University of Cádiz

School of Engineering

Assignment 5: Solving Complex Problems

Francisco Chanivet Sánchez

3rd January 2022

Contents

Tables

Figures

1 What is Simulated Annealing?

Simulated Annealing is *a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a metaheuristic to approximate global optimization in a large search space for an optimization problem.*
The name of the algorithm comes from annealing in metallurgy, a technique where are involving the heat and controlled cooling of a material to alter its physical properties.

Figure 1: Example of annealing in real life

This algorithm belongs to **metaheuristics algorithm group**, specifically **trajectory-based** algorithms.

For this assingment, the main objective to achieve is **obtain the minimum cost** for the **Travelling Salesman Problem**.

2 Proposed problem

As we said before, we are going to apply Simulated Annealing to the Travelling Salesman Problem. Then, we proposed problem is the next:

Given a list of cities and the distances between them, what is the shortest route that starts in a given city, visits every other city exactly once and returns to the origin?

Then, as the problem said, we want to visit every city once and returns to the origin. After, we use the **Simulated Annealing** for obtaining the minimum path, given the coordenates of the cities. For reach the solution, we swap the order of the path of the different cities and the minimum path will be the solution of our problem.

3 Implementation

3.1 How can we read a TSPlib format?

The **TSPlib** is a file format prepared for **Travelling Salesman Problem**, where we have:

- **Name** of the file.
- **Dimension** of the problem.
- **Edge weight type**, in our case **EUC_2D**.
- **Node and coordenates of the city**.

Then, if we have this format of cities, we are going to make a class called **City** for this purpose, because if we save each coordenate in a matrix, we obtain a big time complexity then we have opted for making a class for this purpose.

```
class City{
        public:
                City(unsigned int i, double cx = 0, double cy = 0):id_(i), x_(cx), y_(cy){}
                inline unsigned int id() const{return id_;}
                inline double coord_x() const{return x_;}
                inline double coord_y() const{return y_;}
        private:
                unsigned int id_;
                double x_,y_;
};

```

Then, once we have got the structure of each city, we build a function to translte from TSPlib format to our program. So, we have implemented the next function

```
vector<City> translatorTSP(string file_name){

        string path = "file_instances/";
        file_name += ".tsp";
        path += file_name;

        ifstream file(path);
        string line;
        vector<City> cities;
        unsigned int id_c;
        double x, y;
        while(getline(file,line) && line != "EOF"){
                if(sscanf(line.c_str(),"%u %lf %lf",&id_c,&x,&y) == 3 || sscanf(line.c_str(),"\t%u
                        cities.push_back(City(id_c,x,y));
                }
        }

        file.close();
```

As we can see, we only save the node, the coordenate x and y in a vector of City called **cities**. We do not use the dimension given in the file because we use the function **push_back** implementated in the **vector** library. Also, each city must have three numbers in the line, then there is not any problem to save them. Once we have the vector of cities, we return it.

3.2 Generate an Initial State

When we are going to start to execute the **Simulated Annealing**, we need to generate an initial state for the problem which it can work. Then, for this purpose, we have the next function:

```
vector<City> genInitial(const vector<City>& nodeCities, City origin){

        vector<City> initial = nodeCities;

        auto it = find(initial.begin(),initial.end(),origin);

        initial.erase(it);

        random_shuffle(initial.begin(),initial.end());

        initial[0] = origin;
        initial.push_back(origin);

        return initial;
}

```

At first, we are going to find the origin city for removing it from the vector because we are going to permutate the cities generating a random initial state and we do not have interest on permutating the origin city. Then we pushed the origin city before doing the permutation and once we have the vector permutated, origin city indicating to the program that we want to go since the origin and finish the path in the origin city.
After explaining the generator of initial states, we go to he beginning of the algorithm where we declare the values for the temperature and more things.

3.3 Beginning of the algorithm

At the start of the algorithm, we are going to declare the values of the **temperature** and the **minimum temperature**. As we quoted before, the **Simulated Annealing algorithm** simulates an annealing process, so we need this two parameters for obtaining the best state. We choose for **temperature** value **9.32** and for the **minimum temperature 0.543** because we can obtain better cost with more frequency.

```
        double T = 9.32, T_min = 0.543;
        size_t city = 1, maxAttemps = 0;
        double deltaE,p;

        srand(time(NULL));

        vector<City> currentState = genInitial(cities,origin); //We generate the initial state
        vector<City> newState;
        double deltaEaux = numeric_limits<double>::max();
```

Also, we declare variable **city** because we are going to save the position of the vector of state which we are going to swap with a random position **except the first and the last element of the vector**. After generating the initial state, we can see that we define variables **deltaE** and **deltaEaux**, where **deltaE** is the **difference between the cost of the new state and the current state**. Then we save the **maximum difference** in **deltaEaux** for obtaining the best possible solution for this problem.

