)                        : 35
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0239714
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.000574627

Final set of parameters            Asymptotic Standard Error
=======================            =========================
a               = 5.17061e-08      +/- 5.069e-10    (0.9804%)
```



Como podemos ver con este ajuste la varianza residual es prácticamente nulo y concuerda con los resultados teóricos, pudiendo así concluir que efectivamente nuestro algoritmo es del orden $O(n \log n)$.

4. P2 Problema de asignación de aulas

4.1. Diseño del algoritmo

4.1.1. Descripción del problema

En una escuela se tiene que programar n exámenes en un día determinado, y para cada examen, disponemos de su tiempo de inicio y su duración. El centro cuenta con m aulas disponibles, donde m es mayor que N, asegurando así la posibilidad de realizar todos los exámenes incluso en el peor caso. Pero para cada aula reservada se necesita contratar a un vigilante para supervisar dicha aula. Por tanto, se nos pide diseñar un algoritmo que permita garantizar que los exámenes se realicen con el menor costo posible para la escuela, es decir, reservando el menor número de aulas posibles.

4.1.2. Algoritmo greedy

Proponemos una solución al problema que consiste en, una vez ordenados los exámenes según orden de inicio, ir reservando primero aulas para los exámenes que más pronto empiecen. Teniendo en cuenta de que si para el inicio de un examen existe una aula de las ya reservadas que esté libre, entonces se usa dicha aula, en caso contrario, se reserva otra nueva. Notemos que al ir reservando aulas primero para los exámenes que más pronto empiecen, a la hora de reservar aulas para un determinado examen, no va a haber exámenes para los cuales ya se ha reservado aula que empiece después del dicho examen.

Este algoritmo es claramente greedy por el hecho de que toma decisiones locales para solucionar el problema completo y además en cada desión procura reservar el menor número de aulas posible.

4.1.3. Componentes y demostración de validez

Veamos a ver los aspectos que hay que tener en cuenta en el diseño del algoritmo voráz:

- **Conjunto candidato:** Todas las aulas disponibles. Dentro de todas las aulas, seleccionaremos un número determinado de aulas.
- **Conjunto seleccionado:** Las aulas seleccionadas que van ser usadas para realizar los exámenes, que también conforman la solución al problema a resolver.
- **Función solución:** Para que el conjunto seleccionado sea una solución al problema, este debe poder garantizar que se pueda realizar todos los exámenes en su tiempo establecido, así, la función solución debe comprobar que si con las aulas seleccionadas se han podido programar todos los exámenes, independientemente de que sea el óptimo o no.
- **Función selector:** La función selector seleccionará una aula para un examen según el criterio de nuestro algoritmo.
- **Función de factibilidad:** La función de factibilidad comprobará si la aula seleccionada para el examen es compatible con el resto de exámenes ya seleccionadas, es decir, que no haya dos exámenes que se vayan a realizarse en una misma aula en un mismo tiempo, independientemente de que sea el óptimo o no.
- **Función objetivo:** La función objetivo debe indicar el número de aulas seleccionadas, así como la programación de los exámenes.

Ahora vamos a demostrar por inducción que nuestro algoritmo encuentra el número mínimo de aulas necesarias (m aulas) para realizar todos los exámenes (n examenes), con $m \leq n$. Veamos:

- **CASO BASE: (n = 1):** Nuestro algoritmo seleccionaría el primer examen que empieza y es claro que como tenemos uno solo, y este se tiene que realizar, necesitamos reservar un aula. Por tanto, es obvio que este es el óptimo y coincide con la elección de nuestro algoritmo.
- **PASO DE INDUCCIÓN:** Supongamos que nuestro algoritmo para los n exámenes que empiezan primero ha reservado m aulas, y este es el óptimo por hipótesis de inducción. Supongamos ahora que se necesita hacer un examen más (el n+1-ésimo), que es el siguiente según el tiempo

de comienzo, entonces según nuestro algoritmo:

- Si alguna de las aulas reservadas está ya libre para la hora de comienzo del examen n+1-ésimo, entonces realizaremos en esta aula el dicho examen. Por tanto, habremos reservado hasta el momento m aulas que era el óptimo para n éxamenes, luego al no aumentar el número de aulas, es claro que este es óptimo también.
- Si todas las aulas reservadas hasta el momento del comienzo del examen n+1-ésimo están ocupadas, entonces nuestro algoritmo reservará otra aula. Por tanto, tendremos m+1 aulas reservadas. Por reducción al absurdo probemos que este es el número óptimo.
  Si existiese otra solución que reserve menos aulas para n+1 exámenes siendo $\bar{m}$ el número de aulas que se reserva, entonces tenemos que $\bar{m} < m + 1$. Pero como para el comienzo del examen n+1-ésimo todas las m aulas estaban ocupadas, deducimos que había m exámenes realizándose, por tanto, para el inicio del n+1-ésimo habrá m+1 exámenes en proceso. Entonces, como la solución óptima debe permitir que se realicen todos los exámenes, en particular al comienzo del examen n+1-ésimo, entonces $m + 1 \leq \bar{m}$. Juntando las desigualdades tenemos $m + 1 \leq \bar{m} < m + 1$, contradicción. Luego m + 1 aulas era el óptimo para este caso.

Y así, concluimos con que nuestro algoritmo nos da el mínimo número de aulas a reservar para realizar n exámenes, siendo n un número natural arbitrario, es decir, para cualquier número número de exámenes nuestro algoritmo nos da el óptimo número de aulas.

4.2. Implementación y estudio de eficiencias:

4.2.1. Datos de entrada y evaluación de validez:

Formato datos entrada:
Como entrada necesitaremos el número de exámenes a planificar así como la información relativa a cada examen, es decir, la hora de inicio y la duración de cada uno. Para esto último hemos decidido que el formato de la hora de inicio será de la forma horas:minutos (ej. 15:30) y la duración será expresado en minutos. A continuación mostraremos un ejemplo:

```
4
6:00 30
6:10 30
6:30 20
6:40 20
```

4.2.2. Implementación:

A continuación mostramos la implementación del algoritmo descrito anteriormente:
En primer lugar tenemos una estructura $info\_exam$ para gestionar la información relativa a un examen así como conversión entre formatos de tiempo y E/S. Es fácil ver que todos los métodos implementados son de orden constante, es decir, O(1).

```
1
2  using namespace std;
3
4  /**
5   * @brief Estructura que permite gestionar la información relativa a un examen
6  */
7  struct info_exam {
8
9      int start_time;      // [0-minutos_del_día[ , indica en qué minuto
10                          // del día empieza el examen
11     int duration;        // duración del examen
12
13
14     // métodos
15
```

```
    // constructores
    info_exam(int hour=0, int minutes=0, int duration=0){
        set_info_exam(hour, minutes, duration);
    }

    info_exam(int minutes, int duration): duration(duration), start_time(minutes) {}

    /**
     * @brief Modifica la información del examen
     * @param start_time Tiempo de inicio del examen (mínuto del día)
     * @param duration Duración del examen
    */
    void set_info_exam(int start_time, int duration) {
        this->start_time = start_time;
        this->duration = duration;
    }

    /**
     * @brief Modifica la información del examen dado la hora de inicio
     * en formato usual
     * @param h hora del día en que empieza el examen
     * @param m minuto de la hora en que empieza
     * @param duration Duración del examen
    */

    void set_info_exam(int h, int m, int duration) {
        this->start_time = hour2minFormat(h, m);
        this->duration = duration;
    }

    /**
     * @brief Devuelve la hora de finalización del examen en formato "mínuto del día"
    */
    int get_finish_time(void) const {
        return start_time + duration;
    }

    /**
     * @brief Devuelve por referencia la hora de finalición del examen en
     * formato usual
     * @param h hora en que termina el examen
     * @param m minuto de la hora en que termina el examen
    */
    void get_finish_time(int &h, int &m) const{
        int aux = start_time + duration;
        min2hourFormat(aux, h, m);
    }

    /**
     * @brief Indica si el examen @p e puede realizarse/tiene lugar tras
     * la finalización examen implícito
     * @return true si se puede, false en caso contrario
    */

    bool canGoAfter(info_exam e) {
        return get_finish_time() <= e.start_time;
    }

    /**
     * @brief Operador relacional que compara si la hora de inicio del
     * examen @p e es posterior al del implícito
    */

    bool operator<(info_exam e) {
        return this->start_time < e.start_time;
    }


    private:

    /**
```

```
     * @brief Dado los argumentos, @p hour , @p minutes retorna
     * la hora que indican en formato "minutos del día"
     * @param hour la hora de la hora que indica
     * @param minutes los minutos de la hora que indica
     * @return la equivalencia de la hora indicada en formato "mínutos del día"
    */

    int hour2minFormat(int hour, int minutes) const{
        return hour*60 + minutes;
    }

    /**
     * @brief Dado una hora en formato "mínutos del día", retorna
     * por referencia su equivalente en formato usual
     * @param minutes la hora expresada en formato "mínutos del día"
     * @param h la hora de la hora que indica
     * @param m los minutos de la hora que indica
    */
    void min2hourFormat(int minutes, int& h, int& m) const{
        h = minutes/60;
        m = minutes%60;
    }

    public:

    /**
     * @brief Operador de entrada para una estructura info_exam, lee en
     * formato "hora:minutos"
     * @param inputStream el flujo de entrada
     * @param e objeto info_exam a leer
     * @return referencia al flujo de entrada
    */

    friend std::istream& operator>>(std::istream& inputStream, info_exam& e){

        int h, m, duration;
        char c;
        inputStream >> h >> c >> m >> duration;

        e.set_info_exam(h, m, duration);

        return inputStream;
    }

    /**
     * @brief Operador de salida para una estructura info_exam, imprime
     * en formato hora:minutos
     * @param outputStream flujo de salida
     * @param e estructura info_exam a imprimir
     * @return referencia al flujo de salida
    */
    friend ostream& operator<<(ostream& outputStream, const info_exam& e){

        int h, m;
        e.min2hourFormat(e.start_time, h,m);
        outputStream << "[" << h << ":" << setfill('0') << setw(2) << right << m << ",
    " << e.duration << " min]";
        return outputStream;
    }

};
```

A continuación mostramos el algoritmo en sí:

```

/**
 * @brief Algoritmo greedy que obtiene dado unos exámenes obtiene la programación ó
    ptima
 * para utilizar el mínimo número de aulas
 * @param all_exams representa los exámenes que se van a realizar
 * @param used_classroom estructura de dato que va a almacenar la programación
    completa
```

```
 * de los exámenes y sus aulas correspondientes
*/

void greedy(vector<info_exam> & all_exams, vector<list<info_exam>>& used_classroom) {

    // Ordena los examenes según tiempo de inicio

    sort(all_exams.begin(), all_exams.end());

    // (finish_time, classIndex)
    priority_queue <pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>>
    examenes_en_proceso;

    // Asigna a cada aula un examen --> función solución
    // hasta que no se asigne una aula a cada examen no es una solución

    for (int i = 0; i < all_exams.size(); ++i) {

        bool couldFoundAClass = false;

        // Función selector: primero busca una aula libre entre las ya reservadas
        // Función de factibilidad (implícita): si es compatible colocarlo en una
    determinada aula

        // Pregunta al primero que se queda libre

        if (!examenes_en_proceso.empty() && examenes_en_proceso.top().first <=
    all_exams[i].start_time) {

            couldFoundAClass = true;

            // [finish_time, classIndex]
            pair<int, int> examen_acabado = examenes_en_proceso.top();
            examenes_en_proceso.pop();

            // O(nlogn)
            used_classroom[examen_acabado.second].push_back(all_exams[i]);
            examenes_en_proceso.emplace(all_exams[i].get_finish_time(), examen_acabado
    .second);

