TP3: Simulated Annealing and Parallel Tempering

BY MARTINO FERRARI

1 The Travelling Salesman Problem

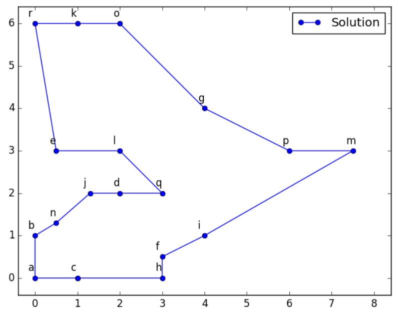


Figure 1. example of a TSP problem

The travelling salesman problem (**TSP**) is a classic and well-known problem, that consists of finding the shortest path connecting a set of cities all interconnected with direct roads (as in figure 1).

Given n cities a solution is $x_i = \{c_1, c_2, ..., c_n\}$ and the solutions space S has size $n!$.

For this reason it's often not feasible to explore S completely and it's only possible to find an acceptable solution trough an heuristic algorithm.

The neighbourhood $N(x_i)$ of a solution x_i is the set of all possible unique permutations of two cities of the itinerary of x_i, so the size of $N(x_i)$ is $\frac{n \cdot (n-1)}{2}$.

2 Simulated Annealing

Simulated Annealing (**SA**) is a metaheuristic introduced in the 1983 by Kirkpatrick originally to solve a physics problem called spin glass. It is inspired by the process of annealing in metallurgy, a technique used to optimize the molecular organization of a metal trough several cycles of controlled heating and cooling of the metal.

While the temperature is high the molecules are free to move around and trough a very slow and controlled cooling an optimized organization will eventually be frozen. If the metal is cooled down too fast the molecular organization will be optimized only locally.

This mechanism is the core of the Simulated Annealing algorithm, where the temperature parameter influences the probability of the selection of a neighbour solution. We can express this mechanism mathematically for a <u>minimization</u> problem as:

$$\begin{array}{rl} t \in R & \text{current temperature} \\ x_i, x_{i+1} & \text{a solution and one neighbour of it} \\ \Delta E & \text{delta fitness } (f(x_{i+1}) - f(x_i)) \\ p(x_{i+1}|\Delta E) = \begin{cases} 1 & \Delta E \leqslant 0 \\ e^{-\frac{\Delta E}{t}} & \Delta E > 0 \end{cases} & \text{probability of selection of } x_{i+1} \end{array}$$

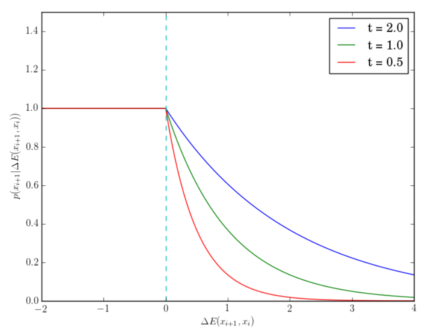


Figure 2. $p(x_{i+1}|\Delta E)$ with different temperatures t

As shown in figure 2 when the temperature t is high the probability that the **SA** will choose a solution with worse fitness than the current one is much higher than that at low temperature. This implicates that at high temperature the algorithm will explore a vast reagion of S (diversification) and the more the temperature decreases the more the algorithm will optimize the solution in a local region (intensification).

As in the real annealing, a too fast decrease of the temperature will probably freeze the research in a wrong area of the solution space S and it would be very unlikely to find a global (or a close to global) solution.

For this reason the theoretical decrease of temperature is logarithmic. However to have a solution in a reasonable time this is approximated with a simple linear decrease (so much faster than the theoretical one).

The decrease of temperature is, unlike in the real annealing process, not continuous. This means that many iterations will be performed with the same temperature and only when certain conditions are met the temperature will decrease.