3.4 Generate Sucessor State

Then, once we begin to search an optimal solution, we are going to need a function with the intention of generating succesor states. Then, we implementing this function with the purpose that we want.

```
vector<City> genSuc(vector<City> currentState, size_t posCity){
        vector<City> newState = currentState;
        size_t pos;
        int direccion = rand() % 2;

        if(posCity == currentState.size() - 2 || (direccion == 0 && posCity > 7)){
                pos = 1 + rand() % (posCity - 1);
        }
        else{
                pos = (posCity + 1) + rand() % (currentState.size() - 2 - posCity);
        }

        newState[posCity] = currentState[pos];
        newState[pos] = currentState[posCity];

        return newState;
}

```

As we can see, we can swap a city that its position is greater than **posCity** or the opposite, this last if only if **posCity** is greather than 7, and in the case that **posCity** is equal to the penultimate position of the vector, we can not go to the "right" of the vector. Then, with this positions, we swap them and we obtain a new succesor state.

3.5 Evaluation function

With this function we can obtain the **total distance of the path**, then we calculate the **Euclid distance** of two cities. Remembering that the **Euclid distance** can be calculated by the expression: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

```
double d_euclid2D(const City& c1, const City& c2){
        return sqrt(pow(c2.coord_x() - c1.coord_x(),2) + pow(c2.coord_y() - c1.coord_y(),2));
}
```

After obtaining the **Euclid distance** of two cities, we sum all the distances and we obtain the total distance of the current state.

```
double fEval(vector<City> currentState){
        double val = 0;

        for(size_t i = 1; i < currentState.size(); i++){
                val += d_euclid2D(currentState[i-1],currentState[i]);
        }

        return val;
}
```

3.6 Main body of the algorithm

Once we have explained the most important functions for working of the algorithm, we pass to explain the main body of the algorithm:

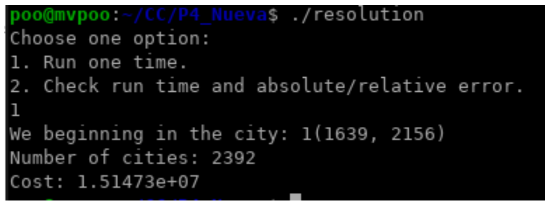
```
        while(T > T_min && maxAttemps < 1000){
                newState = genSuc(currentState,city);
                deltaE = fEval(newState) - fEval(currentState);

                if(deltaE < 0){
                        if(deltaE < deltaEaux){
                                currentState = newState;
                                deltaEaux = deltaE;
                        }

                        ++city;
                }
                else{
                        p = exp(-deltaE/T);
                        if(p > (double)rand()){
                                currentState = newState;
                        }
                }

                T = cool(T);
                ++maxAttemps;

                if(city == currentState.size() - 1){
                        city = 1;
                }
        }

```

The **first step** is **generate a new succesor state**, then once we have generated the succesor, we are going to proceed to obtain the value of **deltaE** substracting the cost of the new state with the current state. If the **deltaE is less than 0**, this means that the new state has better cost than the current state, so the new state converts to the current state and we sum one position to the variable **city**.
Also, if the **difference is positive**, we can accept with a probability **p** the new state. We **cool the temperature** with and we continue to the next iteration until the temperature is less than the minimum temperature.
The function **cool** has the next implementation:

```
double cool(double t){
        return t *0.8;
}
```

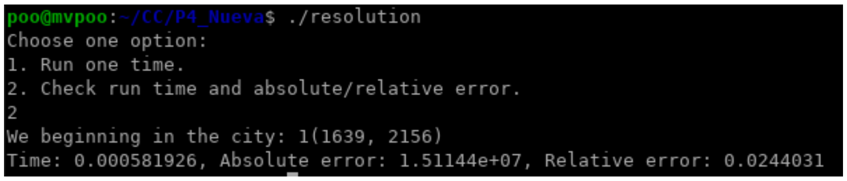
4 How can we run the program?