        }


        // Si no encuentra, reserva una nueva
        if (!couldFoundAClass) {
            list<info_exam> new_classroom;
            new_classroom.push_back(all_exams[i]);
            used_classroom.push_back(new_classroom);

            // Un examen más que se está realizandose
            examenes_en_proceso.emplace(all_exams[i].get_finish_time(),used_classroom.
    size()-1);
        }
    }
}
```

Como dijimos, el algoritmo en primer lugar ordena el vector y seguido selecciona para cada examen un aula según el criterio establecido. Para ello, mantiene un priority_queue o heap donde guarda récord del último examen que se va a realizar en cada aula en el momento, y en el tope está el examen que acaba más proto, facilitando así la elección de aulas.Si el del tope no ha acabado, el resto tampoco habrán acabado, por tanto hay que reservar nuevas aulas. Sin embargo, si el del tope ha acabado entonces ya disponemos de un aula y no hay que reservar más aulas.

En el main podemos ver la función objetivo que imprime la programción de los exámenes por completo si está definida la macro PROGRAMACION e imprime solo el número de aulas necesarias si está definida la macro NUM_AULAS:

```
int main(int argc, char* argv[])
{

    // Lectura de datos: Número de exámenes
```

```
    // y los datos relativos a los exámenes

    int n;

    std::cin >> n;

    info_exam aux;
    vector<info_exam> all_exams;

    for (int i=0; i < n; i++) {
        std::cin >> aux;
        all_exams.push_back(aux);
    }

    vector<list<info_exam>> solution;

    greedy(all_exams, solution);

    // Función objetivo: Muestra la programación completa o el número de aulas según
    lo pedido

    #ifdef PROGRAMACION
    for (int i=0; i < solution.size(); ++i) {
        cout << "Classroom " << i << " : " << endl;
        for (typename list<info_exam>::iterator  it = solution[i].begin(); it !=
    solution[i].end(); ++it) {
            cout << *it << " ";
        }

        cout << endl;
    }
    #endif

    #ifdef NUM_AULAS
    cout << "Number of classrooms: " << solution.size() << endl;
    #endif

    return 0;

}
```

Un ejemplo de la ejecución con opción PROGRAMACIÓN sería:

```
Entrada:
4
6:00 30
6:10 30
6:30 20
6:40 20

Salida:
Classroom 0 :
[6:00,30 min] [6:30,20 min]
Classroom 1 :
[6:10,30 min] [6:40,20 min]
```

4.2.3. Estudio de eficiencia:

Eficiencia teórica:

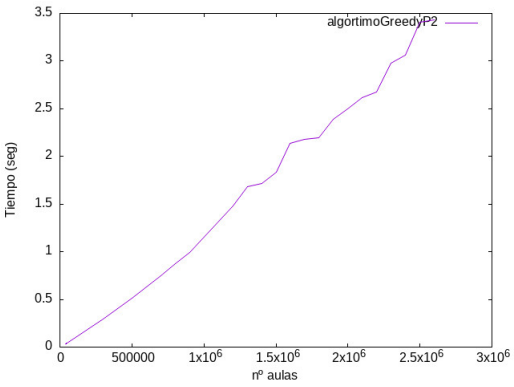
Estudiemos la eficiencia teórica. En primer lugar, la ordenación es del orden $O(n \log n)$. Seguido, en el bucle se realizan n iteraciones, siendo n el número de exámenes. Ahora, cuando se busca una potencial aula vacía de las ya reservadas solo tenemos que consultar el tope del heap, puesto que ese será el primer examen que acabe dentro de los que se están realizándose. Teniendo en cuenta que en

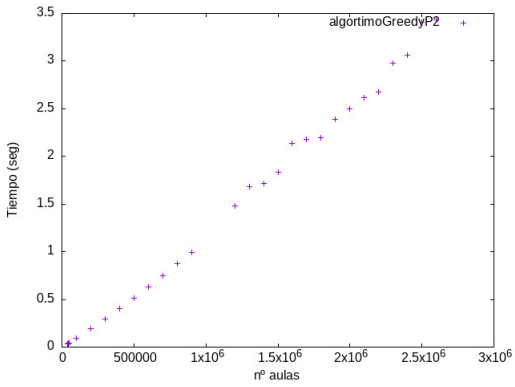
el peor de los casos el heap puede tener i exámenes en cada iteración y la inserción en un heap es del orden logarítmico, concluimos que el orden del bucle for es $O(n \log n)$, ya que el resto de sentencias en ambos condicionales son operaciones simples de orden $O(1)$. Por tanto, aplicando la regla de la suma y del máximo la eficiencia teórica del algoritmo es $O(n \log n)$.

Notemos que este algoritmo obtiene la programación completa de los exámenes, pero eso es equivalente a obtener el número de aulas mínimo, puesto que basta obtener el tamaño del vector solución. Propondremos otra alternativa más eficiente teóricamente pero que no nos ofrece una información tan completa como el estudiado en esta sección.

Eficiencia empírica e híbida:

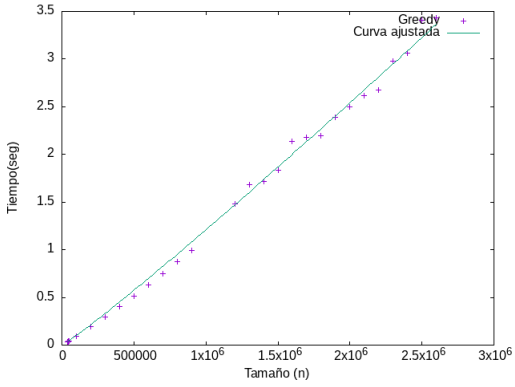
En cuanto a la eficiencia empírica, ejecutando casos de 37000-2600000 exámenes con el formato descrito anteriormente, obtenemos las siguientes gráficas:





Como podemos ver, aunque haya leves fluctuaciones podemos ver que tiene un crecimiento casi lineal, y si lo ajustamos con una curva de regresión del tipo $a_0 n \log n$ obtemos los siguientes resultados:

```
Thu Apr 25 09:51:40 2024


FIT:    data read from datos1
        format = z
        #datapoints = 37
        residuals are weighted equally (unit weight)

function used for fitting: f(x)
        f(x)=a0*x*log(x)
fitted parameters initialized with current variable values

iter      chisq       delta/lim  lambda   a0
   0 1.2544019845e+16   0.00e+00  1.84e+07    1.000000e+00
   5 1.3063857014e-01  -1.06e-10  1.84e+02    8.758471e-08

After 5 iterations the fit converged.
final sum of squares of residuals : 0.130639
rel. change during last iteration : -1.0623e-15

degrees of freedom    (FIT_NDF)                      : 36
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0602399
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00362885

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 8.75847e-08      +/- 5.379e-10    (0.6141%)
```



Como podemos ver con este ajuste la varianza residual es practicamente nulo y concuerda con los resultados teóricos, pudiendo así concluir que efectivamente nuestro algoritmo es del orden $O(n \log n)$.

4.2.4. Alternativas:

Aquí presentamos una solución alternativa que obtiene solo el número de aulas a reservar con eficiencia $O(n \log n)$ también, con n el número de examenes a realizar:

```cpp
/**
 * @brief Algoritmo greedy que determina el máximo número de exámenes que se solapan
 * en el tiempo, que también coincide con el mínimo número de aulas a reservar.
 * @param all_exams Los examenes a realizar
 * @return el número máximo de examenes que se solapan en el tiempo
*/

int greedy(vector<info_exam> & all_exams) {

    // Construir estructura de eventos

    vector<pair<int,int>> events;

    for (int i=0; i < all_exams.size(); ++i) {
        events.emplace_back(all_exams[i].start_time, 1);            // 1: Empieza un examen
        events.emplace_back(all_exams[i].get_finish_time(), -1);    //-1: Termina un examen
    }

    // Se ordenan los eventos
    sort(events.begin(), events.end(), [](pair<int,int> a, pair<int,int> b)->bool{return a.first < b.first;});

    int count = 0;          // Número de exámenes que se solapan en el momento
    int max_count = 0;      // Máximo número de examenes que se han solapado hasta el momento

    for (int i=0; i < events.size(); ++i) {
        count += events[i].second;

        if (count > max_count) {
            max_count = count;
        }
    }

    return max_count;
}
```

Notemos que también es una solución greedy, puesto que max_count, que es el número de aulas a reservar, no reserva más aulas hasta que se necesite más aulas de las que se han reservado hasta el momento. La idea de este algoritmo es saber que el mínimo número de aulas a reservar coincide con el máximo de examenes que se solapan en el tiempo.

5. struct City

Para resolver los problemas 3 y 4 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.

```
void printCycle(const std::vector<int> & cycle, const City & origin, const City v[]){
    int ini = 0;
    while(v[cycle[ini]] != origin) ++ini;
    for(int i=ini; i<(int)cycle.size(); ++i){
        std::cout << v[cycle[i]] << std::endl;
    }
    for(int i=0; i<ini; ++i){
        std::cout << v[cycle[i]] << std::endl;
    }
    std::cout << origin << std::endl;
}
```

6. P3: Problema del camino mínimo

6.1. Definición del problema

Tenemos un conjunto de n ciudades (puntos en un plano), cada una definida por las coordenadas en el mapa (x_i, y_i), con $i = 1, ..., n$. Cada ciudad se puede conectar por carretera con un subconjunto de ciudades, asegurando que la red de vías que se genere crea un grafo conexo.

La distancia entre dos ciudades viene dada por la distancia euclídea entre sus coordenadas.

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

Dada una ciudad de origen, x, y una ciudad de destino, y, se pide encontrar el camino de costo mínimo que permita viajar de x a y.

6.2. Diseño del algoritmo

6.2.1. Algoritmo Greedy

La solución que se describe a este problema se le conoce como algoritmo de Dijkstra, el cual es muy útil para calcular la distancia mínima en un grafo de un nodo a cualquier otro. En este caso concreto se ha modificado ligeramente para también guardar el camino mínimo. El algoritmo consiste en, partiendo de la ciudad origen x, y de que su distancia mínima a cualquier ciudad por el momento es $+\infty$, comprobar su distancia a las ciudades adyacentes (las que estén conectadas por carretera directa) y actualizarla. Luego para cada una de esas ciudades (llamémoslas ciudades intermedias), y para cada una de sus ciudades adyacentes, se comprueba si el camino hacia ellas pasando por la ciudad intermedia es menor que el original que teníamos, actualizando el camino mínimo. Este proceso se repite hasta que ya no queden más ciudades que explorar (habremos explorado todas las ciudades pues se parte de que el grafo es conexo). Para guardar además el camino hacia cada ciudad cada ciudad tendrá asociada en todo momento su ciudad previa en el camino mínimo, y se irá actualizando conforme sea necesario.

6.2.2. Componentes y demostración de validez

Las componentes del algoritmo greedy son:

- **Conjunto candidatos V:** Todos las ciudades de la instancia del problema
- **Conjunto seleccionados S:** Todas las ciudades para las cuales ya se conoce el camino mínimo.
- **Función selección:** En cada paso se añade el nodo con menor coste de entre los no seleccionados `V\S` al conjunto de seleccionados.

A continuación demostraremos por inducción la validez del algoritmo de Dijkstra para el cálculo de caminos mínimos.

La demostración es por inducción.

- **Caso base** ($|S| = 0$): trivial
- **Paso de inducción:** Supongamos que para todo nodo del conjunto `S` se conoce su distancia mínima al nodo origen s que es la calculada por el algoritmo de Dijkstra $d(u)$ $\forall u \in S$. Ahora supongamos que el algoritmo de Dijkstra añade un nodo v al conjunto S. Demostremos por reducción al absurdo. Supongamos que la distancia a v calculada por el algoritmo de Dijkstra $d(v)$ no es la mínima. Entonces sea P el camino mínimo desde s hasta v, y su peso notado como $M(s, v)$. Ahora sea x el último nodo del camino P antes de salirse de S (todos los anteriores son de S y el siguiente no). Sea y el nodo inmediatamente posterior a x en P. Sea entonces $c(x, y)$ el peso del arco (x, y). Se tiene que:

  $$d(v) > M(s, v) = M(s, x) + c(x, y) + M(y, v) \geq M(s, x) + c(x, y) = d(x) + c(x, y) \geq d(y)$$

  Donde en la primera desigualdad hemos usado la suposición de la reducción al absurdo, en la primera igualdad hemos usado que el problema tiene subestructuras optimales, en la segunda

igualdad hemos usado la hipótesis de inducción y la última desigualdad se debe a que no hay pesos negativos.

En definitiva, se ha llegado a que $d(v) > d(y)$, lo cual es una contradicción pues en cada paso el algoritmo de Dijkstra escoge al nodo con menor distancia al conjunto, por lo que el algoritmo de Dijkstra hubiera seleccionado al nodo y en lugar del nodo v. Ello viene de suponer que $d(v) > M(s, v)$, por lo que queda demostrado que $d(v) = M(s, v)$.

6.3. Implementación y estudio de eficiencia

6.3.1. Datos de entrada

El formato de entrada de datos que hemos seguido es el que se muestra en la siguiente función main:

```
int main (int argc, char** argv) {

    // Faster I/O
    ios::sync_with_stdio(false);
    cin.tie(0);

    if (argc < 2) {
        cout << "Uso: ./greedy <input_file>" << endl;
        return 1;
    }

    // INPUT
/*
    Input format:
    3 --> number of cities
    1 2 --> origin and destination indexes
    (1,2) (2,0) (0,0) --> cities
    2 --> number of roads
    0 1 (road between (1,2) and (2,0))
    1 2 (road between (2,0) and (0,0))
*/
    char input_file[80] = "";
    strcat(input_file,argv[1]);
    ifstream fin(input_file,ios::in);

    int n,origin,dest;
    fin >> n >> origin >> dest;

    vector<City> cities(n);
    vector<vector<pair<int,ld>>> roads(n);

    for (int i=0; i<n; i++) {
        City a;
        fin >> a;
        cities[i] = a;
    }

    int m;
    fin >> m;

    for (int i=0; i<m; i++) {
        int a,b;
        fin >> a >> b;
        ld dist = cities[a]-cities[b];

        roads[a].push_back({b,dist});
        roads[b].push_back({a,dist});
    }

    fin.close();

    vector<ld> dist(n,INF);
    vector<int> prev(n,-1);

    Dijkstra(roads,dist,prev,origin);
```

```
56
57      vector<City> path;
58
59      for (int i=dest; i!=-1; i=prev[i]) {
60          path.insert(path.begin(),cities[i]);
61      }
62
63      // OUTPUT
64
65      cout << path << endl;
66
67      return 0;
68  }
```

Los datos de entrada se leen desde un fichero (instancias) que se pasa como parámetro al programa y que contiene los datos en el formato que se muestra comentado en el código: Primero lee un entero N que será el número de ciudades del problema. Luego lee dos enteros entre 0 y $N-1$ que representarán el número de ciudad de origen y destino, respectivamente, de entre las N ciudades que se pasan a continuación, en el formato que indica el operador de entrada del `struct City.h`. Con esto ya se leen las ciudades, a continuación se pasan a leer las aristas del grafo (carreteras). Primero se lee el número de aristas que tiene el grafo M y a continuación se leen M pares de enteros que representan los números asociados a cada ciudad (según el orden en el que se habían introducido, empezando por 0) que indican que en el grafo hay una arista que contiene a esos dos nodos (una carretera que une esas dos ciudades).

El grafo se representa mediante un vector de vectores, en el que el vector `roads[i]` contiene todas las ciudades con conexión directa con la ciudad `cities[i]` así como sus respectivas distancias.

6.3.2. Implementación

A continuación se explica la implementación del algorimo de Dijkstra en detalle el cuál es llamado en la línea 57 de la función main:

```
1  void Dijkstra (const vector<vector<pair<int,ld>>> & g, vector<ld> & dist,
2                 vector<int> & prev, int origin) {
3
4      // Auxiliar priority queue data structure with unreached
5      // nodes (modified to let the first element be the minimum)
6      min_priority_queue<pair<ld,int>> q;
7
8      // First node (distance 0)
9      q.push({0,origin});
10     dist[origin] = 0;
11
12     // While there are unreached nodes in q
13     while(!q.empty()) {
14
15         // Next unreached node (with minimum distance)
16         auto p = q.top();
17         q.pop();
18
19         // Index (distance is not necessary, we only need the minimum distance)
20         int node = p.second;
21
22         // For each adjacent node u
23         for (auto u : g[node]) {
24             int v = u.first; // Index of u
25
26             // Distance from "node" to "v" (euclidean distance previously calculated)
27             ld d = u.second;
28
29             // Compare the original distance "dist[v]" with the
30             // distance through the new path "dist[node] + d"
31             if (dist[node] + d < dist[v]) {
32
33                 // Update distance and previous node if necessary
34                 dist[v] = dist[node]+d;
35                 prev[v] = node;
36
```

```
37                  // Reached v --> explore it
38                  q.push({dist[v],v});
39              }
40          }
41      }
42  }
```

Los parámetros de la función son, el grafo de ciudades y carreteras, que ahora serán referidos como nodos y aristas, como parámetro de entrada; un vector `dist` de manera que al final de la función `dist[i]` guardará la distancia mínima desde el nodo `origin` hasta el nodo `i`; un vector `prev` de manera que en el camino mínimo desde `origin` hasta `i`, antes de llegar a `i`, se pasa por `prev[i]`; y por último el parámetro de entrada `origin` como el nodo origen a partir del cual se calculará el camino mínimo a cada nodo.

Comenzamos por declarar una cola con prioridad `q` que tendrá siempre al frente el siguiente nodo a seleccionar:

```
1  template<typename T>
2  using min_priority_queue = priority_queue< T, vector<T>, greater<T> >;
```

Después, se guarda en la cola el primer nodo `origin` con distancia 0, que también se guarda en `dist`.

A continuación, se entra en un bucle en el cual se procesa siempre el elemento `p` con menor distancia a `origin` (el primero de la cola `q`), empezando claro está por el mismo `origin`, tomando solo su índice `node` (la distancia no hace falta pues ya la tendremos en `dist[node]`). Y ahora para cada nodo `u` adyacente a `node` tomamos su índice `v` y su distancia actual `d` a `origin` (si no se ha procesado, será $+\infty$), y comparamos el peso del camino pasando por `node` y de `node` a `v` con el peso del camino actual a `v` (línea 31) y en caso de que sea menor (la primera vez que se procese siempre lo será) se actualiza el peso del camino a `v`, `dist[v]` y por tanto el previo a `v` en su camino mínimo `prev[v]`. Por último, se añade el nodo de índice `v` con su distancia actualizada `dist[v]` a la cola `q` para procesarlo en futuras iteraciones del bucle. Se añadirán a la cola aquellos nodos para los que se haya encontrado un camino más corto pasando por el nodo intermedio (`node`) por lo que no hay que preocuparse de casos como que se añadan dos nodos en bucle infinito a la cola al ser la distancia una función positiva. Si hubiese pesos negativos se podría dar una situación como la anterior.

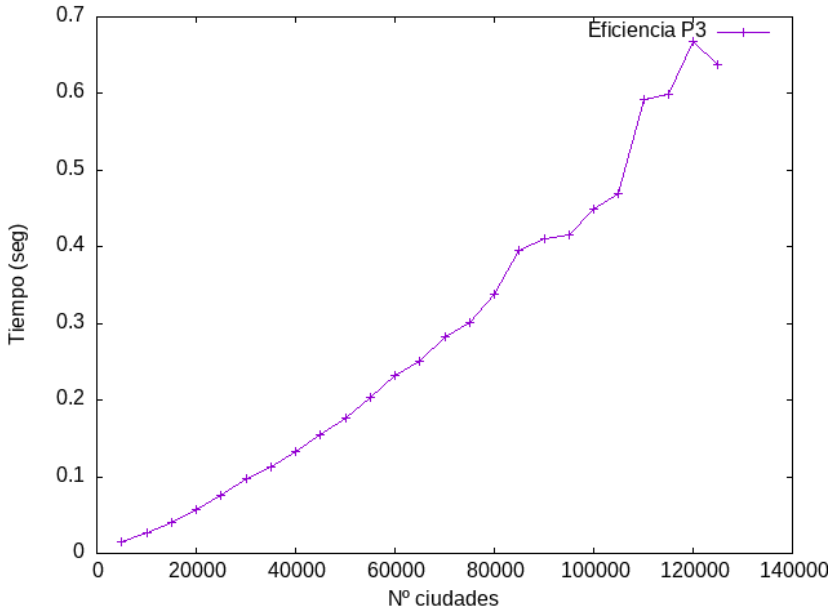
6.3.3. Estudio de eficiencia

Eficiencia teórica:

Analizamos el algoritmo para la eficiencia teórica. En el bucle externo, primero eliminamos el nodo al frente del heap, lo cual tiene una eficiencia de $O(log\ n)$ siendo n el número de nodos. El bucle for interno se ejecuta un máximo de n veces y dentro de él se elimina el nodo v del heap en caso de que sea mejor el camino por *node*. Esto no ocurre más de una vez por arista. Por tanto el algoritmo de Dijkstra tiene una eficiencia del orden de $O((a+n)log\ n)$ siendo a el número de arcos y n el número de nodos. Si nos reducimos al caso de un grafo conexo, que es el nuestro, tenemos que $a \geq n-1$ y por tanto la eficiencia pasa a ser $O(a\ log\ n)$. En el caso de grafos muy densos, la versión de cola con prioridad pierde eficiencia, pues en el caso de un grafo completo sería del orden de $O(n^2\ log\ n)$.

Eficiencia empírica e híbrida:

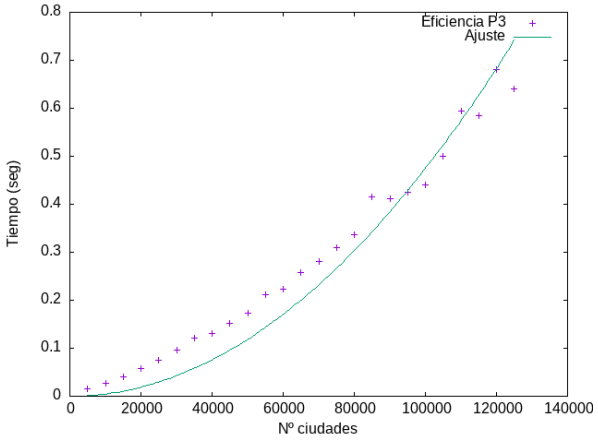
Al ejecutar para diferentes números de ciudades el programa (desde 5000 hasta 125000) obtenemos los siguientes tiempos de ejecución:



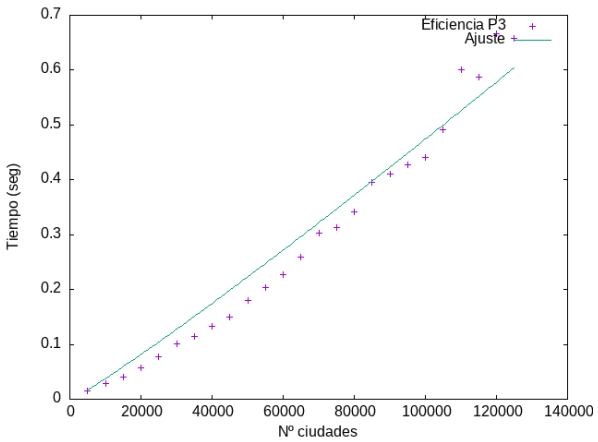
Se puede observar que no sigue una curva uniforma (una cuadrática o una lineal-logarítmica) ya que la eficiencia depende también del número de arcos, que es generado aleatoriamente entre $n-1$ para que sea conexo y n^2.

Si tratamos de hacer una regresión cuadrática o lineal logarítmica obtenemos los siguientes resultados.

