

ALGORITMOS GREEDY

Problema de las cintas

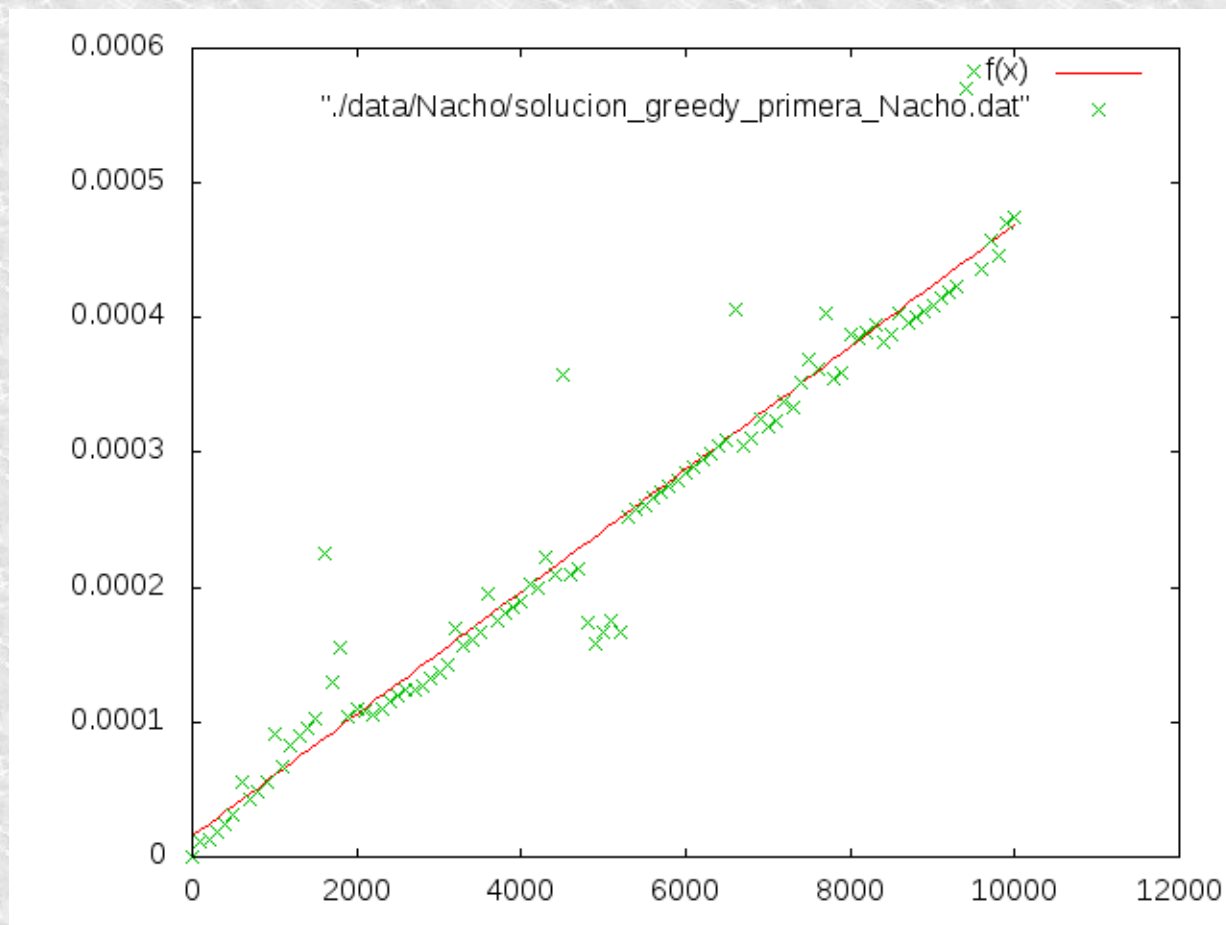
- Cada cinta tiene una longitud y un peso.
- Minimizar tiempo de acceso.
- $T = \sum p_i \sum l_j$

