POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Master of Science in Computer Engineering

Dipartimento di Elettronica e Informazione

Design and implementation of an Ant Colony Optimization algorithm for traffic assignment

Relatore: Ing. Matteo MATTEUCCI

Correlatore: Prof. Lorenzo MUSSONE

Tesi di Laurea di:

Marco Roland FERENCZI Matr. 734687

Anno Accademico 2011 - 2012

To My Family.

Computers are incredibly fast, accurate, and stupid. Human beings are incredibly slow, inaccurate, and brilliant. Together they are powerful beyond imagination.

Albert Einstein

Contents

List of Figures

List of Algorithms / Code Snippets

List of Tables

Abstract

Swarm Intelligence (SI) is a concept used by artificial intelligence to solve decision making problems. A peculiar trait of SI algorithms is the use of multiple agents that interact locally and use simple rules to find a globally valid solution. There is no centralized control and the agents have only a limited knowledge of the entire system. In this work we develop and implement a software in Java that extends the basic Ant Colony System (ACS) algorithm, a particular algorithm of the SI family, to model traffic distribution over transportation networks. The objective is to have a software that can work on networks with separable and non-separable cost link functions, searching for a deterministic or stochastic user equilibrium. The flow assignment process is possible over large and real networks with multiple flow origins and destinations, vehicle categories, and limited traffic zones. To achieve this, careful development is done to have an efficient memory consumption, a critical aspect in every software written in Java that uses large amount of data, while maintaining a good computational speed. The software is optimized to work on many threads in parallel, giving a huge increment in performance on multi-core systems. Finally, the software performance and solutions quality is confronted with other commercial software used to solve this type of problems.

Estratto

La *swarm intelligence* (SI, traducibile come: teoria dello sciame intelligente) è un concetto usato in intelligenza artificiale per risolvere problemi di decisione. Un tratto di questa famiglia di algoritmi è l'uso di agenti multipli che interagiscono localmente e usano delle semplici regole per trovare una soluzione globalmente valida. Non c'è alcun controllo centralizzato e gli agenti hanno solo una conoscenza limitata dell'intero sistema in cui si muovono e agiscono. In questo lavoro viene sviluppato e implementato un software in Java che estende l'algoritmo chiamato *Ant Colony System* (ACS), un particolare algoritmo della famiglia degli SI, per modellare da distribuzione del traffico su diverse reti. L'obiettivo è avere un software che può lavorare su diverse reti, con funzioni di costo degli archi dipendenti sia dal traffico sullo stesso arco che su archi incidenti, cercando una soluzione con un equilibrio deterministico o non deterministico. La ricerca di equilibrio è possibile su grandi reti prese dal mondo reale, con diverse origini e destinazioni per il traffico, categorie di veicoli e zone a traffico limitato. Si è data molta attenzione sia alla velocità di esecuzione sia al consumo di memoria, che deve rimanere il più efficiente possibile sopratutto in considerazione del fatto che il software è scritto in Java e lavora su grandi quantità di dati. Il software è ottimizzato per lavorare in parallelo usando diversi processi indipendenti, in modo da avere un ottimo incremento di prestazioni su sistemi multi processore.

Chapter 1

Introduction

In the last century, thanks to mass production of automobiles, there has been an exponentially increasing amount of cars circulating in large urban centers and on highways. This created a great concern over traffic congestion, which is a condition where the volume of traffic on a road is near the road maximum capacity. High level of congestion causes to the drivers a big economical impact in the form of increased delays and travel time. While many different measures have been taken into consideration to contrast the problem, like promoting an efficient public transportation, a careful development of the road network can reduce traffic congestion. That is why many transport analysts, economists, mathematicians and, later, computer scientists have started to investigate how to cope with road congestion, finding models for traffic distribution and how to plan a good road network to avoid it.

First work that introduced the concept of traffic equilibrium was done by Frank Knight in 1924, where he presented an argument about how the taxation of roads can reduce the congestion to its efficient levels [1].

Based on Frank Knight work, in 1952 John Glen Wardrop introduced his First and Second principles of equilibrium [2]. The first principle was derived from game theory and formalized the notion of traffic equilibrium, based on the concept that agents selfish behavior degrades the system efficiency and leads to models with Nash Equilibrium. The second principle introduced the alternative behavior where the system optimal equilibrium is reached through the minimization of the average travel costs and therefore of the total costs.

The first basic formulation for the Transportation System Theory, that uses Wardrop's first and second principles of equilibrium, was done by Beckman in 1968. He introduced the concept of “network” to model the connections between traffic demand and supply on a territory and the interactions between demand and supply, while the traffic demand was still an independent variable not related to the system [3].

In the same year a German mathematician, called Dietrich Braess, demonstrated a paradox where adding one link to a network decreased the overall performance of the network, due to the selfish behavior of the agents and the consequent Nash equilibrium [4]. This paradox can happen in real world road networks as many successive studies demonstrated. For reference see the work of Springer and Verlag [5], where it is described how a new traffic section in Stuttgart was not effective in reducing traffic congestion until some sections where closed or [6], a New York Times article written in 1990 where it is described how closing a road decreased overall traffic congestion.

A better analysis to construct mathematical models for transportation demand was later given by Daniel McFadden [7]. Here the fundamentals for the Transportation System Theory were a number of hypotheses and relations that represented the supply of transportation services, the behavior of travelers and how supply and demand interact [8].

Today, many other studies have been done to improve the Transportation System Theory to determine facility needs, costs and benefits. The main problem in these studies is the traffic assignment, over a network, needed to simulate how additions made to a network affect the overall efficiency. Major urban centers that provide the infrastructures to monitor traffic have made very easy to collect large amount of data about traffic going in and out the city. Many of the older algorithms used to solve this problem, like Frank–Wolfe algorithm [9] or method of successive averages (MSA), suffer of high computation costs and slow convergence to equilibrium over complex networks. This made a primary concern the development of new algorithms that can reach equilibrium in a relatively low amount of time.

Finding a good algorithm is the subject of this thesis. To achieve this goal we use a particular branch of artificial intelligence derived from observation of natural biological systems, called Swarm Intelligence [10].

1.1 From swarm intelligence to flow assignment

Studies in biology about behavior of large communities of insects, like ants or bees, have given surprising results on their ability to solve problems through cooperation of many elements. Without cooperation this kind of problems would be otherwise impossible to approach because of the low intelligence or ability of a single member of the community. An interesting characteristic of these communities is the lack of a central authority that makes decisions for every member. The members simply relieve to a set of rules depending on the role taken inside a colony. For example, in the case of ants, it has been observed that an ant with the role of forager will only move outside the colony to bring back food if enough scout ants have returned. To accomplish this, the members of the community need the ability to communicate between them. Continuing the previous example, ants use the antennae to know if the other ants they are interacting with are members of the same colony and the role they assume inside the colony. The interactions are always simple and done locally, this means that ants rely only on local information to take decisions about what to do. Inspired by this idea computer scientists started to create mathematical procedures that imitate insects, flock of birds, fish schooling or bacterial growth behaviors to solve particular human problems like routing, scheduling and optimization in general.

Swarm intelligence and traffic supply/demand over a network have some interesting common features. Both describe the collective behavior of agents that interact locally within the environment, without any centralized control and with independent decision process from each other. Considering a static network where travel time is not affected by the numbers of agents traveling on a route, the Nash and optimal equilibrium is the same and it is reached when all the agents follow the shortest path. Finding the shortest path is a problem that has been successfully resolved by a particular swarm intelligence algorithm called Ant Colony System (ACS), that simulate the behavior of cooperating ants in finding and following the shortest path between a source of food and the nest [23]. This is done through a mechanism called stigmergy, used by ants to communicate locally using pheromone.

The aim of the thesis is to develop and implement a software that extends the ACS algorithm and makes agents search routes over a network where travel time is in relation with traffic intensity, i.e. travel time on roads is related to traffic intensity on the same road or other roads that intersect with it. Another important difference from standard ACO algorithm is the presence of multiple traffic supply sources and destinations that affect traffic over roads. The software has been tested on different traffic networks to analyze their behavior and to simulate the traffic distribution on real large cities like Milan or Naples. The result we obtained is, under different assumptions, to converge to user equilibrium in a reasonable time.

1.2 Outline of the thesis

The thesis is organized as follows:

- In **Chapter 2** we formalize the traffic assignment problem and discuss how equilibrium can be reached, particularly focusing on the Wardrop equilibrium and to some extensions to it. Next, we describe some algorithms used to find the equilibrium, like the Frank-Wolfe algorithm, and describe the principal algorithm classes used to resolve traffic assignment problems.
- In **Chapter 3** we first explain how the ant colony optimization algorithms works, and then describe a particular algorithm called Ant System. Next, we discuss how we extended the algorithm into a new algorithm called Ant Colony System for Traffic Assignment (ACS-TA) to resolve the traffic assignment problem, the problems encountered during the simulations and how we resolved them making various optimization to the algorithm.
- In **Chapter 4** we focus on the software implementation, explaining the software requirements, and what data we expect at the end of the algorithm execution. The software implementation is divided in two parts, the former is the transportation network model, the latter is the previously described ACS-TA algorithm and sub-algorithms implementation.
- In **Chapter 5** we describe the characteristics of the networks used to test the software execution. After, we compare and analyze the results of different simulation that used various parameters and sub-algorithms, to

find how they affect the algorithm in finding the equilibrium and the computation time to reach it.

- In **Chapter 6** we report the conclusions of the whole work, discussing on the results obtained. Next, we give several indications on future lines of works that can be followed both in extending the ACS-TA algorithm to achieve better solutions, and the possible software improvements.
- In **Appendix A**, we provide a manual to correctly configure and run the software, explaining what files are needed and how all the configurable parameters change the behavior of the algorithm. It is also explained how to do the best tuning for the Java Virtual Machine memory occupation, given a network.
- In **Appendix B**, we provide the class diagram of the software described.
- In **Appendix C**, we provide the network graphical visualization for every network used.

Chapter 2

Preliminary concepts

2.1 Introduction

In this chapter all the concepts needed to understand the scope of this thesis work are introduced. First the definition of the traffic assignment problem is given and the most common models of the problem are explained. Finally the most common algorithms used to find a solution to the problem are described.

2.2 The Traffic Assignment Problem

The problem to determine how users choose different routes between origins and destinations over transportation networks, taking in consideration the congestion on the passed roads, is called Traffic Assignment Problem. In the transportation realm, congestion usually relates to an excess of vehicles with respect to a portion of roadway at a particular time resulting in speeds that are much slower than normal or "free flow" speeds.

An instance of the traffic assignment problem is given by the transportation supply, and demand. The transportation supply is usually represented by the network topology, road geometry, road capacity and arc link travel cost functions

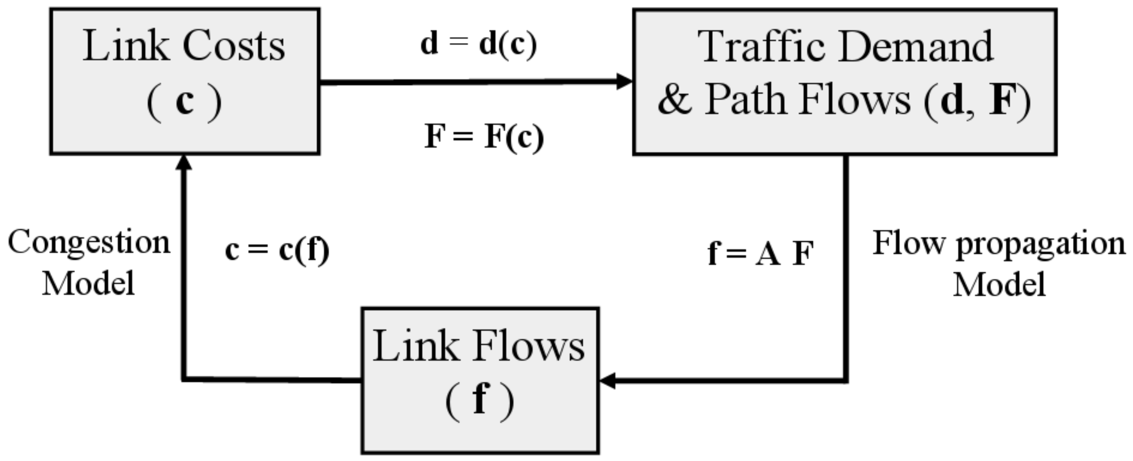


Fig. 2.1: A model of transportation system describing the equilibrium relationship between traffic demand,flows and costs. The cost c for traveling on a certain link depends on the observed traffic f generated by traffic demand. Traffic demand d, in turn, is distributed on links according to the cost vector c.

while the transportation demand is represented by the list of OD pairs and their demand rates. A transportation planner, who is generally involved in evaluating, assessing and designing the transportation facilities[1], needs to find or estimate all the elements that comprise the model. The topology of the network is usually digitized from maps, if it is not already available. Link travel cost functions are calibrated from historical information using tabulated functions that relate geometry of the road to capacity. One may need also to add tolls or other costs to the arcs, which, in most cases, can be converted to the same units by using the average value of time for the population. The transportation supply can usually be estimated from socioeconomic information coming from census data. The transportation demand can be measured directly or may come from historical OD matrices that can be calibrated using up-to-date traffic counts [15].

To formalize the Traffic Assignment Problem, we consider an instance where the network topology is modeled as a directed graph $G = (N, L)$ consisting a set of N

1 Examples of transport facilities are streets, highways, bike lanes and public transportation lines.

nodes and a set of L links. The transportation demand is modeled by a set of origin-destination (OD) pairs. Each node that is part of a pair is called centroid. The flow demand rate that must be routed from the corresponding origin to its destination is considered arbitrarily divisible and for each $k \in M$ it is equal to d_k. The set of routes connecting an OD pair in G is enumerated in I_k for every $k \in M$ while R is the union of all the possible routes. Each origin-destination demand d_k generates a set of network path flows F_i, with $i \in I_k$. For a given link $l \in L$, the sum of all path flows crossing this link is called

the propagation model:

$$f_l = \sum_{i \in I_k} a_{li} F_i \tag{1}$$

where a_{li} is 1 if the path i contains the link l and 0 otherwise. In matrix form:

$$f = AF \tag{2}$$

where F and f are the vectors of path and link flows, respectively with a dimension equal to the number of paths $|I_k| = n_{path}$ and to the number L of links, while the link-path incidence matrix A is made up of the a_{li}.

The model of a transportation system that we consider describes the behavior of the traffic demand, i.e., the average number of users moving between centroids in M, and its relationship with link flows according to the scheme of *Figure 2.1*. In particular let c_l be a function, called cost, with values in $\mathbb{R}_{\geqslant 0}$ that represent the travel time over an arc I, and C_i the total traveling time on a certain path i, depending on the observed traffic (f or F). Assuming that all costs on links are additive, the relationship between the c vector of link costs and the C vector of path costs can be written as:

$$C = A^T c \tag{3}$$

Traffic demand d is distributed on links according to the cost vectors c and C; in particular, the relationships between *F*, *f* and *d* are:

$$F = P(C(f))\,d(C(f)) \tag{4}$$

$$f = AP(C(f))\,d(C(f)) \tag{5}$$

where *P* is the path choice probability matrix, also known as path choice map of dimensions (n_{paths}, $|M|$). Each element p_{ij} of *P* expresses the probability that traffic demand d_i (of the *i-th* od-couple) is routed on path *j.* Its value is zero when path *j* does not start in O or does not end in *D*, otherwise it depends on the cost of traveling on path *j.* Its functional form can vary according to the distribution of the cost itself. The equilibrium solutions F^* and f^*, for Equation 4 and 5 can be written as:

$$F^* = P(A^T c(AF^*))\,d(A^T c(AF^*)) \tag{6}$$

$$f^* = AP(A^T c(f^*))\,d(A^T c(f^*)) \tag{7}$$

Equations 6 and 7 describe the circular dependencies upon which the equilibrium problem of *Figure 2.1* is based; these equations represent the fixed point solution, i.e., the equilibrium, of Equations 4 and 5. The dynamics of the system, and the transient until the equilibrium is reached, are not specified by the model we are focusing on; it is assumed that when the system reaches equilibrium it is steady, and this is the state we are interested to analyze. Equilibrium can be analyzed by taking into consideration another element regarding d_k, that is, demand elasticity. The demand d_k may be rigid, in the sense that the increasing costs due to congestion affect only the choice of the path; in this case the vector *d* is assumed invariant to link costs. Vector *f* or *F* are then defined by the equations:

$$F^* = P(C(AF^*))d \quad where \quad C(AF^*) = A^T c(AF^*) \tag{8}$$

$$f^* = AP(C(f^*))d \quad where \quad C(f^*) = A^T c(f^*) \tag{9}$$

Otherwise, if demand is elastic, that is, it depends on congestion costs as well as on system attributes, then Equations 6 and 7 must be used.

An example of rigid demand could be railway commuters whereas elastic demand could refer to weekend or holiday travelers. In general rigid demand assumptions can be adopted when we analyze mono-modal (that is, single mode) networks in standard conditions. Indeed, in these conditions user choices are related only to path choice considering fixed any other demand dimension (mode and/or destination choice). Therefore, elastic demand hypotheses have to be assumed in the case of multi-modal (that is, with at least two different modes) networks or in the case of unusual conditions, that is, when user reconsider path choice jointly with mode/destination choice.

2.3 Wardrop equilibria

A common assumption in the study of transportations systems is that a traveler choose the route that he perceives as the fastest (or least expensive) to reach his destination, taking in consideration the traffic congestion on the roads [11].

The consequences of these individual decisions are that travelers cannot reduce the travel time choosing unilaterally a different route, creating a condition called Wardrop equilibrium.

The first principle of Wardrop that describe this equilibrium is here enunciated [2, p. 345]:

<< The journey times on all the routes (of the same origin/destination couple) *actually used are equal, and less* (or equal) *than those which would be experienced by a single vehicle on any unused route. >>*

The same concept was already enunciated by Kohl [11] and Knight [1] in precedent works. This principle regarding the path choosing has been accepted as simple and sound to describe the behavior of the travelers that have to choose the route to take under traffic congestion conditions [12].

The Wardrop equilibrium is considered as the result of a transitory phase where the travelers iteratively change the chosen route until the situation becomes stable, which means that everyone travels the fastest route perceived and there are not any more variations of traffic flow on the transportation network.

A mathematically formalized Wardrop equilibrium in the context of transportation networks was done by Beckmann, McGuire and Winsten in 1952 [13], and it has become the most used by network planners to predict the decision taken by travelers on real networks [14, 15]. This model have remained valid until today to estimate how the traffic is redistributed after a change on the transportation network, like adding a road, a bridge or introducing of tolls.

Wardrop's first principle can be interpreted as requiring that flow travels along the shortest paths because no single user can change his own route reducing his travel cost. That means Wardrop equilibrium can be studied by means of the following variational inequalities:

$$C(F^*)^T(F-F^*)\geq 0 \,\forall F\in S_F \tag{10}$$

where S_F is the set of admissible path flow vectors. Equivalent variational inequality models are based on link flow leading to:

$$c(f^*)^T(f-f^*)\geq 0 \,\forall f\in S_f \tag{11}$$

where S_f is the set of admissible link flows vectors.

Beckmann et al. [13] proved that such a flow always exists by considering the following min-cost multicommodity flow problem with separable objective function:

$$f^*=arg\,min\sum_{l\in L}\int_0^{f_l} c_l(z)dz : f\in S_f \tag{12}$$

The previous problem is convex because the objective is the integral of non-decreasing function and, since its domain is a compact set, the problem solution attains its optimum. If cost functions c_l are strictly increasing, f is unique. Computationally, equation (12) implies that an equilibrium can be calculated using general convex optimization techniques.

Charnes and Cooper [17] were the first to notice that the concepts of Nash and Wardrop equilibria are related, while Haurie and Marcotte [18] proved that a Nash equilibrium in a network game with a finite number of players converges to a Wardrop equilibrium when the number of players increases.

For this reason, although the solution concepts are different, a Wardrop equilibrium can be viewed as an instance of a Nash equilibrium in a game with a large numbers of players. De Palma and Nesterov [19] looked at generalizations and alternative definitions of the basic model and established conditions that guarantee the existence of equilibria. For example, Wardrop equilibria still exist if cost functions are lower semicontinuous[1]. Marcotte and Patriksson [20] also discussed alternative definitions of equilibria in network games and the relationships between them. Since the Wardrop equilibrium considers that users unilaterally choose their routes to minimize their route cost, the solution is not necessarily efficient.

The second Wardrop principle gives a definition that leads to a social optimal equilibrium [2]:

<< *At equilibrium the average journey time is minimum.* >>

It states that users minimize the total travel time in the system, a system optimum f^* is an optimal solution to the min-cost multicommodity flow problem:

1 A definition of lower semicontinuous function can be found at http://en.wikipedia.org/wiki/Semi-continuity.

$$min\, C(f) : f \in S_f \tag{13}$$

As general equilibria typically do not minimize the social cost, Koutsoupias and Papadimitriou [21] proposed to analyze the inefficiency of equilibria from a worst-case perspective; this led to the notion of "price of anarchy" [22], which is the ratio of the worst social cost of a Nash equilibrium to the cost of an optimal solution.

2.4 Extensions to Wardrop equilibria model

The Wardrop equilibrium is also a Deterministic User Equilibrium (DUE) because it makes the following assumptions:

- all travelers are perfectly aware of the travel times on the network;
- all travelers are always capable of identifying the shortest travel time route;
- network travel times are deterministic for a given flow pattern.

This assumption will not stand in the real world, where we cannot realistically assume the drivers have an exact idea of the length of every possible route connecting an origin to its destination or about the real topology of the network. A more realistic model would introduce some uncertainty in the decision making process. The way to deal with this issue is to assume that the drivers estimate the travel time of the routes, i.e. that their perception is affected by some random errors.

To overcome the limitations of the deterministic model, some researchers have proposed different Stochastic User Equilibrium (SUE) models to leave aside the assumption of perfect knowledge of network travel times and to take into account dispersion among users, user perception errors, and modeling errors [25, 26, 27]. Stochastic user equilibrium models date back to the 1970s, when Dial [24] proposed a model where the demand on each OD is distributed among routes (with random lengths) according to a logit distribution, in the case of uncongested traffic networks. In the same work Dial tried to reduce the route enumeration, taking in consideration for the flow distribution only "efficient routes" (see paragraph 3.4.2).

