1 Overview

For this TP, we recall the Traveling Salesman Problem (TSP) that was solved in TP1, which is defined as follows: given a set of n cities, and the distance between every pair of cities, find the shortest possible route that visits every city exactly once and returns to the starting point. As done in TP1, we will represent the TSP as a complete and symmetric graph $G(V, E)$, where V is the set of vertex (cities) and E the set of edges. As G is complete, there is an edge $(i, j) \in E$ for all $i, j \in V$ such that $i \neq j$, in other words, there is a path between every pair of cities. The cost of an edge $e_{ij} \in E$ is $c(e_{ij})$ and the symmetry ensures that $c(e_{ij}) = c(e_{ji})$.

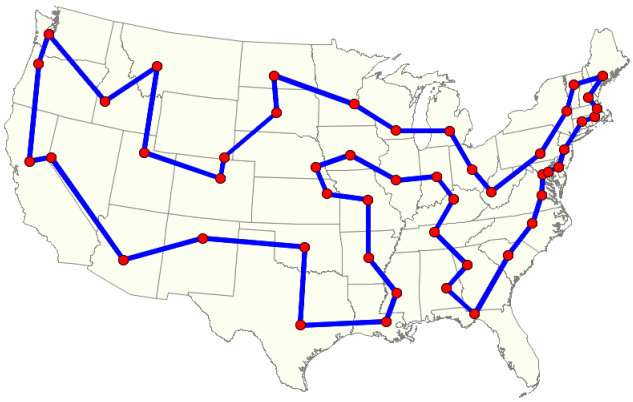

Figure 1: TSP illustration [1].

As seen in TP1, exactly solving the TSP can be hard even for small size instances. Your job for this TP is to use a type of "swarm intelligence" to optimize the TSP. You are expected to create a heuristic algorithm able to find good TSP solutions for instances with more than 4000 cities.

2 Ant Colony Optimization [1]

Ant Colony Optimization (ACO) is a nature-inspired metaheuristic proposed in 1992 by Marco Dorigo in his PhD thesis. ACO was designed for solving hard combinatorial problems and is inspired in the optimization process performed by ants in order to find the shortest path from their nest to food. Real ants are capable of finding this shortest path without using visual cues by exploiting pheromone information. While walking, ants deposit *pheromone* on the ground and follow, in probability, pheromone previously deposited by other ants.

[1] This section is heavily based in the work of Marco Dorigo et al (1997) [5] and Marco Dorigo et al (2006) [3], for which the interested student is referred to.

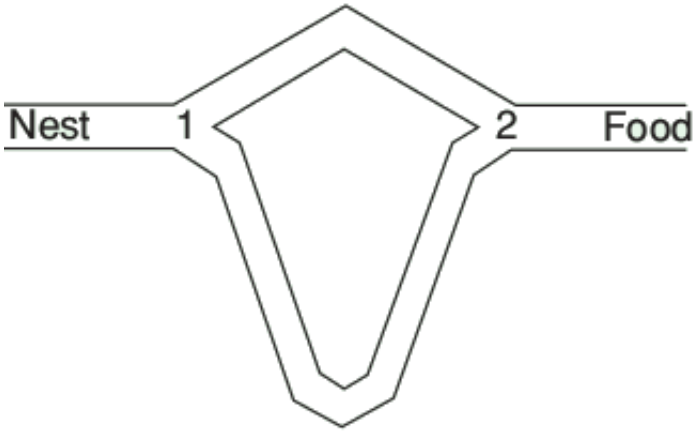

Figure 2: Illustration of a decision that ants need to make in order to find the shortest path to the food [3].

Consider the image above: when the first group of ants arrive at the bifurcation 1 they have to decide whether to turn left (top path) or right (bottom path). Since they have no clue about which is the best choice, they choose randomly (it can be expected that, on average, half of the ants decide to turn left and the other half to turn right). Assuming that all ants are walking with the same speed, the ants that have decided to turn left in 1, will reach the food much faster, and thus return to the nest first. As the second group of ants approach the decision in 1, the level of pheromone is higher on the top path once the ants that took that path already deposited the pheromone twice (once on the way to the food, and again on the return), so there is a higher probability of this second group also choosing to turn left. Once more ants will visit the top path, and therefore pheromone accumulates faster, after a short transitory period the difference in the amount of pheromone on the two paths is sufficiently large so as to influence the decision of new ants coming into the system. From now on, new ants will prefer in probability to choose the top path, since at the decision point they perceive a greater amount of pheromone on the top path. This in turn increases, with a positive feedback effect, the number of ants choosing the top, and shorter, path. Very soon all ants will be using the shorter path[2].