Algoritmo 1

```
template <typename T>
void swapFollowing(size_t i, vector<T>& v){
    T aux;
    aux=v[i+1];
    v[i+1] = v[i];
    v[i] = aux;
}

void SolucionGreedyPrimera(vector<int>& tam, vector<double>& pesos)
{
    for(size_t i = 0; i < tam.size()-1;++i)
    {
        if(tam[i]*pesos[i]>=tam[i+1]*pesos[i+1])
        {
            swapFollowing(i,tam);
            swapFollowing(i,pesos);
        }
    }
}
```

Algoritmo 1



Suma al cuadrado de los residuos = $1.2507e-07$

Teorema de optimalidad

Si $\pi_1 \cdot l_1 \leq \pi_2 \cdot l_2 \leq \dots \leq \pi_n \cdot l_n$
entonces la ordenación $i_j = j$, $1 \leq j \leq n$
minimiza $\sum_{k=1}^n \pi_{i_k} \sum_{j=i}^k l_{i_j}$ sobre
todas las posibles permutaciones de i_j .

Algoritmo 2

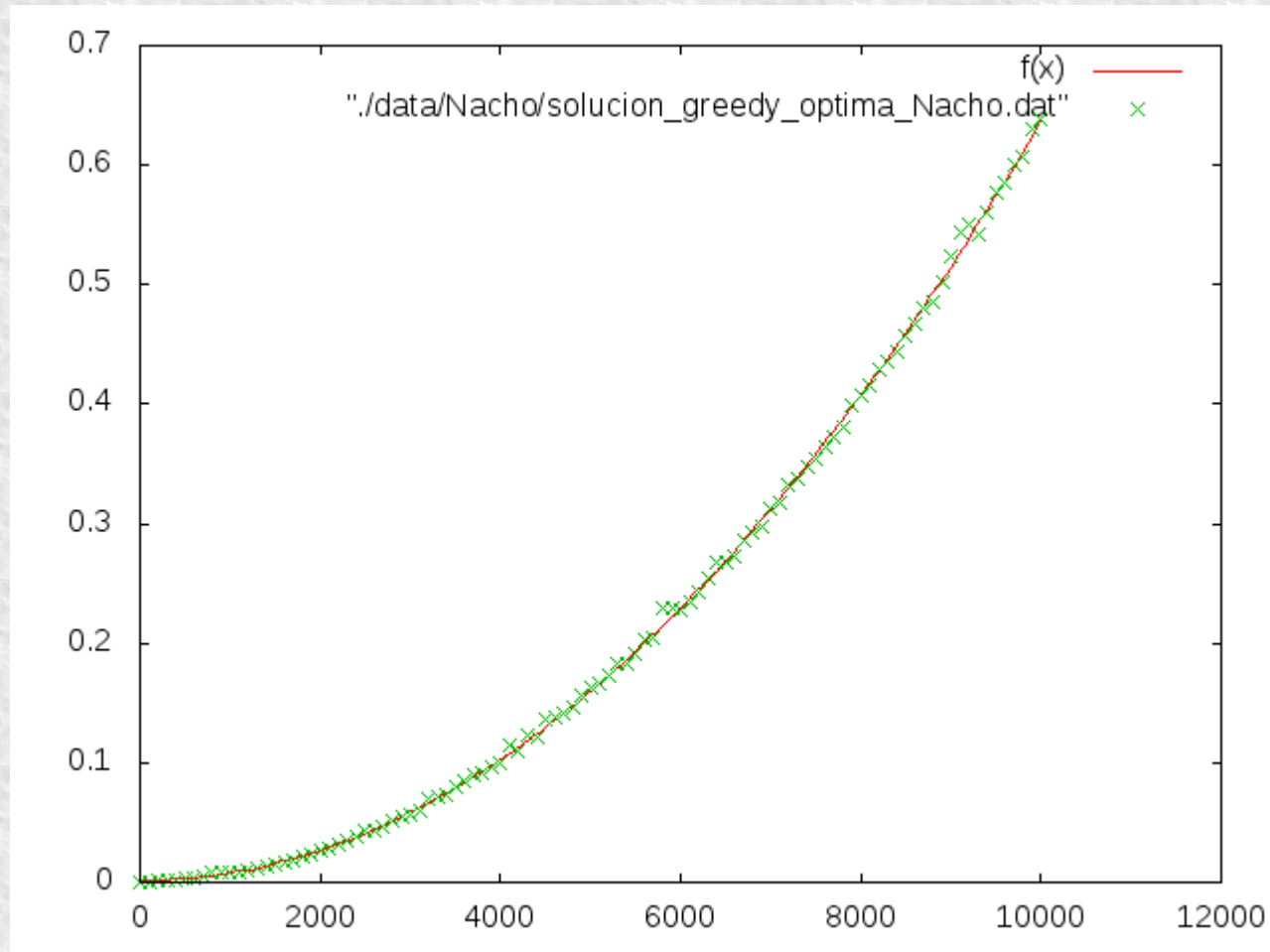
```
void solucionGreedy(vector<int>& tam, vector<double>& pesos)
{
    std::vector<int> aux1;
    std::vector<double> aux2;

    while (! tam.empty()) {
        size_t index = 0;
        for (size_t i = 0; i < tam.size(); i++) {
            if(tam[i]*pesos[i]<tam[index]*pesos[index])
                index = i;
        }

        aux1.push_back(tam[index]);
        aux2.push_back(pesos[index]);

        tam.erase(tam.begin()+index);
        pesos.erase(pesos.begin()+index);
    }
    tam = aux1;
    pesos = aux2;
}
```