Analogously to the deterministic equilibrium, an optimization problem can also be formulated for SUE and, in case the Jacobian of the cost function is symmetric, it can be written as:

$$f^{*}=arg\,min\sum_{k\in M} d_k\, s_k\left(-\Delta_k^T c(f)\right)+c(f)^T f+\int_0^{f_l} c_l(z)\,dz : f\in S_f \qquad (14)$$

where f is the link flow, f^* is the link flow which minimizes the objective function, c is the link cost, d is the demand, and s is a measure of demand satisfaction (or utility).

Another unrealistic assumption in the Wardrop equilibria model is that travel time on a road is not affected by the congestion level on other roads that intersect with it. This cannot be true if we consider roads ending with non-signalized intersections such as T-intersections and roundabouts, still many algorithms used for resolution use convex optimization techniques need strictly increasing cost functions. The reason behind the difficult to abandon this assumption is that models with highly general and realistic equilibrium foundations with non-separable link cost functions suffer from computational difficulties [29] and some tradeoff between theoretical consistency and computational tractability is needed, particularly for urban-scale travel demand analyzes. Research has been done to find models that extend Wardrop equilibria releasing some of the more strict assumptions while maintaining a good computational speed. Examples of these extensions allow tradeoffs among cost components in route choice [30], contain temporal dynamics [31] and allow for imperfect decision-making and information [32].

2.5 Frank-Wolfe algorithm

Frank-Wolfe algorithm can be applied on convex optimization problems of the form:

$$\min_{x \in D} f(x) \tag{15}$$

In the traffic assignment problem, *f* is the cost function of the roads and it needs to be continuously differentiable in the domain *D*, while *x* is a vector containing for each link the flow of an OD. The algorithm follows these steps for each OD:

1. Choose an initial solution $x^{(0)} \in D$.

   In traffic assignment problem this means choosing randomly how the traffic distributes on the network for each OD. For example, an initialization can be choosing the shortest route considering the cost when no flow is present on the network.

2. Determine a search direction p_k

   In the Frank-Wolfe algorithm one determines p_k through the solution of the approximation of the problem (15) that is obtained by replacing the

Algorithm 2.1: Frank-Wolfe conditional gradient method (1956)

```
Initialize $k = 0$
Let $x^{(0)} \in D$
do
    Compute $p_k := y_k - x_k$ that minimize $f(x_k) + \nabla f(x_k)^T (y - x_k)$
    Determine step length $\alpha_k \in [0,1]$ that minimize $f(x_k + \alpha p_k)$
    Calculate $x_{k+1} = x_k + \alpha_k p_k$
    Increment $k$
while $(f(x_k) - z_k(y_k)) / |z_k(y_k)| \le \epsilon$
```

function *f* with its first order Taylor expansion around x_k: therefore, solve the following problem:

$$min\{z_k(y) := f(x_k) + \nabla f(x_k)^T (y - x_k)\} \tag{16}$$

The value of *y* needs to be in the domain *D*. This is an LP problem, and it gives an extreme point, y_k, as an optimal solution. The search direction is $p_k := y_k - x_k$, that is the direction vector from the feasible point x_k towards the extreme point. Observe that this is a feasible direction, since both x_k and y_k belong to *D* and *D* is convex.

3. Determine a step length α_k, such that:

$$f(x_k + \alpha_k p_k) < f(x_k) \tag{17}$$

Here, we must limit the step length to be at most 1, because for $\alpha > 1$ the solution becomes infeasible; the line search therefore has the form:

$$min_{a \in [0,1]} \{f(x_k + \alpha p_k)\} \tag{18}$$

4. New iteration point:

$$x_{k+1} = x_k + \alpha_k p_k \tag{19}$$

5. Check the stop condition:

$$(f(x_k)-z_k(y_k))/|z_k(y_k)| \leq \epsilon \tag{20}$$

If it is fulfilled then stop, else go back to point 2

2.6 Other algorithms for Traffic Assignment

The Frank-Wolfe algorithm is an example of *decomposition algorithm*, which separate the main problem into subproblems. It performs well for problems with separable objective function (12), but sometimes it shows poor convergence because it tends to move around the equilibrium solution.

Another famous decomposition approach is done in the Bar-Gera's algorithm [33], where flows are separated by node of origin whereby every iteration assigns all destinations for each origin at the same time. This algorithm has proven to be one of the most efficient to compute Wardrop equilibria.

Partial linearization is a class of algorithms that try to simplify the objective function to be able to find a search direction. An example is the Partan (parallel tangents), developed by Leblanc, Helgason, and Boyce that determines the descent direction using the results of two consecutive iterations, diminishing the oscillations around the equilibrium [34].

The class of *column generation* algorithms deal with a path formulation of the model. These algorithms are necessary used when there are constraints based on paths, because an arc formulation is not powerful enough to represent the problem, or when costs along routes are not addictive. Instead of keeping track of all the possible routes, new routes are added only when discovered or needed during the search direction procedure. The path formulation (12) uses only the discovered routes.

The class of *simplicial decomposition* algorithms are similar to the Frank-Wolfe algorithm, but at each iteration instead of searching through all the possible solutions, they use a subset of the restricted set of solutions computed from the previous iterations. Since all the route information previously computed are badly utilized by algorithms that perform line search, this class can solve problems more efficiently at the cost of doing more work per iteration.

Last is the *method of successive averages,* a heuristic method used for computing equilibrium in complex models where exact techniques are not available. It starts by computing the costs on all arcs for an arbitrary feasible flow. After that, it iteratively computes a new solution using an auxiliary linear program that keeps costs fixed, and updates the current solution by averaging it with the new one using a factor that depends on the iteration.

Most commercial software packages that resolve traffic assignment problem use the algorithms described here. A non-exhaustive list of software implementations is Aimsun [45], Cube [46], CONTRAM [47], DynaMIT [48], DYNASMART [49], Emme/4 [50], Paramics [51], TransCAD [52], transims [53], TSIS-CORSIM [54], SATURN [55], Vistro [56] and VISTA [57].

Chapter 3

Ant colony optimization for the Traffic Assignment

3.1 Introduction

In this chapter we first introduce how the basic ant colony optimization algorithm works and the advantages of using this type of algorithms to solve combinatorial optimization problems. Next we explain the Ant System algorithm and finally show how the algorithm has been modified to solve the traffic assignment problem.

3.2 Ant Colony Optimization

Ant Colony Optimization (ACO) is a paradigm for designing meta-heuristic algorithms for combinatorial optimization problems, ranging from quadratic assignment to protein folding or routing vehicles. The first algorithm which can be classified within this framework was presented by Dorigo in 1991 [35, 36] and, since then, many different variants of the basic principle were reported in the literature.

Ant Colony Optimization algorithms have been originally inspired by Dorigo observation of real ants behavior during the search for food, where he discovered that after sufficient time, ants tend to find and follow the shortest path between the nest and the food source. This is done thanks to stigmergy, a mechanism of indirect coordination between agents [37], where an action done by an agent leaves a trace in the environment and stimulates the performance of another action by the same or different agents. Subsequent actions tend to reinforce and build on each other, leading to the spontaneous emergence of coherent, apparently systematic activity. This mechanism is used by ants during the exploration around the nest in search for food, where they release pheromones that make the path more likely followed by them or by other ants. the more pheromone is present on a path, the more likely that path will be preferred over other paths. The shortest path to a food source will accumulate pheromone faster than the longer ones, and this will increase the number of ants choosing it. Finally, pheromone evaporates over paths, and this leads ants to avoid choosing longer paths over the shorter one because, given enough time, only the shortest path will have pheromone. This behavior has a weakness: when all ants follow the shortest path if a new shorter path becomes available it will be probably ignored.

The principal trait of ACO algorithms is the use of meta-heuristic to find a solution using information from previous iterations. This is possible either starting from a null solution and adding elements to build a good complete one, or making a local search starting from a complete solution and iteratively modifying some of its elements in order to achieve a better one. The meta-heuristic approach allows to search over a wide number of solutions, possibly avoiding local optima. The use of elements found in previous iterations is combined using a Monte Carlo approach to find a better solution.

The particular way of defining components and associated probabilities is problem-specific, and can be designed in different ways, facing a trade-off between the specificity of the information used for the conditioning and the number of solutions which need to be constructed before effectively biasing the probability distribution to favor the emergence of good solutions. Another advantage over simulated annealing and genetic algorithm approaches of optimization problems is that ACO algorithms can be run continuously and adapt to changes in real time, like in the case of network routing and urban transportation systems. Lastly, ACO algorithms can take advantage of using several constructive computational threads that do a parallel search to find a problem solution. Every thread uses local problem data and a dynamic memory structure containing information on the quality of previously obtained results. The collective behavior emerging from the interaction of the different search threads has proved effective in solving combinatorial optimization (CO) problems.

3.3 Ant System

The first algorithm of the ant colony optimization paradigm to resolve combinatorial optimization problems was developed by Dorigo and is called Ant System (AS) [35].

A combinatorial optimization problem is defined over a set $C := c_1, \ldots, c_n$ of basic components. A subset *S* of components represents a solution of the problem; $F \subseteq 2^C$ is the subset of feasible solutions, thus a solution *S* is feasible if and only if $S \in F$. A cost function *z* is defined over the solution domain, $z: 2^C \rightarrow R$, the

Algorithm 3.1: Ant system (1991)

```
Initialize $\tau_{ij}^{\alpha}$ and $\eta_{ij}^{\beta}, \forall (ij)$
do
   for each ant k (currently in state i) do
     repeat
         choose in probability the state to move into.
         append the chosen move to the k-th ant's set $tabu_k$ .
      until ant k has completed its solution.
   end for
   for each ant move (ij) do
      compute $\Delta\tau_{ij}$
      update the trail matrix.
   end for
while found better solution
```

objective being to find a minimum cost feasible solution S^*, i.e., to find $S^*: S^* \in F$ and $z(S^*) \leq z(S) \forall S \in F$. To find a solution for this type of problems, AS uses a set of concurrent and asynchronous agents called ants, that move through states of the problem corresponding to partial solutions. Each move of an ant from a state *i* to a state *j* is chosen through a stochastic local decision that is based on 2 parameters:

- the *attractiveness* η_{ij} of the move that indicates a fixed desirability of the move;
- the *trail level* τ_{ij} of the move, indicating the quantity of pheromones released by ants that chose this move in the past. The more pheromones are present, the better was the solution found by the ants that chose this move, leading to increase move desirability.

The move probability distribution used by an ant *k* to move from a state *i* to a state *j* is the following:

$$p_{ij}^{k} = \left\{ \begin{array}{cc} \frac{\tau_{ij}^{\alpha} + \eta_{ij}^{\beta}}{\Sigma_{(ij) \notin tabu_k} (\tau_{\Im}^{\alpha} + \eta_{\Im}^{\beta})} & if\,(ij) \notin tabu_k \\ 0 & otherwise \end{array} \right\} \quad (21)$$

where $0 \leq \alpha, \beta \leq 1$ are user defined parameters that give more weight to trail or attractiveness and $tabu_k$ is the set of not permitted movements for ant *k*.

The ants continue to change the state in a loop until a complete solution is found. At this point, every ant evaluates the solution and releases the pheromone, while some pheromone previously released evaporates. The trail update formula is the following:

$$\tau_{ij}(t) = \rho \tau_i(t-1) + \Delta \tau_{ij} \tag{22}$$

where $\Delta \tau_{ij}$ represents the sum of the contributions of all ants that have used move (*ij*) to construct their solution and ρ is a user-defined parameter called evaporation coefficient that assumes a value between 0 and 1. The ants contributions are problem dependent, proportional to the quality of the solutions achieved, i.e., the better is a solution found by an ant, the higher is the trail contributions added to the moves used by the ant. For example, if we want to find the shortest path in a graph, the pheromone released would be inversely proportional to the travel distance.

The main loop where *m* ants construct in parallel their solutions and release pheromones continues until there are not many variations on the solutions found by the ants and no better solution is found after some iterations.

3.4 Ant Colony System for Traffic Assignment

The Ant System algorithm described in the previous paragraph was applied to traffic assignment problems by Matteucci and Mussone in [38, 39], where they studied the influence that parameters such as pheromone or heuristic information have on the ACO meta-heuristic performance. Leveraging from there, D’Acierno proposed an MSA algorithm for SUE simulation based on the ant colony optimization paradigm [46]. His work is particularly relevant since it states that, under some specific hypotheses, the ant system originally proposed is capable of solving a particular SUE formulation of the traffic assignment problem.

The algorithm that we use, called Ant Colony System for Traffic Assignment (ACS-TA) is a generalization of the D'Acierno analysis, with the following assumption made on the traffic assignment problem:

1. The travel time (cost) over a road can be dependent from the travel time over other different roads, like in the case of roundabouts. This leads to non separable cost functions that are not generally monotonically increasing.
2. There can be uncertainty in the path decision making process, to simulate the imperfect perceptions of the drivers when they estimate the travel time of the routes. This leads to a Stochastic User Equilibrium (SUE).
3. There are many different origin/destination pairs, each of them can generate flow for different categories of vehicles.
4. Some roads can be traveled only by a subset of vehicle categories, to simulate the common situation where roads are reserved to particular vehicles like taxi.
5. Some roads are part of restricted traffic zones that have a toll for particular vehicles categories.

The first and second assumption are of great importance and need to be further analyzed because they go against the conditions needed to reach Wardrop equilibrium. To better understand the implications, let's consider the traffic assignment as a problem where we have to find the fixed point that is solution of:

$$f=P(c),c=H(f) \tag{23}$$

where f is the path flow, c is the path cost and P and H are continuous functions. Under proper assumptions on P and H, equation (23) becomes a composed fixed point problem that can be written as $f = P(H(f))$ or $c = H(P(c))$, where P is the user choice model and H is the link cost function like the one reported later in equation (55).

Cascetta and Cantarella in their works [41, 42] demonstrated that the existence of the solution for such fixed point problems is guaranteed to exist for SUE and it has at least a solution if path choice probability functions and cost functions are continuous. Uniqueness is guaranteed when link cost functions are strictly monotonically increasing. No assumptions over equilibrium can be made for cost functions that are not monotonically increasing as in the case of non separable cost. For DUE, the same conditions obviously hold, keeping in mind that there is no uncertainty in the path choice policy since it depends only on costs. Hence, with DUE, existence is guaranteed if link cost functions are continuous, while uniqueness is guaranteed when cost functions are monotonically increasing.

The third and forth assumptions are easily handled in the ant colony algorithms, because every origin/destination/category combination can be considered as an independent entity, handled by his own thread that see only the portion of the network he has access to. Roads not accessible by a given category are simply ignored by the colony as if they do not exist. Independent entities can easily be computed in parallel, drastically increasing the algorithm performance.

Algorithm 3.2: Ant colony system for traffic assignment (ACS-TA)

```
Initialize the network and the pheromone trails
do
   for each colony do
      for each ant do
         find the path from origin to destination (sequence of nodes and links)
         deposit pheromones on the path
      end for
   end for
   Evaporate the pheromones
   Assign flow
   Calculate cost on links
while convergence not reached
```

Finally the fifth assumption is easily implemented considering a toll as a generalized cost added to the total travel cost of a route that runs through a limited traffic zone. The amount of the added cost is dependent on the vehicle category.

As in Ant System, ACS-TA is a meta-heuristic which uses many iterations and information found on previous iterations to determine how the flow distributes on a directed graph that models the topography of a city. Each link in the graph contains the following information: the travel cost and, for each origin/destination/category, the flow and the pheromone released by ants. The distribution of the pheromone and the traffic flow for each origin/destination/category combination is elaborated by an independent ant colony, that uses his own pheromone trails. At the end of the iteration the travel cost on each link is obtained using the cost function and the sum of flows on the current link, or the links that intersect with it in the non separable cost case. In the next sections the various steps of the algorithm, used for each colony, will be analyzed.

3.4.1 Network initialization

An important concept in ACS-TA is that when ants choose the next link to travel, only the pheromone trail level comes into play in the decision process. To achieve this, an initial pheromone distribution is needed on each link if we want to guarantee some degree of initial exploration. If this is not done, after the first iteration only links that have pheromone, which are all the links that have been previously chosen by ants, can be chosen again.

When assigning the initial pheromone on the links, we have to consider that every origin/destination pair can have very different average path costs. Initializing with an arbitrary value of pheromone on each link for every origin/destination couple is very inefficient and should be avoided. If we use too much initial pheromone on a link in respect to the average pheromone released by ants, the released pheromones will not affect the desirability of a link until the initial pheromone evaporate enough and reach an order of magnitude similar to the one of the released pheromone. Using too few initial pheromone will cause a high pheromone difference between the links that have been chosen by ants during the first iteration and the links which have not, making very unlikely for ants to use a new path that does not contain a link previously chosen.

To avoid this, for every origin/destination couple, we use the shortest path obtained using Dijkstra's algorithm to find a path cost which can be used to simulate how much pheromone will be released by the ants during the normal algorithm execution. Using that cost, we can initialize the links with a value that is at least of the same order of magnitude of the future released pheromone.

The pheromone initialization is done in two steps:

1. First the cost C_{min} of the shortest route F^*_{min} from the origin to the destination is calculated:

$$C_{min} = \sum c_{i,j}(Cap_{i,j} \cdot Const_f, 0) \ \forall \ c_{i,j} \in F^*_{min} \tag{24}$$

where $c_{i,j}(f,t)$ is the cost of link that goes from node *i* to node *j* at time *t* having a flow of *f*, $Cap_{i,j}$ is the capacity of the same link, F^*_{min} is the set of links that are part of the shortest route, and $Const_f$ is a value between 0 and 1 that we can choose: the higher the value, the less pheromone will be released, making the pheromone evaporate faster and decreasing the exploration; the lower the value, the more pheromone will be released making the desiderability less affected by ants decision on the first iterations, increasing exploration and execution time.

2. The pheromone on each link is initialized with the amount that would be released by an ant choosing a path with cost C_{min} :

$$\tau^c_{i,j}(0) = Const \cdot R(C_{min}) \ \forall i,j : l_{i,j} \in L \tag{25}$$

where *Const* is another constant to increase pheromone and leads to better exploration at cost of slowest convergence, and $R(C_{min})$ is the pheromone release function, that will be explained in detail in the next paragraph.

The first iteration of the algorithm is done with the network without any traffic, so the flow is initialized to 0 in each link. For the purpose of traffic distribution, we need to save the total released pheromone $\check{\tau}^C_{tot}(0)$ and the released pheromone on each link $\check{\tau}^C_{i,j}(0)$, both initialized to 0 because no pheromone has been released by ants at the beginning.

3.4.2 Ant exploration and travel from origin to destination

In ACS-TA each ant builds a solution, that is, a path from the origin to the destination of its colony, by selecting at each node *i* of the graph the next link $l_{i,j}$

to be added to the path. This choice is made according to a decision probability table $P_i^c = [p_{i,j}^c(t)]_{|L_i|}$, where the probability of selecting a forwarding link $l_{i,j}$ depends on the pheromone trail left on the same link by the preceding ants belonging to the same colony *c*. The basic function used is the following:

$$p_{i,j}^c(t) = \frac{\tau_{i,j}(t)}{\sum_{i,j\in L_i} (\tau_{i,j}(t))} \quad \forall i,j : l_{i,j} \in L \tag{26}$$

where $\tau_{i,j}(t)$ is the amount of pheromone trail on link (*i, j*) at time *t* and L_i is the set of outgoing links from node *i*.

Another possible function introduces some uncertainty to the pheromone perceived by ants, making it a stochastic process; the perception error is determined using a Gaussian function:

$$p_{i,j}^c(t) = \frac{max(0, \tau_{i,j}(t) + N(0,\sigma)\cdot\tau_{i,j}(t))}{\sum_{i,j\in A_i} max(0, \tau_{i,j}(t) + N(0,\sigma)\cdot\tau_{i,j}(t))} \quad \forall i,j : l_{i,j} \in L \tag{27}$$

Increasing error in perception has a double effect: it can increase the probability to choose a link with low pheromone increasing path exploration, and by adding a perception error on the usefulness of a link, it leads to a stochastic user equilibrium. A high degree of exploration is an advantage on smaller networks to try many different paths, but it has a drawback on more complex network.

An important rule is that a path between two nodes, with a total cost of C_1, which crosses a node more than once, it will always contain a sub-path without repeated nodes and with a total cost $C_2 < C_1$. Taking into consideration this rule, we know that if an ant chooses a link that leads to a node it already passed, it will find a

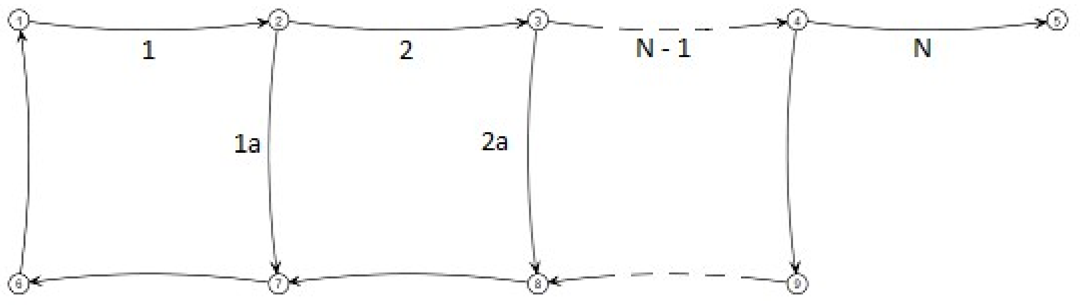


Fig. 3.1 : an example of network where traffic moves from node 1 to node 5. Only one possible path without repeated nodes is possible.

sub-optimal solution. To avoid this we have considered two possible improvements:

- **Solution 1:** When an ant moves on a node already passed, it forgets the route taken and starts again from the beginning.
- **Solution 2:** When an ant choose what link it will take next, it only considers links that lead to nodes he never passed. If no possible link can be selected, he forgets the route taken and starts again from the beginning.

Both solutions need that ants memorize the nodes they already passed, and both have some advantages and disadvantages. The first solution has the decision probability table P^c computed only once at the beginning of the iteration and used by all the ants. The second solution cannot use a fixed probability table, because the probability to choose a link is measured at every node using only a subset of A_i where all the links that lead to an already passed node are excluded. An example, where we can measure which method is more convenient, is given in *Figure 3.1*. This network has one path that connects origin and destination and *(N-1)* paths that come back to the origin node where ants block and restart. Let's examine the computation complexity for both solutions on the first iteration, when every link has the same probability to be chosen. In this example, consider the cost of determining the probability to choose a node being comparable with the cost of moving on links.