The above behavior of real ants inspired ACO, an algorithm in which a set of artificial ants cooperate to the solution of a problem by exchanging information via pheromone deposited on graph edges. The pseudo-code below presents this general idea of the ACO metaheuristic. In general terms, ACO algorithm defines K ants, each of them responsible to construct a solution for the problem, using pheromone information as guide to obtain good solutions. When an ant finds a solution, it can be improved using a heuristic. After all ants have finished constructing their solution, the concentration of pheromone is updated taking into account which was the best solution found. The next section uses the TSP to explain in details how each of these steps must the performed.

[2]The interested students are referred to the video `https://www.youtube.com/watch?v=HisgmcLaHoY` about an ant colony that lives in the Montreal's insectarium.

Algorithm 1 General idea of ACO algorithm

1: **while** stop condition is not met **do**
2: Send K ants to construct K solutions.
3: Apply an local search algorithm to improve the solutions found.
4: Update the level of pheromone according to the best solution.
5: **end while**

2.1 ACO in the context of TSP

In ant colony optimization, the problem is tackled by simulating a number of artificial ants moving on a graph, in which there is a variable $\tau(e)$ called pheromone associated with each edge $e \in E$, which can be read and modified by ants. In each iteration of the ACO algorithm, a number K of artificial ants are considered. Each of them builds a TSP solution by walking from vertex to vertex on the graph with the constraint of not visiting any vertex that it has already visited in her walk. At each step of the solution construction, an ant selects the following vertex to be visited according to a stochastic mechanism that is biased by the pheromone: when in vertex i, the next vertex is selected stochastically among the previously unvisited ones (see figure below). In particular, if j has not been previously visited, it can be selected with a probability that is proportional to the pheromone $\tau(e_{ij})$ associated with edge e_{ij}.

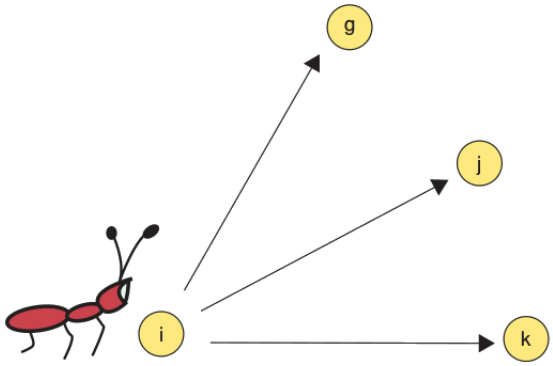

Figure 3: Illustration of an ant in city i choosing the next city to visit [3].

At the end of an iteration, on the basis of the quality of the solutions constructed by the ants, the pheromone values are modified in order to bias ants in future iterations to construct solutions similar to the best ones previously constructed.

These are the three base stages in the ACO algorithm for TSP:

1. **Construct Ant Solutions:** Formally, let S_k be the solution being built by the ant k, with $V(S_k) \subseteq V$ the set of visited vertex and $\overline{V}(S_k) = V \backslash V(S_k)$ the set of unvisited vertices.