Algoritmo 2



Suma al cuadrado de los residuos = 0.00154463

Problema del viajante de comercio

- Tenemos un grafo con las ciudades.
- Queremos recorrerlas todas minimizando la distancia y volviendo a la ciudad de origen.
- Datos de prueba kroA100.tsp

Clase TSP

```
#ifndef TSP_H_INCLUDED
#define TSP_H_INCLUDED

#include <fstream>
#include <vector>
#include <utility>
#include <vector>

struct City{
    int ciudad;
    double coord_x;
    double coord_y;
    City& operator=(const City&);
};

using namespace std;

class TSP {

private:
    int nCiudades;
    vector<City> ciudades;
    vector<City>::iterator menorDistancia(City c, vector<City>& candidatos);
    void find_max_edge(vector<City>& l, vector<City>::iterator&);
    void find_nearest_point(const vector<City>& orig, vector<City>::iterator& it, vector<City>& searching, vector<City>::iterator&);

public:
    TSP(char* cadena);

    pair<double, vector<City>::iterator> DevuelveMenorDistancia(City c, vector<City>& candidatos);
    void Dijkstra(vector<City>& res);
    void TSP_vecino_mas_cercano(vector<City>& vec);
    void TSP_triangles(vector<City>& solucion);
    void TSP_RandomSwap(int n, vector<City>& solucion);
    void TSP_WriteBack(ofstream& os, vector<City> sol);

};
#endif // TSP_H_INCLUDED
```