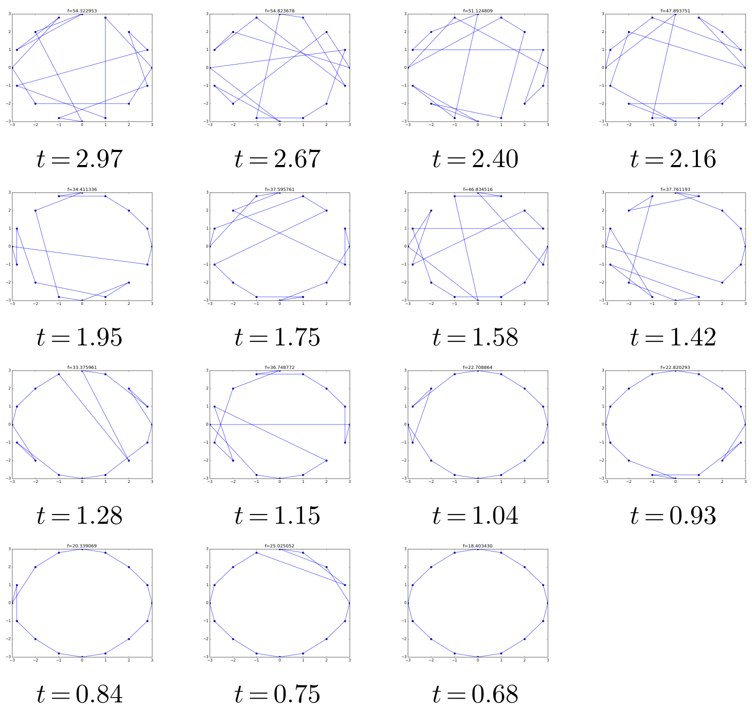


Figure 3. Results at every temperature decrease

At any temperature step the solutions explored will be closer together, as explained before, and when no new solutions are accepted the algorithm will terminate its research. Figure 3 shows the temperature steps and the solutions found of my implementation of the algorithm.

3 The Simulate Annealing and the TSP

As for any other metaheuristics we have to define some elements:

- The research space S
- A start solution x_0
- The neighbourhood of a solution $N(x)$
- The exploration operator U
- The stop condition

As we are implementing the **SA** for the **TSP** problem the research space S and the neighbourhood of a solution $N(x)$ is the one defined in section 1.

The start solution x_0 is simply a random solution extracted from S.

The exploration operator U has already been described in section 2, but we still need to define:

- The initial temperature t_0
- The conditions of temperature decrease
- The decrease of temperature function

The initial temperature t_0 will be defined empirically: computing the average difference between the fitness ($\Delta E_{N(x_0)}$) of 100 random samples from $N(x_0)$ and x_0. Then imposing a probability of exploration of 0.5:

$$\begin{aligned} e^{-\frac{\Delta E_{N(x_0)}}{t_0}} &= 0.5 \\ -\frac{\Delta E_{N(x_0)}}{t_0} &= \log(0.5) \\ t_0 &= \frac{-\Delta E_{N(x_0)}}{\log(0.5)} \end{aligned}$$

The decrease of temperature will happen on every $100 \cdot n$ attempted selections or $12 \cdot n$ successful selections and will follow the rule: $t_{i+1} = 0.9 \cdot t_i$.

Finally the stop (or freezing) condition is: either $t_i \approx 0$ or no solution has been selected with the previous temperature.

Those are not rules but more like steps of a recipe that can be adapted depending on the context.

4 Implementation

I implemented the algorithm in C++ using extensively the object-oriented features of the language. To improve the performance of the algorithm I chose to compute the ΔE directly without computing the total fitness of the neighbours. ΔE for a neighbour that swaps the ith city with the jth of the original itinerary is computed as follows:

$$\Delta E(i,j) = \sum_{k \in -1,1}^{i+k \neq j} (\|(i+k) - j\| - \|(i+k) - i\|) + \sum_{k \in -1,1}^{j+k \neq i} (\|(j+k) - i\| - \|(j+k) - j\|)$$

So the ΔE can be computed in only 4 operations independently from the problem size n.