   In the construction of a solution, ants select the following city to be visited through a stochastic mechanism. When ant k is in city i and has so far constructed the partial

solution S_k, the next city j to be visited is obtained by expression (1):

$$j = \begin{cases} \arg\max_{j \in \overline{V}(S_k)} \left\{ \dfrac{\tau(e_{ij})}{c(e_{ij})^{\beta}} \right\}, \text{if } q \leq q_0 \\ \\ \text{randomly selected in the probability distribution } D \end{cases} \tag{1}$$

where β is a parameter which determines the relative importance of pheromone versus distance ($\beta > 0$), q is a random number uniformly distributed in [0...1], q_0 is a parameter ($0 \leq q_0 \leq 1$), and the probability distribution D contains the probability $p_k(e_{il})$ on moving to city j given by:

$$p_k(e_{ij}) = \begin{cases} \dfrac{\tau(e_{ij})/c(e_{ij})^{\beta}}{\displaystyle\sum_{l \in \overline{V}(S_k)} \tau(e_{il})/c(e_{il})^{\beta}}, \text{if } j \in \overline{V}(S_k) \\ \\ 0, \text{otherwise} \end{cases} \tag{2}$$

To understand this construction, consider the example with $q_0 = 0.9$. Then there is a 90% of chance that the next city is the one that maximizes the relation between the level of pheromone and the cost of the edge (if two or more cities have this maximum value, choose randomly among them). If we fall in the case with remaining 10% of chance, we will select a unvisited city j with probability $p_k(e_{ij})$.

2. **Apply Local Search.** Once solutions have been constructed, and before updating the pheromone, it is common to improve the solutions obtained by the ants through a local search. This steps can help the ACO to faster converge to a good solution.

   For the TSP for example, these improvement heuristics can start from a given route and attempt to reduce its length by exchanging edges chosen according to some heuristic rule until a local optimum is found (i.e., until no further improvement is possible using the heuristic rule). One of the most used and well-known improvement heuristics for TSP is 2-opt [2].

   Consider a solution S as the sequence in which the vertices were visited. For a vertex i, let i' be the immediate successor of i in the sequence S. The 2-opt heuristic work as follows: for every pair of non-consecutive vertexes i, j, check if removing the edges $e_{ii'}$ and $e_{jj'}$ and replacing them by e_{ij} and $e_{i'j'}$ results in a improvement for the solution cost. If so, perform this swap. This process repeated until there are no more profitable exchanges. The 2-opt heuristic is illustrated in the figure below.

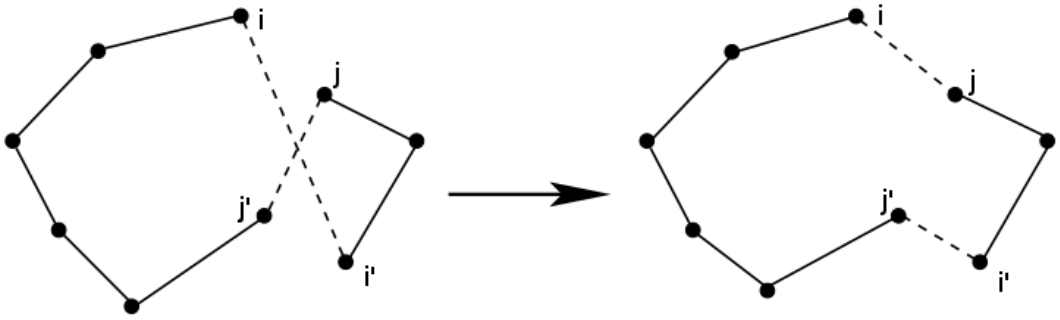

Figure 4: 2-opt heuristic illustration. Adapted from [4].

3. **Update Pheromones:** The aim of the pheromone update is to increase the pheromone values associated with good or promising solutions, and to decrease those that are associated with bad ones. Usually, this is achieved (i) by decreasing all the pheromone values through pheromone evaporation, and (ii) by increasing the pheromone levels associated with the best solution found.

   There are two types of updates:

   - **Global Updating:** After all ants have finished constructing their routes, it is necessary to update the level of pheromone in the edge e_{ij} using the expression below:

   $$\tau(e_{ij}) = (1-\rho)\tau(e_{ij}) + \rho\Delta\tau(e_{ij}) \tag{3}$$

   where ρ $(0 < \rho < 1)$ is the evaporation rate,

   $$\Delta\tau(e_{ij}) = \begin{cases} \frac{1}{L_{gb}}, \text{if } e_{ij} \in \text{global-best tour}, \\ 0, \text{otherwise.} \end{cases} \tag{4}$$

   It is the cost of best solution

   and L_{gb} is the is the length of the globally best tour from the beginning of the algorithm. Using expression (3) and (4), the edges that belong to the best tour have their pheromone level increased, while all the other edges the pheromone is evaporated.