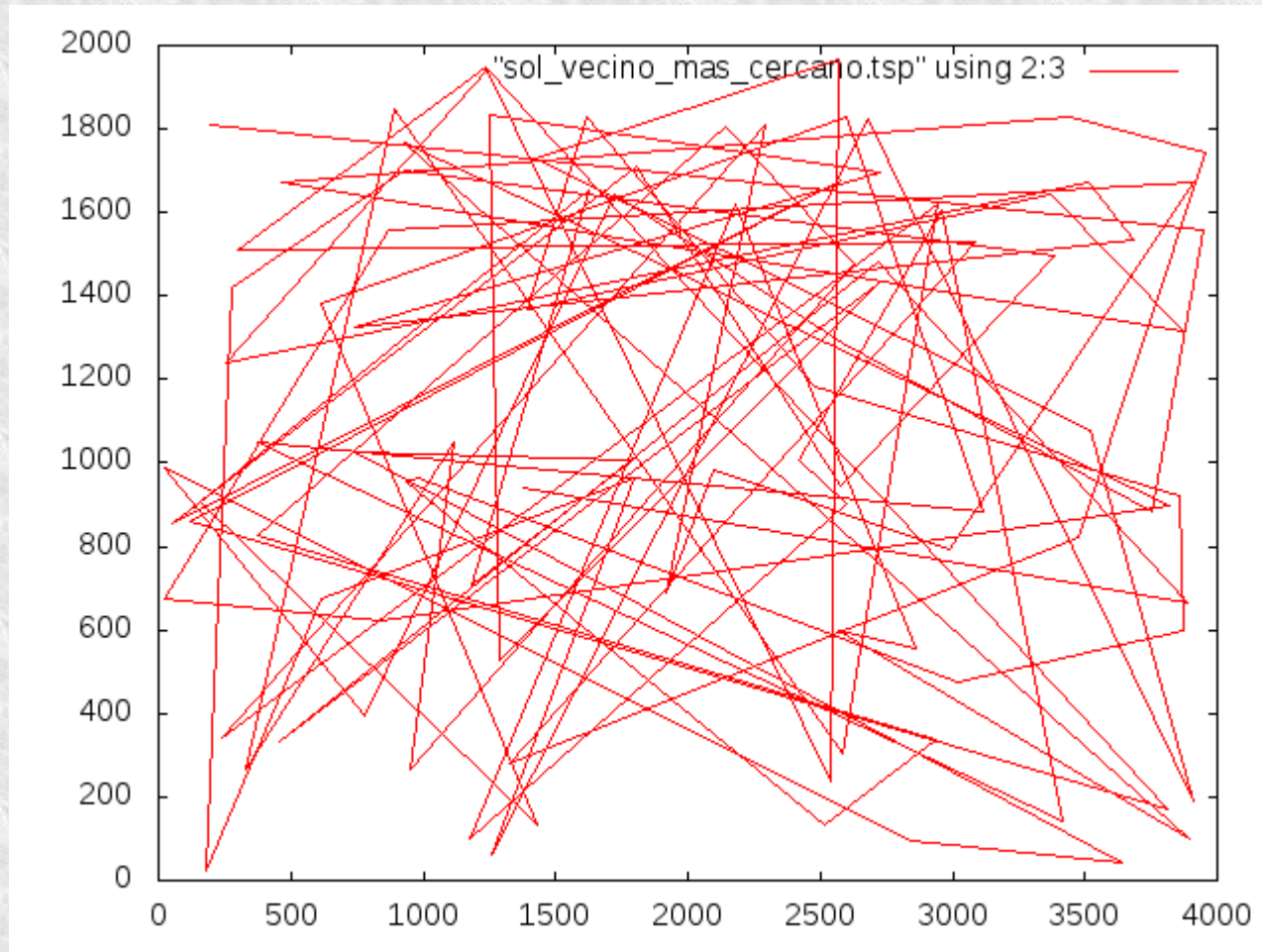
Vecino más cercano

```
void TSP::TSP_vecino_mas_cercano(vector<City>& solucion)
{
    solucion.push_back(ciudades[0]);

    vector<City> candidatos(ciudades);
    candidatos.erase(candidatos.begin());

    while((int)solucion.size() < nCiudades)
    {
        vector<City>::iterator it = menorDistancia(solucion.at(solucion.size()-1), candidatos);
        solucion.push_back(*it);
        candidatos.erase(it);
    }
}
```