Also my stop condition is either $t_i \leqslant 10^{-5}$ or no solution has been selected or no fitness improvement within the last 3 temperatures steps.

I was also asked to implement a simple **Greedy** heuristic to be able to compare the performance and results of the **SA**.

The **Greedy** heuristic is about choosing an itinerary starting from a first random city and then choosing the closest city between the remaining ones as second and repeating this mechanism till the end.

This means that given n cities with no equidistant ones, this algorithm will always produce a maximum of n different itineraries, depending only on the starting city.

5 Results

As in the last TP, before analysing the performance and results of the algorithm, I wanted to understand the problem and its solution space S better.

In order to do so I used the problem file *cities.dat* as reference and I explored the solution space generating 1,000 random solutions and its fitness.

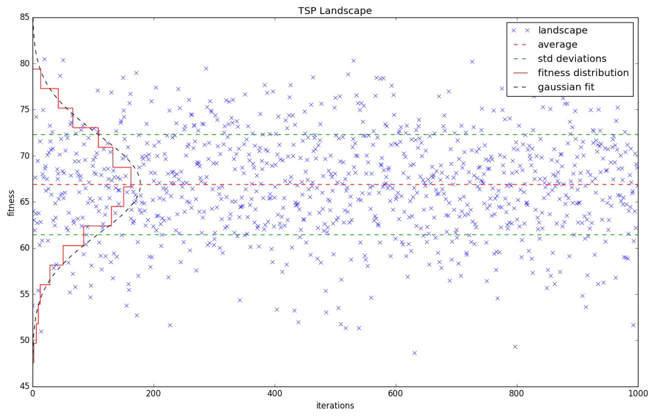


Figure 4. TSP landscape of 1,000 random samples of *cities.dat*

The results of this exploration is shown in figure 4. The analysis of the landscape distribution reveals an average length of the itinerary of 66.89 and a standard deviation of 5.42.

Performing multiple runs of my implementation of the **SA** over this data results in an itinerary length of 27.5154 as the best result, which is very far from the average distribution of the landscape. The probability to find an itinerary with this length by chance is only 6.96×10^{-9}.

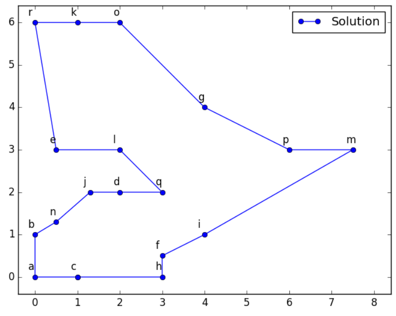
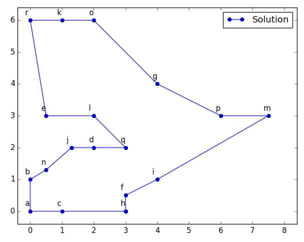


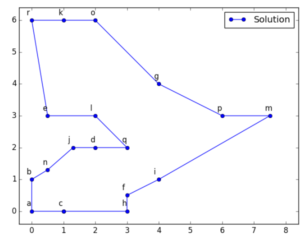
Figure 5. Global minimum for *cities.dat*

Figure 5 shows that the solution with fitness 27.5154 is as well the global minimum of the problem.