```
*******************************************************************************
Sun Apr 28 22:35:49 2024


FIT:    data read from DATOS
        format = z
        #datapoints = 25
        residuals are weighted equally (unit weight)

function used for fitting: f(x)
        f(x)=a*x*x
fitted parameters initialized with current variable values

iter      chisq       delta/lim  lambda   a
   0 1.3460281249e+21   0.00e+00  7.34e+09    1.000000e+00
   5 5.7157562654e-02  -1.43e-08  7.34e+04    4.745548e-11

After 5 iterations the fit converged.
final sum of squares of residuals : 0.0571576
rel. change during last iteration : -1.42766e-13

degrees of freedom    (FIT_NDF)                       : 24
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0488013
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00238157

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 4.74555e-11      +/- 1.33e-12     (2.803%)
```




```
*******************************************************************************
Sun Apr 28 22:39:07 2024


FIT:    data read from DATOS
        format = z
        #datapoints = 25
        residuals are weighted equally (unit weight)

function used for fitting: f(x)
        f(x)=a*x*log(x)
fitted parameters initialized with current variable values

iter      chisq       delta/lim  lambda   a
   0 1.8036088599e+13   0.00e+00  8.49e+05    1.000000e+00
   4 3.6704728907e-02  -1.86e-01  8.49e+01    4.121644e-07

After 4 iterations the fit converged.
final sum of squares of residuals : 0.0367047
rel. change during last iteration : -1.85936e-06

degrees of freedom    (FIT_NDF)                       : 24
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0391071
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00152936

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 4.12164e-07      +/- 9.208e-09    (2.234%)
```



Observamos que con ambos ajustes se obtiene un error no despreciable porque el número de aristas, que se está tomando de forma arbitraria, se tiene en cuenta también a la hora de calcular la eficiencia.

7. P4: Problema del Viajante de Comercio

7.1. Definición del problema

Tenemos un conjunto de n ciudades (puntos en un plano), cada una definida por las coordenadas en el mapa (x_i, y_i), con $i = 1, \ldots, n$. 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 no podemos esperar encontrar una solución óptima al mismo, lo que se pretende en esta práctica es utilizar la estrategia del greedy para encontrar una solución aproximada que puede ser de utilidad en situaciones como las que se plantea en este problema.

7.2. Versión 1

7.2.1. Diseño e implementación del algoritmo voraz

Diseño

La primera versión de algoritmo greedy para resolver este problema es la más sencilla de todas, y es la que se le puede ocurrir a uno de forma más directa, aplicando la heurística (*greedy*) a los nodos del grafo. Se trata de en cada paso ir a la ciudad más cercana (closest neighbour) hasta que hayamos visitado todas las ciudades.

Dado que el problema es NP-Difícil, esta estrategia no da con la solución óptima, pero sí es cierto que consigue un resultado aproximado de manera fácil y rápida.

Implementación

```
void TSP_greedy_v1(int n, int home_ind, const City v[], vector<int> & path){

    // Not visited cities
    vector<int> not_visited(n);

    // Initializes with all cities
    for(int i=0; i < n; i++) {
        not_visited[i] = i;
    }
    // Start with home city
    int current = home_ind;

    while (not_visited.size() > 0) {
        // Add current city
        path.push_back(current);

        // Remove it from not_visited cities
        remove(not_visited,current);

        // Calculate the closest neighbour
        ld min_dist = INF, dist;
        int next;
        for (int city : not_visited) {
            dist = v[current]-v[city];
            if (dist < min_dist) {
                min_dist = dist;
                next = city;
            }
```

```
        }

        // Next city
        current = next;
    }
    path.push_back(home_ind);
}
```

En la función, `n` es un entero que indica el número de ciudades y `home_ind` indica el índice la ciudad desde la que se parte en el vector `v`, que almacena todas las ciudades haciendo uso de la `struct City`. Finalmente, la solución se va guardando en forma de secuencia de índices en el vector `path`, que tendrá como primer y último elemento a `home_ind`.

Se comienza inicializando un vector que guardará los índices de las ciudades que no se hayan visitado. Este vector irá decreciendo en tamaño durante la ejecución de la función. A continuación se inicializa dicho vector con las ciudades en orden y se declara `current` a `home_ind`, donde `current` indicará en todo momento la ciudad que se está visitando. Después en cada iteración del bucle se añade la ciudad actual a `path` y se elimina de `not_visited`. Luego, para cada ciudad de entre todas las ciudades no visitadas se calcula aquella con menor distancia a la actual, y se selecciona para la siguiente iteración.

La función `remove` es la que se muestra a continuación:

```
template<typename T>
void remove(vector<T> & v, const T & elem) {
    auto it = find(v.begin(), v.end(), elem);
    if (it != v.end()) {
        v.erase(it);
    }
}
```

Eficiencia de remove: lineal.

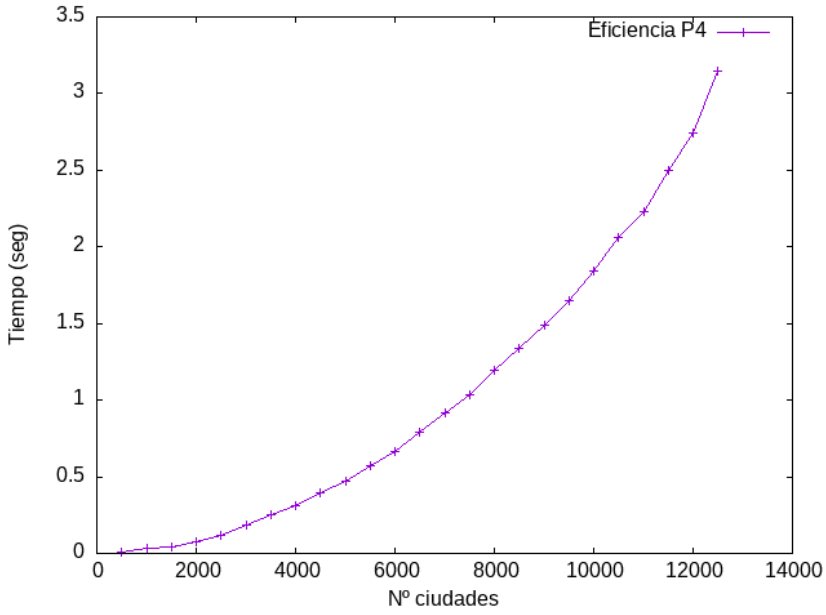
7.2.2. Análisis de eficiencia

Eficiencia teórica

La eficiencia teórica es sencilla en este caso. Para cada ciudad, se elimina la ciudad del vector de no visitados en orden lineal, y se calcula la distancia mínima en el orden del número de ciudades que queden por visitar. Aplicando la regla del máximo, visitar cada ciudad supone un costo lineal, y como se visitan todas las ciudades el orden de eficiencia es $O(n^2)$.

Eficiencia empírica

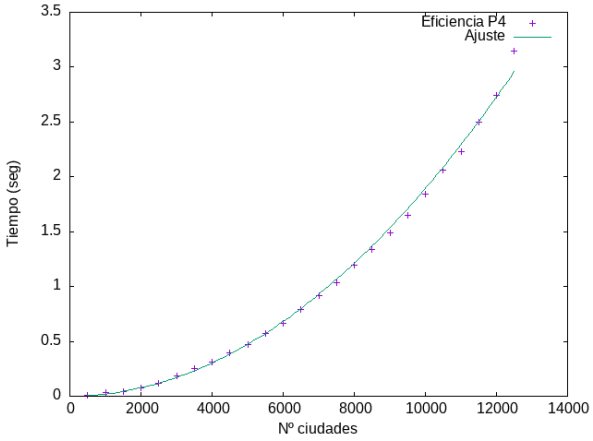
Al ejecutar el programa para números de ciudades desde 500 hasta 12500 con saltos de 500 se obtienen los siguientes resultados:



Eficiencia híbrida

A la hora de hacer un ajuste cuadrático se obtiene el siguiente resultado.

```
******************************************************************************
Mon Apr 29 14:37:39 2024


FIT:    data read from DATOS
        format = z
        #datapoints = 25
        residuals are weighted equally (unit weight)

function used for fitting: f(x)
        f(x)=a*x*x
fitted parameters initialized with current variable values

iter      chisq       delta/lim  lambda   a
   0 1.3460280721e+17   0.00e+00  7.34e+07    1.000000e+00
   5 5.5495672092e-02  -5.00e-11  7.34e+04    1.963912e-08

After 5 iterations the fit converged.
final sum of squares of residuals : 0.0554957
rel. change during last iteration : -5.00139e-16

degrees of freedom    (FIT_NDF)                        : 24
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0480866
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00231232

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 1.96391e-08      +/- 1.311e-10    (0.6674%)
```



La función obtenida es:

$$f(x) = 1.96391 \cdot 10^{-8} \cdot x$$

La cual como vemos, se ajusta prácticamente a la perfección a la gráfica obtenida, reafirmando nuestra conclusión en el análisis teórico de que la eficiencia de la función es $O(n^2)$.

7.3. Versión 2

7.3.1. Diseño e implementación del algoritmo voraz

Diseño

En esta versión de la solución, hemos optado por aplicar la heurística voraz (*greedy*) a los arcos del grafo en vez de a los nodos, al contrario de la versión anterior. La solución se resume en dos sencillos sencillos pasos:

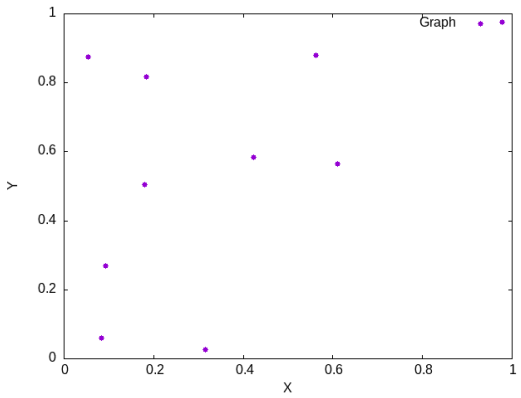
a) Construir el árbol recubridor mínimo (*Minimum Spanning Tree o* ***MST***).

b) Listar el árbol construido en preorden, transformándolo así en un ciclo que recorre todas las ciudades una única vez.

Ejemplo. Dadas las siguientes ciudades y sus coordenadas:

```
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)
```

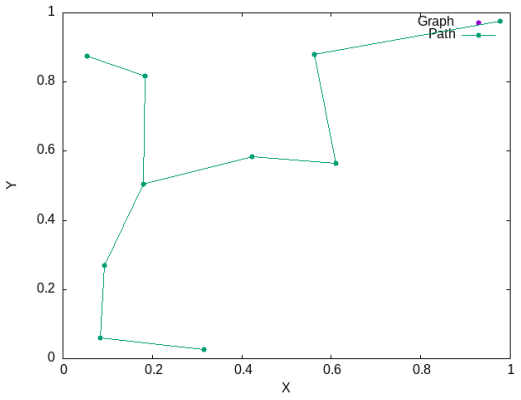
(a) Input data



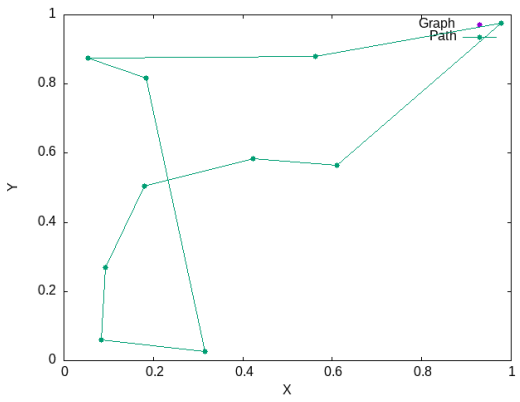
(b) Representación gráfica

Obtenga el ciclo de coste mínimo que recorra todas las ciudades una única vez (partiendo y acabando en la primera ciudad).

Solución



(a) Árbol recubridor mínimo (MST)



(b) Recorrido en preorden (solución)

Calidad de la solución

Si eliminamos un arco a la solución óptima del problema del viajante de comercio (llamémosla TSP), obtenemos un árbol. Por definición de árbol recubridor mínimo (llamémosle TSP), sabemos que es el árbol cuya suma del peso sus arcos es mínima. Por tanto, dicha suma es menor que la suma del peso todos los arcos de TSP menos uno, (y por tanto menor que la de todos).

Si quisiéramos recorrer todos los nodos del grafo con tan solo los arcos del MST, en el peor de los casos deberíamos recorrer cada arco dos veces. Este caso se daría si todos los nodos estuvieran conectados al mismo y partiéramos del nodo raíz como se puede observar en la figura (10).

Como nuestra solución lo que hace es listar el MST en preorden, tenemos pues que si un nodo tiene más de un hijo, a la hora de visitar el siguiente hijo, en vez de volver a recorrer el arco que une al hijo anterior con el padre para después recorrer el arco que une al padre con el siguiente hijo, nuestro algoritmo se limita a ir directamente de un hijo al otro.

Dado que nuestro problema en particular trabaja con la distancia euclídea, la cual cumple la **desigualdad triangular**, es decir, la distancia de ir de un nodo a otro es menor o igual que la de ir

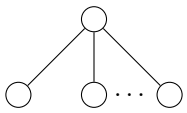


Figura 10: MST en el que hay que recorrer todos los arcos para recorrer todos los nodos

primero a un nodo intermedio para después ir al nodo destino. Formalmente sería

$$\forall u, v, w \in V \;\; dist(u, v) \leq dist(u, w) + dist(w, v) \quad \text{(Desigualdad Triangular)}$$

donde V es el conjunto de los nodos del grafo.

Tenemos pues que haciendo uso de dicha desigualdad, el coste de la solución proporcionada por nuestro algoritmo es menor o igual que el coste de recorrer MST dos veces. Y por definición de MST, esto es menor o igual que el coste recorrer la solución óptima dos veces, donde por coste entendemos la suma de las distancias. Matemáticamente, notando nuestra solución por SOL, tenemos

$$cost(SOL) \leq 2 \cdot cost(MST) \leq 2 \cdot cost(TSP)$$

En conclusión, tenemos que el coste de la solución dada por este algoritmo es, en el peor de los casos, **a lo sumo 2 veces el coste de la solución óptima**.

Implementación

El algoritmo previamente explicado ha sido implementado de la siguiente manera:

```
void TSP_greedy_v2(int origin, int n, City cities[], vector<int> & ans) {
    // Construction of the Minimum Spanning Tree with Prim's algorithm
    vector<vector<int>> mst;
    Prim(origin,n,cities,mst);

    // The given tree is listed in preorder (turning the tree into a cycle)
    ans.reserve(n);
    dfs(origin,-1,mst,ans);
}
```

Para construir el arbol de recubrimiento mínimo se ha escogido el ***algoritmo de Prim*** debido a que para grafos completos (en los que existe un arco entre todos los nodos) como es nuestro caso (o simplemente grafos densos[1]) resulta más eficiente que el *algoritmo de Kruskal*.

Esto se puede comprobar fácilmente. Llamanado n al número de nodos del grafo y m al número de arcos, tenemos que la eficiencia de dichos algoritmos[2] es la que viene en la tabla (1).

Prim	$O(n^2)$
Kruskal	$O(mlogn)$

Cuadro 1: Eficiencia algoritmos de Prim y Kruskal

Como hemos mencionado antes, nuestro grafo es completo, por tanto sabemos que el número de arcos del grafo es $m = \frac{n(n-1)}{2}$.

Por tanto, sustituyendo tenemos que la eficiencia en de ambos algoritmos en nuestro caso es la que viene en la tabla (2). Claramente vemos que nos conviene más utilizar $Prim$ que $Kruskal$.

Veamos ahora la implementación de `Prim` con más detalle.

[1] Se llama grafos densos a aquellos que tienen muchos arcos respecto a su número de nodos

[2] Hay varias implementaciones para el algoritmo de Prim. Nos hemos decantado por la que nos es más favorable en este caso.

Prim	$O(n^2)$
Kruskal	$O(n^3)$

Cuadro 2: Eficiencia algoritmos de Prim y Kruskal

```
void Prim(int origin, int n, City cities[], vector<vector<int>> & mst){
    // Adjacency matrix of the graph (the distances of all nodes between each other)
    vector<vector<ld>> dist(n,vector<ld>(n));
    for(int i = 0; i < n; ++i){
        for (int j = 0; j < n; ++j){
            dist[i][j] = cities[i] - cities[j];
        }
    }
```

En primer lugar, vemos que se construye la **matriz de adyacencia del grafo**. Esto no forma parte del *algoritmodePrim* como tal, y de hecho es algo innecesario en nuestro caso. No obstante, se ha añadido para facilitar la lectura del código, evitar repetir cálculos (calcular todo el rato la distancia entre los mismo nodos) y aportar generalidad a la función, de forma que en caso de cambiar el grafo o la distancia entre los nodos, simplemente haría falta modificar la matriz de adyacencia.

```
    // visited[i] = whether we have already visited city i or not
    vector<bool> visited(n,false);
    visited[origin] = true; // Initially we have only visited the origin city

    // nearest_visited_city[i] = the nearest city to city i of all the cities we have
    already visited
    // Initially we have only visited the origin city, so thats the nearest visited
    city to
    // every city
    vector<int> nearest_visited_city(n,origin);

    // Memory allocation for the answer graph
    mst.assign(n,vector<int>());
```

Necesitaremos un vector de booleanos `visited` para saber que ciudades se han visitado y cuales no (puesto que solo se puede pasar por cada ciudad una **única** vez). Esto es si `visited[i] == true` entonces la ciudad i-ésima ha sido visitado y si `visited[i] == false` no. Al principio solo se ha visitado la ciudad de origen.

También hará falta tener un vector que tenga almacenada para cada ciudad no visitada la ciudad más próxima de todas las ciudades visitadas. Como en principio la única ciudad visitad es la ciudad origen, inicializamos el vector entero a la ciudad origen.

No nos olvidamos de asignar la memoria necesaria a la **lista de adyacencia** en la cual almacenaremos el MST.

Pasemos ahora a explicar el *algoritmo de Prim* en cuestión.

```
    for(int i = 1; i < n; ++i){ // We visited the remaining n-1 cities
        int v = -1; // Of all the remaining cities, the nearest one to the visited
    cities (we choose one of the visited cities)
        for (int u = 0; u < n; ++u){
            if(!visited[u] && (v == -1 || dist[u][nearest_visited_city[u]] < dist[v][
    nearest_visited_city[v]])){
                v = u;
            }
        }
```

Para formar el MST necesitamos visitar las $n-1$ ciudades restantes (recordamos que la de origen ya la hemos visitado), ahí el bucle `for` de $n-1$ iteraciones. En cada iteración, localizamos la ciudad más cercana a las ciudades visitadas. Es decir, si definimos la distancia de una ciudad no visitada a las ciudades visitadas como el mínimo del conjunto de las distancias de la ciudad no visitada a cada ciudad visitada, entonces nos quedamos con la ciudad no visitada que minimiza esta distancia.
Formalmente sería de la siguiente forma:

Si U es el conjunto de las ciudades no visitadas, es decir,

$$U = \{u \in C : visited[u] = false\}$$

donde C es el conjunto de las ciudades, y V es el de las ciudades visitadas

$$V = \{v \in C : visited[v] = true\}$$

definimos

$$dist(u, V) = min\{dist(u, v) : v \in V\} \quad \forall u \in U$$

y buscamos $u_0 \in U$ tal que

$$dist(u_0, V) = min\{dist(u, V) : u \in U\}$$

```
        visited[v] = true; // We visit the selected city and add it to the solution
        mst[v].push_back(nearest_visited_city[v]);
        mst[nearest_visited_city[v]].push_back(v);
```

Una vez elegido la siguiente ciudad que vamos a visitar, la marcamos como visitada y la añadimos a nuestro árbol solución del problema, dicho de otro modo, añadimos el arco formado por la ciudad seleccionada y la ciudad visitada más cercana a ella a la lista de adyacencia de nuestro árbol solución.

```
        // Update the nearest visited city to all cities (in case the newest visited
    city
        // is closer to them than their current nearest visited city)
        for (int to = 0; to < n; ++to){
            if(dist[v][to] < dist[to][nearest_visited_city[to]]){
                nearest_visited_city[to] = v;
            }
        }
```

Por último, como hemos visitado una ciudad más, iteramos por todas las ciudades y actualizamos su ciudad visitada más cercana a la ciudad recién añadida en caso de que su distancia a esa sea menor a la distancia a su ciudad visitada más cercana actual.

7.3.2. Análisis de eficiencia

Eficiencia teórica
Calculemos la eficiencia teórica del algoritmo, es decir, de la función `TSP_greedy_v2`.

```
void TSP_greedy_v2(int origin, int n, City cities[], vector<int> & ans) {
    // Construction of the Minimum Spanning Tree with Prim's algorithm
    vector<vector<int>> mst;
    Prim(origin,n,cities,mst);

    // The given tree is listed in preorder (turning the tree into a cycle)
    ans.reserve(n);
    dfs(origin,-1,mst,ans);
}
```

Se ve claramente que la eficiencia de dicha función es igual a la suma de la eficiencia de la función `Prim` (que ya se ha visto implementa el *algoritmo de Prim*) y la de la función `dfs` (que implementa una *búsqueda en profundidad* de un árbol) más 1 por la reserva de memoria del vector solución, pero esto es despreciable.

- **Prim:** Análicemos la eficiencia teórica de la función `Prim`.

```
    // Adjacency matrix of the graph (the distances of all nodes between each
    other)
    vector<vector<ld>> dist(n,vector<ld>(n));
    for(int i = 0; i < n; ++i){
        for (int j = 0; j < n; ++j){
            dist[i][j] = cities[i] - cities[j];
        }
    }
```

Inicialmente, se crea la **matriz de adyacencia** de nuestro grafo[3]. El código se trata de dos bucles `for` anidados que realizan n iteraciones cada uno[4] y por tanto su eficiencia es $O(n^2)$.

```
    vector<bool> visited(n,false);
    visited[origin] = true; // Initially we have only visited the origin city
```

```
    vector<int> nearest_visited_city(n,origin);
```

```
    mst.assign(n,vector<int>());
```

Después se crean e inicializan las correspondientes arrays, lo cual supone $O(1)$.