Vecino más cercano



Suma = 166422, Suma óptima = 21282

Algoritmo Triángulo

```
void TSP::TSP_triangles(vector<City>& solucion){

    vector<City> candidatos(ciudades);

    vector<City>::iterator minb = candidatos.begin();
    for (vector<City>::iterator it = candidatos.begin(); it != candidatos.end(); it++) {
        if ((*minb).coord_x > (*it).coord_x)
            minb = it;
    }

    solucion.push_back(*minb);
    candidatos.erase(minb);

    vector<City>::iterator maxb = candidatos.begin();
    for (vector<City>::iterator it = candidatos.begin(); it != candidatos.end(); it++) {
        if ((*maxb).coord_x < (*it).coord_x)
            maxb = it;
    }

    solucion.push_back(*maxb);
    candidatos.erase(maxb);

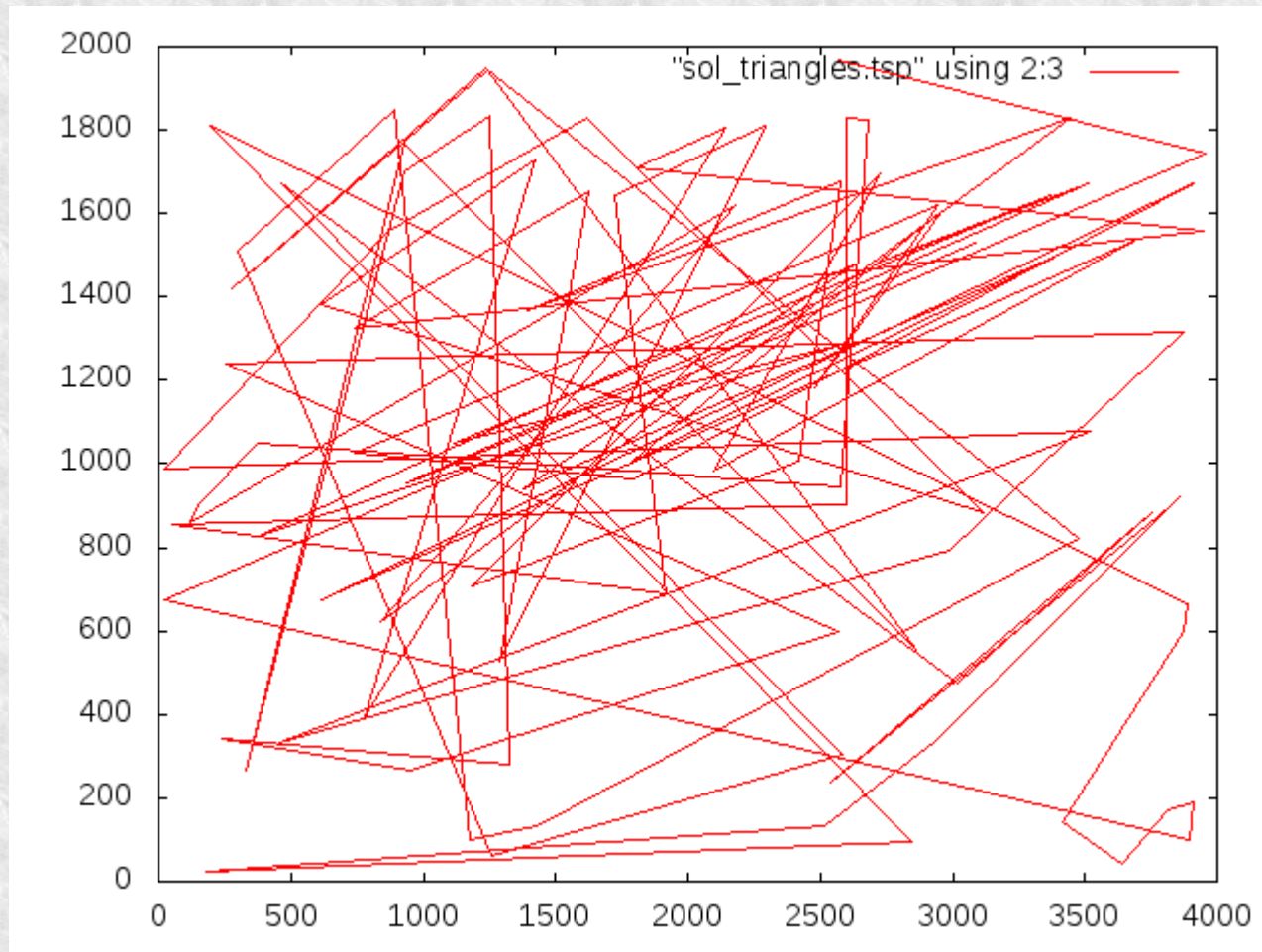
    vector<City>::iterator maxh = candidatos.begin();
    for (vector<City>::iterator it = candidatos.begin(); it != candidatos.end(); it++) {
        if ((*maxh).coord_y < (*it).coord_y)
            maxh = it;
    }

    solucion.push_back(*maxh);
    candidatos.erase(maxh);

    City aux;
    while (!candidatos.empty()){
        vector<City>::iterator mayor_lado, nearest;
        find_max_edge(solucion, mayor_lado);
        find_nearest_point(solucion, mayor_lado, candidatos, nearest);
        aux = *nearest;
        solucion.insert(mayor_lado, aux);
        candidatos.erase(nearest);
    }

}
```