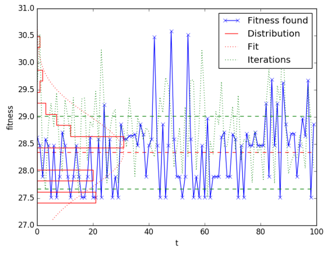
I was asked to compare the results of my **SA** algorithm with the **Greedy** heuristics over 10 runs. However I chose to run it 100 times to have more relevant data. The results for *cities.dat* were the following:



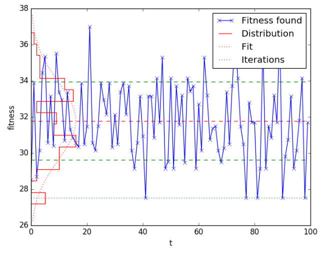
SA best solution



Greedy best solution



SA solutions distribution



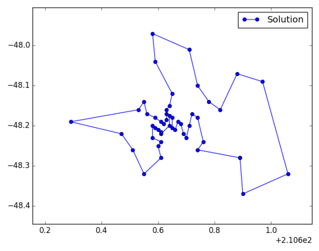
Greedy solutions distribution

Figure 6. Results of 100 runs over *cities.dat*

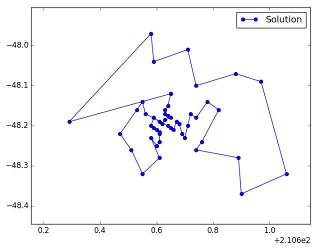
It can be seen in figure 6 that the two algorithms are capable to find the best solution. In this case the **SA** found it in 21% of the runs and the **Greedy** only in 5% of the runs. These results are reinforced by the distribution of the solutions of the two algorithms. The **SA** solutions have an average length of 28.35 and a standard deviation of only 0.67, whereas the **Greedy** solutions have an average length of 31.79 and a standard deviation of 2.17.

However the computational performance of the two algorithms is very different. To compute 100 solutions it took the **SA** algorithm 5.9 s (or 59 ms per solution) whereas it took the **Greedy** algorithm only 12 ms (or 120 μs per solution).

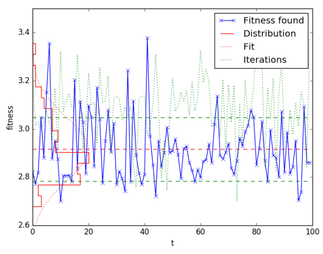
I made similar comparison for the file *cities2.dat*, the results were the following:



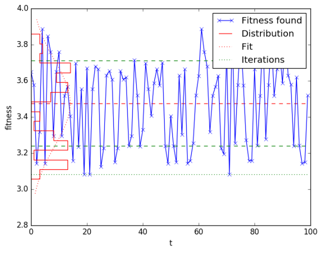
SA best solution



Greedy best solution



SA solutions distribution



Greedy solutions distribution

Figure 7. Results of 100 runs over *cities2.dat*

In this more complex case, where n is 49, the **SA** is performing much better then the **Greedy**. The best solutions of the two algorithms, as shown in figure 7, are now different. For the **SA** the best solution has a length of 2.70 while the one found by the **Greedy** algorithm has a length of 3.08.

More importantly, the distribution of the solutions of the two heuristics is very different. The **SA** has an average of 2.92 and standard deviation of 0.13, while the **Greedy** has an average of 3.48 and standard deviation of 0.24.

Again the performance of the two algorithms is very different. The **Greedy** needs only 82 ms to perform 100 runs and the **SA** needs 14.2 s. It's interesting to see that the implementation of the **SA** using the ΔE formula with a problem size of $n_2 = 49$ (*cities2.dat*) is performing only 2.4 times worse than with a problem size of $n_1 = 18$ (*cities.dat*). This difference in performance is linear ($\frac{n_2}{n_1} = 2.7$) while the solution space size increases factorially. It's important to note that due to the implementation of the ΔE, the execution time difference doesn't depend on the computation of each step but on the fact that with bigger n and a rougher solution space S the algorithm will decrease the temperature more before freezing.