Solution 1: the decision probability table P^c is calculated only once at the beginning of the iteration *{1}* and it will be used by all the *m* ants. On the first iteration every neighbor link has the same probability to be chosen and this gives, in this particular network, a probability of $p_o=1-1/2^N$ to choose a path that brings back to the origin node and a probability $p_o=1/2^N$ to choose the only path that leads to destination. The average number of times an ant will choose a path that goes back to the origin is $1/p_0=2^N$, and every time it happens he will move on an average of $\sum_{k=1}^{N}(\frac{1}{k}k)=N$ nodes before arriving to the origin and reset *{2}*. When it chooses the path that go to destination, he will go through *N* nodes *{3}*.

$$\underset{\{1\}}{O(N)}+O(m(\underset{\{2\}}{O(2^N N)}+\underset{\{3\}}{O(N)})) = O(2^N N m) \tag{28}$$

Solution 2: as in the first solution, an ant will choose the path that goes back to the origin with an average of *N* times, but in this case it moves to the next node, it will have to recalculate the probability to choose the out links *{1}*.

$$O(m(O(2^N N)\underset{\{1\}}{O(1)}+O(N)\underset{\{2\}}{O(1)})) = O(2^N N m) \tag{29}$$

In this example, solutions 1 and 2 have the same computational complexity, because the larger term, that is the cost of ants moving on the nodes, dominates on the cost given by the probability calculation term.

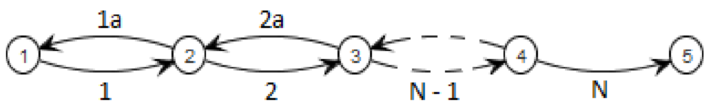


Fig 3.2: another example of network where traffic moves from node 1 to node 5.

Let's now examine a second example given in *Figure 3.2*. In this example, there is again only one path that leads to destination without repeated nodes and *(N-1)* paths that come back to the origin node.

Solution 1: we can make the same considerations of the first example, the only difference is that an ant does not have to come back to the beginning to restart when it chooses a wrong path. The computation complexity is then of $O(2^N m)$.

Solution 2: all the links that are part of a path that would reset the ant are excluded, only one link remain to choose so the probability is never computed and it takes only one iteration for each ant to find the destination node. The computation complexity is $O(N m)$.

As we can see, the performance depends on the network topology. The second solution can achieve a much better performance on networks with many back links, but its complexity is the same as that of Solution 1 in the worst-case scenario. An observation we can make about the topology of the real word transportation networks like the ones that we will use for the experiments, is that it is very common to have back links to nodes that were already passed like in the examples shown in *Figure 3.1* and *Figure 3.2*. This is easily understandable, as real word road networks are usually designed to make simple to reach any place starting from any position using roundabouts, intersections where it is possible to do a U-turn or taking a secondary road parallel to a one-way road.

Another possible improvement, using the second solution, is to save in a blacklist all the found paths that lead an ant to reset and share this information among all the ants of the colony. The ants can then avoid links that would result in choosing a path contained in the blacklist. Excluding a large number of links in the decision process, without having to enumerate all possible paths, can lead to a good increment of computation performance.

Let's consider again the network in *Figure 3.1*, where Solution 2 had the same performance as Solution 1. When an ant finds one of the *(N-1)* paths that brings back to the origin, it examines the chosen links starting from the last one and moves backward until it was possible to choose between more than one link, then it saves the link sequence from the start by putting it into a blacklist. An example of sequence is 1→2→2a; if the sequence is into the blacklist when an ant does the movement sequence 1→2, it will not consider the link 2a when calculating the probability to move to the adjacent nodes. To analyze the computation complexity, let's consider the worst case scenario, where the first ant puts into the blacklist every path that go back to the origin, starting from the longest path, before he can reach the destination node *{1}*. The following ants will reach the destination node using the remaining path *{2}*. The computation complexity is the following:

$$\begin{gathered} O(\overset{\{1\}}{\textstyle\sum_{i=1}^{N}}(2(N-i)))O(1)+O((m-\overset{\{2\}}{1})O(N)) = \\ O(\textstyle\sum_{i=1}^{N}(N-i))+O((m-1)N) = \\ O(N^2)+O((m-1)N) \underset{m\leq N}{=} O(N^2) \end{gathered} \tag{30}$$

On networks with many nodes, if we keep the number of ants less or equal to the number of nodes, the complexity becomes $O(N^2)$ that is comparable to the Solution 2 in the best case scenario. If we consider that the blacklist can be saved and used in the following iterations, or even in the following algorithm executions, being it dependent on the network topology, the advantages of using a blacklist can increase even more. In the example used above, the computation cost of reading and saving the blacklist is *O(1);* this assumption cannot be possible in the real application and the efficiency of the blacklist largely depends on how it is implemented. We will examine better how the implementation is done in Chapter 4.4.3. In a worst-case scenario a blacklist basically enumerate the complement of all the possible paths between an origin and destination, without repeated nodes. On a network as simple as the one in *Figure 3.1* the enumeration has a cost of $O(N^2)$, but on a hypothetical network where all nodes are intersected with every other node the computation cost is $O(N^N)$, making the blacklist not enough to decrease the computation time and ensure the ants reach the destination node by filtering not useful paths. Without a blacklist the probability to choose a path that reaches the destination node remains extremely low and, as some simulations in real networks demonstrated, ants cannot find the path to reach the destination after many thousands iterations.

To address the problem of helping ants to find a path to the destination, we need to sacrifice some of the exploration and increase the probability to choose links that are part of paths that lead to destination:

$$\begin{aligned} p_{i,j}(t) &= \frac{\left(\tau_{i,j}(t)\cdot b_{i,j}^{l}\right)}{\sum\limits_{i,j\in P_{n_i}}\left(\tau_{i,j}(t)\cdot b_{i,j}^{l}\right)} \\ l_{k+1} &= \begin{cases} 0 & \text{if } k=0 \text{ or all the ants arrived at destination} \\ l_k & \text{if an ant arrive at destination at the } k+1 \text{ try} \\ l_k+1 & \text{if an ant is blocked at } k+1 \text{ try} \end{cases} \end{aligned} \tag{31}$$

where *k* is the number of failed tries done by ants to reach the destination node during an iteration of the algorithm and b is the bias applied to the path. The bias is increased exponentially as the ant continues to reset, until it reaches a value where the ant can finally reach the destination node. This mechanism makes possible for ants to reach the destination while keeping the most possible exploration and is most effective on networks with many origin/destination pairs that need different value of bias or no bias at all, depending on where the pair is positioned.

To calculate the value of the bias, a possible solution is to apply it to the shortest path found using Dijkstra shortest path algorithm:

$$b_{i,j} = \begin{cases} Const_1 & \forall\, i,j : l_{i,j} \in F^*_{min} \\ 1 & \forall\, i,j : l_{i,j} \notin F^*_{min} \wedge l_{i,j} \in L \end{cases} \tag{32}$$

where F^*_{min} is the shortest path and $Const_1 > 1$ is a value we can choose. The higher is the value of $Const_1$, the faster the bias value will reach a point where ants start to find the destination, while the smaller it is, the most exploration is ensured when the bias value is found, but it can take many more iterations to find it. The shortest path is calculated only when needed, maximum once every iteration because the shortest path can change only if the flow distribution changes. The computation cost of the Dijkstra shortest path algorithm is, in a network with a set of N nodes and A arcs, $O(|A| + |N| \log |N|)$.

Another possible solution to help ants to find a path to the destination, is to give a bias to the links that are part of Dial definition of "efficient routes". A link is part of the set E of efficient links only if the following condition is met:

$$\begin{array}{l} l_{i,j} \in E \Rightarrow r(n_i) < r(n_j) \wedge s(n_i) > s(n_j) \\ r(n_i)\text{: the smallest cost from origin node } r \text{ to node } i \\ s(n_i)\text{: the highest cost from node } i \text{ to destination node } s \end{array} \tag{33}$$

When we have found the set *E* of efficient links, the bias value is calculated as in the best path example:

$$b_{i,j} = \begin{cases} Const_1 & \forall\, i,j : l_{i,j} \in E \\ 1 & \forall\, i,j : l_{i,j} \notin E \wedge l_{i,j} \in L \end{cases} \qquad (34)$$

Using a bias on efficient links guarantees a much higher exploration on complex networks, at the cost of a much higher computation complexity, because the Dijkstra shortest path algorithm needs to be used on every node twice instead of only once for origin and destination, leading to a computation complexity of $O(O(A+N\log N)2\mathrm{N}) = O(AN+N^2\log N)$

An example of efficient routes is shown in *Figure 3.3*.

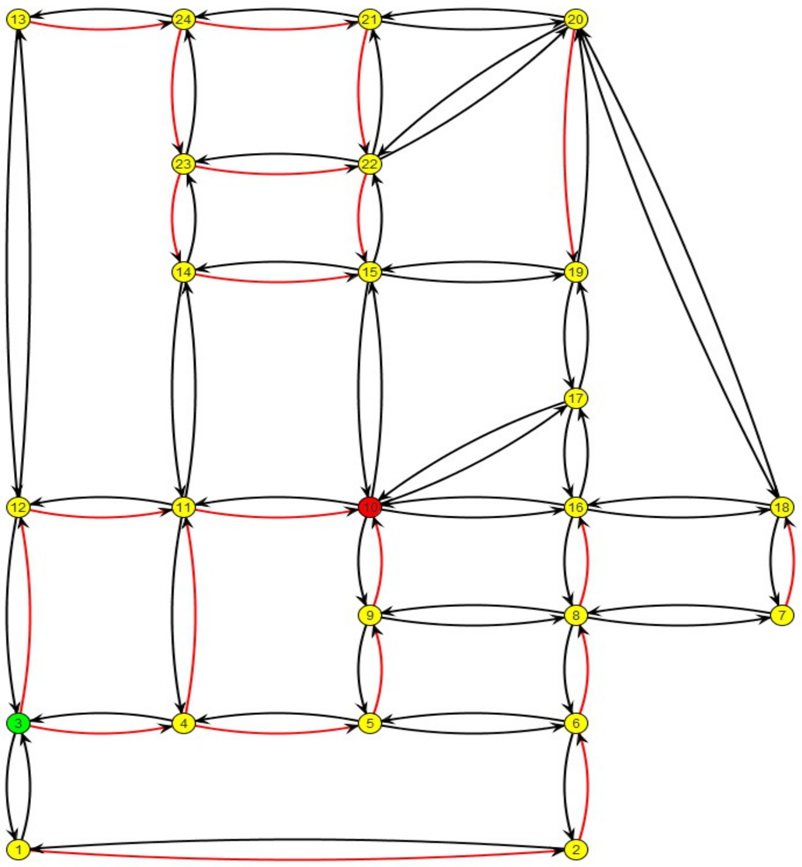


Fig 3.3: The Sioux-Falls network, with highlighted in red the efficient links, having as origin the node 3 and as destination the node 10.

3.4.3 Pheromone distribution and evaporation

Once an ant reaches the destination, it adds an amount of pheromone $\Delta\tau_{i,j}^{c}(t)$ to the links that are part of the followed path. The value of the pheromone released is obtained using the previously introduced $R(C)$ function, the more pheromone is released, the better is the path. A measure involved in the evaluation of the path goodness is the total cost of the path used by ant *n*:

$$C_n(t) = \sum c_{i,j}(f_{i,j}(t), t) \;\; \forall i, j : \; l_{i,j} \in F_n^* \tag{35}$$

where F^*_n is the path chosen by an ant *n*.

The basic $R(C)$ function used in ACS-TA is deterministic and it simply release the pheromone proportionally to the cost of the path chosen by the ant:

$$\tau^c_{i,j,n}(t) = \begin{cases} 1/C_n & \forall i,j : l_{i,j} \in F^*_n \\ 0 & \forall i,j : l_{i,j} \notin F^*_n \wedge l_{i,j} \in L \end{cases} \tag{36}$$

An alternative to the basic function is to add a perception error modeled as a Gaussian distribution, leading to a stochastic model for the pheromone distribution and consequently to a stochastic user equilibrium:

$$\tau^c_{i,j,n}(t) = \begin{cases} \dfrac{1}{max(0\ ,\ N(0,\sigma) \cdot C_n(t))} & \forall i,j : l_{i,j} \in F^*_n \\ 0 & \forall i,j : l_{i,j} \notin F^*_n \wedge l_{i,j} \in L \end{cases} \tag{37}$$

Finally, we can simulate the Logit stochastic user equilibrium. As proved by D'Acierno in [40], this is possible by using an exponential utility function:

$$\tau^c_{i,j,n}(t) = \begin{cases} e^{\left(-C_n(t)/\theta\right)} & \forall i,j : l_{i,j} \in F^*_n \\ 0 & \forall i,j : l_{i,j} \notin F^*_n \wedge l_{i,j} \in L \end{cases} \tag{38}$$

where θ is a parameter related to the variance of the random residuals of the perceived utility. Using an exponential utility function can lead to very different pheromone release for each Origin/Destination path. This can be a problem on paths with high cost, because the pheromone value is numerically too low and can end up approximated to 0, making necessary an high θ value. If this approximation is done on the shortest path, no pheromone will be ever released causing the algorithm to not work correctly. To avoid this problem, it is possible to normalize the exponential utility function, by subtracting the path cost with the cost of the fastest path when no flow is assigned:

$$\tau^c_{i,j,n}(t) = \begin{cases} e^{\left(-C_n(t) - C_{min} / \theta\right)} & \forall i,j : l_{i,j} \in F^*_n \\ 0 & \forall i,j : l_{i,j} \notin F^*_n \wedge l_{i,j} \in L \end{cases} \qquad (38b)$$

Once all the ants have reached their destination and they have distributed their pheromone, the pheromone trails evaporate on every link. This is obtained by implementing the following rule:

$$\tau^c_{i,j}(t) = \tau_{i,j}(t-1) \cdot (1-\rho(t)) + (\sum_{n=1}^{N} \tau^c_{i,j,n}(t)) \cdot \rho(t) \quad \forall i,j : l_{i,j} \in L \qquad (39)$$

where $\rho \in (0,1]$ is the pheromone trail decay coefficient. The higher is the value of the decay, the faster the information about previously found solution will be forgotten, leading to a faster convergence time but less precision of the solution. A possible weakness introduced by the evaporation is that after some iterations, links that never have been used and the relative paths that contain them, will continue to decrease in desirability even if ants never had a chance to check the goodness of the path. This can lead to use only a subset of feasible solutions decided by how the ants moved in the first few iterations of the algorithm, making the solution dependent by the seed used for the random link choosing. To prevent this, it is possible to evaporate pheromone only on links that have been used by an ant, without reducing the desirability of unused links:

$$\tau^c_{i,j}(t) = \begin{cases} \tau^c_{i,j}(t-1) \cdot (1-\rho(t)) + (\sum_{n=1}^{N} \tau^c_{i,j,n}(t)) \cdot \rho(t) & \forall i,j,n : l_{i,j} \in F^*_n \\ \tau^c_{i,j}(t-1) & \forall i,j,n : l_{i,j} \notin F^*_n \wedge l_{i,j} \in L \end{cases} \qquad (40)$$

Another update we need to do at this point, is the value of released pheromone on each link and the total released pheromone:

$$\check{\tau}^{c}_{tot}(t) = \tau_{tot}(t-1)\cdot(1-\check{\rho}) + (\sum_{n=1}^{N}\sum_{i,j\in F^{*}_{n}} \tau^{c}_{i,j,n}(t))\cdot\check{\rho} \tag{41}$$

$$\check{\tau}^{c}_{i,j}(t) = \check{\tau}^{c}_{i,j}(t-1)\cdot(1-\check{\rho}) + (\sum_{n=1}^{N} \tau^{c}_{i,j,n}(t))\cdot\check{\rho} \quad \forall i,j : l_{i,j}\in L \tag{42}$$

where $\check{\rho}\in(0,1]$ is the pheromone trail decay coefficient. The higher is value of the decay, the faster the traffic flow will adapt to follow the solution found by the ants, but this can lower the quality of the solution because there will be more oscillations of the flow when the algorithm is near the equilibrium.

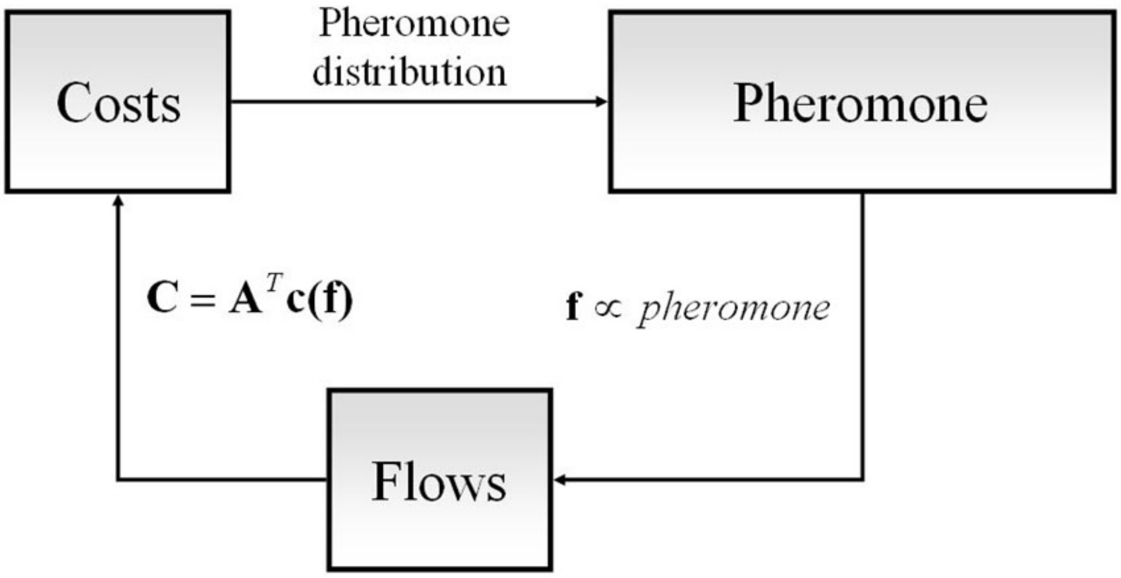

Costs
Pheromone distribution
Pheromone
$\mathbf{C} = \mathbf{A}^T\mathbf{c}(\mathbf{f})$
$\mathbf{f} \propto$ *pheromone*
Flows

Fig. 3.4: Circular relationships between pheromone distribution, flows and costs: the cooperation among ants in the same colony acts as user decision optimization; the competition among colonies can be compared to the congestion effect in road traffic. This circular relationship mimic the circular relationships of equilibrium between traffic demand, flows and costs.

3.4.4 Flow assignment and link cost update

This section of the algorithm completes the circular relationship shown in *Figure 3.4*. In ACS-TA the pheromone trails, besides being a means to guide the ants in the building of their paths, are used to distribute the traffic flow on the network. For each link, a quantity of flow proportional to the pheromone present on it is assigned; this turns into a variation of the costs that the following ants will experience in their paths. This is obtained by having on link (*i, j*) at time t a quantity of flow equal to:

$$f_{i,j}^{c}(t) = \sum_{c=1}^{N_{OD}} d^{c} \frac{\check{\tau}_{i,j}^{c}(t)}{\check{\tau}_{tot}^{c}(t)} \tag{43}$$

where d^c is the flow demand of colony c, N_{OD} is the number of colonies, $\check{\tau}^c_{i,j}(t)$ are the released pheromone of colony c on link *(i, j)* and $\check{\tau}_{tot}$ is the total sum of released pheromone. We use the added pheromone instead of the current pheromone on a link because they are initialized at a value higher than 0 and the evaporation of the current pheromone on links can happen only if they are used by ant ant, leading to an inhomogeneous evaporation.

The cooperation among ants in a same colony acts as user decision optimization, while the competition among colonies can be compared to the congestion effect in road traffic. All this can be viewed in the same manner as the circular relationships of equilibrium between traffic demand, flows and costs explained previously where pheromone substitutes traffic demand (see *Figure 3.4*). This view is commonly proposed for the SUE case but it can also be extended to the DUE case (preferably solved using the Frank-Wolfe algorithm) since the latter is a particular case of SUE.

To avoid using released pheromone instead of the current pheromone on a link, there are two other possible solutions. The first is to enumerate all the possible paths that have pheromone and distribute the flow proportionally on each path:

$$f^c_{i,j}(t) \quad = \quad \sum_{l_{y,i} \in F_k} f_{y,i}(t) \cdot \sum_{l_{i,j} \in F_k} \left(\frac{\tau_{i,j}}{\sum_{\forall l_{i,x} \in L} \tau_{i,x}} \right) \quad \forall F_k \in F, \quad \forall i,j : l_{i,j} \in L \qquad (44)$$

Starting on the origin node, we divide the flow that goes on the connected nodes proportionally to the pheromone on the link that connects the nodes, creating a tree with all the paths having the destination node as the leaf. The problem of this algorithm is that having initialized every link with some pheromone, it ends up enumerating all the paths that connect the origin and destination, which is a computationally expensive operation we usually want to avoid. As an alternative without path enumeration, we can use again the ants to simulate how the flow is distributed. In this case, the probability table used by the ants to choose a link is proportional to the pheromone on it:

$$p_{i,j}^{c}(t) \quad = \quad \frac{(\tau_{i,j}^{c}(t))}{\sum_{i,j \in L_i} (\tau_{i,j}^{c}(t))} \quad \forall i,j : l_{i,j} \in L \tag{45}$$

where L_i are all the outgoing links from the node *i.* The colony release *k* ants which use different paths that reach the destination. To find the flow on a link, we simply compare the total number of ants with the number of ants that used the link:

$$f_{i,j}(t) = d^{c} \cdot \frac{k_{i,j}(t)}{k} \forall i,j : l_{i,j} \in L \tag{46}$$

Using the current pheromone on the links to calculate the flow has the advantage of consuming less memory, but the computation complexity of both solutions is considerably higher than the one using released pheromone.