Algoritmo Triángulo

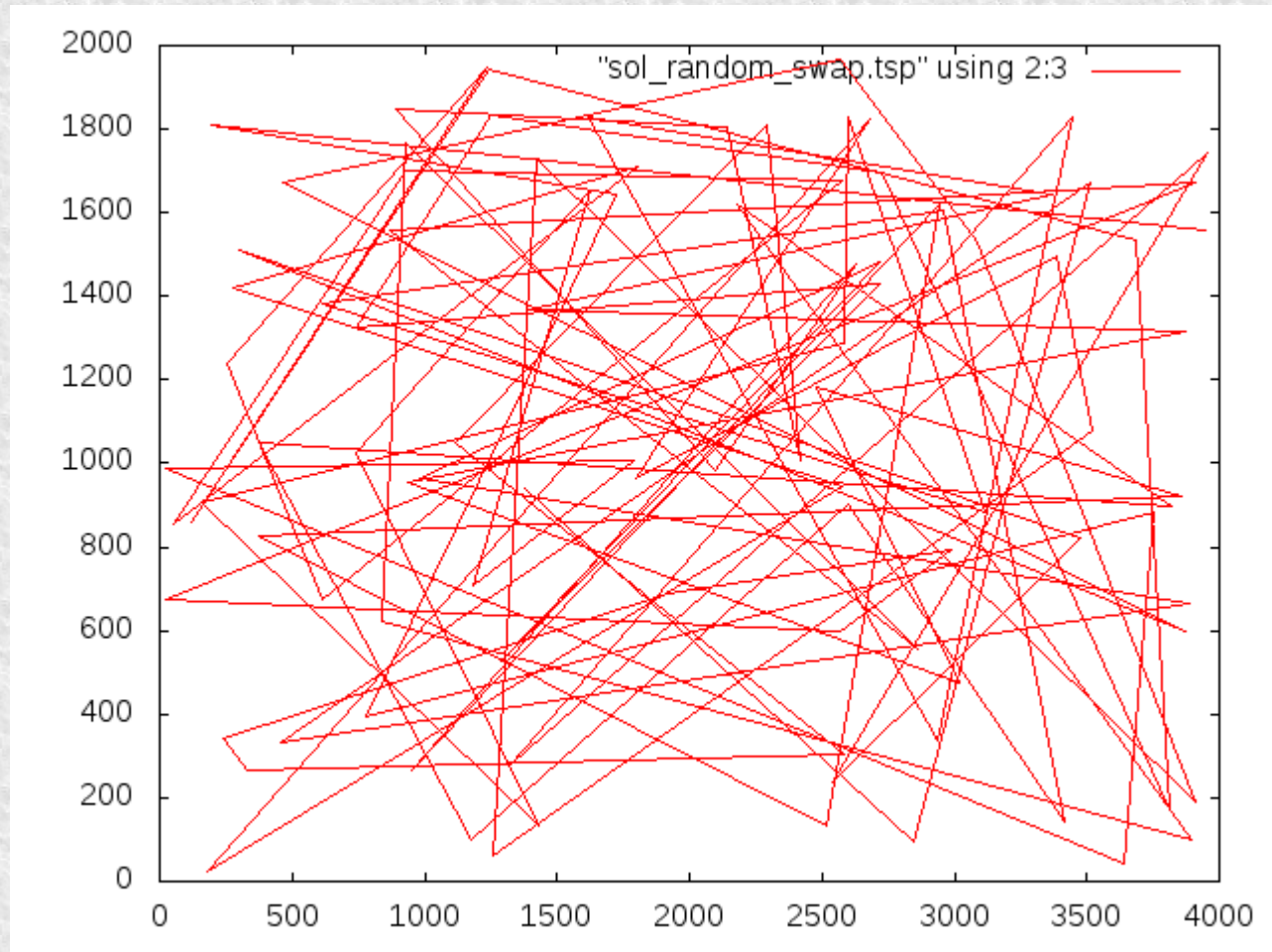


Suma = 150742, Suma óptima = 21282

Algoritmo Random Path

```
void TSP::TSP_RandomSwap(int n, vector<City>& solucion){
    City aux1,aux2,aux3;
    srand(time(0));
    int j,k;
    double distant,distlueg;
    for(vector<City>::iterator it = ciudades.begin(); it!=ciudades.end();it++){
        solucion.push_back(*it);
    }
    for(int i = 0; i < n; i++){
        j = nCiudades*rand()/(RAND_MAX + 1.0);
        do{
            k = (nCiudades)*rand()/(RAND_MAX + 1.0);
        }while((j > nCiudades - 3 && k < 3)||((k > nCiudades - 3 && j < 3)));
        j = j%nCiudades;
        k = k%nCiudades;
        distlueg = distant = 0;
        for(vector<City>::iterator it = ciudades.begin(); it!=ciudades.end()-1;it++){
            vector<City>::iterator it2 = it;
            ++it2;
            distant += distancia((*it).coord_x,(*it2).coord_x,(*it).coord_y,(*it2).coord_y);
        }
        distant += distancia(ciudades[0].coord_x,ciudades[nCiudades-1].coord_x,ciudades[0].coord_y,ciudades[nCiudades-1].coord_y);
        aux1 = ciudades[j];
        aux2 = ciudades[j+1];
        aux3 = ciudades[j+2];
        ciudades[j] = ciudades[k];
        ciudades[j+1] = ciudades[k+1];
        ciudades[j+2] = ciudades[k+2];
        ciudades[k] = aux1;
        ciudades[k+1] = aux2;
        ciudades[k+2] = aux3;
        for(vector<City>::iterator it = ciudades.begin(); it!=ciudades.end()-1;it++){
            vector<City>::iterator it2 = it;
            ++it2;
            distlueg += distancia((*it).coord_x,(*it2).coord_x,(*it).coord_y,(*it2).coord_y);
        }
        distlueg += distancia(ciudades[0].coord_x,ciudades[nCiudades-1].coord_x,ciudades[0].coord_y,ciudades[nCiudades-1].coord_y);
        if(distant < distlueg){
            aux1 = ciudades[j];
            aux2 = ciudades[j+1];
            aux3 = ciudades[j+2];
            ciudades[j] = ciudades[k];
            ciudades[j+1] = ciudades[k+1];
            ciudades[j+2] = ciudades[k+2];
            ciudades[k] = aux1;
            ciudades[k+1] = aux2;
            ciudades[k+2] = aux3;
        }
    }
}
```