If we want to **check the running time, absolute and relative error** or **run one time**, we can choose the mode that we want in the terminal.
But at first, we need to compile the program, then you introduce in the terminal **make** to compile the program and after you introduce **./resolution** for executing the program.
Then, once the program is running, you can choose between the two options that we mentioned before.



```
poo@mvpoo:~/CC/P4_Nueva$ ./resolution
Choose one option:
1. Run one time.
2. Check run time and absolute/relative error.
1
We beginning in the city: 1(1639, 2156)
Number of cities: 2392
Cost: 1.51473e+07
```

Figure 2: Option 1 of our program



```
poo@mvpoo:~/CC/P4_Nueva$ ./resolution
Choose one option:
1. Run one time.
2. Check run time and absolute/relative error.
2
We beginning in the city: 1(1639, 2156)
Time: 0.000581926, Absolute error: 1.51144e+07, Relative error: 0.0244031
```

Figure 3: Option 2 of our program

5 Plots time

5.1 Running time

If we want to check the running time of the algorithm for each **number of cities**, this fragment of code is responsible of this task:

```
120         c.activar();
121         do{
122                 vector<City> final = SimulatedAnnealing(cities,*it);
123                 costs.push_back(fEval(final));
124                 //cout<<"The cost for the instance "<<name<<".tsp is: "<<fEval(final)<<endl;
125                 ++r;
126         }while(c.tiempo() < e_abs_t/e_abs_t + e_rel_t);
127         c.parar();
```

Before we execute the loop, we activate the object **cronometro**, that it means in English **chronometer** and after activate the chronometer, we execute the Simulated Annealing function so much as being necessary. If we want to know the cost of the path on each iteration, we discomment the line 124. In addition, the vector costs still work for calculating the absolute and relative error.

Then once the loop finish, we stop the **chronometer** and the results are showed in the terminal. So, if we collect this times depending on the number of cities.

Number of cities	Running time
51	1.98393e-05
76	4.235292e-05
100	5.51515e-05
101	3.05035e-05
105	5.23675e-05
130	6.30636e-05
225	5.6892e-05
442	9.75121e-05
1002	0.000366257
2392	0.000552596

Table 1: Running time for each number of cities

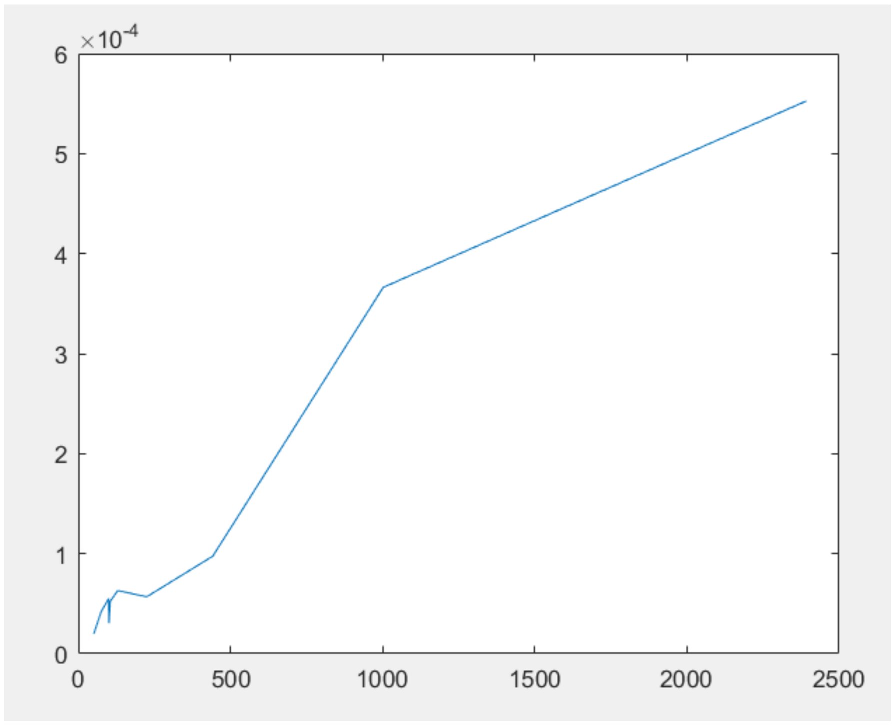


Figure 4: Running time graphic

5.2 Absolute error

For calculating the **absolute and relative error**, before we must calculate the **optimal cost for the path** that we have. Then, we programm a function where we can read a TSPlib format in **opt.tour** format, and we use the **fEval** function.

```
double optSolution(string file_,vector<City> coord){
        //We open the optimal opt.tour to compute the absolute and relative error
                ifstream file(file_);
                vector<unsigned int> optimalState;
                vector<City> optimalPath;
                string line;
                unsigned int id_c, origin;
                size_t cont = 0;

                while(getline(file,line) && (line != "EOF" && line != "-1")){
                        if(sscanf(line.c_str(),"%u",&id_c) == 1){
                                optimalState.push_back(id_c);
                                if(cont == 0){
                                        origin = id_c;
                                        cont++;
                                }
                        }
                }
                file.close();
                optimalState.push_back(origin);

                for(size_t i = 0; i < optimalState.size(); i++){
                        City c(optimalState[i]);
                        auto it = find(coord.begin(),coord.end(),c);
                        optimalPath.push_back(*it);
                }

        //Compute the optimal cost of the path
                return fEval(optimalPath);
}
```