When all the parallel threads finish to calculate the flow of a colony on each link, we obtain the sum of the flows on every link and update the cost of the link using the cost function:

$$c_{i,j}(f_{i,j}(t),t) \quad \forall i,j : l_{i,j} \in F \tag{47}$$

3.4.5 Rho update

The value ρ , the pheromone trail decay coefficient, can have a huge impact on the precision of the found solution and the time to reach convergence. Ideally, we want a high value of ρ at the beginning, to reach the convergence in less time, and reduce the value as we arrive near the solution, to achieve a better precision.

Having a small ρ means that the variations of pheromone and of the flow after every iteration are small, and there are less oscillations near the solution.

This behavior can be obtained by checking the variations of the solution, the sum of cost integrals on the links, over the last *I* iterations, and if the solution does not have much variation, decrease the value of ρ . This check should be done at the end of each iteration after calculating the solution:

$$\begin{aligned} C^I(t) &= \sum c^I_{i,j}(f_{i,j}(t),t) \ \forall i,j : a_{i,j} \in A \\ C^I_{avg} &= \sum_{i=0}^{I} \frac{C^I(t-i)}{I} \\ \rho(t) &= \begin{cases} \rho(t-1)/q & \text{if } C^I_{avg}\cdot(1-d) \le C^I(t) \le C^I_{avg}\cdot(1+d) \wedge \rho(t-1) > \rho_{min} \\ \rho(t-1) & \text{otherwise} \end{cases} \end{aligned} \tag{48}$$

where $d \in (0,1)$ is the maximum distance variation over the last *I* iterations between the cost integral average and the cost integral last value, q is used to decide how much the ρ will decrease if the conditions are met and ρ_{min} is the minimum value after which the algorithm stop decreasing the ρ value.

3.4.6 Stop condition

After we find the cost on each link, we have to decide if we stop the algorithm or proceed with a new iteration. To measure the quality of a solution we use the sum of the cost integral on each link. On real networks we cannot know what is the value of the solution, which we know to exist and is unique only in the case of deterministic user equilibrium. That is why we use the variance of some variables of interest observed in the previous *K* iterations. If in the last iterations the variance is below a certain threshold, we assume that we are near the solution. The variables of interest that we use in our algorithm are the following:

Cost Variance

1. For each link the average cost in the last *K* iterations is found:

$$M_{i,j}(t) = \sum_{k=0}^{K} \frac{c_{i,j}(f(t-k), t-k)}{K} \quad \forall i,j : l_{i,j} \in L \tag{49}$$

2. Using the average we can easily compute the variance:

$$\sigma_{i,j}^2(t) = \sum_{K=0}^{K} \frac{(c_{i,j}(f(t-k), t-k) - M_{i,j}(t))^2}{K} \quad \forall i,j : l_{i,j} \in L \tag{50}$$

3. To stop the algorithm, the variance should remain below a certain threshold on every link. The condition to be verified is the following:

$$\forall l_{i,j} \in F : \frac{\sigma_{i,j}(t)}{c_{i,j}(f(t), t)} < \varepsilon \tag{51}$$

Flow Variance

1. For each link the average flow in the last *K* iterations is found:

$$M_{i,j}(t) = \sum_{k=0}^{K} \frac{f_{i,j}(t-k)}{K} \quad \forall i,j : l_{i,j} \in L \tag{52}$$

2. Using the average we can easily compute the variance:

$$\sigma_{i,j}^2(t) = \sum_{K=0}^{K} \frac{(f_{i,j}(t-k) - M_{i,j}(t))^2}{K} \; t.c \quad \forall i,j : l_{i,j} \in L \tag{53}$$

3. To stop the algorithm, the variance should remain below a certain threshold on every link. The condition to be verified is the following:

$$\forall l_{i,j} \in F : \frac{\sigma_{i,j}(t)}{f_{i,j}(t)} < \varepsilon \tag{54}$$

To avoid premature stopping in the experiments, we adopt a third criterion to calculate the number of iterations to convergence based on the final value of flow after a maximum number of 1000 iterations. When we present the experimental results we assume the algorithm has reached convergence when 90% of the flows are inside the interval defined by the values below and above 1% of the final value. This criterion can only be applied off-line, but provides a non optimistic estimate of the number of iterations the algorithm needs to reach convergence.

3.4.7 Further optimization

As explained in chapter 3.4.2, a lot of computation time on larger networks is wasted in ants that end up choosing sub-optimal paths and do a reset. To further reduce the probability to select a sub-optimal path, it is possible for every colony, to do an optimization on the network topology that filters links which will surely lead to choosing a sub-optimal path. The best optimization is possible by enumerating all the optimal paths for a colony and, during the link choosing, ignoring the links that are not in any optimal path. The problem is that enumerating all the paths on big networks is a very long operation and should be avoided; a tradeoff is needed to find a good optimization algorithm that is executed in an acceptable amount of time. The algorithm we use analyzes every link and excludes it if one of the following conditions is met:

1 connects the destination node to another node;

2 connects a node to the origin node;

3 enters in a node that does not have out links, with the exception of the destination node;

4 exits from a node that does not have any entering link, with the exception of the origin node.

5 is part of a ring of connected nodes where, with the exception of only one node, all the other nodes have only one outgoing link.

Algorithm 3.3: Network optimization

```
for each colony do
    remove visibility of links coming out from the destination node;
    remove visibility of links entering into the origin node;
    do
        For each node that is not origin or destination do
            if node does not have visible exiting links then
                remove visibility of links entering in the node
            end if
            if node does not have visible entering links then
                remove visibility of links exiting in the node
            end if
            if node has only one visible out link then
                set the current node as the first node;
                while next node has one out link and is not the destination node do
                    if next node is the first node then
                        remove visibility of the link that connect to next node
                        break;
                    end if
                    select next node
                end while
            end if
        end for
    while no visibility is removed from any link;
end for
```

Last three points are repeatedly checked on every link until there are not any more excluded links, because the exclusion of a link can create the conditions to have new links that can be excluded. Using this algorithm on the network of *Figure 3.1* would leave visible only the links used to reach the destination, making impossible to choose a sub-optimal path. In *Figure 3.5* there is an example of real network with highlighted the links excluded after the algorithm application.

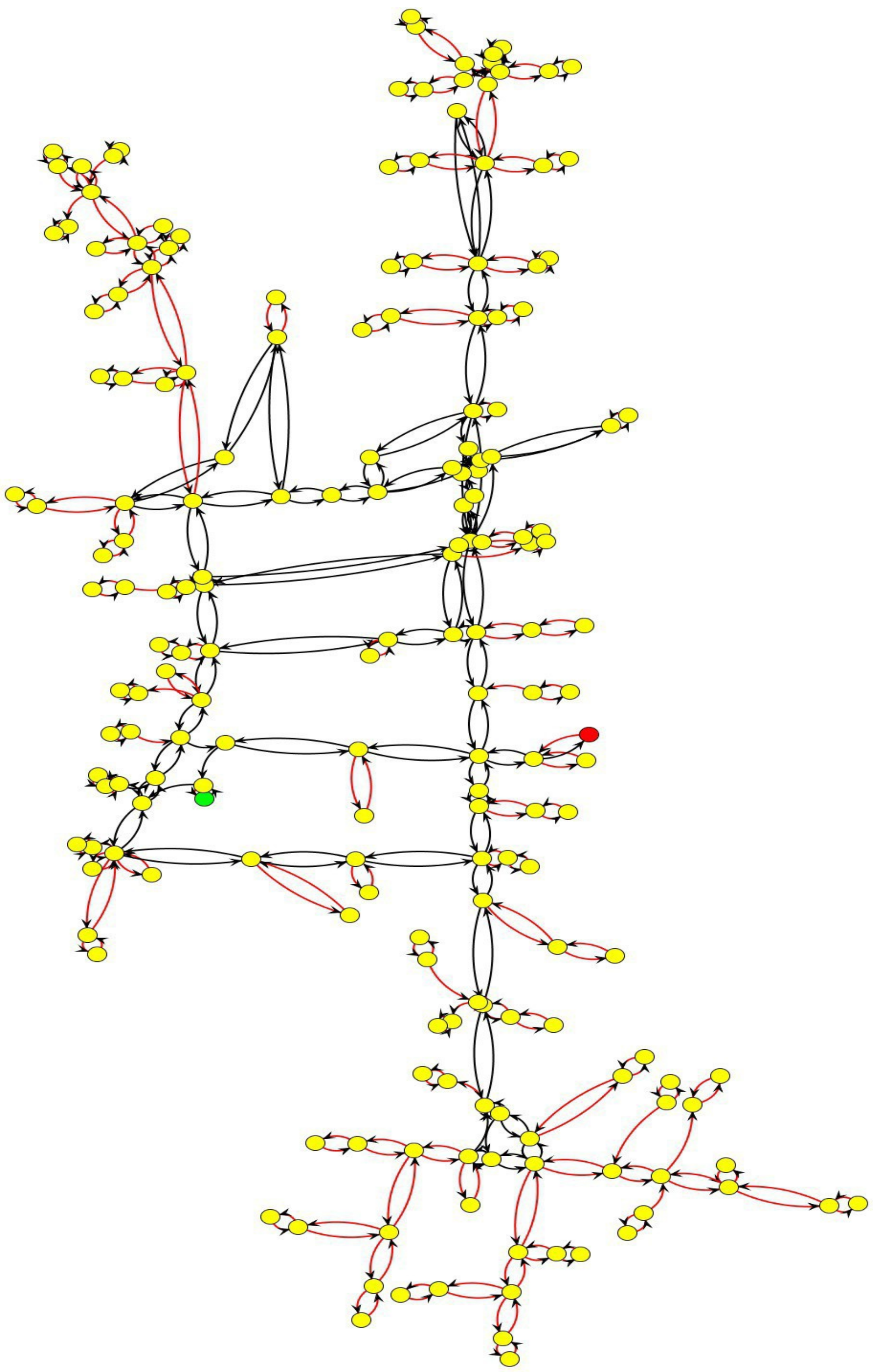


Fig. 3.5: The Area Maggi network with highlighted in red the ignored links, for a particular origin/destination pair, after the network optimization.

Chapter 4

Design and implementation

4.1 Introduction

In this chapter we explain how the software that runs the algorithm is designed and implemented. First we perform an analysis of software requirements, then we describe how the transportation networks are coded in files and how they are parsed to create a model of the network in memory. Next we explain how the ACS-TA algorithm is implemented and how we use the model of the network to search for a solution. Finally we explain how the data obtained during the algorithm execution and the final results is saved and ready to be analyzed.

4.2 Software requirements overview

During the design of the software the following requirements have been discussed and accepted:

- the software runs on the most common operative systems, using single or multi-core processors, with different size of memory available;
- it works on any network given, as long as it is coded in files that use a

specific syntax. The networks need to follow the specifications discussed in paragraph *3.4*;

- it implements the basic ACS-TA algorithm and all the variants discussed in chapter *3*, with the possibility to select a different variant in a configuration file or as an input parameter;
- it gives the possibility to choose which data we are interested and save the data on files using a specific syntax;
- it gives the possibility to graphically visualize the network and show how the traffic is distributed for a specific origin/destination/category combination;
- it makes simulations repeatable by passing a seed as input parameter.

The programming language that we use is Java, which can work on many different operating systems and system specifications, and it provides many functions to easily implement parallel computing on different threads. The library we use to easily implement and visualize the network model is the Java Universal Network/Graph Framework (JUNG)[1]. One weakness of using Java is the low control given on memory management which, although it makes easier the code writing, needs lot of attention to avoid memory leak and optimal usage of memory. Considering the amount of data on larger networks, every little memory leak or inefficiency can cause huge performance loss or lot of avoidable memory occupation.

To make possible the use of the algorithm on large networks we have to make some tradeoff between performance and memory consumption. This is caused by the amount of additional data we need on every link on the network, like

1 See http://jung.sourceforge.net/

pheromone and released pheromone for every origin/destination/category. A good tradeoff is done by avoiding the use of indexes on the links, because they would increase by a significant amount the memory consumption. The search for a specific link and the data regarding it is done by a binary search on an ordered list of link ids, to occupy less memory while still using a fast search algorithm. The Java Runtime Environment (JRE) where the simulations run is the 1.7 version. It is possible to use JRE 1.6, but it turns out to be slower.

4.2.1 Input data

Most of the input data is divided into different files which are parsed by the software to create the network model and configure how we want the algorithm to work. The location of the files is given as an argument when lunching the program; by default the configuration file is expected with the name "*parameters_ant.txt"*. It is possible to specify a different location, to make possible the execution of many simulations using a batch that tries different configurations for the same network. For a detailed description of the arguments that can be used and the parameters in the configuration files see the "Use Manual" (Appendix A).

In the next section is provided the description of the files that contain the data regarding the network. In each file, if not specified otherwise, the first row is the header where the column names are indicated, so that the order of the columns is not important. Each column is separated by a *tab* and each row by a *newline*.

Nodes: the file name is specified in the parameters file, and it needs three columns. The first one is ***ID***, used to give an identification number for each node and it is also shown in the graphic representation; the last two columns are ***X*** and ***Y*** which indicate the position of the node, used in the graphic representation.

Links: the file name is specified in the parameters file, and it can have various formats depending on the cost function $c_{i,j}(f,t)$, used to determine the travel time through a link given the flow. It needs the two columns: ***A***, which for each link indicate the origin node identification number, and ***B***, which indicates the destination node identification number. After this, the remaining columns depend from the cost function;

- *Type 2* (***A; B; L_i; V_0; Cap_u_i; delta; gamma***): described by Cascetta (equation 2.3.3 in [41]) and it writes as:

$$tr_l(f_i, f_{l^*}) = \frac{L_l}{v_{01}} + \delta\left(\frac{L_l}{v_{cl}} - \frac{L_l}{v_{01}}\right)\left(\frac{f_l + f_{l^*}}{Q_{ll^*}}\right)^{\gamma} \tag{55}$$

  where L_l (column ***L_i***) is the length of link l, v_{01} (***V_0***) is the free-flow average speed, v_{cl} (***V_c***) is the average speed with flow equal to capacity (***Cap_u_i***), and δ , γ (***delta, gamma***) are two additional parameters for cost function calibration; the parameter referring to the link in the opposite direction is denoted by l^* and the overall capacity in both directions by Q_{ll^*} (obtained multiplying the column ***Cap_u_i*** with ***n_lanes_i***).

- *Type 3 separable cost* (***A; B; L_i; alpha; beta; C; green, f***): the cost function of type 3 is the BPR (Bureau of Public Roads) and it writes as:

$$tr_l(f_l) = tr_{01}\left(1 + \alpha\left(\frac{f_l}{Q_l}\right)^{\beta}\right) \tag{56}$$

  where tr_{01} (obtained as 3.6 * ***L_i*** / ***V_0***) is the free-flow average travel time for the link *l*, and parameters α , and β (***alpha, beta***) are calibration parameters of the cost function; Q_l (***C***) is the capacity of link *l*. The value of **green** is always 1, indicating that it has separable cost. It is possible to use symbolic links that have no cost, setting to 1 the value of column ***f***.

- *Type 3 non separable cost* (***A; B; L_i; alpha; beta; C; green; f; links_number; link_{n}_in and link_{n}_out***): in this case, the value of green is 0 indicating that it is a non-separable cost link. For this kind of links the TRB cost function is used, introducing a delay due to all conflicting links:

$$tr_{n-sep}(f_{conf})=\exp(-0.2661+0.3967\cdot\ln(f_{conf})) \qquad (57)$$

  where f_{conf} is the sum of flows on conflicting links. To specify what are the links that are used to calculate f_{conf} , we use the column ***links_number*** that indicate the number *n* of links used in the sum and then for each link we use the columns ***link_{n}_in*** and ***link_{n}_out*** that indicate the origin node and destination node that identify the links.

- *Type 3 with limited access* (***A; B; alpha; beta; TIMEBASE; CAPACITY; CURVA_DEFL; JURISDICTIO; VL15***): it is possible to use the type 3 cost function in networks with limited access links and reserved links. In this case the value of tr_{01} is obtained from the column ***TIMEBASE*** and Q_l from the column ***CAPACITY***. To indicate a link with no cost, the value of column ***CURVA_DEFL*** need to be set to 0. The reserved links are identified using the column ***JURISDICTIO***, that contains an identification number used to decode the cost to travel through the link for each category. The limited access links are identified using the column ***VL15***, that contains an identification number used to decode the accessibility of a link for each category

Origin/Destination/Category: the file name is specified in the parameters file, it needs the two columns ***O*** and ***D.*** The former indicates the origin node identification number, the latter indicates the destination node identification number. If only one category is used, the column ***F*** is used to indicate the value of the flow generated for each O/D couple. If more than one vehicle category is present, we use a column named with the category identification number (starting from 0 and incrementing by 1 for each successive category), and each column contains the flow for the Origin / Destination / Category combination.

Accessibility: the file is named "*accessibility.txt*", it contains a matrix where the first line is the column ***CategoryId***, followed by the columns named with the identification numbers used in the column ***VL15*** of the file containing the links. The following rows contain, under the column ***CategoryId***, the categories used in the file that contains the Origin / Destination /Categories combinations. Under the other columns, there is the value "*t*" if the category have access to the links that have the column name in the value of column ***VL15***, "*f*" otherwise.

Reserved links: the file is named "*reserved_link_cost.txt*", it contains a matrix where the first line is the column ***reservedId***, followed by the column names of the categories used in the file that contains the Origin / Destination /Categories combinations. The next rows contain, under the column ***reservedId***, the identification numbers used in the column ***JURISDICTIO*** of the file containing the links. Under the other columns, indicating the vehicle category identification number, there is the value of the travel time cost penalty that need to be added if a path uses a link that has this identification number in the value of column ***JURISDICTIO***.

Seeds: the file name is specified in the parameters file, it does not have a header in the first row, and each row contains a different seed used by the Origin / Destination / Category combination ant colony. If there are not enough rows for each ant colony, new values are generated using the current time and saved in the file.

4.2.2 Output data

The output data is saved in a sub directory called logs. In the same directory there are two files, the first one called "*logEvents.txt*" that contains the events we want to log (only those which do not start with # are considered). The possible events are:

- **executionTime:** the software writes in a file named "*log.log*" the execution time of the pheromone distribution, flow distribution and total execution time for every ant colony;
- **flowUpdate:** it writes in a file named "*flows.csv*", a matrix that contains the flow value on each link, at each iteration;
- **costUpdate:** it writes in a file named "*costs.csv*", a matrix that contains the cost of each link, at each iteration;
- **flowEnd:** it writes in a file named "*flows_end.csv*", the flow value on each link at the last iteration;
- **costEnd:** it writes in a file named "*costs_end.csv*", the cost on each link at the last iteration;
- **checkEndCondition:** it writes in a file named "*variance.csv*", the maximum cost (or flow) variance calculated at each iteration;
- **costIntegralUpdate:** it writes in a file named "*cost_integral.csv*", the sum of the cost integral of each link, for every iteration;
- **costIntegralEnd:** it writes in a file named "*cost_integral_end.csv*", the sum of the cost integral of each link at the last iteration;
- **antBlockUpdate:** it writes in a file named "*ant_block_update_<colony>.csv*", how many ants have been reset

because they ended up blocked at each iteration for a specific ant colony;

- **chooseUpdate:** it writes in a file named "*choose_table_<colony>.csv*", the probability to choose a link when it changes, for a specific ant colony. Only used when a probability table is calculated at the beginning of the iteration;
- **pheromoneUpdate:** it writes in a file named "*pheromone_<colony>.csv*", a matrix that, for each link and for each iteration, contains the pheromone for the colony indicated in the file name;
- **newPheromoneUpdate:** it writes in a file named "new*_pheromone_<colony>.csv*", a matrix that for each link and for each iteration, contains the pheromone released by the ants of the colony indicated in the file name. A special value "*-1*" is used for links that have been used by one or more ants, but the ants always ended up blocked and had to be reset;
- **rhoChange:** it writes in a file named "*rho.csv*", a matrix that contains the value of the pheromone trail decay coefficient ρ , at each iteration;
- **rhoUpdateRequest:** it writes in a file named "*log.log*" the new value of ρ when it changes, and the iteration number where the change happened;
- **blackListUpdate:** it writes in a file named "*black_list_update_<colony>.csv*", all the paths that are added to the blacklist of the colony in the file name. The paths are represented as a set of links;
- **flowInLinksEnd:** it writes in a file named "*flow_links_end_<colony>.csv*", the total flow that is present on each link, and the contribution given by each colony to the total flow, only for

the last iteration.

A second file called “authorizedFlows.*txt*” is needed here. The first row has the header with the column names **O**, **D** and **C**; starting from the second row it contains the Origin / Destination / Category combination that we want to be logged for the events that happen inside a colony (*antBlockUpdate, chooseUpdate, pheromoneUpdate, newPheromoneUpdate, blackListUpdate*). If there is only one category, the value 0 is used under **C**. Saving too much data on files can slow down the algorithm execution time, so it is better to carefully choose what data we are interested in instead of saving everything.

4.3 The Network Model design and implementation

As explained in the software requirements, all the data needed to create the model is contained in different files. The parameters contained in the configuration file are parsed and saved in a utility class called *ParameterParser,* and passed as an argument for the network model initialization. As an alternative it is possible to specify the path where the configuration data is located and let the model build the parser. The class that models the transportation network is called *GraphTrafficModel.* The initialization is composed of various steps:

1. Creates a *RowParser* object for every file used by the network model (Nodes, Links, Origin/Destination/Category combinations, Accessibility and Reserved links). This object parses the assigned file and creates a map between the header and the value in each row.
2. Creates the nodes of the network using the class called *TrafficNodeFactory.* This class initialization takes as argument a row read

by the node file parser and creates a *TrafficNode* object using the data in each row. Every node uses a numeric identification number which is long 10 decimals.