```
    for(int i = 1; i < n; ++i){ // We visited the remaining n-1 cities
        int v = -1; // Of all the remaining cities, the nearest one to the visited
     cities (we choose one of the visited cities)
        for (int u = 0; u < n; ++u){
            if(!visited[u] && (v == -1 || dist[u][nearest_visited_city[u]] < dist[
    v][nearest_visited_city[v]])){
                v = u;
            }
        }

        visited[v] = true; // We visit the selected city and add it to the
    solution
        mst[v].push_back(nearest_visited_city[v]);
        mst[nearest_visited_city[v]].push_back(v);

        #ifdef MST
            cout << cities[v] << endl;
            cout << cities[nearest_visited_city[v]] << endl;
            cout << endl;
        #endif

        // Update the nearest visited city to all cities (in case the newest
    visited city
        // is closer to them than their current nearest visited city)
        for (int to = 0; to < n; ++to){
            if(dist[v][to] < dist[to][nearest_visited_city[to]]){
                nearest_visited_city[to] = v;
            }
        }

    }
```

Finalmente, el *algoritmo de Prim* consiste en un bucle `for` que realiza $n-1$ iteraciones, compuestas de dos bucles `for` secuenciales de n iteraciones cada uno más operaciones constantes de asignación de variables y reasignaciones de memoria que son $O(1)$.
Sumado (por ser secuenciales), nos queda que la eficiencia dentro del bucle es $O(2n+1)$, y por tanto la del bucle en sí nos queda $O(2n^2+n)$.
Por tanto, la eficiencia de la función `Prim` nos queda:

$$T(n) = n^2 + 1 + 2n^2 + n = 3n^2 + n + 1$$

Lo que implica que el orden de eficiencia de la función[5] es: $O(n^2)$

- **dfs (Depht First Search):** Análicemos la eficiencia teórica de la función `dfs`.

```
void dfs(int node, int parent,vector<vector<int>> & tree, vector<int> & ans){
    ans.push_back(node); // Add the node to the answer
```

[3]Esto ya vimos que es innecesario, pero se realiza para evitar tener que estar calculando constantemente la distancia entre las mismas ciudades (repetir cálculos) y facilitar la legibilidad del código. En breve se verá que no afecta al orden de eficiencia del algoritmo en lo más mínimo.

[4]Podríamos aprovechar que la distancia euclídea es simétrica y ahorrarnos la mitad de iteraciones del segundo bucle, no obstante, dado que como ya se verá más adelante esta mejora no afecta al orden de eficiencia del algoritmo, se ha optado por no realizar dicha mejora y a cambio dotar de mayor generalidad al código, siendo fácilmente reutilizable en caso de cambiar las distancias del problema.

[5]Tal y como se previó, la creación de la matriz de adyacencia no ha modificado el orden de eficiencia de la función en lo más mínimo

```
3       for(int neighbour : tree[node]){ // Visit the node's neighbours
4           // If the neighbour is not the node's parent, then it means it hasn't been
        visited
5           // (since the given graph is a tree) so it is visited
6           if(neighbour != parent){
7               dfs(neighbour,node,tree,ans);
8           }
9       }
10  }
```

Se trata de una función recursiva, que tiene una operación que es $O(1)$ al principio (recordamos que se reservó memoria para el vector `ans`) y luego tiene tantas llamadas recursivas como vecinos tenga el nodo en cuestión descontando al nodo padre.
Como se sabe que está función va a actuar sobre un árbol, se ve que no entrará en un ciclo infinito, es decir, aunque no haya un caso base claramente diferenciado, sabemos que en cuanto se llegue a un nodo hoja, es decir, un nodo cuyo único vecino sea su padre, no se realizarán más llamadas recursivas. Además tenemos que si n es el número de nodos, entonces el número de arcos es $n-1$, por tratarse de un árbol.
Es evidente pues, que la función se dedica a recorrer todos los nodos del árbol una **única** vez (por como están estructurados los árboles) y por tanto la eficiencia de la función es $O(n)$.
Comprobemos que esto es cierto de manera más formal.
Tenemos pues que, llamando n al número de nodos del árbol, la función de eficiencia para la función `dfs` es:

$$T(n) = \sum_{i=1}^{nhijos} T(subarbol_i) + 1 \quad \forall n \in \mathbb{N}$$

donde $nhijos$ es el número de vecinos del nodo raíz y $subarbol_i$ es el número de nodos del subárbol del hijo i-ésimo.

Teorema 7.1. *Sea $T : \mathbb{N} \longrightarrow \mathbb{R}^+$ la función previamente definida. Entonces*

$$T(n) = n \quad \forall n \in \mathbb{N}$$

Demostración. Razonando por el **segundo principio de inducción**:

- Caso base $n = 1$:
  Obviamente $T(1) = 1$ puesto que al ser un árbol de un único nodo, este no tiene vecinos (y en consecuencia hijos) y por tanto la función no realiza llamadas recursivas.
- Supuesto cierto $T(k) = k \ \forall k < n$ tal que $k, n \in \mathbb{N}$ probemos que es cierto para n.
  Obviamente, si tenemos un árbol de n nodos, la suma de los subárboles de sus hijos debe ser $n-1$ ya que son todos los nodos del árbol inicial menos el nodo raíz, es decir:

  $$\sum_{i=1}^{nhijos} subarbol_i = n-1 \implies subarbol_i \leq n-1 < n \ \forall i \in \mathbb{N} \text{ tal que } 1 \leq i \leq nhijos$$

  Por tanto, podemos aplicar la hipótesis de inducción a $subarbol_i \ \forall i \in \mathbb{N}$ tal que $1 \leq i \leq nhijos$
  De esta forma:

  $$T(n) = \sum_{i=1}^{nhijos} T(subarbol_i) + 1 = \{\text{hip. ind.}\} = \sum_{i=1}^{nhijos} subarbol_i + 1 = n - 1 + 1 = n$$

□

Queda pues demostrado que la función `dfs` es $O(n)$.

En conclusión, sumando la eficiencia de ambas funciones, puesto que se llaman secuencialmente, tenemos que la eficiencia de la función `TSP_greedy_v2` es:

$$O(n) = O(\texttt{Prim}) + O(\texttt{dfs}) + O(1) = O(n^2) + O(n) + O(1) = O(n^2 + n + 1) = O(n^2)$$

La cual nos indica que el orden de eficiencia es $O(n^2)$.

Eficiencia empírica
Realicemos ahora el estudio de la eficiencia empírica.Para ello, el programa ha sido ejecutado con valores desde $n = 100$ hasta $n = 3250$ cada 150 siendo n el número de ciudades.
Los tiempos obtenidos para n ciudades se puede observar en la tabla (3).

n	Tiempo (seg)
100	0.000541
250	0.003491
400	0.008968
550	0.017105
700	0.026252
850	0.040499
1000	0.054733
1150	0.074964
1300	0.095075
1450	0.117382
1600	0.142658
1750	0.171322
1900	0.20185
2050	0.232115
2200	0.267185
2350	0.308353
2500	0.353059
2650	0.397252
2800	0.44067
2950	0.490691
3100	0.558724
3250	0.587015

Cuadro 3: Tiempos para la versión 2 del TSP

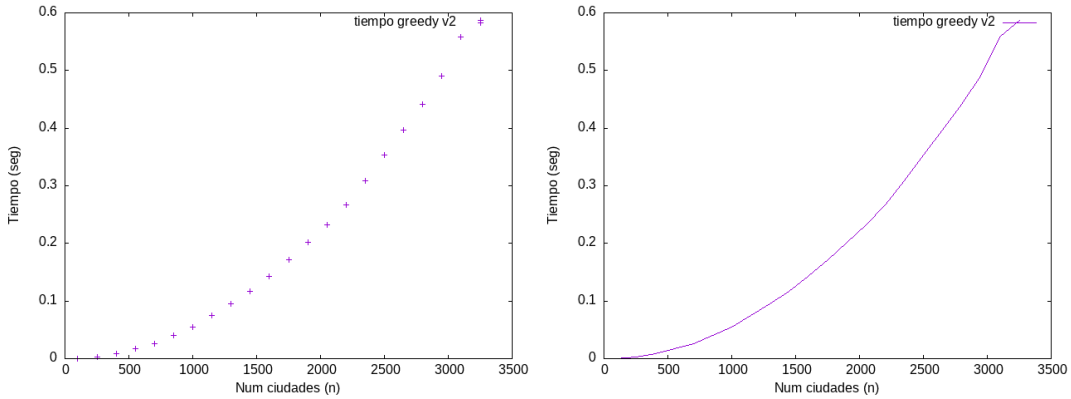


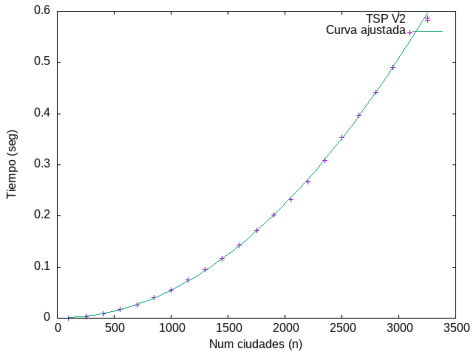
Figura 11: Eficiencia empírica de la versión 2 del TSP

Como se puede observar en la figura (11), se trata de una función claramente cuadrática, lo que coincide con el resultado demostrado en el análisis teórico.

Eficiencia híbrida
Previamente en el análisis teórico se vio que la eficiencia del algoritmo es $O(n^2)$ y en el análisis empírico constatamos que efectivamente el algoritmo tiene complejidad cuadrática.

Para obtener las constantes ocultas de la ecuación de eficiencia y verificar que en efecto la función que representa la complejidad del algoritmo se trata de una función cuadrática hemos realizado una

regresión cuadrática por mínimos cuadrados sobre los puntos obtenidos empíricamente con `gnuplot`, es decir, con una recta de la forma $f(x) = ax^2 + bx + c$.



(a) Gráfica ajustada de la versión 2 del TSP

```
After 10 iterations the fit converged.
final sum of squares of residuals : 0.000403926
rel. change during last iteration : -5.28759e-07

degrees of freedom    (FIT_NDF)                        : 19
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.00461078
variance of residuals (reduced chisquare) = WSSR/ndf   : 2.12593e-05

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 5.77329e-08      +/- 1.217e-09    (2.109%)
b0              = -4.19136e-06     +/- 4.207e-06    (100.4%)
c0              = 0.00139113       +/- 0.003051     (219.3%)