   - **Local Updating:** The main goal of the local update is to diversify the search performed by subsequent ants during an iteration of the ACO algorithm. This is done by decreasing the pheromone concentration on the traversed edges, so ants encourage subsequent ants to choose other edges and, hence, to produce different solutions. This makes it less likely that several ants produce identical solutions during one iteration.

     The local pheromone update is performed by all the ants after each construction step (after including a city to their route). Each ant applies it only to the last edge added:

   $$\tau(e_{ij}) = (1-\varphi)\tau(e_{ij}) + \varphi\tau_0(e_{ij}). \tag{5}$$

where $\varphi \in (0, 1]$ is the pheromone decay coefficient, and $\tau_0(e_{ij})$ is the initial value of the pheromone in edge e_{ij}. For this TP, these values are given.

Question. How do you associate the ant colony optimization algorithm with reinforcement learning?

3 Time to code

Note: all resources files and *skeleton* classes for this TP are provided in C++ and are available at the course web page on moodle. If you are not familiar with C++, feel free to choose any programming language that you might be more comfortable with.

- **Graph.** The `Graph.h` is a ready-to-use header file that contains the class Graph used to represent the problem. An object of this class has the following 3 useful methods:
  - `Graph(dataset_file)` the constructor method that receives a string with the dataset file name and constructs the graph itself;
  - `getN()` that returns the number of vertices n;
  - `getEdge(i,j)` that returns the object edge e_{ij} itself.

  An **edge** object e has 3 attributes:
  - `e.source:` source vertex;
  - `e.destination:` destination vertex;
  - `e.cost:` cost of the edge.

- **Solution.** The Solution class (defined in *Solution.h*) has 4 variables:
  - `visited:` list of visited cities;
  - `not_visited:` list of unvisited cities;
  - `cost:` cost of the solution.
  - `graph:` the problem graph (*Graph* instance). This attribute is used by some methods of *Solution*.

  and 4 useful methods:
  - `Solution(g):` the constructor of class Solution. It receives the *Graph* g as parameter and initializes all the solution attributes. This method will be used to create an **empty** solution.
  - `Solution(sol):` the copy constructor. This method receives another *Solution* instance, *sol*, and copies all its value to the solution being created. T
  - `addEdge(v,u):` adds the edge (v,u) to the solution and updates its cost. v is an already visited vertex (index of the current city from the node being explored). u is an unvisited vertex being added to the route.

- `print()`: prints the solution's edges and its cost.

- **ACO** This is the class that implements the ACO algorithm.

  There are several variables for this class:

  - `parameter_q0`: parameter q_0;
  - `parameter_beta`: parameter β;
  - `parameter_rho`: parameter ρ;
  - `parameter_phi`: parameter φ;
  - `parameter_K`: number of ants;
  - `parameter_init_phoromone`: the values of $\tau_0(e)$ for each edge $e \in E$;
  - `phoromone`: vector that stores the current value of phoromone in each edge;
  - `best`: solution object that stores the best solution found so far.

  Now, it is your time to code. Implement the methods below of class ACO:

  - `getNextCity(sol)`: returns the next city to be added into solution `sol` using the expression (1) and (2).

    Execution. Now test your code using the test function `testNextCity()` provided in the file `Tests.cpp`

  - `heuristic2opt(sol)`: applies the local search 2-opt in the solution `sol`.

    Execution. Now test your code using the test function `testHeuristic2opt()` provided in the file `Tests.cpp`

  - `globalUpdate(best)`: performs the global updating step using the best solution found, as showed in (3) and (4).

    Execution. Now test your code using the test function `testGlobalUpdate()` provided in the file `Tests.cpp`

  - `localUpdate(sol)`: perform sthe local updating step using in the solution `sol`, as showed in (5).

    Execution. Now test your code using the test function `testLocalUpdate()` provided in the file `Tests.cpp`

  - `runACO(numberIteration)`: run the ACO algorithm and returns the best solution found. The parameter `numberIteration` is the maximum number of iterations. Here you must implement the workflow of the ACO algorithm, i.e., creation of K ants; construction of the solution; execution of the local search routine; and updating the pheromone.

