TP3: Simulated Annealing and Parallel Tempering

by Martino Ferrari

1 The Travelling Salesman Problem

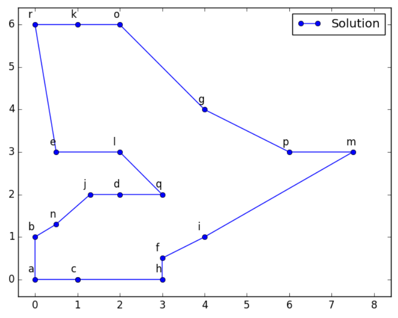


Figure 1. example of a TSP problem

The travelling salesman problem (**TSP**) is a classic and well-known problem, that consist in 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 if often not possible to explore completely S 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 permutation 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 a optimized organization is frozen. If the metal is cooled down to 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. For a <u>minimization</u> problem , we can express it mathematically:

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

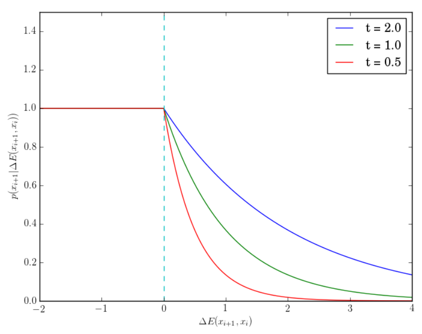


Figure 2. $p(x_{i+1}|\Delta E)$ with different temperatures t

As shown in the Figure 2, the temperature t affect the probability of a solution to be selected, with higher temperatures (in the figure, the line blue) the algorithm will more-likely choose a neighbours with a worst fitness (or energy) of the current solution then a low temperatures. This implicate that at high temperature the algorithm will explore S (diversification) and more the temperature decrease more the algorithm will optimize the solution in the 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, but to have a solution in a reasonable time this is approximate with a simple linear decrease (so much faster than the theoretical one).

The decrease of temperature, however, 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 reached the temperature will decrease.

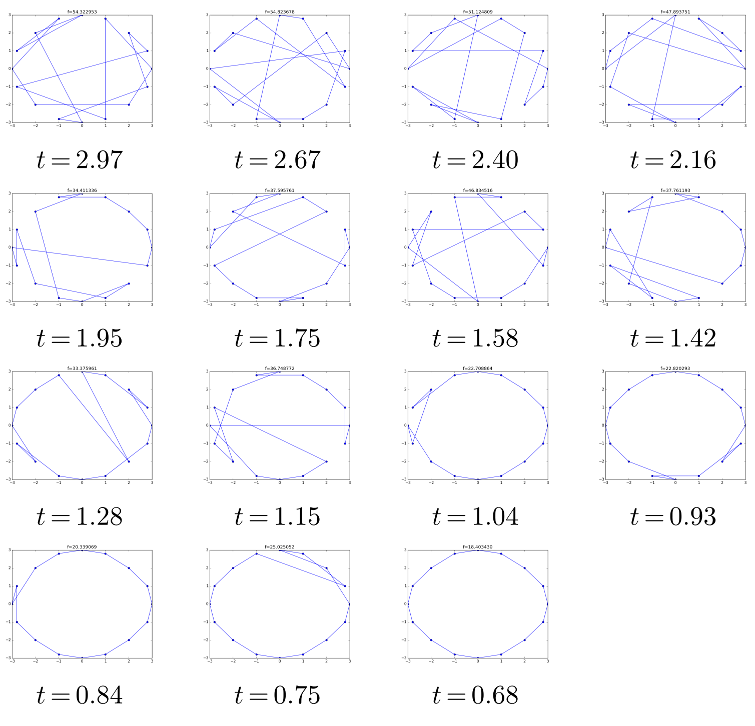


Figure 3. Results at every temperature decrease

At any temperature step the solutions explored will be more close together, as explained before, and when no new solutions are accepted the algorithm will end. The 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 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 the Section 1.

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

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

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

The initial temperature t_0 will be defined empirically: computing the average difference of fitness ($\Delta E_{N(x_0)}$) between 100 random samples from $N(x_0)$ and x_0 and 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 every $100 \cdot n$ attempted selections or $12 \cdot n$ successfully selections, and $t_{i+1} = 0.9 \cdot t_i$.