Algoritmo Random Path



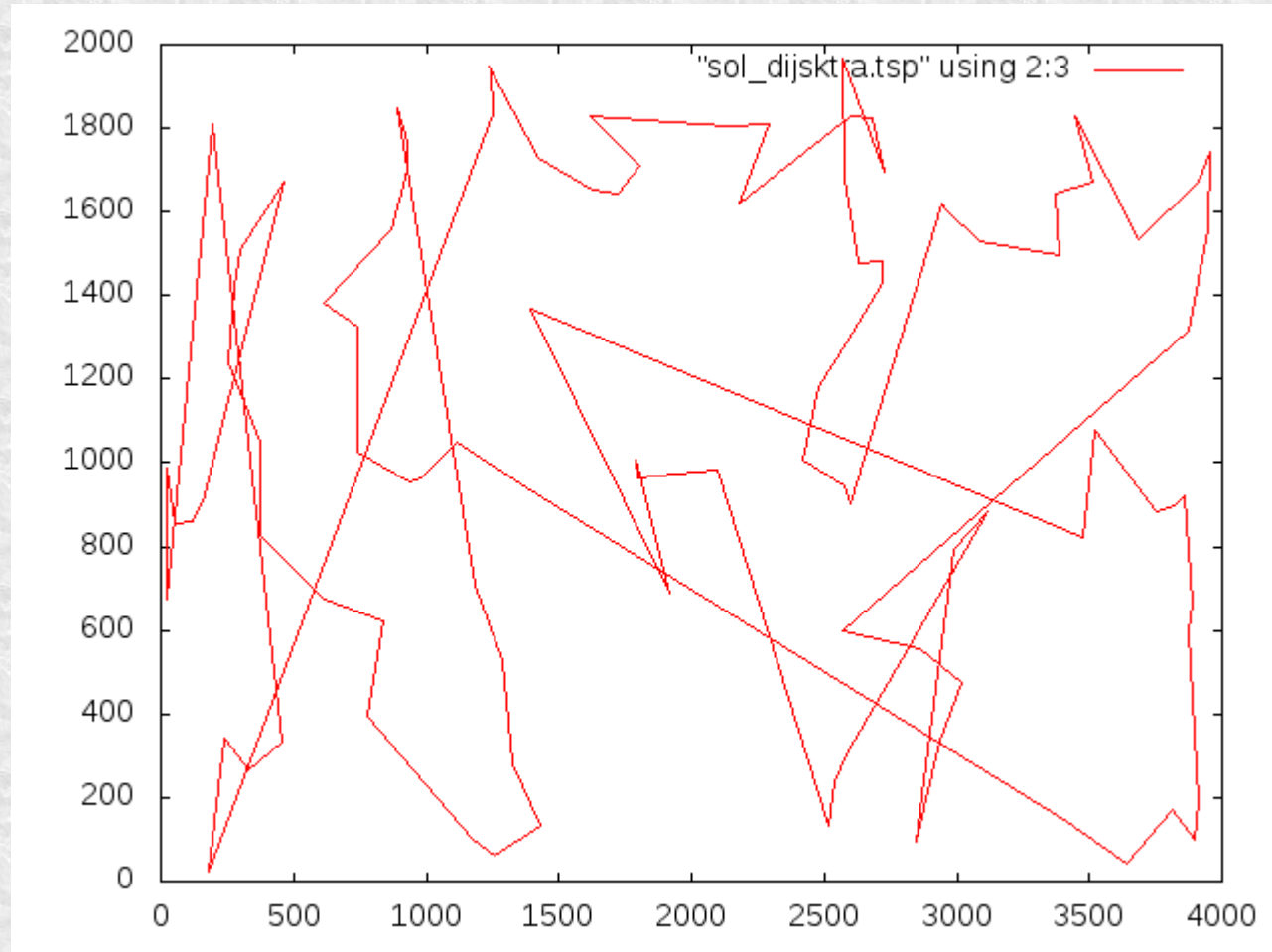
Suma = 188750, Suma óptima = 21282

Algoritmo Dijkstra

```
void TSP::Dijkstra(vector<City>& res)
{
    vector<City> candidatos(ciudades);
    res.push_back(candidatos[0]);
    candidatos.erase(candidatos.begin());

    while(candidatos.size() != 0)
    {
        double dist = INT_MAX;
        vector<City>::iterator min_dist;
        for(vector<City>::iterator it = res.begin(); it != res.end(); ++it)
        {
            pair<double, vector<City>::iterator> f = DevuelveMenorDistancia(*it, candidatos);
            if(dist > f.first)
            {
                min_dist = f.second;
                dist = f.first;
            }
        }
        res.push_back(*min_dist);
        candidatos.erase(min_dist);
    }
}
```


Algoritmo Dijkstra



Suma = 30952.4, Suma óptima = 21282