correlation matrix of the fit parameters:
                a0     b0     c0
a0              1.000
b0             -0.969  1.000
c0              0.758 -0.874  1.000
```

(b) Regresión lineal de la versión 2 del TSP

La función obtenida es:

$$f(x) = 5.77329 \cdot 10^{-8}x^2 - 4.19136 \cdot 10^{-5}x + 0.00139113$$

Como podemos observar en la figura (12b) la varianza residual es minúscula lo que nos indica que el ajuste realizado es el correcto, tal y como se ve en la gráfica (12a) y por tanto el algoritmo es, en efecto, $O(n^2)$.

7.4. Versión 3

7.4.1. Diseño e implementación del algoritmo voraz

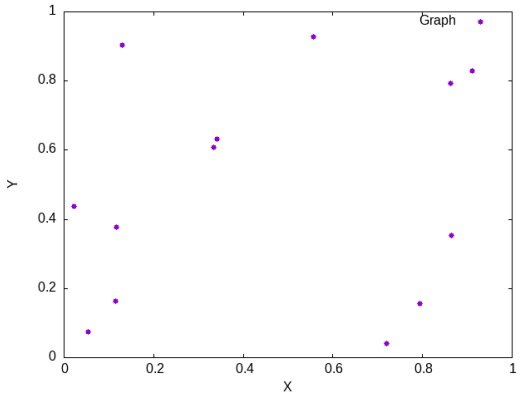
Diseño

En esta versión, aplicamos la misma heurística que en el caso anterior pero en vez de constuir el *Minimun Spanning Tree* y después listarlo en preorden, añadimos en cada paso la arista de menor coste directamente, comprobando que no se generen ciclos antes de tiempo y que no haya nodos de grado mayor que dos. Dadas estas recurrentes comprobaciones, este enfoque será menos eficiente que el anterior (lo cual comprobaremos), pero veremos que en la mayoría de los casos las soluciones son de mayor calidad. Veamos un ejemplo de cálculo de camino:

Ejemplo. Dadas las siguientes ciudades y sus coordenadas:

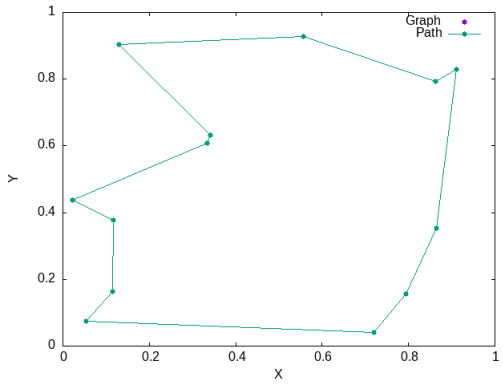
```
13
(0.341416,0.632643)
(0.130035,0.90411)
(0.334507,0.608363)
(0.794827,0.155829)
(0.556065,0.928201)
(0.0531666,0.074904)
(0.11578,0.163737)
(0.911695,0.829456)
(0.862248,0.792853)
(0.0231408,0.436502)
(0.865406,0.353367)
(0.720413,0.040622)
(0.11688,0.377968)
```

(a) Input data



(b) Representación gráfica

Solución



(a) Camino obtenido con la heurística greedy

Implementación

Describimos cómo hemos implementado el algoritmo. La idea que hemos utilizado para controlar que no se generen ciclos es que un ciclo se genera si y solo si se unen con una arista en el grafo dos ciudades que están en la misma componente conexa. Así, en cada paso controlamos esta condición usando la estructura de datos del Union Find. El grado queda claramente controlado usando un vector.

La función principal del algoritmo es la siguiente:

```
typedef pair<int,int> Edge;
```

```
void TSP_greedy_v3(int origin, int n, City cities[], vector<int> & ans) {
    vector<Edge> edges;
    edges.reserve((n*(n-1)/2));
    for(int i = 0; i < n; ++i){
        for(int j = i+1; j < n; ++j){
            edges.push_back({i,j});
        }
    }

    sort(edges.begin(),edges.end(),[&cities](Edge& a, Edge & b){
        return cities[a.first] - cities[a.second] < cities[b.first] - cities[b.second
    ];
    });

    int n_edges = edges.size();
    int ind_edge = 0;
    int n_sets = n;
    vector<int> parent(n),rank(n),degree(n,0);

    // Initialize connected components
    for(int i = 0; i < n; ++i){
        make_set(i,parent,rank);
    }

    vector<vector<int>> tree(n); // Adjacency list
    while(n_sets > 1){
        Edge edge = edges[ind_edge++];
        int u = edge.first;
        int v = edge.second;
        if(degree[u] < 2 && degree[v] < 2){
            if(union_sets(u,v,parent,rank)){
                --n_sets;
                ++degree[u];
                ++degree[v];
                tree[u].push_back(v);
                tree[v].push_back(u);
            }
        }
    }
```

```
39
40      dfs(origin,-1,tree,ans);
41  }
```

La primera parte (hata la línea 17) es clara: inicializamos un vector de aristas y lo ordenamos de menor a mayor en cuanto a longitudes (distancias entre ciudades: usamos el operador diferencia del tipo City). Los vectores `parent` `rank` `degree` guardan respectivamente: un índice al nodo representante de la componente conexa del nodo que indexa, el rango del nodo que indexa (que es la profundiad aproximada del árbol que forma la componente conexa de la cual es representante), y el grado del nodo que indexa. El uso de `rank` en este algoritmo tiene como único objetivo agilizar la búsqueda de representantes como veremos en la función `union_sets`. Nótese también que la estrucutura donde almacenaremos finalmente el grafo para dar una solución es en un `vector<vector<int>>tree`, en forma de lista de adyacencia. La primera función que usamos, inicializa una componente conexa por nodo, para posteriormente ir añadiendo aristas y conectando componentes como corresponda hasta quedar solo con una:

```
1  void make_set(int node,vector<int> & parent, vector<int> & rank){
2      parent[node] = node;
3      rank[node] = 0;
4  }
```

En el bucle tan solo insertamos en el grafo la siguiente arista más corta controlando el grado y que no se generen ciclos, ahí entra la función `union_sets` :

```
1  int find_set(int node, vector<int> & parent){
2      if(parent[node] == node) return node;
3      return parent[node] = find_set(parent[node],parent);
4  }
5
6  bool union_sets(int u, int v, vector<int> & parent, vector<int> & rank){
7      int pu = find_set(u,parent);
8      int pv = find_set(v,parent);
9
10     if(pu == pv) // Nothing to join here
11         return false;
12
13     if(rank[pu] < rank[pv])
14         swap(pu,pv);
15     parent[pv] = pu;
16     if(rank[pu] == rank[pv])
17         ++rank[pu];
18     return true;
19 }
```

La función `find_set` no hace más que llamarse recursivamente para encontrar el representante de la componente conexa de un nodo `node`, consultando quién es el padre del nodo que se está procesando en cada paso hasta encontrar uno que sea su propio padre: el representante. El hecho de que la asignación se haga en la propia llamada en la línea 3 es un leve ajuste para ahorrar reiteradas asignaciones de los padres tras cada actualización: si tenemos un nodo `v` a cuyo padre `pv` cambiamos el padre, también estamos cambiando indirectamente el padre de `v`, pero este cambio no se refleja en el vector `parent[v]` hasta que no se pregunta por él, cuando ya es necesario.

La función primero encuentra los padres (representantes de las respectivas componentes conexas) de los nodos que unimos, y antes de nada comprueba si están en la misma componente conexa (si es el caso, al unir se generarían ciclos, y entonces no lo hacemos). Ahora bien, a la hora de escoger cuál será el nuevo representante de la componente conexa, consultamos el `rank` de ambos nodos, y enganchamos (ponemos como padre) al que tenga mayor `rank`, pues de lo contrario el árbol de parentesco que representa las componentes conexas resultaría innecesariamente profundo, lo que supondría llamadas extra en la función `find_set`.

Hecho esto, unimos como se espeararía añadiendo ambos nodos a la lista de adyacencia, para terminar una vez tengamos solo una componente conexa (`n_sets == 1`). Es en este momento que llamamos a la función `dfs` para construir el camino usando el árbol de parentesco:

```
1  void dfs(int node, int parent,vector<vector<int>> & tree, vector<int> & ans){
2      ans.push_back(node); // Add the node to the answer
3      for(int neighbour : tree[node]){ // Visit the node's neighbours
```

```
        // If the neighbour is not the node's parent, then it means it hasn't been visited
        // (since the given graph is a tree) so it is visited
        if(neighbour != parent){
            dfs(neighbour,node,tree,ans);
        }
    }
}
```

Esta función tan solo recorre el árbol (única componente conexa) en preorden, añadiendo en cada paso el nodo no visitado (cada uno los vecinos que no sea el padre del nodo que se esté procesando) a un vector de índices sobre el vector de ciudades que será nuestra solución (nótese que en este vector el índice de la primera ciudad no está repetido, es decir, que el ciclo no está cerrado: el trato del último paso se delega en la impresión de resultados con la función `printCycle()` ya explicada en problemas anteriores).

7.4.2. Análisis de eficiencia

Eficiencia teórica

Procedemos al análisis de la eficiencia teórica de la función `TSP_greedy_v3`. Primero nos econtramos con una inicalización del vector de aristas, que ya es de coste $O(n^2)$. Tras esto, notamos que la función `sort` aplicada sobre las aristas ya es de orden $O(n^4)$ en el peor caso, pues sabemos que `sort` utiliza internamente el algoritmo `quick_sort` y tenemos un total de $\frac{n^2-n}{2}$ aristas. Sin embargo, en la práctica, `quick_sort` se comporta como un $O(n\log(n))$, y entonces el orden de nuestra ordenación de aristas será de $O(2n^2\log(n))$.

Ahora bien, vemos que en el bucle `while` en el que vamos añadiendo aristas a nuestra solución (que se ejecuta del orden de n^2 veces) se llama a la función `union_sets`. Veamos que esta función es del orden $O(n)$ para concluir que nuestro algoritmo es del orden $O(n^3)$.

```
int find_set(int node, vector<int> & parent){
    if(parent[node] == node) return node;
    return parent[node] = find_set(parent[node],parent);
}

bool union_sets(int u, int v, vector<int> & parent, vector<int> & rank){
    int pu = find_set(u,parent);
    int pv = find_set(v,parent);

    if(pu == pv) // Nothing to join here
        return false;

    if(rank[pu] < rank[pv])
        swap(pu,pv);
    parent[pv] = pu;
    if(rank[pu] == rank[pv])
        ++rank[pu];
    return true;
}
```

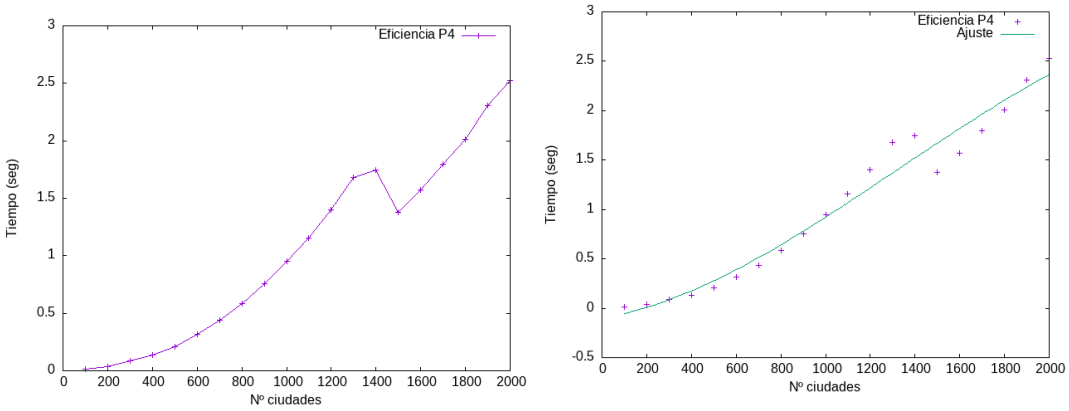
Como vemos, salvo asignaciones y comprobaciones $O(1)$ la función es $O(n)$ por su llamada a `find_set` que en el peor de los casos tiene que dar del orden de $O(n)$ pasos para encontrar la ciudad *padre* o representante de la componente conexa. Por tanto, concluimos por la regla del máximo, que la eficiencia teórica de nuestro algoritmo es $O(n^3)$.

Eficiencias empírica e híbrida

Procedemos ahora a realizar un estudio de la eficiencia empírica del algoritmo. Los datos de tiempos desde $n = 100$ hasta $n = 2000$ con saltos de 100 son los siguientes:

```
100 0.006324
200 0.030369
300 0.069987
400 0.134143
500 0.211006
600 0.318191
```

```
700 0.436856
800 0.585133
900 0.756034
1000 0.952759
1100 1.15567
1200 1.40932
1300 1.6817
1400 1.63723
1500 1.38287
1600 1.57021
1700 1.79021
1800 2.01373
1900 2.30574
2000 2.55001
```



(a) Regresión por mínimos cuadrados

Figura 15: Eficiencias empírica e híbrida de la versión 3 del TSP

Observamos que alrededor de la zona de 1400 ciudades hay importantes saltos en los tiempos, quizá por el hardware sobre el que estamos ejecutando (en cuanto a reservas de memoria y demás gestiones de recursos) o por, incluso, que en esas instancias concretas aparezca el caso límite del `quick_sort`. Para obtener las constantes ocultas que determinan la eficiencia del algoritmo hemos usado la herramienta `gnuplot` y hemos obtenido los siguientes resultados:

```
function used for fitting: f(x)
        f(x)=a*x*x*x + b*x*x + c*x + d
fitted parameters initialized with current variable values

iter      chisq       delta/lim  lambda   a             b             c             d
   0 2.1670269281e+20   0.00e+00  1.64e+09    1.000000e+00   1.000000e+00   1.000000e+00   1.000000e+00
  13 4.3070885733e-01  -1.38e-05  1.64e-04   -2.169039e-10   8.606762e-07   3.779800e-04  -9.965655e-02

After 13 iterations the fit converged.
final sum of squares of residuals : 0.430709
rel. change during last iteration : -1.37957e-10

degrees of freedom    (FIT_NDF)                      : 16
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.164071
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.0269193

Final set of parameters            Asymptotic Standard Error
=======================            =========================
a               = -2.16904e-10     +/- 2.47e-10     (113.9%)
b               = 8.60676e-07      +/- 7.878e-07    (91.53%)
c               = 0.00037798       +/- 0.0007211    (190.8%)
d               = -0.0996565       +/- 0.1792       (179.8%)

correlation matrix of the fit parameters:
                a      b      c      d
a               1.000
b              -0.988  1.000
c               0.929 -0.974  1.000
d              -0.732  0.807 -0.905  1.000
```

Es decir, la función representada en la gráfica 15a que aproxima la eficiencia de nuestra función es:

$$f(x) = -2.17 \cdot 10^{-10}x^3 + 8.607 \cdot 10^{-7}x^2 + 3.78 \cdot 10^{-4}x - 0.0997$$

7.4.3. Comparativa de eficiencia

Analicemos la eficiencia de las tres versiones proporcionadas para dar una solución aproximada al problema del viajante del comercio.

Eficiencia teórica

Hemos visto previamente que las eficiencias teóricas son las de la tabla (4).

Versión 1	$O(n^2)$
Versión 2	$O(n^2)$
Versión 3	$O(n^3)$

Cuadro 4: Comparación eficiencia teórica TSP

Tenemos pues que las versiones 1 y 2 tienen un orden de eficiencia **cuadrático** y son por tanto más rápidas que la versión 3, que es de orden **cúbico**.

Eficiencia empírica

Para comparar la eficiencia de las tres versiones empíricamente, los tres códigos han sido ejecutados para los mismos casos de prueba, cuyos tamaños van desde $n = 100$ hasta $n = 1500$ cada 100.

Los tiempos obtenidos se pueden ver en la tabla (5) y las gráficas comparativas en la figura (16).

Num ciudades (n)	Tiempo (seg) V1	Tiempo (seg) V2	Tiempo (seg) V3
100	0.000103	0.000589	0.002734
200	0.000363	0.002207	0.012647
300	0.000756	0.005126	0.027018
400	0.00161	0.008774	0.051717
500	0.002085	0.012729	0.081984
600	0.002915	0.018929	0.124607
700	0.00393	0.025662	0.170133
800	0.005131	0.033922	0.224846
900	0.006374	0.044317	0.289857
1000	0.007957	0.054267	0.364425
1100	0.009524	0.067381	0.443214
1200	0.01134	0.082201	0.536435
1300	0.013196	0.097048	0.641631
1400	0.01524	0.109277	0.743729
1500	0.017893	0.125745	0.858521

Cuadro 5: Tiempos de las tres versiones del TSP

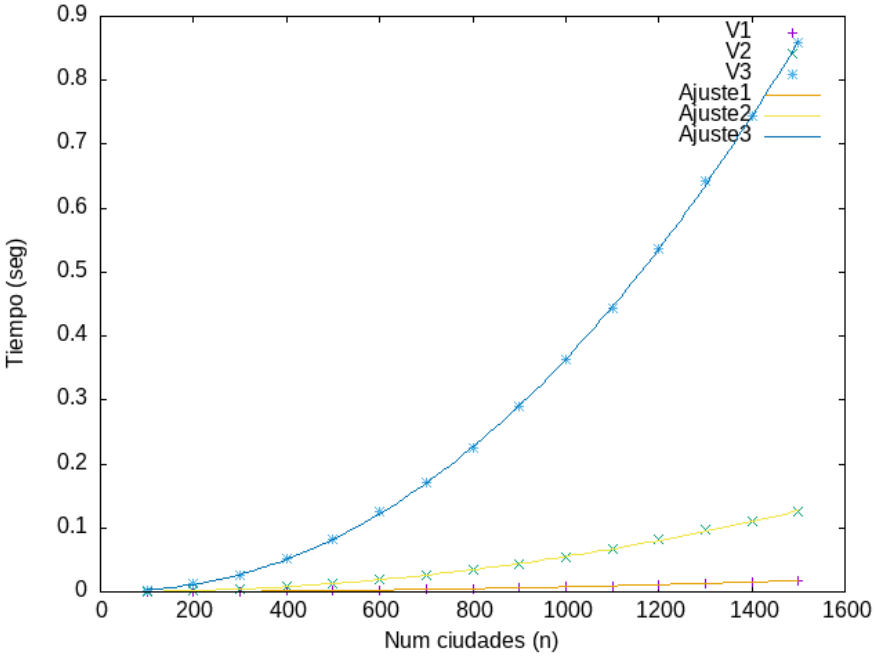


Figura 17: Comparativa eficiencia híbrida TSP

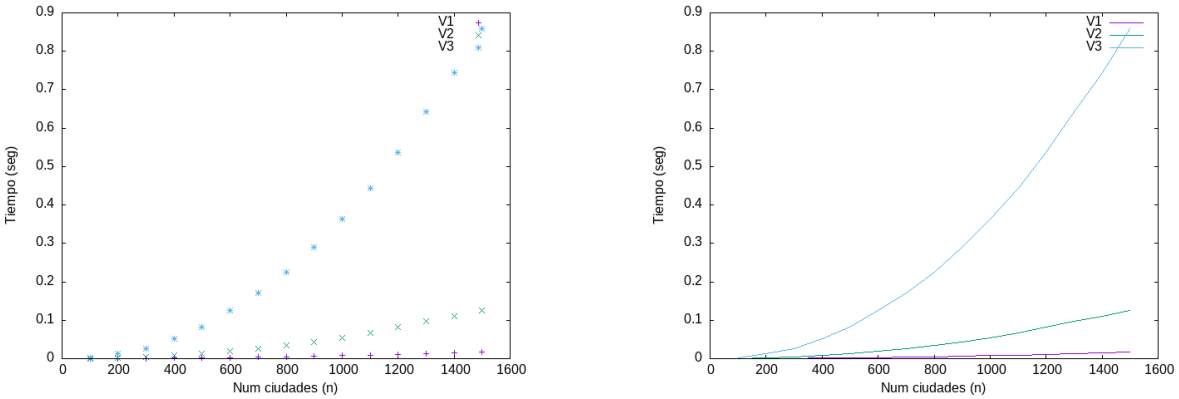


Figura 16: Gráficas comparativas eficiencia empírica TSP

Podemos visualizar claramente que la versión 1 (closest neighbour) es la más eficiente de las tres a pesar de tener el mismo orden de eficiencia que la versión 2 (prim). Además salta a la vista que la versión 3 (kruskal) tiene un orden de eficiencia superior al de las otras dos.

Eficiencia híbrida

Finalmente presentamos una gráfica comparativa con las curvas de eficiencia de cada versión adecuadamente ajustadas en la figura (17).

Aquí confirmamos las conclusiones obtenidas en la eficiencia empírica.

7.4.4. Comparativa de la calidad de las soluciones

Dado que el problema del viajante de comercio es un problema NP-Díficil, no se nos es posible aportar una solución que tarde tiempo polinómico en resolver el problema. Por este motivo, todos nuestros heurísticos se limitan a dar una solución **aproximada** a la solución verdaderamente óptima del problema en cuestión.

Es por esto no solo nos interesa escoger un algoritmo eficiente, sino que también uno que nos proporcione una solución relativamente "óptima". Comparamos en la figura (18) las soluciones aportadas por nuestro algoritmos.

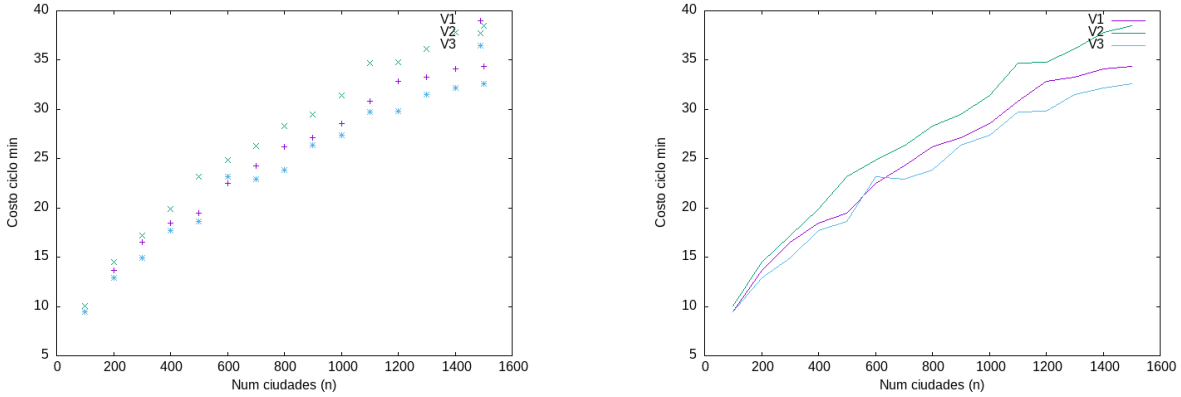


Figura 18: Gráficas comparativas eficiencia empírica TSP

De ellas observamos que, las soluciones dadas por los tres algoritmos no difieren significativamente para valores pequeños de n. El algoritmo que proporciona una solución más óptima es la versión 3 (kruskal). Después iría la versión 1 (closest neighbour) y por último la versión 2 (prim).

8. Conclusiones

A lo largo de esta práctica, hemos explorado y comprobado la efectividad y aplicabilidad del paradigma de **algoritmos voraces** en la resolución de problemas complejos de computación, complementando así los contenidos teóricos de nuestra formación.

La experiencia nos ha permitido no solo aplicar concretamente los principios teóricos, sino también observar las ventajas en términos de eficiencia que esta estrategia algorítmica ofrece. Durante el proceso, hemos enfrentado retos que destacan la importancia de la estructura de los datos de entrada y las particularidades de cada implementación en los resultados obtenidos, validando que los *algoritmos voraces* son especialmente útiles para ciertos tipos de problemas que permiten un entendimiento favorable

En particular, hemos notado la manifestación de conceptos teóricos clave en la construcción de nuestros algoritmos, tales como:

- La importancia de demostrar correctamente la validez y viabilidad de nuestros algoritmos voraces
- La observación de que las diferencias en los tiempos de ejecución al cambiar de plataforma (ya sea por hardware o software) suelen influir principalmente en una constante de proporcionalidad, pero no alteran la eficiencia comparativa entre diferentes enfoques de solución.

Esta práctica ha resultado ser una oportunidad invaluable para aplicar la teoría en un contexto práctico, enfrentando desafíos reales y desarrollando un entendimiento más profundo de los principios de diseño de algoritmos. Consideramos que esta experiencia ha sido exitosa, enriqueciendo nuestra preparación para enfrentar futuras problemáticas en nuestro camino académico y profesional, y proporcionando herramientas y perspectivas esenciales para identificar casos en los cuales la estrategia de **algoritmos voraces** es adecuada y cómo implementarla efectivamente.