As we can see, we save the origin city in a variable called **origin** because we push it again after saving in the vector the **city node**. After we find the city nodes in the **coord vector** and we save the **coordenate x and y**.

Then, once we obtain **different costs** after executing the loop of the chronometer and obtaining the **optimal solution**, we calcule the absolute error with the next function:

```
double absoluteError(vector<double> costs, double optSol){
        vector<double> costsAux = costs;
        vector<double> costs_;
        vector<size_t> frecs;
        double cost;

        while(!costsAux.empty()){
                cost = costsAux.front();
                frecs.push_back(count(costsAux.begin(),costsAux.end(),cost));
                costs_.push_back(cost);
                for(size_t i = 0; i < costsAux.size(); i++){
                        if(costsAux[i] == cost){
                               costsAux.erase(costsAux.begin() + i);
                               --i;
                        }
                }
        }

        double val = 0;

        for(size_t i = 0; i < costs_.size(); i++){
                val += (costs_[i]*frecs[i]);
        }

        return (val/costs.size()) - optSol;
}
```

As we can see in this function, we obtain the frequence of the value that we obtain, then we use the function **count** for counting this and we remove each element that is equal. Then, once that we have got the frequences we calculate the mean and we substract the **mean with the cost of the optimal solution**, we obtain the absolute error.

After explaining the functioning of our function, we obtain the absolute error for each number of cities.

Number of cities	Absolute error
51	981.53
76	1864.84
100	143109
101	2583.38
105	111088
130	39419.4
225	37513.4
442	723843
1002	6.2857e04
2392	1.49658e07