    Execution. Now test your code using the test function `testRunACO()` provided in the file `Tests.cpp`

4 Experiments

4.1 Tuning the parameters

As required by many artificial intelligence methods, it is first necessary to tune the set of parameters so that the algorithm can achieve its best performance.

Let $P = \{q_0, \beta, \rho, \varphi, K\}$ be our set of parameters. A simple procedure to tune a parameter $p \in P$ is to fix the others $|P| - 1$ parameters, and run multiple experiments changing the value of p in order to observe the algorithm convergence.

For example, if you want to tune the q_0 parameter, you can set its value starting in 0, run an experiment and observe the algorithm behavior. Next you increment its value to 0.1 and re-run your experiments. Repeat the process until $q_0 = 1$ (its upper bound), and identify which was its best value and update q_0 with it. To help you, we provide a set of initial values that you can use to fix the parameter before optimizing it:

q_0	β	ρ	φ	K
0.9	2	0.1	0.1	10

Now, use the data set **N500.data.txt** for tuning your parameters. You can set the maximum number of iterations to 1000. For each parameter p, you must report two plots: p×best solution cost; and p×execution time. Since the algorithm has a stochastic step, run the algorithm 5 times, for each value of p, and report the average value, i.e., the average best solution cost for the first plot, and the average execution time for the second plot.

4.2 Running your algorithm

Now it is time to optimize a larger instance of TSP. Using your set of optimized parameters, you will run your algorithm for the two national instances of TSP showing in the table below.

Instance	**Cities**	**Optimal cost**
Uruguay	734	79114
Canada	4663	1290319

For the rest of the TP you will need the file accessible on the following link: `https://drive.google.com/drive/folders/1emMiFl9eq4GkvVdOCWqGuxdCPhvXJ36f`

For each instance, you must run the algorithm five times using 1000 iteration as stop condition. For each execution you must report your results as showing in Table 1, where after every 100 iterations you must inform the:

gap_{best}	given by $100\% \times \frac{ACO_{best} - OPT}{OPT}$ where ACO_{best} is the best solution found so far and OPT is the instance's optimal cost.
iterations without improvement	the number of iteration that have occurred since the last time the best solution was improved
CPU time	the execution time since the beginning of the algorithm

Table 1: Execution i for instance x

iteration	**gap_{best}**	**iterations without improvement**	**CPU time**
1			
100			
200			
300			
400			
500			
600			
700			
800			
900			
1000			

For example, if for some execution solving the Uruguay's instance, you find a solution with cost 90330 after your first iteration, you may have a table that starts as

iteration	**gap_{best}**	**iterations without improvement**	**CPU time**
1	14.17%	0	0.01s
⋮	⋮	⋮	⋮

To finish your experiments, you must summarize your results in the table below:

Instance	**Best gap**	**Worst gap**	**Average gap**	**Average time**
Canada				
Uruguay				

5 Bonus

How can you improve your results? Find a way to amend your ACO algorithm and report your gains.

6 Directives de remise

Le travail sera réalisé en équipe de deux ou trois. Vous remettrez un fichiers .zip par personne, nommés TD1_nom_prenom_matricule.zip. Vous devez également remettre un fichier pdf contenant une explication de vos implémentations ainsi que les réponses des questions. Tout devra être remis avant le **2 Décembre à 23h55**. Tout travail en retard sera pénalisé d'une valeur de 10% par jour de retard.

Barème :

implémentations:	30 pts
résultats:	30 pts
rapport :	40 pts

*Ce TP a été imaginé par Daniel Aloise et développé par Rodrigo Randel

References

[1] Robert Allison. What's new in v9.4 sas/graph... [online] robslink.com. available at: http://robslink.com/sas/democd62/new_94_sas.htm [accessed in 1 nov. 2017].

[2] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958.

[3] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[4] Marco Dorigo and Gianni Di Caro. Ant colony optimization: a new meta-heuristic. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 2, pages 1470–1477. IEEE, 1999.

[5] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation*, 1(1):53–66, 1997.