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

Those are not rules but more like steps of a recipe, that can be adapted for 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 language I choose to compute directly the ΔE without computing the total fitness of the neighbours. ΔE is for a neighbour swapping the i city with the j one in the itinerary is computed as follow:

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

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

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

Was also asked to implement a simple **Greedy** heuristic to be able to confront the performance and results of the **SA**.

This simply consists in an heuristic that choose an itinerary starting from a first random city and than choosing the closer 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 from the starting city.

5 Results

As in the last TP, before analysing the performances and results of the algorithm I wanted to understand better the problem and it's solution space S.

To do so I used as reference the problem file *Cities.dat* and I explored the solution space generating 10000 random solutions and its fitness.

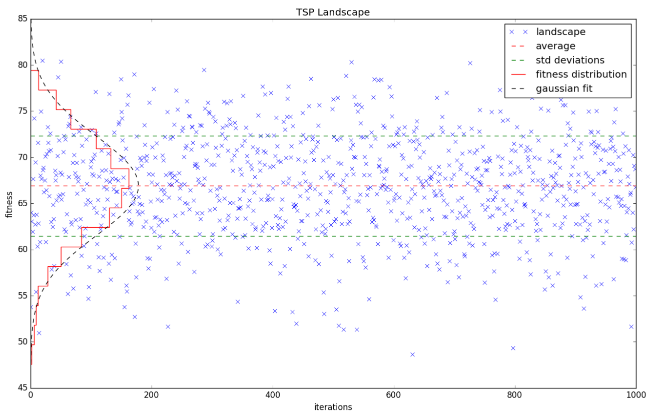


Figure 4. TSP landscape of 1000 random samples of *cities.dat*

The results of this explorations is shown in Figure 4, where also the landscape distribution is analysed, the average length of the itinerary is 66.89 and the standard deviations is 5.42.

Performing multiple runs of my implementation of the **SA** over this data gave as best result an itinerary length 27.5154, very far from the average distribution of the landscape, the probability of found an itinerary with this length is only 6.96×10^{-9}.

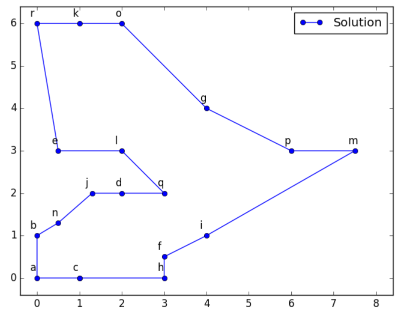
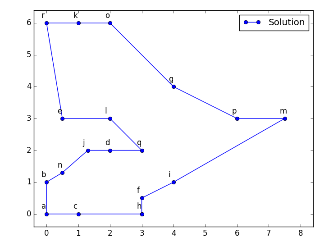


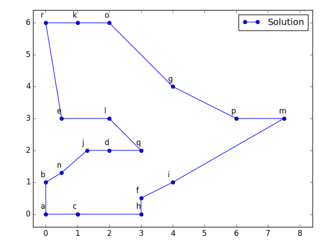
Figure 5. Global minimum for *cities.dat*

The solution with fitness 27.5154, shown in the figure 5, is as well the global minimum of the problem, as is possible to see graphically.

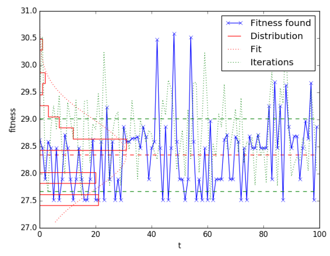
As asked I compared the results of my **SA** algorithm with the **Greedy** heuristics over 10 runs of the two, however I choose to run it 100 times to have more relevant data. The results for *cities.dat* were the following:



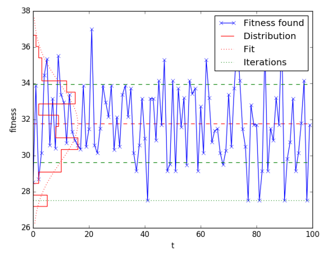
SA best solution



Greedy best solution



SA solutions distributions



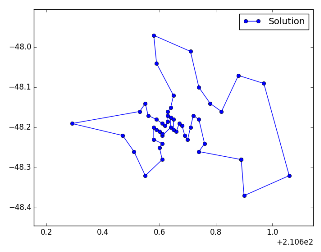
Greedy solutions distributions

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

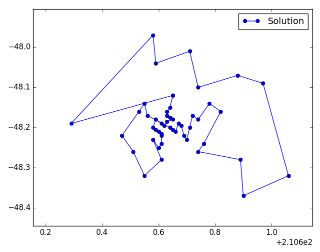
As possible to see in the Figure 6, the two algorithms are capable to find the best solution, in this case the **SA** found it in the 21% of the runs and the **Greedy** only in the 5% of the runs. This results is confirmed by the distribution of the solutions of the two algorithms, the **SA** solutions have an average length of 28.35 and a standard deviation of only 0.67, where the **Greedy** solutions have an average length of 31.79 and a standard deviation of 2.17.

However the computational performances of the two algorithm are very different, for computing this 100 solutions the **SA** algorithm spent 5.9 s (or 59ms per solution) where the **Greedy** only 12 ms (or 120μs per solution).

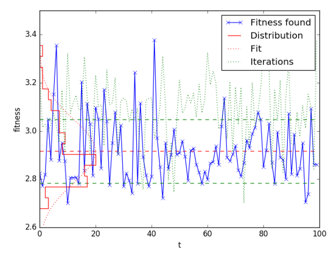
I made similar comparison for the file *cities2.dat*, the results were the following:



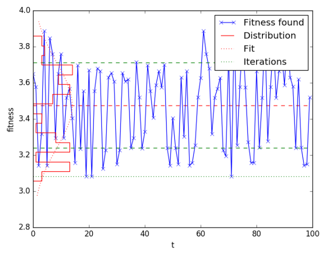
SA best solution



Greedy best solution



SA solutions distributions



Greedy solutions distributions

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

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

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

Again the performance of the two algorithms are very different, with the **Greedy** that need only 82 ms to perform 100 runs and whit the **SA** that need 14.2 s. It's interesting to see that the implementation of the **SA** using the ΔE formula with a problem of size $n_2 = 49$ (*cities2.dat*) is performing only 2.4 then with a problem of size $n_2 = 18$ (*cities.dat*), this difference in performance is only linear ($\frac{n_2}{n_1} = 2.7$) while the solution space size increase factorially. It's important to note that due to the implementation of the ΔE, the execution time difference is not due to the time need to compute each steps but to the fact that with bigger n and a more rough solution space S the algorithm will decrease the temperature more before freezing.

The following table will resume the results of the previous experiments, plus the one made over random problems of size 50, 60 and 80 (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 shows us that not always the performance of the **SA** and the **Greedy** depends from the landscape of the problem, and from it size as well, for bigger n probably the stop conditions and the fast decrease of the temperature is conducting the **SA** to a sub optimal solution, while the **Greedy** still perform correctly. Also the landscape rugosity influence the results, for a problem like the one showed in the Figure 3, the **Greedy** will perform very well, always finding the best solution, and very fast compared to the **SA**.

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

6 Parallel Tempering

As 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 every step there are n **SA** replicas with fixed temperature (but different) exchanging the solution when there is some sort of equilibrium.

Every machine 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 choose to use a geometrical distribution of the other temperatures as the energy jump of high temperature are bigger of the one at lower temperatures, so it make sense to don't distribute the temperature evenly.

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

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

Where β_i and β_j are the inverse of the respective temperature of the two **SA** while E_i and E_j are the fitness of the current selected solutions of the two **SA**.

In my implementation only two close **SA** will exchange solutions between them (so $j = i + 1$) and at every equilibrium status will only try to exchange solutions of a random pair, not all the possible pairs.

The stop condition is simply a maximum number of possible exchanges that I fixed to 100.

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 is very similar to the ones of the normal **SA** but in the less then half of the time of the original algorithm.

I believe that with further optimization and a real parallelization of the processes would be possible to improve even more the performance, both in results and speed, of this algorithm.

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 compare the SA and TP over a dat file
→/stepper	runs the SA step by step and visualize or save the single steps
→/landscape	explores randomly the landscape of a problem
→/generator	creates a random problem and save 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 be also found on my github repository: Metaheuristic.