3. Creates the links of the network using the class called *TrafficLinkFactory.* This class initialization takes as argument a row read by the links file parser and creates a *TrafficLink* object using the data in the row and in the configurations file. Every link uses an identification number. To use less possible memory, the identification number is saved as a *long* primitive (64 bit), where the first 10 decimals are the destination node id, the last 10 decimals are the origin node id.
4. Creates the links of the network using the parser of the Origin / Destination / Category file. For every combination, a *FlowGeneratorNode* object is created, which is a special node that contains how much flow is generated. The node is connected with his origin node through a link that does have any cost (*FlowGeneratorLink*). Every *FlowGeneratorNode* uses a numeric identification number to differentiate each other, where the first 2 decimals are the category id, the next 10 decimals are the destination node id and the last 10 decimals are the origin node id.
5. Each link needs to know which Origin / Destination / Category combination can go through it. To obtain this information two operations are attempted: first is an attempt to parse a file, which name is obtained from the configuration file, that contains which flow can go through each link. If the file exist, each row is parsed and used to save the O/D/C combinations that are authorized to use the link related to the row. If the file does not exists, the accessibility file is parsed and for each link is determined if a particular category can go through it, filtering every *FlowGeneratorNode* with a category that cannot access the link. After

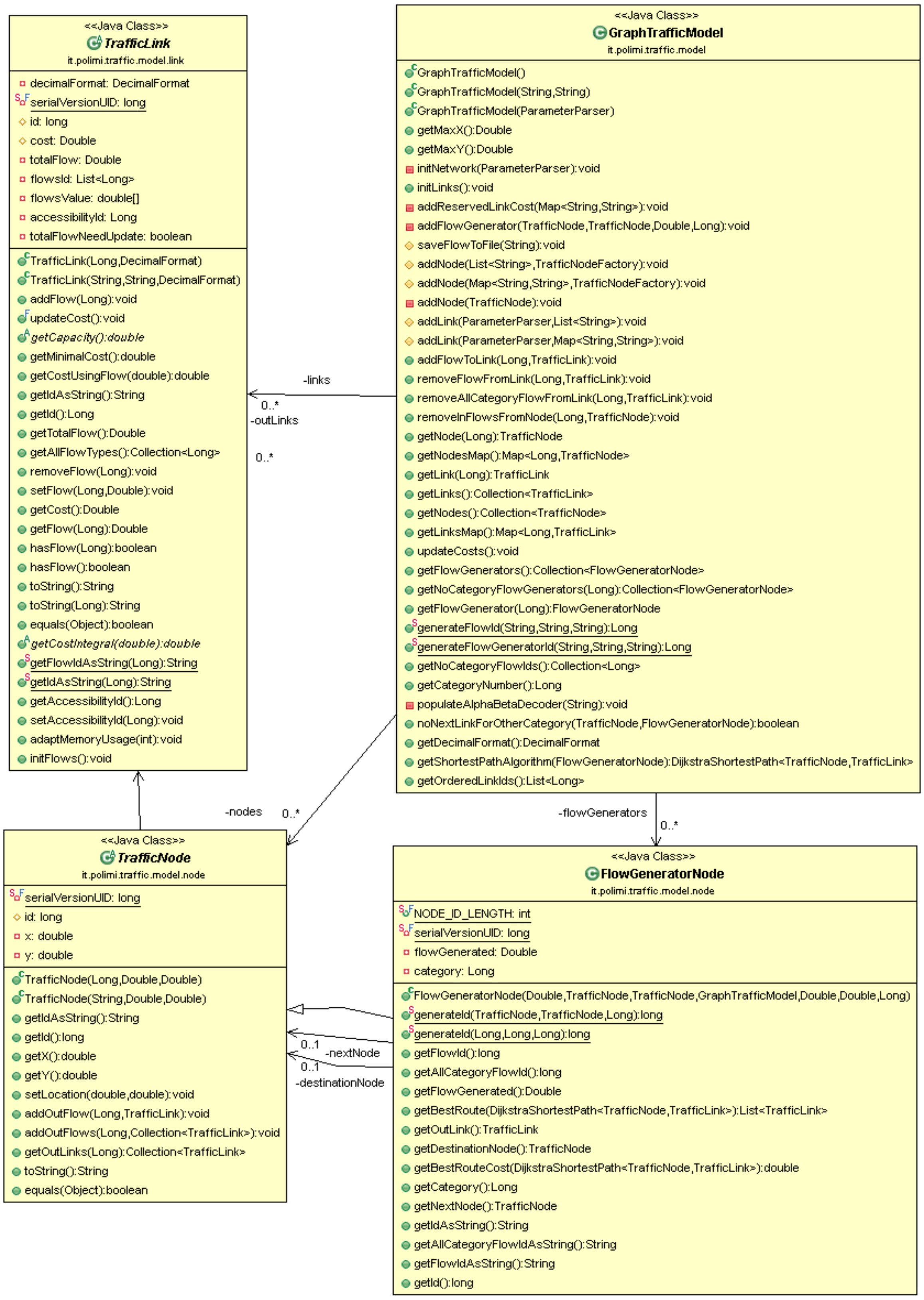

<<Java Class>>
TrafficLink
it.polimi.traffic.model.link
decimalFormat: DecimalFormat
serialVersionUID: long
id: long
cost: Double
totalFlow: Double
flowsId: List<Long>
flowsValue: double[]
accessibilityId: Long
totalFlowNeedUpdate: boolean
TrafficLink(Long,DecimalFormat)
TrafficLink(String,String,DecimalFormat)
addFlow(Long):void
updateCost():void
getCapacity():double
getMinimalCost():double
getCostUsingFlow(double):double
getIdAsString():String
getId():Long
getTotalFlow():Double
getAllFlowTypes():Collection<Long>
removeFlow(Long):void
setFlow(Long,Double):void
getCost():Double
getFlow(Long):Double
hasFlow(Long):boolean
hasFlow():boolean
toString():String
toString(Long):String
equals(Object):boolean
getCostIntegral(double):double
getFlowIdAsString(Long):String
getIdAsString(Long):String
getAccessibilityId():Long
setAccessibilityId(Long):void
adaptMemoryUsage(int):void
initFlows():void
<<Java Class>>
GraphTrafficModel
it.polimi.traffic.model
GraphTrafficModel()
GraphTrafficModel(String,String)
GraphTrafficModel(ParameterParser)
getMaxX():Double
getMaxY():Double
initNetwork(ParameterParser):void
initLinks():void
addReservedLinkCost(Map<String,String>):void
addFlowGenerator(TrafficNode,TrafficNode,Double,Long):void
saveFlowToFile(String):void
addNode(List<String>,TrafficNodeFactory):void
addNode(Map<String,String>,TrafficNodeFactory):void
addNode(TrafficNode):void
addLink(ParameterParser,List<String>):void
addLink(ParameterParser,Map<String,String>):void
addFlowToLink(Long,TrafficLink):void
removeFlowFromLink(Long,TrafficLink):void
removeAllCategoryFlowFromLink(Long,TrafficLink):void
removeInFlowsFromNode(Long,TrafficNode):void
getNode(Long):TrafficNode
getNodesMap():Map<Long,TrafficNode>
getLink(Long):TrafficLink
getLinks():Collection<TrafficLink>
getNodes():Collection<TrafficNode>
getLinksMap():Map<Long,TrafficLink>
updateCosts():void
getFlowGenerators():Collection<FlowGeneratorNode>
getNoCategoryFlowGenerators(Long):Collection<FlowGeneratorNode>
getFlowGenerator(Long):FlowGeneratorNode
generateFlowId(String,String,String):Long
generateFlowGeneratorId(String,String,String):Long
getNoCategoryFlowIds():Collection<Long>
getCategoryNumber():Long
populateAlphaBetaDecoder(String):void
noNextLinkForOtherCategory(TrafficNode,FlowGeneratorNode):boolean
getDecimalFormat():DecimalFormat
getShortestPathAlgorithm(FlowGeneratorNode):DijkstraShortestPath<TrafficNode,TrafficLink>
getOrderedLinkIds():List<Long>
-links
0..*
-outLinks
0..*
-nodes
0..*
-flowGenerators
0..*
<<Java Class>>
TrafficNode
it.polimi.traffic.model.node
serialVersionUID: long
id: long
x: double
y: double
TrafficNode(Long,Double,Double)
TrafficNode(String,Double,Double)
getIdAsString():String
getId():long
getX():double
getY():double
setLocation(double,double):void
addOutFlow(Long,TrafficLink):void
addOutFlows(Long,Collection<TrafficLink>):void
getOutLinks(Long):Collection<TrafficLink>
toString():String
equals(Object):boolean
0..1
-nextNode
0..1
-destinationNode
<<Java Class>>
FlowGeneratorNode
it.polimi.traffic.model.node
NODE_ID_LENGTH: int
serialVersionUID: long
flowGenerated: Double
category: Long
FlowGeneratorNode(Double,TrafficNode,TrafficNode,GraphTrafficModel,Double,Double,Long)
generateId(TrafficNode,TrafficNode,Long):long
generateId(Long,Long,Long):long
getFlowId():long
getAllCategoryFlowId():long
getFlowGenerated():Double
getBestRoute(DijkstraShortestPath<TrafficNode,TrafficLink>):List<TrafficLink>
getOutLink():TrafficLink
getDestinationNode():TrafficNode
getBestRouteCost(DijkstraShortestPath<TrafficNode,TrafficLink>):double
getCategory():Long
getNextNode():TrafficNode
getIdAsString():String
getAllCategoryFlowIdAsString():String
getFlowIdAsString():String
getId():long

Fig. 4.1: Class diagram for the network model

this, the network model is used to initialize a *TrafficOptimizer,* which creates a *TrafficDeadEndsRemover* object for each *FlowGenerator* and then execute them in different threads. Each *TrafficDeadEndsRemover* does the operations listed in Chapter 3.4.7 to optimize the network. Finally the file containing the permitted flows on each link is saved, to avoid the filter and optimization operation next time the software is run for this network.

6. Creates an ordered list of the *TrafficLink*, using the id as key, excluding the links used to connect the *FlowGeneratorNode* to its origin node. The purpose of the list is to be a main index which easily retrieves the position of a *TrafficLink* with a binary search. Every data that is associated to a link but is not part of the model, i.e. the pheromone, can be saved using only the position in this list, without the use of a Map between the link id and the value.
7. Having the network model initialized, it is now possible to use it for the traffic assignment problem. The public methods of *TrafficModel,* which is an interface implemented by *GraphTrafficModel,* gives a complete view of the network, with his nodes, links, and O/D/C combinations. Any algorithm can use this interface to distribute the flow and calculate how the link cost changes after the flow assignment using the method *updateCosts(). A* utility method *getShortestPathAlgorithm()* gives the instruments to easily find the links part of the shortest path for an O/D/C combination.

Routine 4.1: Network model initialization

```
Parse the files containing the network data using different *RowParser* objects
**for each** row in nodes *RowParser* **do**
   create a *TrafficNode* object using the data in the row and adds it to an array
**end for**
**for each** row in links *RowParser* **do**
   create a *TrafficLink* object using the data in the row and adds it to an array
**end for**
**for each** row in origin/destination/category *RowParser* **do**
   create a *FlowGeneratorNode* object using the data in the row and add it to an
   array
   connect the *FlowGeneratorNode* with his origin node using a *NoCostLink*
**end for**
**if** authorized flows file exists **then**
  **for each** row in authorized flow *RowParser* **do**
     retrieve the *TrafficLink* associated to the row
     **for each** *FlowGeneratorNode* **do**
        **if** row contains *FlowGeneratorNode* identification number **then**
           add the *FlowGeneratorNode* to the *TrafficLink* authorized flows
        **end if**
     **end for**
  **end for**
**else**
  **if** exists accessibility file **then**
      **for each** row in the accessibility *RowParser* **do**
        retrieve the link associated to the row
        **for each** *FlowGeneratorNode* **do**
           **if** *FlowGeneratorNode* has the accessibility id of the link **then**
              authorize the *FlowGeneratorNode* to use the link
            **end if**
        **end for**
      **end for**
  **end if**
  **for each** *FlowGeneratorNode* **do**
      create a *TrafficRoutesOptimizer* assigning the *FlowGeneratorNode*
  **end for**
  run all the *TrafficRoutesOptimizer* in different threads
  create a new authorized flows file
  **for each** *TrafficLink* **do**
     save the authorized flows in a new row of the file
  **end for**
**end if**
order the *TrafficLink* array using the link identification number
```

4.3.1 Network nodes

The network nodes are generated through the class *TrafficNodeFactory* and extend the abstract class *TrafficNode.* This abstract class contains the basic information, for a node: an identification number, the coordinates where it is located, and a collection of the links going out from it. Only two implementations of the node exist:

- *FlowGeneratorNode*: contains the origin, destination, category and the flow generated. There is also a method for formatting the identification number in a human readable format for the data logging;
- *NoCostNode:* the basic nodes of the network, with no additional data.

For the complete class diagrams see Appendix B.

4.3.2 Network links

All the links are generated through the class *TrafficLinkFactory* and extend the abstract class *TrafficLink.* This abstract class contains the basic information, for a link: an identification number, the cost, the total flow, which O/D/C combination can use it, the O/D/C combination flow and the accessibility id used to determine which categories can go through it during the network initialization. The implementation of this class provide functions to obtain the cost integral, and the link capacity. There are various possible implementations returned by the factory, depending the data read in the row and the link type in the configuration file (see Chapter 4.2.1):

- *Type2Link:* implements the cost as the type 2 link (see Chapter 4.2.1) using the parameters passed.

- *Type2LimitedAccessLink:* same as *Type2Link*, with added a limited access for certain categories that need to apply an additional path cost if they pass through it.
- *Type3Link:* implements the cost as the type 3 link (see Chapter 4.2.1) using the parameters passed.
- *Type3LimitedAccessLink:* same as *Type3Link*, with added a limited access for certain categories that need to apply an additional path cost if go through it.
- *Type3NotSeparableLink: it is* a non separable cost link type (see Chapter 4.2.1) that needs, during the creation, the set of links which flows have an influence on the link cost.
- *NoCostLink:* a link with no cost, used in some particular cases as explained in Chapter 4.2.1.

Another type of link, not created through the factory, is the *FlowGeneratorLink,* which connect the *FlowGeneratorNode* with his origin node. For the complete class diagrams see Appendix B.

4.3.3 Network visualization

To visualize the network, the *GraphTrafficModel* extends a class of the Jung library called *DirectedSparseMultigraph*, while the nodes extend *TrafficNode* which in turn extends *Point2D.* The model, with a selected *FlowGeneratorNode,* is used by a class named *Visualizer*, which creates another Jung class called *VisualizationViewer* that can be used by a frame (J*Frame*) to visualize the network. In the *Visualizer*, before showing the network, various transformations are applied on the network to correctly visualize all the information needed, using utility classes of the Jung library:

- *TrafficLinkPredicate:* filters all the links that do not have any possible flow passing through or are a *FlowGeneratorLink;*
- *TrafficNodePredicate:* filters all the *FlowGeneratorNode* type of nodes;
- *TrafficLinkLabelTransformer:* does not show any label on the links;
- *TrafficNodeLabelTransformer:* shows the node identification number;
- *TrafficLinkToolTipTransformer:* shows a tooltip on the links that contains the identification number, flow passing through it and cost;
- *TrafficNodeToolTipTransformer:* shows a tooltip on the node that contains the identification number;
- *TrafficLinkPaintTransformer:* changes the color of the link, that will range from blue where the passing flow is negligible, to red where the passing flow is near the maximum generated by the selected *FlowGeneratorNode*. The links that cannot be used by the current *FlowGeneratorNode* are black;
- *TrafficNodePaintTransformer:* changes the color of the node. The origin node is green, the destination node is red and all the nodes where flow is passing through are yellow;
- *PositionTransformer:* returns the *TrafficNode,* which contains the node coordinates;
- *TrafficLinkStrokeTransformer:* shows a dash line for links that do not have any flow passing through for the selected *FlowGeneratorNode,* a dotted line otherwise.

Various examples of visualized networks are shown in Appendix C.

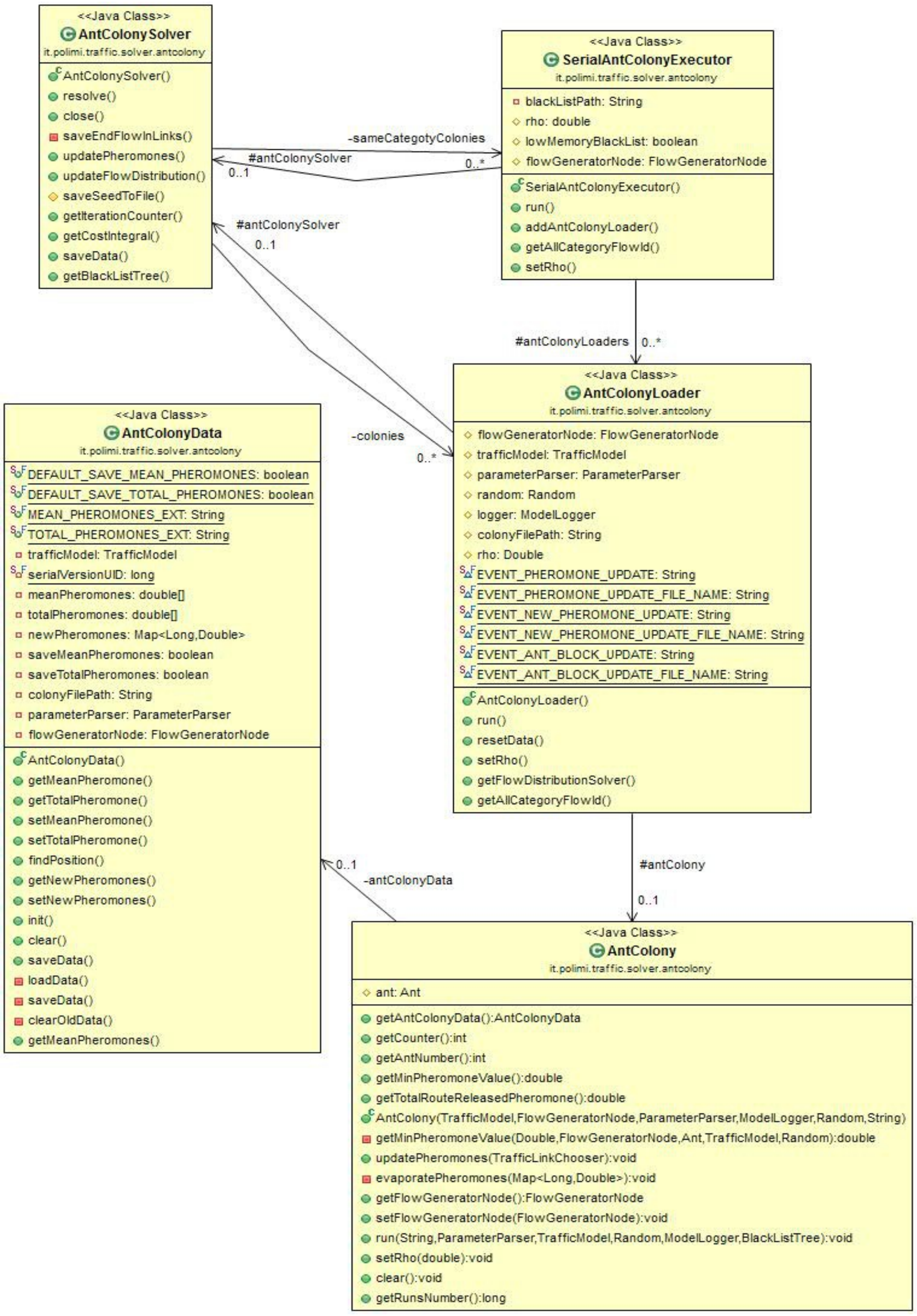
4.4 The ACS-TA algorithm design and implementation

The algorithm implementation is done by the class *AntColonySolver.* For the initialization, it needs a *TrafficModel* and a *ParameterParser,* that contains all the settings, sub-algorithms to be used and algorithm parameters. Using this data it creates an *AntColonyLoader* for each *FlowGeneratorNode* in the model, assigning a different seed obtained by the seeds file or, if not available, generating a new one and adding it to the file. The *AntColonyLoader* contains an *AntColony,* which is the core of the algorithm; it contains all the data necessary for the pheromone release, the flow distribution, and all the selected sub-algorithms used by ACS-TA (a more detailed description is given in Chapter 4.4.1).

To avoid too much overhead during the algorithm parallel execution, with the creation of too many threads, and to have the possibility to share information among flows that have the same Origin / Destination, all the colonies with the same O/D are grouped in containers called *SerialAntColonyExecutor.* The *AntColonySolver* initialization also selects a stop condition and a ρ update (see Chapter 3.4.5 and 3.4.6), using two classes called *StopConditionFactory and RhoUpdaterFactory*. They read the information contained in the *ParameterParser* and respectively generate a *StopCondition* and a *RhoUpdater.*

After the initialization, the *AntColonySolver* runs the algorithm execution procedure, which does the following operations:

- runs all the *SerialAntColonyExecutor* on different threads and wait that all



<<Java Class>>
AntColonySolver
it.polimi.traffic.solver.antcolony
AntColonySolver()
resolve()
close()
saveEndFlowInLinks()
updatePheromones()
updateFlowDistribution()
saveSeedToFile()
getIterationCounter()
getCostIntegral()
saveData()
getBlackListTree()
<<Java Class>>
SerialAntColonyExecutor
it.polimi.traffic.solver.antcolony
blackListPath: String
rho: double
lowMemoryBlackList: boolean
flowGeneratorNode: FlowGeneratorNode
SerialAntColonyExecutor()
run()
addAntColonyLoader()
getAllCategoryFlowId()
setRho()
-sameCategotyColonies
0..*
#antColonySolver
0..1
#antColonySolver
0..1
#antColonyLoaders
0..*
-colonies
0..*
<<Java Class>>
AntColonyData
it.polimi.traffic.solver.antcolony
DEFAULT_SAVE_MEAN_PHEROMONES: boolean
DEFAULT_SAVE_TOTAL_PHEROMONES: boolean
MEAN_PHEROMONES_EXT: String
TOTAL_PHEROMONES_EXT: String
trafficModel: TrafficModel
serialVersionUID: long
meanPheromones: double[]
totalPheromones: double[]
newPheromones: Map<Long,Double>
saveMeanPheromones: boolean
saveTotalPheromones: boolean
colonyFilePath: String
parameterParser: ParameterParser
flowGeneratorNode: FlowGeneratorNode
AntColonyData()
getMeanPheromone()
getTotalPheromone()
setMeanPheromone()
setTotalPheromone()
findPosition()
getNewPheromones()
setNewPheromones()
init()
clear()
saveData()
loadData()
saveData()
clearOldData()
getMeanPheromones()
<<Java Class>>
AntColonyLoader
it.polimi.traffic.solver.antcolony
flowGeneratorNode: FlowGeneratorNode
trafficModel: TrafficModel
parameterParser: ParameterParser
random: Random
logger: ModelLogger
colonyFilePath: String
rho: Double
EVENT_PHEROMONE_UPDATE: String
EVENT_PHEROMONE_UPDATE_FILE_NAME: String
EVENT_NEW_PHEROMONE_UPDATE: String
EVENT_NEW_PHEROMONE_UPDATE_FILE_NAME: String
EVENT_ANT_BLOCK_UPDATE: String
EVENT_ANT_BLOCK_UPDATE_FILE_NAME: String
AntColonyLoader()
run()
resetData()
setRho()
getFlowDistributionSolver()
getAllCategoryFlowId()
#antColony
0..1
0..1
-antColonyData
<<Java Class>>
AntColony
it.polimi.traffic.solver.antcolony
ant: Ant
getAntColonyData():AntColonyData
getCounter():int
getAntNumber():int
getMinPheromoneValue():double
getTotalRouteReleasedPheromone():double
AntColony(TrafficModel,FlowGeneratorNode,ParameterParser,ModelLogger,Random,String)
getMinPheromoneValue(Double,FlowGeneratorNode,Ant,TrafficModel,Random):double
updatePheromones(TrafficLinkChooser):void
evaporatePheromones(Map<Long,Double>):void
getFlowGeneratorNode():FlowGeneratorNode
setFlowGeneratorNode(FlowGeneratorNode):void
run(String,ParameterParser,TrafficModel,Random,ModelLogger,BlackListTree):void
setRho(double):void
clear():void
getRunsNumber():long

Fig. 4.2: Class diagram for ACS-TA algorithm

the ants on every colony release the pheromone;

- for each *AntColony* assigns the flow by running on different threads a *FlowDistributionSolver* returned by each *AntColony* (see Chapter 4.4.4);
- updates the value of ρ , which will be eventually used for the next flow assignment (see Chapter 4.4.5);
- checks if the stop conditions are met. If not, it starts again with the pheromone distribution (see Chapter 4.4.6).