Table 2: Absolute error for each number of cities

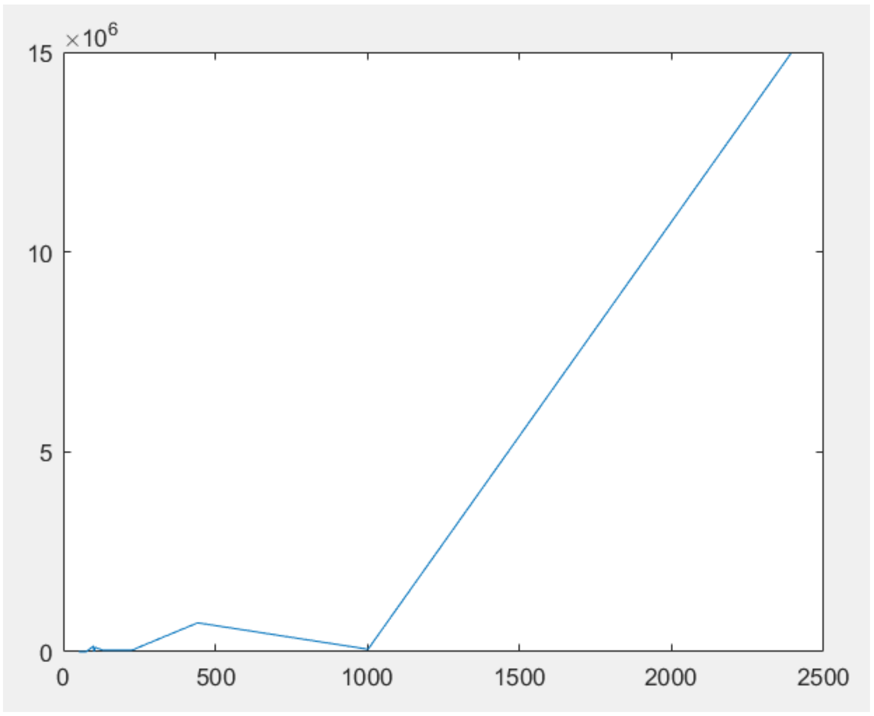


Figure 5: Running time graphic

5.3 Relative error

As we explained before, we use the **optimal solution** function for calculating the relative error in this case. So, if we want to calculate the relative error, we need the next function.

```
double relativeError(vector<double> costs, double optSol){
        vector<double> costsAux = costs;
        vector<double> costs_;
        vector<size_t> frecs;
        double cost;

        while(!costsAux.empty()){
                cost = costsAux.front();
                frecs.push_back(count(costsAux.begin(),costsAux.end(),cost));
                costs_.push_back(cost);
                for(size_t i = 0; i < costsAux.size(); i++){
                        if(costsAux[i] == cost){
                                costsAux.erase(costsAux.begin() + i);
                                --i;
                        }
                }
        }

        double val = 0;

        for(size_t i = 0; i < costs_.size(); i++){
                val += (costs_[i]*frecs[i]);
        }

        return optSol/(val/costs.size());
}
```

In this function we do the same operations as we can see in the **absolute error**, but in this case, we return the division between the **cost of the optimal solution** and the **mean**.

Then, once that we explained about the **relative function**, we are going to take samples for each number of cities.

Number of cities	Relative error
51	0.304614
76	0.22628
100	0.129478
101	0.199123
105	0.114632
130	0.134215
225	0.0932748
442	0.06555
1002	0.0165369
2392	0.0246393

Table 3: Relative error for each number of cities

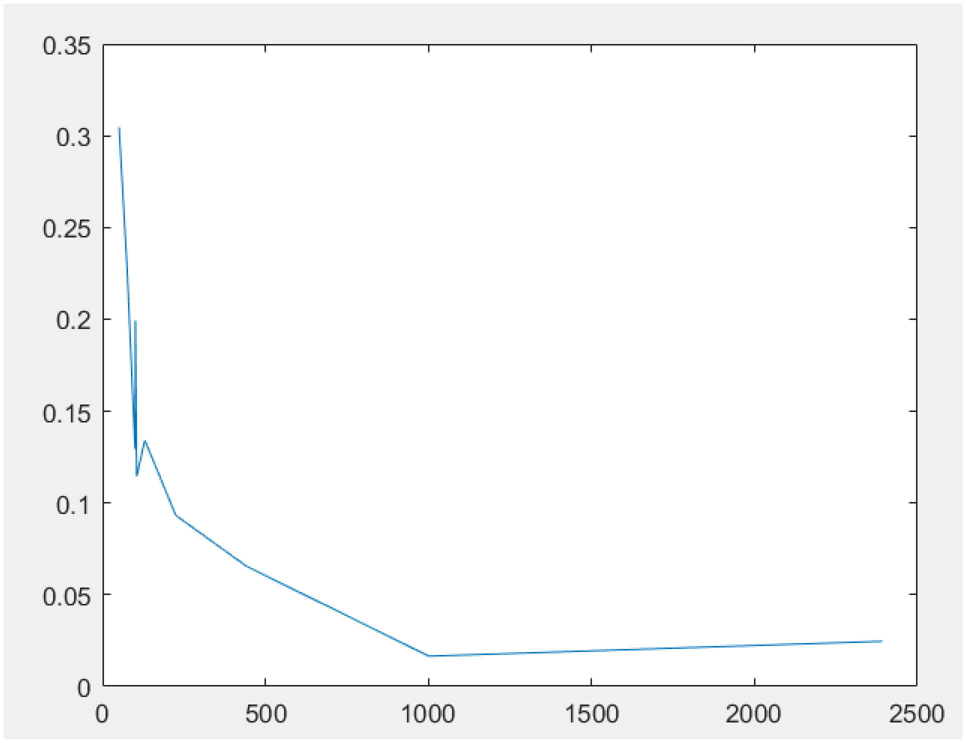


Figure 6: Relative error graphic

6 Conclusion

After realising this assignment, we have achieved the next objectives:

- Learn about the **TSPlib** format.
- Use the **Simulated Annealing** for solve the **Travelling Salesman Problem**.

We start with the **first objective**. This is the base of the program because we need the coordenates of each city for calculating the minimum path of the list of cities that we have, given the origin city.

After, we have the **second objective**, that is solving the **Travelling Salesman Problem** using this algorithm. We can obtain bad results sometimes because the temperature reach the minimum before that we obtain a better result than we have obtained. Also, the succesor function use random positions and in the algorithm, a solution can be accepted with a probability p as current state and this can be worse than the the result that we have.

7 References

[1] *Simulated Annealing.*
URL: `https://en.wikipedia.org/wiki/Simulated_annealing`.

[2] *TSPLIB.*
URL: `https://www.maplesoft.com/support/help/maple/view.aspx?path=Formats%2FTSPLIB`.