The following table will summarise the results of the previous experiments, plus the ones made over random problems of size 50, 60, 80 and 100 (always over 100 runs):

File	n	Algorithm	Best length	Average length	Std deviation	Execution time
cities.dat	18	SA	27.52	28.35	0.67	5928 ms
		Greedy	27.52	31.79	2.17	12 ms
cities2.dat	49	SA	2.70	2.92	0.13	14168 ms
		Greedy	3.08	3.48	0.24	82 ms
cities50.dat	50	SA	76.83	88.04	6.16	19359 ms
		Greedy	78.09	87.19	4.73	97 ms
cities60.dat	60	SA	109.78	121.34	5.33	25399 ms
		Greedy	129.76	127.48	4.57	119 ms
cities80.dat	80	SA	157.48	184.67	11.79	29195 ms
		Greedy	144.93	158.28	7.90	214 ms
cities100.dat	100	SA	77.99	87.20	4.45	37430 ms
		Greedy	72.60	82.23	4.21	320 ms

This results show us that the performance of the **SA** is not always better than the one of the **Greedy** algorithm. This depends on the landscape of the problem and on its size as well. Probably for bigger n the stop conditions and the fast decrease of the temperature is conducting the **SA** to a sub optimal solution, while the **Greedy** still performs accurately. Also the landscape roughness influences the results. For a problem like the one showed in figure 3, the **Greedy** performs very well, always finds the best solution and is very fast compared to the **SA**.

The table also shows that the execution time is growing linearly with a coefficient of ≈ 0.36 (with time expressed in seconds).

6 Parallel Tempering

As a bonus we were asked to implement the Parallel Tempering (**PT**) metaheuristic. This algorithm is a variation of the **SA** where instead of having a single **SA** decreasing temperature there are n **SA** replicas with fixed temperatures (but different) exchanging the solution when there is some sort of equilibrium.

Every **SA** will have a different temperature T_i where $T_1 < T_2 < \ldots < T_i < \ldots < T_n$, and where T_n has the same value of the initial temperature t_0 of a normal **SA**.

I chose to use a geometrical distribution of the other temperatures as the energy jumps of high temperatures are bigger than the ones at lower temperatures. Therefore it makes sense to not distribute the temperatures evenly.

The exchange of the solutions of two neighbours **SA** (i, j) has the following probability:

$$p(i,j) = \min\left(1, e^{(\beta_i - \beta_j)(E_i - E_j)}\right)$$

β_i and β_j are the inverses of the respective temperatures of the two **SA** while E_i and E_j are the fitness of the currently selected solutions of the two **SA**.

In my implementation only two close **SA** will exchange solutions between each other (so $j = i + 1$) and at every step it tries to exchange solutions of a single random pair of **SA**.

The stop condition is simply a maximum number of possible exchanges that I set to 1,000.

Running my implementation of the **PT** over the file *cities.dat* with 2, 5, 10 and 20 replicas gave me the following results:

n of replicas	Minimum length	Average length	Std deviation	Execution time
2	29.92	49.94	10.16	761 ms
5	29.07	39.15	5.89	1199 ms
10	28.50	32.35	3.16	1755 ms
20	27.52	29.62	1.26	2683 ms

With 20 replicas the results are very similar to the ones of the normal **SA** but were determined in less than half of the time of the original algorithm.

I believe that with further optimization and a real parallelisation of the processes it would be possible to improve the performance even more, both, in terms of correctness of results and speed.

7 Project structure

src/	contains all the source codes (above the core codes)
→/tsp.*	is the code that defines the TSP solution, neighbours, etc
→/annealing.*	is the code that defines the SA metaheuristic
→/greedy.*	is the code that defines the Greedy heuristic
src/bin/	contains all the binary, dat files and the compilation script
→/compile.sh	is the compilation script
→/compile_basic.sh	is a compilation script without the plots functionality
→/analyse	analyses and compares the SA and TP over a dat file
→/stepper	runs the SA step by step and visualizes or saves the single steps
→/landscape	explores the landscape of a problem randomly
→/generator	creates a random problem and saves it to a dat file
→/parallel_tempering	runs the Parallel Tempering test code

All the executables if executed without arguments will show the basic usage.

The codes and report can also be found on my github repository: Metaheuristic.