4.4.1 The ant colony

The flow and pheromone distribution for a single Origin / Destination / Category combination is implemented by the *AntColony* class. The data used by the algorithm, such as the pheromone on links and the released pheromone, are all contained in a class called *AntColonyData*, which uses the network model sorted list as index to retrieve the correct value for each link. There is also a mechanism that tries to retrieve the data on files if not already in memory. Using this method, every *AntColony* can read the data at the beginning of the pheromone distribution and releases the occupied memory after the completion, saving everything in a file, at the cost of a loss of performance for the read/write overhead time. Instead of keeping the data for every *AntColony* in memory, only the instance under execution will occupy memory, using a very low amount of it. During the *AntColony* initialization, there is also the selection of the sub-algorithms, based on the passed parameters, that will be used during the pheromone distribution and the flow assignment:

- *Ant*: it is the interface used to determine the amount of released pheromone (chapter 3.4.3)

Routine 4.2: Pheromone distribution

```
for each ant do
   reset ant to origin node
   while ant not on destination node do
      try to move the ant to the next node using a link chooser
      if ant is blocked then
         increment block counter
         if total block counter reach max value then
            exit the program with an error
            break
         end if
         if block counter reach max value then
            apply bias to link chooser
            reset block counter
            reset ant to origin node
         end if
      end if
   end while
   ant release the pheromone on links used to reach the destination
end for
evaporate pheromone on links
```

- *TrafficLinkChooser: it is* the interface used to determine what link chooses an ant from the node he is staying (Chapter 3.4.2)
- *FlowDistributionSolver*: it is the interface used to determine how the flow distributes using the pheromone on the links (Chapter 3.4.3)

To avoid unnecessary memory consumption, only one *Ant* is kept in memory for each *AntColony*, since no parallel computing is done on the ants, and it is always reset to beginning when it reaches the destination. Finally, all the initialization operations needed for the ACS-TA algorithm explained in Chapter 3.4.1, are completed in the *AntColony* initialization.

After the initialization, the *AntColony* provides the methods to run a pheromone distribution (see “Routine 5.2”) and the flow assignment (see Chapter 4.4.4).

4.4.2 Pheromone release

The pheromone release sub-algorithm is done through the implementation of the *Ant* interface, returned by an *AntFactory,* which uses the settings in *ParameterParser* to determine which implementation has to be returned. The category is also needed to apply the correct added cost to the path if the ant goes through a limited access link. The possible *Ant* implementations are the following:

- *AntDue*: implements the deterministic user equilibrium, as in equation (36);
- *AntProbit*: it adds perception error to the cost in an *AntDue* implementation, as in equation (37);
- *AntLogit*: implements the Logit stochastic user equilibrium, as in equation (38).
- *AntLogitNormalized*: implements the Logit stochastic user equilibrium with normalized cost, as in equation (38b).

4.4.3 Link choosing

The ants link choosing sub-algorithm is done through the implementation of the *TrafficLinkChooser* interface, returned by a *TrafficLinkChooserFactory,* which uses the settings in *ParameterParser,* to determine which implementation has to be returned. The possible *TrafficLinkChooser* implementations are the following:

- *StandardTrafficLinkChooser*: it is the basic implementation, that randomly chooses one of the links going out the node where the ant stays. It uses a probability table, calculated at the beginning of an iteration and used by all the ants in equation (26);

- *PheromoneGaussTrafficLinkChooser:* extends the basic implementation, but instead of using a probability table*,* calculates the probability to choose a link every time an ant moves, adding a perception error to the pheromone (27);
- *BestBiasTrafficLinkChooser*: extends the basic implementation, adding the capability to increase the bias towards the links part of shortest path, when ants get stuck too many times, as in (31) and (32);
- *EfficientBiasTrafficLinkChooser*: extends the previous implementation, but instead of using the shortest path, the links with the applied bias are those part of the efficient set, as in (33) and (34),.
- *BlackListChooser:* extends the basic implementation, but it does not use a probability table and, during the link choosing it, filters all the links that lead to a path in the blacklist or leads to a node already used by the ant (see Chapter 3.4.2);
- *BestBiasBlackListChooser:* extends the best bias algorithm*,* adding the blacklist functionality;
- *EfficientBiasBlackListChooser:* extends the efficient bias algorithm adding the blacklist functionality.

The three blacklist implementations are delegated[1] to a helper class called *BlackListChooserHelper*, since the three classes already extend a flow chooser implementation and cannot have a common parent to implement the blacklist. The utility class contains a tree, in an object of type *BlackListTree,* with all the blocked paths. A path in the *BlackListTree* is composed by a set of

1 Delegation pattern, see http://en.wikipedia.org/wiki/Delegation_pattern for reference

BlackListNode objects, which corresponds to a link of the network. Each origin/destination couple has a common *BlackListTree,* which means that different categories have a common tree. This is done to greatly reduce the memory occupation of the blacklist, assuming that most of the links will be accessible for every category. The blacklist is read and saved to file by the *AntColonySolver* using the utility class *BlackListUtil.* In the configuration file it is possible to select between keeping the tree in memory during all the algorithm execution, or to save to file, when not used, to free as much memory as possible. To avoid using all the memory during the software execution, there is a minimum value of memory that needs to be available when adding a new path. At the end of the algorithm execution, the tree is always saved to a file.

The root node of the blacklist always exists and corresponds to the link connecting the *FlowGeneratorNode* to the origin node. To allow the filtering of links, every time an ant moves on a link of the network, it also moves on the blacklist tree, but only if the next node exists in the tree. When the links going out a node are selected, two conditions need to be met:

- the link does not lead to a node already used by the ant;
- if the next child node (associated to the link) of the blacklist tree exists, it is not a leaf.

When an ant blocks, the path is added to the blacklist tree only if there are not links going out the node that are not accessible by the current *AntColony* category, and are accessible by any other vehicle category. When adding a path to the blacklist tree, it is also checked if the previous node does not have all the possible out links filtered. In that case, all the leafs are deleted and the node become a new leaf. In Figure 5.3 it is possible to see a trial network with the corresponding blacklist tree. To decrease the computation time the blacklist is used only during the first iteration, because most of the ant blocks are

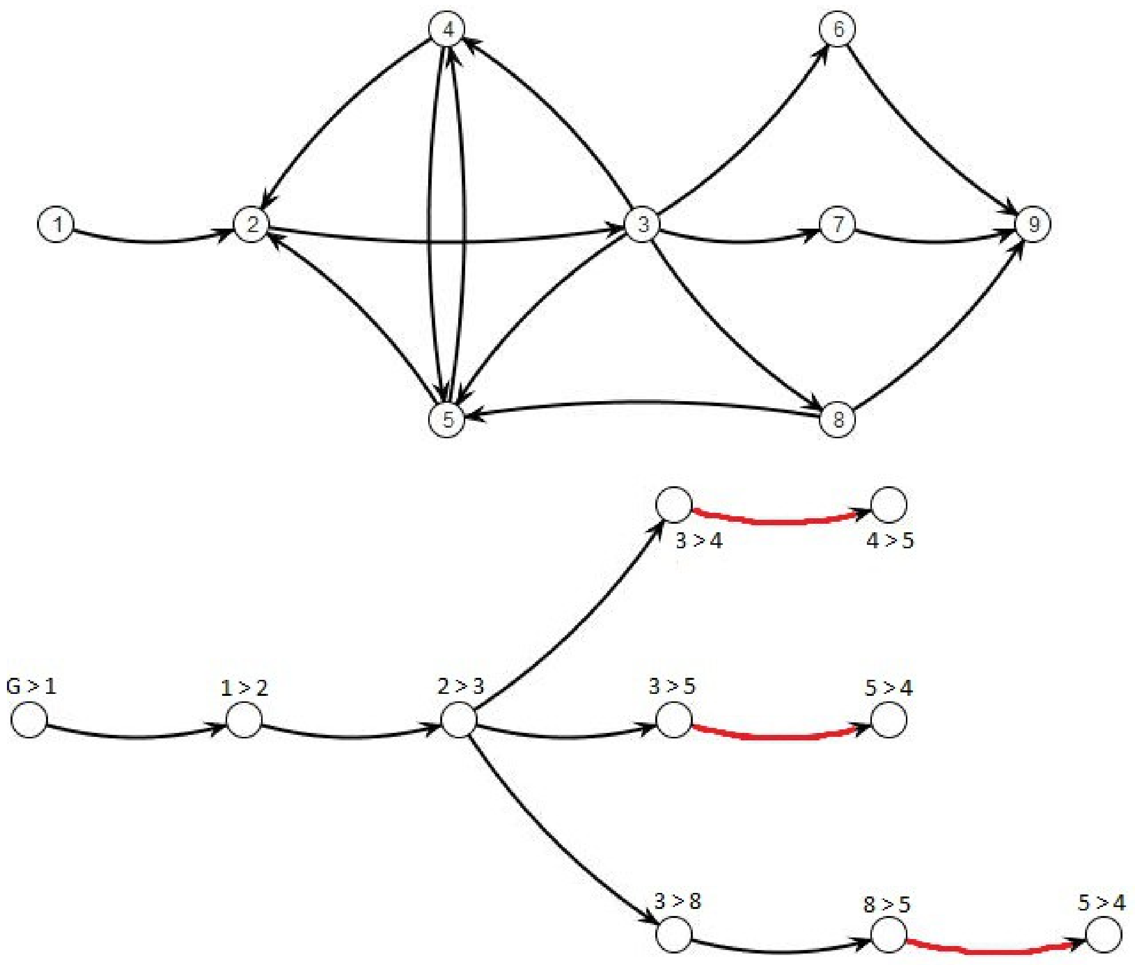


Fig 4.3: A network where origin is in node 1 and destination in node 9. Below there is the blacklist tree obtained from it. In red are the links connecting the removed nodes during the blacklist tree execution.

concentrated here. Starting from the second it is used the parent chooser[1].

1 *EfficientBiasTrafficLinkChooser* instead of *EfficientBiasBlackListChooser, BestBiasTrafficLinkChooser* instead of *BestBiasBlackListChooser* and *StandardTrafficLinkChooser* instead of *BlackListChooser.*

4.4.4 Flow assignment

The flow distribution sub-algorithm is done through the implementation of the *FlowDistributionSolver* interface, returned by an *FlowDistributionFactory,* which uses the settings in *ParameterParser* to determine which implementation has to be returned. The possible *FlowDistributionSolver* implementations (see chapter 3.4.4) are the following:

- *PheromoneDistribution*: assigns the flow using the distributed pheromone on the links, uses the equation (43);
- *RoutesTreeDistribution*: assigns the flow enumerating all the paths, uses the equation (44);
- *Ant distribution*: use ants to distribute the flow using the equations shown in (45) and (46).

4.4.5 Rho value update

The rho update sub-algorithm is done through the implementation of the *RhoUpdater* interface, returned by a *RhoUpdaterFactory,* which uses the settings in *ParameterParser* to determine what implementation return. The possible *RhoUpdater* implementations are the following:

- *CostIntegralAvgRhoUpdater*: implements the rho update if the cost integral average change below a set threshold in the last iterations as in equation (48);
- *NoRhoUpdate*: the value of rho remains always the same.

4.4.6 Stop condition

The stop condition sub-algorithm is done through the implementation of the *StopCondition* interface, returned by a *StopConditionFactory,* which uses the settings in *ParameterParser* to determine which implementation has to be returned. The possible *StopCondition* implementations are the following:

- *CounterCondition*: stops if the iteration number reach a certain value specified in the parameters.
- *CostVarianceCondition*: stops if the flow variance is below a threshold specified in the parameters (51).
- *FlowVarianceCondition*: stops if the flow variance is below a threshold specified in the parameters (54).

During the stop condition, there is always a check that the current iteration number is between a configured minimum and maximum using the *CounterCondition*. After that, another condition can be selected using the parameter file. If the iteration number condition is the only one needed, an implementation called *NoStopCondition* is used for the second check.

4.4.7 Data log and final results save

The log of all the data is done by two classes:

- *Logger*: implements the basic methods to format into strings and save to file;
- *ModelLogger:* logs the data during the ACS-TA algorithm execution.

Both of them use some parameters taken from the *ParameterParser,* such as the label to apply to the file names. During the initialization, *AntColonySolver*, every

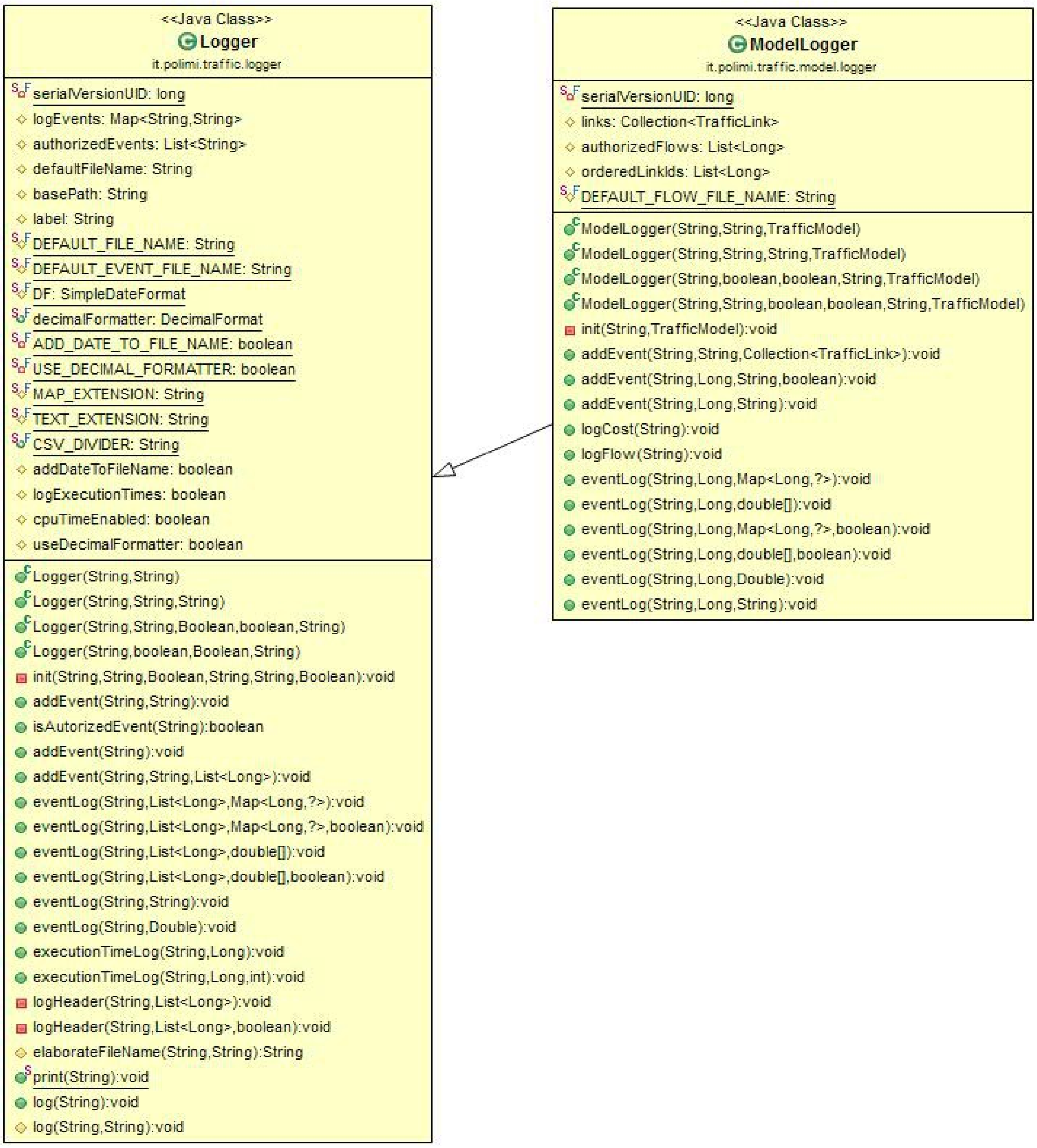

<<Java Class>>
Logger
it.polimi.traffic.logger
serialVersionUID: long
logEvents: Map<String,String>
authorizedEvents: List<String>
defaultFileName: String
basePath: String
label: String
DEFAULT_FILE_NAME: String
DEFAULT_EVENT_FILE_NAME: String
DF: SimpleDateFormat
decimalFormatter: DecimalFormat
ADD_DATE_TO_FILE_NAME: boolean
USE_DECIMAL_FORMATTER: boolean
MAP_EXTENSION: String
TEXT_EXTENSION: String
CSV_DIVIDER: String
addDateToFileName: boolean
logExecutionTimes: boolean
cpuTimeEnabled: boolean
useDecimalFormatter: boolean
Logger(String,String)
Logger(String,String,String)
Logger(String,String,Boolean,boolean,String)
Logger(String,boolean,Boolean,String)
init(String,String,Boolean,String,String,Boolean):void
addEvent(String,String):void
isAutorizedEvent(String):boolean
addEvent(String):void
addEvent(String,String,List<Long>):void
eventLog(String,List<Long>,Map<Long,?>):void
eventLog(String,List<Long>,Map<Long,?>,boolean):void
eventLog(String,List<Long>,double[]):void
eventLog(String,List<Long>,double[],boolean):void
eventLog(String,String):void
eventLog(String,Double):void
executionTimeLog(String,Long):void
executionTimeLog(String,Long,int):void
logHeader(String,List<Long>):void
logHeader(String,List<Long>,boolean):void
elaborateFileName(String,String):String
print(String):void
log(String):void
log(String,String):void
<<Java Class>>
ModelLogger
it.polimi.traffic.model.logger
serialVersionUID: long
links: Collection<TrafficLink>
authorizedFlows: List<Long>
orderedLinkIds: List<Long>
DEFAULT_FLOW_FILE_NAME: String
ModelLogger(String,String,TrafficModel)
ModelLogger(String,String,String,TrafficModel)
ModelLogger(String,boolean,boolean,String,TrafficModel)
ModelLogger(String,String,boolean,boolean,String,TrafficModel)
init(String,TrafficModel):void
addEvent(String,String,Collection<TrafficLink>):void
addEvent(String,Long,String,boolean):void
addEvent(String,Long,String):void
logCost(String):void
logFlow(String):void
eventLog(String,Long,Map<Long,?>):void
eventLog(String,Long,double[]):void
eventLog(String,Long,Map<Long,?>,boolean):void
eventLog(String,Long,double[],boolean):void
eventLog(String,Long,Double):void
eventLog(String,Long,String):void

Fig. 4.4: Class diagram for the data loggers

AntColony and every sub-routine register to the logger, indicating the events that will be launched (i.e. pheromone update, cost update,...), the format to use to save the data regarding the event, the file name and, if present, the *AntColony* lunching the event. During the registration the *Logger* enables only the events selected in the file “*logEvents.txt”* file and check that the requiring colony Origin / Destination / Category combination is in the “*authorizedFlows.txt*” file. Finally, during the execution, events are launched and the data associated to these events is passed to the logger*,* which proceeds to log to file only if the event was enabled during the registration.

Chapter 5

Simulations and results

In this chapter we present the experiments worked out to evaluate the ACS-TA algorithm and the implemented software. The experiments were worked out on a system with 48 AMD cores with 64bit technology and a working frequency of 2300MHz. We used 40 parallel threads during the elaboration, and a maximum of 13GB of memory. The Java Virtual Machine was version 1.7.

5.1 Networks overview

Five different transportation network have been chosen to run simulations. Some of the physical and functional characteristics are shown in table 5.1, and the networks topology is shown in Appendix B. Here is a short description for each of them:

- **Non Separable Costs**: it is a very simple trial network with 28 links, which 7 of them use a type 3 non separable cost functions. There is only one vehicle category and no restricted traffic zones or tolls;
- **Sioux-falls**: a well-known test network first introduced by LeBlanc in his PhD thesis in 1975 [58], it is still used to test traffic assignment

Table 5.1 Physical and functional characteristics of the networks used in the experimental validation.

	Cost Function Type	#Links	#Nodes	# O/D/Cs	Demand [vehicles]
Non Separable Costs	3	28	12	8	23,000
Sioux Falls	3	76	24	24	360,600
Area Maggi (Milan, Italy)	3	273	189	332	~ 40,000
Extra urban Naples (Italy)	2	1363	994	772	~ 45,800
Area Bastioni (Milan, Italy)	3	3919	1779	17448	~ 46,500

algorithms. For this network, the demand is given, the optimal objective function value (that is the average vehicular cost at equilibrium) is known to be 42.31 minutes;

- **Area Maggi**: it is a quite large area in the southern part of Milan. The demand used is that of the morning peak hour (8:00-9:00am); it is characterized by a limited number of paths and by a high number of O/D couples (332), with only one vehicle category and no limited traffic zones.
- **Extra Urban Naples**: represents the very large area of the extra-urban network of Naples (with 1363 links); this network was the subject of an Italian national PRIN grant project named "Road transport systems in the information society: monitoring, simulation and preparation of dynamic information databases" [59] which produced and studied the demand for each day in the course of a year; also in these experiments we used the demand of a typical weekday at 8 a.m.
- **Bastioni**: a very large central area of Milan. The demand used is that of the morning peak hour (8:00-9:00am); it is characterized by a very complex road network with a huge number of O/Ds. There are 4 different vehicle categories[1], restricted traffic zones to different subsets of vehicle categories and limited traffic zones where tolls are applied.

1 Vehicle categories: car, motorbike, light, heavy

5.2 ACS-TA algorithm performance analysis

The first set of experiments were conducted to evaluate the performance of the different ant link choosing policies. The policies without bias where not considered, because on complex networks ants could not reach the destination even after 10000 tries, and the algorithm can't obtain convergence. The blacklist was pre-populated with the paths that blocked the first 1000 ants in the networks, and then different experiments were run to test the bias applied on the efficient or best route links, with or without the use of a blacklist. Only the deterministic case was considered, using 100 ants and the pheromone trail decay coefficient ρ was kept fixed to 0.8. During all the experiments, the value of the decay coefficient for the total released pheromone $\check{\rho}$ was maintained to 1. For the stop condition, we considered the flow variance in the last 10 iterations, and blocked only when the value of ε was below 0.1. If the number of iterations is 1001, it means that after 1000 iterations the convergence was not yet achieved. The quality of the solution is measured using the using the total time spent by the vehicles in the network [veh*min].

The results are shown in Table 5.2. As we can see, the effectiveness of the various link choose methods are very dependent on the network size. On small networks, like the non separable cost, both the usage of blacklist and bias on efficient links makes harder to converge. This can be explained as more paths are used by ants, leading to more flow variations on the links. On medium sized networks like Area Maggi and Sioux-Falls, both using a blacklist or a bias on efficient links performs better than using a bias on the fastest route. This is possible because many paths that lead to ant blocks are filtered and mostly good paths are chosen. Having paths filtered has the consequence of needing less bias to reach the destination, which leads to a better exploration of paths with more

Table 5.2 Performance analysis of different link choosers

	Bias on links & blacklist	#Iterations	Time to converge[s]	Total time spent in the network [veh*min]	#blocks
Non Separable Costs	best	130	0.6	1,893,044	32,516
	efficient	1001	2.9	1,889,256	225,373
	best+blacklist	1001	2.6	1,890,257	160,037
	efficient+blacklist	1001	2.7	1,897,329	159,180
Sioux Falls	best	14	3.7	274,397	122,087
	efficient	13	3.5	269,960	122,364
	best+blacklist	13	3.3	268,935	123,772
	efficient+blacklist	13	3.9	269,019	122,896
Area Maggi (Milan, Italy)	best	60	13.8	311,468	2,263,747
	efficient	15	15.1	309,403	988,652
	best+blacklist	23	9.7	309,362	861,422
	efficient+blacklist	27	11.7	307,281	952,242
Extra urban Naples (Italy)	best	1001	142.2	359,802	152,214
	efficient	1001	174.0	359,881	151,839
	best+blacklist	1001	141.2	359,843	150,525
	efficient+blacklist	1001	181.6	360,124	150,939

ant blocks as side effect. The final effect is a convergence using less iterations and less time, finding a better solution. On big networks like Naples, the added computation time for finding the efficient links becomes noticeable, while no benefits are given for the convergence. The blacklist fails to decrease the computation time, because the number of possible routes is too much to have a noticeable reduction of ant blocks, although a reduction is present. Considering that the blacklist is easily saved and reused, it should be possible to increase the blacklist performance by doing many more iterations to populate it.

Next set of experiments were conducted to evaluate the effects of three different parameters on the number of iterations needed to converge, the amount of time to converge and the quality of the solution measured. The evaluated parameters are the following:

- Ant number: varied between 100 and 1000;
- Pheromone trail decay coefficient ρ : varied between 0.1 and 0.8
- Traffic assignment model: tested using DUE and SUE Logit with normalized costs. In the case of SUE Logit, standard deviation of their probability density function distributions θ value was varied between 1 and 100.

For the stop condition, we considered the flow variance in the last 10 iterations, and blocked only when the value of ε was below 0.1. The link choosing policy was the bias on the best route without a blacklist. The results of the experiments are shown in Table 5.3. If the cost integral is not present, it means that the value of released pheromone was too low and was approximated to 0, making impossible to distribute the flow. The number of ants has an impact on two aspects, the accuracy of the final solution and the time needed to escape local minima. While the latter might be obvious, the former comes from the Montecarlo interpretation of the ACS-TA algorithm: each ant is a sample from the final flow distribution and thus the more the ants, the more accurate the sample-based estimate of the true flow. The evaporation coefficient ρ significantly affects the convergence of the algorithm: the higher the ρ coefficient, the faster is the convergence. This calls for an accurate setting of this parameter since high values of ρ might induce premature convergence or prevent, due to oscillations, the final convergence to the true value of flow. This usually happens with a value of ρ that is too low. The parameter θ deserves some discussion as well. In SUE assignments, it represents the user uncertainty in cost perception and thus, the higher its value, the less deterministic the user choice. A too low value of θ prevents early convergence, but having a too much low value has proven to give numerical instability. That is because of too few pheromone

Table 5.3 Flow variance convergence over different networks

				Non Separable Costs		Sioux Falls		Area Maggi		Naples	
	Ants	θ	ρ	#iter	Total time spent in the network [veh*min]	#iter	Total time spent in the network [veh*min]	#iter	Total time spent in the network [veh*min]	#iter	Total time spent in the network [veh*min]
SUE	100	1		1	-	1	-	1	-	1001	354,689
Logit	100	10		1	-	577	268,677	1	-	1001	354,807
	100	100		1001	2,309,759	22	305,837	1001	325,764	1001	356,397
	1000	1	0.1	1	-	1	-	1	-	1001	354,746
	1000	10		1	-	187	269,284	1	-	1001	354,784
	1000	100		1001	2,281,969	26	305,773	71	302,586	1001	354,957
DUE	100	-		1001	1,895,234	30	295,671	459	310,282	1001	358,503
	1000	-		1001	1,856,982	15	295,365	42	313,933	1001	355,171
SUE	100	1		1	-	1	-	1	-	1001	354,746
Logit	100	10		1	-	79	267,942	1	-	1001	354,805
	100	100		1001	8,525,308	15	295,261	1001	315,669	1001	356,584
	1000	1	0.4	1	-	1	-	1	-	1001	354,717
	1000	10		1	-	37	269,792	1	-	1001	354,783
	1000	100		1001	2,657,262	17	290,282	142	301,407	1001	354,974
DUE	100	-		1001	1,893,628	15	279,552	863	312,235	1001	359,109
	1000	-		1001	1,856,577	15	275,097	48	309,741	1001	355,353
SUE	100	1		1	-	1	-	1	-	1001	354,717
Logit	100	10		1	-	17	268,487	1	-	1001	354,807
	100	100		1001	3,858,955	14	286,012	1001	304,451	1001	356,180
	1000	1	0.8	1	-	1	-	1	-	1001	354,751
	1000	10		1	-	21	267,115	1001	805,986	1001	354,783
	1000	100		1001	2,522,362	14	281,110	59	300,043	1001	354,959
DUE	100	-		130	1,893,044	13	274,276	60	311,468	1001	359,802
	1000	-		37	1,861,989	13	270,178	49	309,045	1001	355,370

released, which usually end up approximating the released pheromone to 0 and prevent the flow distribution.

In the next experiment, a varying ρ was used to address the convergence and the accuracy problem of having a high ρ . The more the cost integral converges, the more the value of ρ decreases to have a better precision and reduce oscillations. The experiments where conducted using 100 and 1000 ants, DUE traffic assignment model, a starting ρ of 0.8, that is halved every time the update conditions explained in equation (48) are met, until it reaches the value of 0.01.

Table 5.4 Convergence analysis using variable ρ

			Non Separable Costs		Sioux Falls		Area Maggi		Naples	
Ants	d	ρ	#iter	Total time spent in the network [veh*min]	#iter	Total time spent in the network [veh*min]	#iter	Total time spent in the network [veh*min]	#iter	Total time spent in the network [veh*min]
100	0.1	0.8	1001	1,893,413	13	274,737	1001	313,014	1001	358,367
100	0.005		1001	1,891,877	13	274,465	1001	312,938	1001	358,207
100	0.001		1001	1,890,401	13	274,882	721	313,413	1001	358,910
1000	0.1		83	1,863,055	13	270,340	28	309,912	1001	355,311
1000	0.005		33	1,866,123	13	270,047	37	309,338	1001	355,245
1000	0.001		37	1,861,989	13	270,170	49	309,045	1001	355,278

The d value of equation (48) was varied between 0.1 and 0.001, while the average windows was kept to 10. The final results are visible in Table 5.4.

From the table we can see that a varying ρ helps in reducing the number of iterations needed to have convergence only in some scenarios, like in Area Maggi, using 1000 ants. If a lesser ants are used, a decreasing ρ makes convergence more difficult to reach. Analyzing how the ρ decrease, we saw that its value started to decrease too soon, when the convergence was still distant. To resolve this problem, a lower d should be used. Varying ρ also does not affect the convergence when it is reached after a low number of iterations. Experiments show also that, using 1000 ants, a too high value of d is like using a fixed ρ , because its value is decreased only when the algorithm is too near the convergence.

During the last experiments, it was clear that a convergence using the flow variance is not effective in finding a good solution. The first cause is the early convergence on networks like Sioux-Falls. The second cause is the missing convergence on bigger networks or networks with non separable costs, where the variations on flows are usually higher. To find a better convergence, a new run of simulations was done for one SUE and DUE scenario. This time, the

Table 5.5 A-posterior convergence analysis

				Non Separable Costs		Sioux Falls		Area Maggi		Naples	
	Ants	θ	ρ	#iter	Total time spent in the network [veh*min]	#iter	Total time spent in the network [veh*min]	#iter	Total time spent in the network [veh*min]	#iter	Total time spent in the network [veh*min]
SUE	1000	100	0.8	1001	2,522,361	961	264,701	110	300,063	106	367,670
DUE	1000	100		1001	1,856,625	497	264,405	89	308,842	110	355,198

convergence analysis was conducted a-posterior over 1000 iterations and the convergence is reached when the 90% of flows are less then 1% from the flow at the end of the 1000 iterations. Table 5.5 shows the obtained the results. With the new convergence system, we obtained an effective convergence on all the Networks with separable cost links. Using this convergence, we can do a performance comparison between ACS-TA and the CUBE5 Voyager [46] commercial software. This commercial software can develop a deterministic multi-class equilibrium assignment and it is very commonly used especially in public administrations for transport network analysis and planning. For comparison we have considered the best configuration (in the sense of number of iterations and time to convergence) for ACS-TA in a DUE assignment of the Naples network: number of ants equal to 1000, $\theta=100$ and $\rho=0.8$; the criteria of convergence are the same as presented in the previous section. Table 5.6 reports the main indexes of performance. The number of iterations of ACS-TA is twice the number of iterations of CUBE, but the quality of the final equilibrium is better. Unfortunately, we could not carry out a performance comparison for SUE assignments where ACS-TA should be much more effective with respect to other algorithms being CUBE suitable only for DUE assignment.

Finally, an experiment was done on the most complex network, Bastioni. The tested scenario was a SUE using 1000 ants, $\theta=100$ and $\rho=0.8$. In this network the second convergence criteria was met in 1071 minutes, after 687 iterations,

Table 5.6 Performance comparison between ACS-TA and CUBE-Voyager

	Iterations	Time to convergence (s)	Total time spent in the network [veh]*min
CUBE-Voyager	50	9	390,901
DUE	110	64.9	355,198

with a final total time spent in the network of 601,731 veh*min.

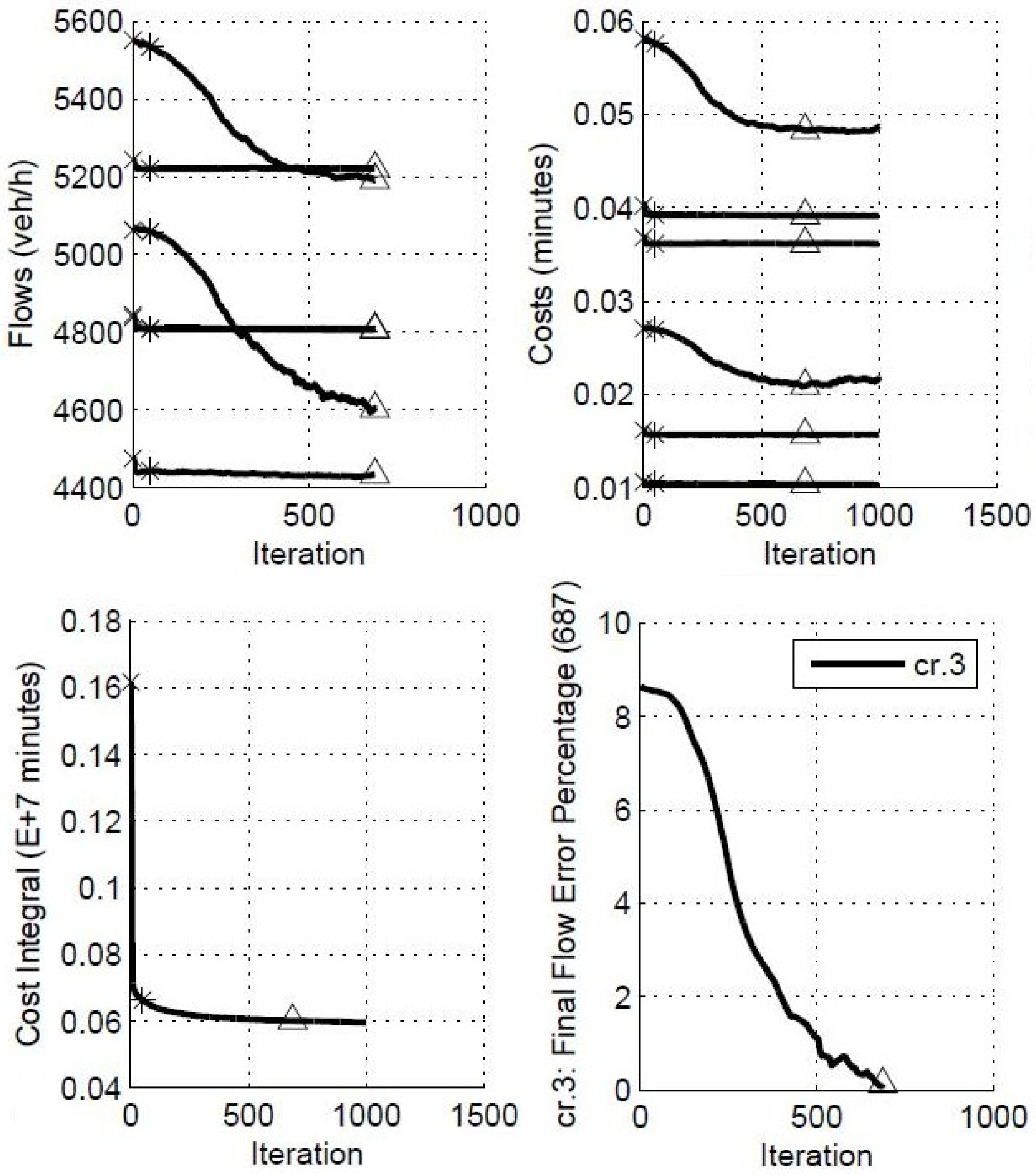


Fig 5.1: Bastioni network (SUE-Logit assignment,1000 ants, $\theta=100$ *and* $\rho=0.8$ *). These plots are representative of the results obtained. The figures show the link flow and the link cost of the six highest flow links (top left and top right respectively in the figure), the total cost (cost integral in* 10^7 *minutes, bottom left) and, the second criteria plot. In brackets is reported the number of iterations at convergence.*

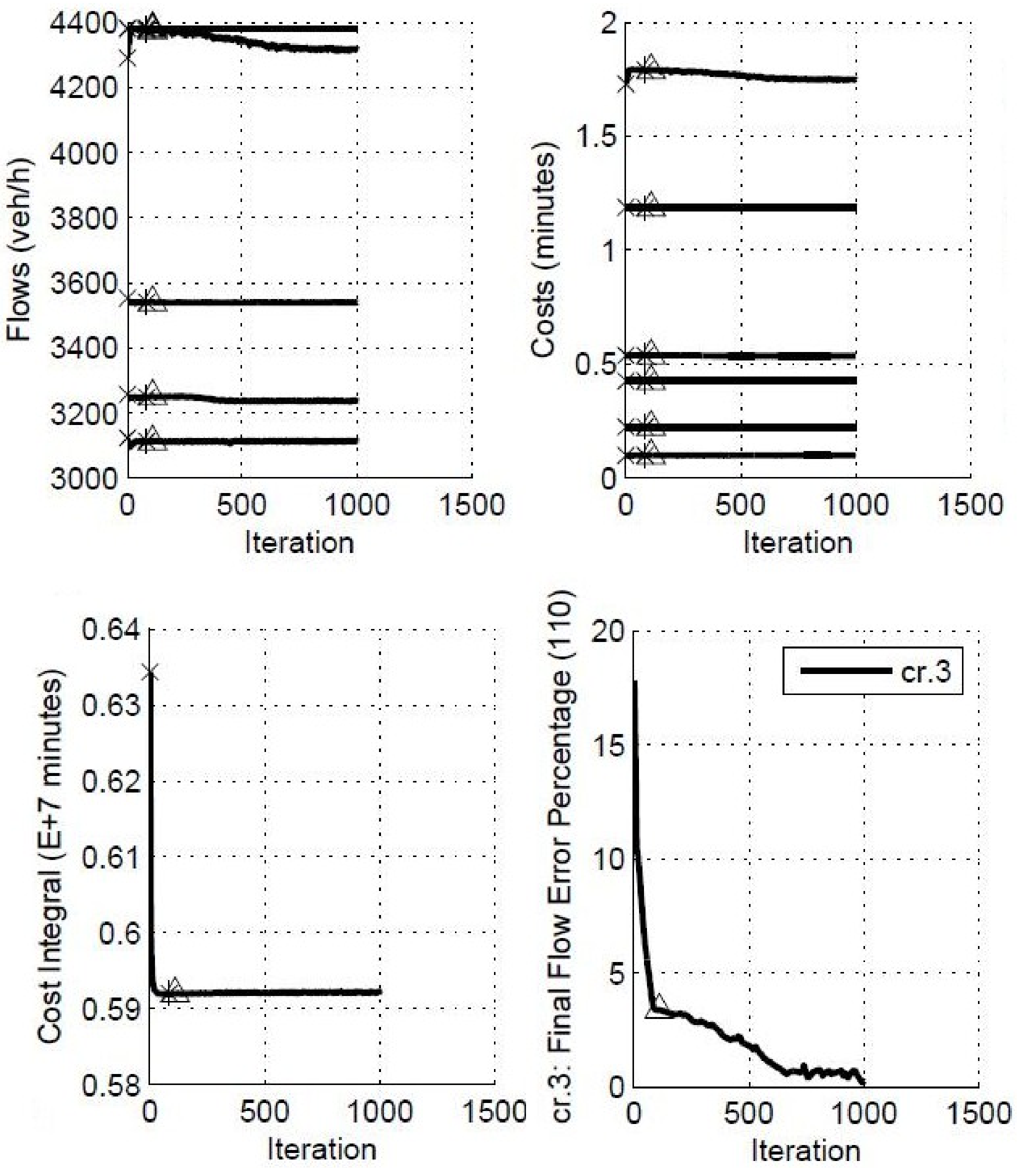


Fig 5.2: Napoli network (DUE assignment,1000 ants and $\rho=0.8$). These plots are representative of the results obtained. The figures show the link flow and the link cost of the six highest flow links (top left and top right respectively in the figure), the total cost (cost integral in 10^7 minutes, bottom left) and, the second criteria plot. In brackets is reported the number of iterations at convergence.

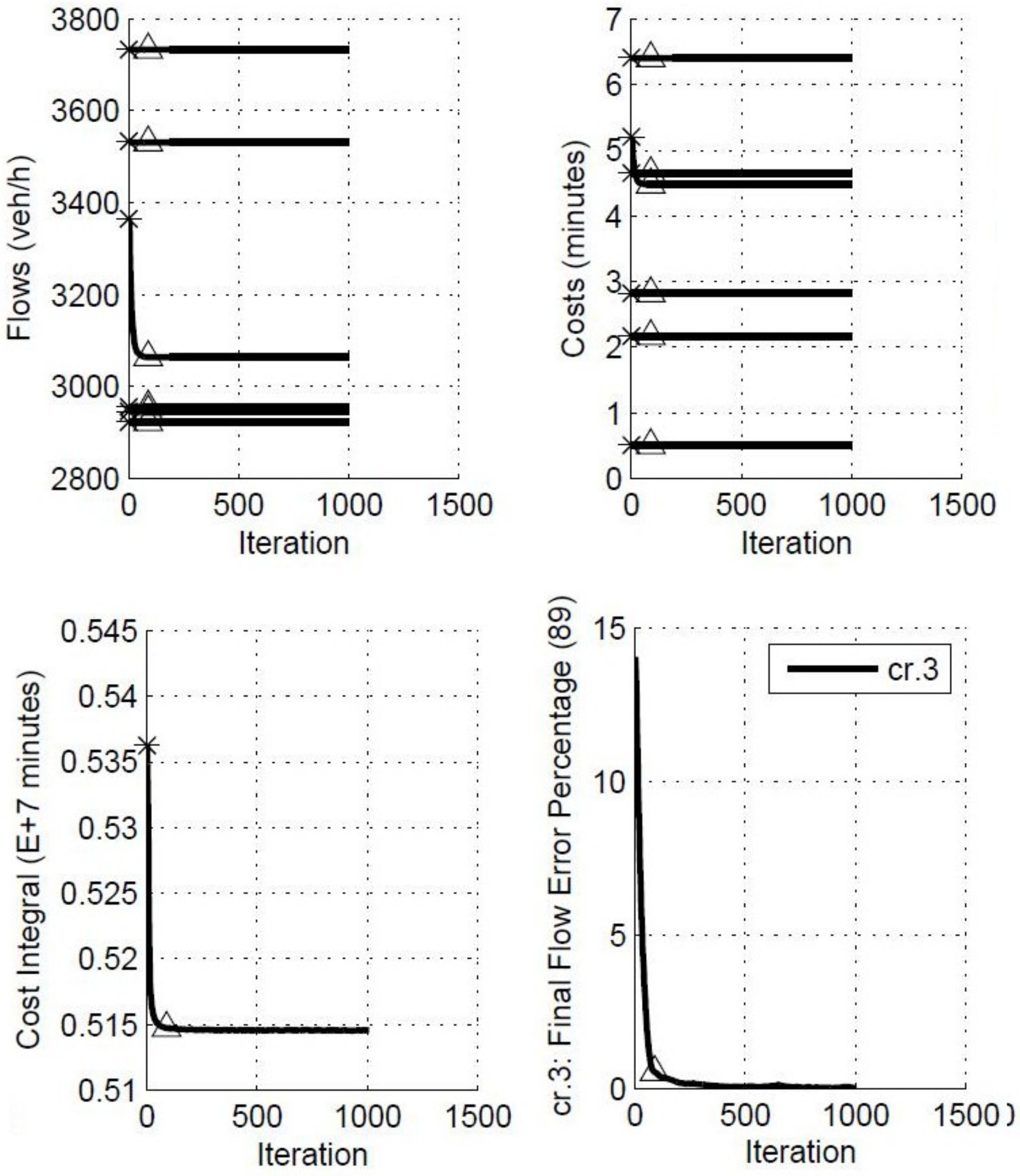


Fig 5.3: Maggi network (DUE assignment,1000 ants and ρ=0.8). These plots are representative of the results obtained. The figures show the link flow and the link cost of the six highest flow links (top left and top right respectively in the figure), the total cost (cost integral in 10^7 minutes, bottom left) and, the second criteria plot. In brackets is reported the number of iterations at convergence.

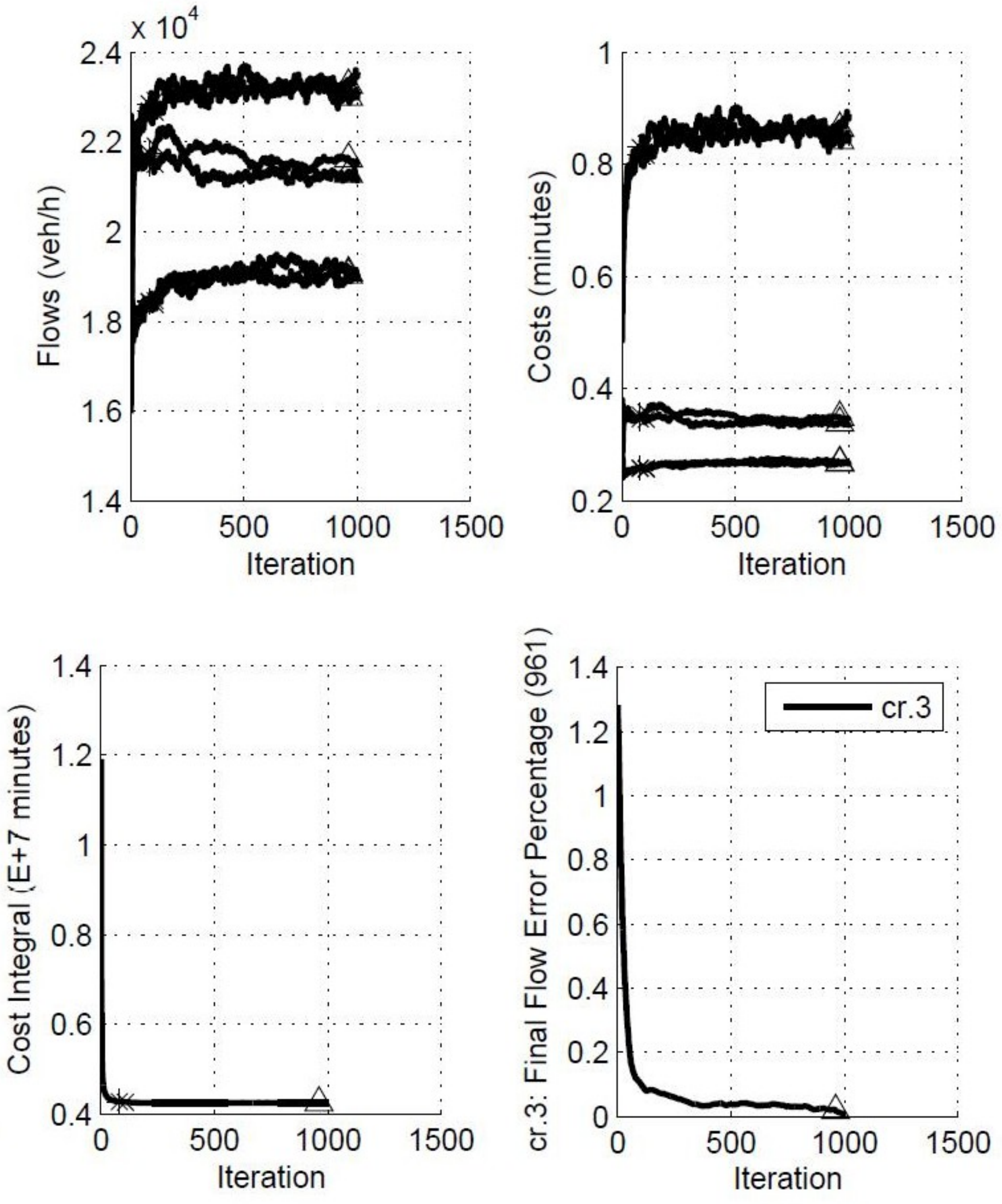


Fig 5.4: Sioux network (SUE-Logit assignment,1000 ants, $\theta=100$ *and* $\rho=0.8$ *). These plots are representative of the results obtained. The figures show the link flow and the link cost of the six highest flow links (top left and top right respectively in the figure), the total cost (cost integral in* 10^7 *minutes, bottom left) and, the second criteria plot. In brackets is reported the number of iterations at convergence.*

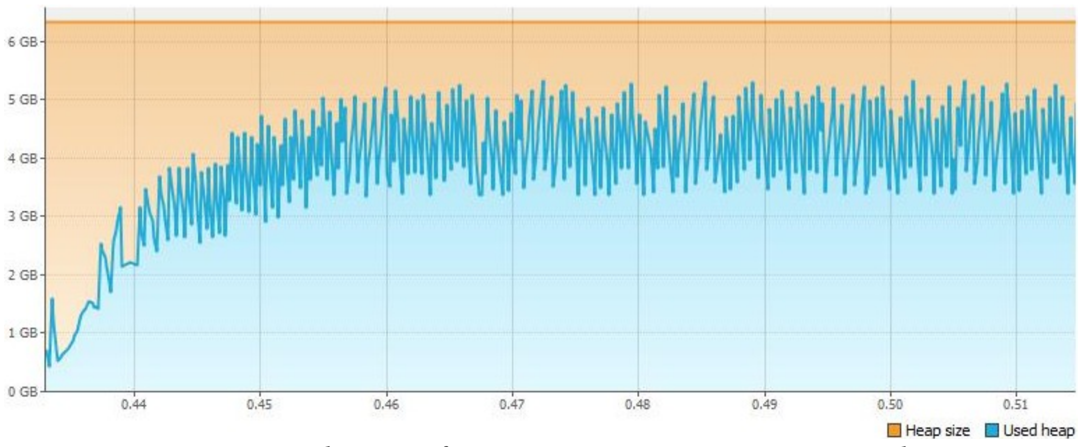


Fig 5.5: Memory usage during software execution on Bastioni network

5.3 Memory management

Another aspect that was considered during the experiments, that can be critical on bigger networks, is memory consumption. In Figure 5.1 it is shown the memory consumption during the algorithm execution on Bastioni, taken with a profiling software called VisualVM[1]. The memory consumption proved to be stable, without any memory leaks during the execution. It is present a fixed amount of occupied memory, around 3.4GB. This is not surprising, because in the case of 8 byte primitives, every information on links regarding a O/D/C combination will take 3919·17448·8=547,029,696 bytes of memory. That means that only keeping in memory the flow, the list of permitted O/D/C that can access a link, pheromone present on links and released pheromone on links, will consume 2,188,118,784 bytes. The remaining amount is used for indexes, class addressing and other model data like identification numbers or link costs. It would be possible, at the cost some performance loss, to save the pheromone on file, freeing another 1,094,059,392 bytes of memory. The rest of the used memory show in Figure 5.1, is used during the algorithm execution for temporary data that is frequently garbage collected. The amount of this memory is

1 See http://visualvm.java.net/

maximum 2GB and was fixed using JVM parameters (see Appendix A, “Java Virtual Machine parameters” for more details).

Chapter 6

Conclusions and future work

At the end of this work, we can affirm that we successfully designed and implemented the ACS-TA algorithm, an extended version of Ant Colony System, to solve DUE and SUE traffic assignment problems. ACS-TA is particularly versatile and suitable for application in many real cases without assuming simplifying hypotheses on cost functions. Differently from classical traffic assignment algorithms, the applicability of ACS-TA does not depend on the shape of the objective function and hence the particular cases of non-separable cost link function and multi-class demand can also be tackled easily and successfully. Moreover, the user choice model is implicitly defined through the definition of a suitable pheromone update formula. Different models, ranging from the classical Logit to more sophisticated ones that use perception error, could be defined by changing this updating formula. Applications to real networks show a computation time that is short enough, with respect to traditional approaches, for converging also in complex networks cases.

The impact of pheromone decay was also analyzed and we can suggest that its effects depend strongly on network structure and on cost functions. Generally, we expect that by increasing the value of ρ oscillations increase, but this holds true only if the feasible set of possible paths is wide. For the SUE model, the choosing of a good value of θ is critical, as the performance becomes better as the value decrease, but with a too low value the released pheromone are too few and close to 0.

Usually a few iterations are sufficient for the algorithm to converge, also in complex networks, and probably a better tuning of parameters could reduce the number of iterations even more. Particularly, using a varying ρ and a populated blacklist can decrease the convergence time. The impact of standard deviation in cost perception distribution has been investigated and the results show how the stochastic nature of ACS can solve SUE faster than DUE problems.

More research is required to investigate convergence mechanisms, especially when the existence and uniqueness of convergence cannot be theoretically demonstrated. A future step of this research will be the application of ACS-TA to the study of very large networks (that is, ones with more than 40,000 links) to test all the above-mentioned features and evaluate the time reduction obtainable by parallel implementation. On very large networks, memory consumption becomes a critical aspect. This issue was already partially taken into consideration in this work, with the possibility to save data on disk. This feature was not used in these experiments because memory size was sufficient to contain all the data.

A future possible works on the software is to rewrite the algorithm in a faster language, like c or c++, to increase the computation speed and reduce memory consumption. To easily implement the network data visualization, it should be relatively simple to save the network final state on disk. A software designed in a high level language, like Java using Jung library, should be able to visualize the network and provide tools to easily access all the useful data, which for now is only saved on files. Using files is convenient for small to medium networks, but for very huge networks having the possibility to easily select the interesting data becomes a critical aspect. Finally, it is possible to increase parallelization, by implementing a client/server architecture with a centralized software that assigns work to multiple clients in a network.

Appendix A

Software use manual

This manual explains how the software need to be configured to run all the simulations used in the thesis, and the configurations needed to run it on any network.

General definitions

Here are the definitions that will be used in the manual:

A	set of links of the network
N	set of nodes of the network
$n_i \in N$	node in the network
$a_{i,j} \in A$	link that connects two nodes n_i and n_j
A_i	set of links that go out node n_i
$f_{i,j}(t)$	flow on a link at time t
$c_{i,j}(f,t)$	cost to travel through a link with flow f at time t
$Cap_{i,j}$	capacity of link $a_{i,j}$
F	all the possible paths between an Origin/Destination couple
$F^* \in F$	all the path between an Origin/Destination couple choose by ants
F^*_{min}	path with less cost between an Origin/Destination couple
F^*_n	path choose by an ant
M	total number of ants
$M_{i,j}$	number of ants that go through link $a_{i,j}$

Requirements

The software needs the following files and directories to work correctly:

AntColonyFlowAssignment.jar
<network_directory>/parameters_ant.txt
<network_directory>/seeds.txt
<network_directory>/arcs.txt
<network_directory>/nodes.txt
<network_directory>/OD.txt
<network_directory>/logs/logEvents.txt
<network_directory>/logs/authorizedFlows.txt
<network_directory>/tmp/

After having installed Java (the tested version are JRE 1.6 and JRE 1.7), with the software compiled into the jar file, it is possible to start the elaboration with the command:

```
java -jar AntColonyFlowAssignment.jar [-network {path to network
directory}] [-noresolve] [-parameters {path to file with
parameters}] [-nogui] [-flow {O} {D} {C}] [-p {parameterName}
{parameterValue}]*
```

The parameters used have the following meaning:

- **network**: indicates the path to the folder where are stored the files of the network;
- **noresolve**: creates the network model without any flow assignment;
- **parameter**: indicates the path/filename containing the network parameters. If not specified, it uses a default value “parameters_ant.txt” in the folder indicated in the network parameter;
- **nogui**: disables the network visualization at the end of the flow assignment;
- **flow**: indicates the origin/destination/category combination that is visualized on the network at the end of the elaboration. Link colors have a gradation ranging from blue, when no flow is using the link, to red, when all the flow is going through the link. The origin and destination node will be respectively green and red;

- **p**: adds the specified parameter, eventually overwriting the parameter written in the parameters file. Multiple parameters can be specified.

Data log during execution

The file *log/logEvents.txt* contains various events where data is saved to file, while *log/authorizedFlows.txt* contains the flow that we want to be logged. The possible events are described in chapter 4.1.2.

Settings and parameters

All the parameters are defined in the file *parameters_ant.txt,* divided into various subsections. A parameter name is always preceded by "-", the next line contains the value of the parameter. Every line that does not respect this convention is considered a comment. In the parentheses are indicated the default values, used when the parameter is not defined in the file.

Basic Parameters

- **Seed_file** (seeds.txt): file name containing the seeds used for the generation of casual values;
- **OD_File** (OD.txt): file name containing the O/D/C combinations and flow generated by them. If a flow value is 0 or less, it will be ignored;
- **Nodes_File** (nodes.txt): file name containing the nodes and their position;
- **Coordinates_divisor** (100): the value used to divide the nodes position coordinates. Used to correctly show the network in the graphic visualization;
- **Coordinates_add_x** (0): added value to the coordinate X, to correctly show the network in the graphic visualization;
- **Coordinates_add_y** (0): added value to the coordinate Y, to correctly show the network in the graphic visualization;
- **Thread_number** (4): number of threads to use during parallel computation;
- **Thread_timeout** (1000): maximum execution time, in seconds, after which a thread is considered to be in an endless loop and the software

return an error;
- **Add_date_to_log** (true): if 'true', adds the date to the log file names, to avoid overwriting different tests;
- **Use_decimal_formatter** (false): if 'true', uses the exponential format to visualize long values in the logs;
- **Label**: if present, puts the label before the log file name. Useful to differentiate a set of test runs on a batch;
- **Arcs_File** (arcs.txt): file name containing the arcs and their parameters;
- **Arcs_Type** (3): type of cost functions used for the arcs, described in chapter 4.2.1;
- **Save_pheromone_on_file** (false): if 'true', at each iteration after the calculation of how much pheromone are released by an ant colony, the values are saved to disk and the memory is released to have enough of it for the next elaboration. At the next iteration, the pheromone will be read from file. Useful for huge networks on systems that does not have enough memory to contain all the data. If the value is false, the pheromone are always kept in memory to have a faster execution time;
- **Save_total_pheromone_on_file** (false): as the parameter before, but for the total released pheromone;
- **Low_memory_blacklist** (true): if 'true', at each iteration after the calculation of how much pheromone are released by an ant colony, the blacklist is saved on disk and the memory released to have enough for the next elaboration.

Network optimization

The network optimization, described in 3.4.7, can work really well on networks like Maggi, to reduce the number of possible routes. It is important to run again the algorithm if an O/D/C combination is added or removed, or if any link is added or removed. Not doing it, can lead to a wrong solution calculation, because the ants ignore links that could be used.

- **Use_optimization** (true): if 'true', applies the network optimization explained in chapter 3.4.7;
- **Recalculate_flow** (false): recalculates the flow even if there is already a file containing a previous calculation;
- **Flow_File** (link_flow.txt): file name containing which colonies can use

each link, created after the optimization algorithm.

Network initialization

The parameters listed here are used during the network initialization, described in chapter 3.4.1.

- **Initial_flow_modifier** (1.0): value of *Const_f* used to calculate the cost of the fastest path:

$$C_{min}(0) \quad = \quad \sum c_{i,j}(Cap_{i,j} \cdot Const_f, 0) \ \forall \ c_{i,j} \in F^*_{min}$$

- **Ant_type** (Due): used for the pheromone initialization on the links. Can assume the following values:
  - ***Due:***

$$\tau_{i,j}(0) \quad = \quad Const_1 \cdot {}^1\!/_{C_{min}} \quad \forall i,j : a_{i,j} \in A$$

  - ***Due_2:***

$$\tau_{i,j}(0) \quad = \quad Const_1 \cdot {}^1\!/_{C^2_{min}(0)} \quad \forall i,j : a_{i,j} \in A$$

  - ***Due_3:***

$$\tau_{i,j}(0) \quad = \quad Const_1 \cdot {}^1\!/_{C^3_{min}(0)} \quad \forall i,j : a_{i,j} \in A$$

  - ***Probi:***

$$\tau_{i,j}(0) = Const_1 \frac{1}{max(0.1\ ,\ Const_2 \cdot N(0,\sigma) \cdot C_{min}(0))} \forall i,j : a_{i,j} \in A$$

  - ***Logit:***

$$\tau_{i,j}(0) \quad = \quad Const_1 \cdot e^{(-C_{min}(0)/\theta)} \quad \forall i,j : a_{i,j} \in A$$

  - ***LogitNormalized:***

$$\tau_{i,j}(0) \quad = \quad Const_1 \ \forall i,j : a_{i,j} \in A$$

- **Initial_flow_modifier** (1.0): value of *Const_f*;

- **Initial_pheromone_modifier** (1.0): value of $Const_1$;
- **Probit_aleatory_width** (0.1): value of $Const_2$;
- **Probit_sigma** (1.0): value of σ;
- **Theta** (10): value of θ.

Path choosing

The parameters listed here are used during the path choosing, described in chapter 3.4.2.

- **Chooser_type** (Standard):
  - ***Standard:*** basic path choosing algorithm (26)
  - ***PheromoneGaussWithBias:*** standard with perception error on pheromone as in (27);
  - ***CostGaussWithBias:*** standard with perception error on link cost
  - ***BestBias:*** bias on fastest path as in (31) and (32);
  - ***EfficientBias:*** bias on efficient links as in (33) and (34);
  - ***BlackList*****:** standard algorithm using blacklist;
  - ***BestBiasBlackList:*** bias on the best route using blacklist;
  - ***EfficientBlackList:*** bias on efficient links using blacklist;
- **Bias_value** (2): value of $Const_1$, which is the bias to add in *BestBias* and *EfficientBias* algorithms;
- **Aleatory_width** (0.1): value of σ^2, the Gaussian variance;
- **Add_black_list_route** (false): if 'false', no new paths will be added to the blacklist;
- **Max_memory_usage_percent** (50): new paths will be added only if the memory is at least this percentage free.

Pheromone distribution

The parameters listed here are used during the pheromone distribution, described in chapter 3.4.3.

- **Ant_type** (Due):
    - ***Due***:

$$\tau_{i,j,n}(t) = \begin{cases} 1/C_n & \forall i,j: a_{i,j} \in F_n^* \\ 0 & \forall i,j: a_{i,j} \notin F_n^* \wedge a_{i,j} \in A \end{cases}$$

    - ***Due_2***:

$$\tau_{i,j,n}(t) = \begin{cases} 1/C_n^2 & \forall i,j: a_{i,j} \in F_n^* \\ 0 & \forall i,j: a_{i,j} \notin F_n^* \wedge a_{i,j} \in A \end{cases}$$

    - ***Due_3***:

$$\tau_{i,j,n}(t) = \begin{cases} 1/C_n^3(t) & \forall i,j: a_{i,j} \in F_n^* \\ 0 & \forall i,j: a_{i,j} \notin F_n^* \wedge a_{i,j} \in A \end{cases}$$

    - ***Probit***:

$$\tau_{i,j,n}(t) = \begin{cases} \dfrac{1}{max(0.1, Const_1 \cdot N(0,\sigma) \cdot C_n(t))} & \forall i,j: a_{i,j} \in F_n^* \\ 0 & \forall i,j: a_{i,j} \notin F_n^* \wedge a_{i,j} \in A \end{cases}$$

    - ***Logit***:

$$\tau_{i,j,n}(t) = \begin{cases} e^{(-C_n(t)/\theta)} & \forall i,j: a_{i,j} \in F_n^* \\ 0 & \forall i,j: a_{i,j} \notin F_n^* \wedge a_{i,j} \in A \end{cases}$$

    - ***LogitNormalized***

$$\tau_{i,j,n}(t) = \begin{cases} e^{(-C_n(t) - C_{min}/\theta)} & \forall i,j: a_{i,j} \in F_n^* \\ 0 & \forall i,j: a_{i,j} \notin F_n^* \wedge a_{i,j} \in A \end{cases}$$

- **Probit_aleatory_width** (0.1): value of *$Const_1$* in *Probit*;
- **Probit_sigma** (1.0): value of σ;
- **Theta** (10): value of θ;

Pheromone evaporation

The parameters listed here are used during the pheromone evaporation, described in chapter 3.4.3.

- **Evaporate_unused_link** (false): if 'true', pheromone evaporates on every link as in (39), else the pheromone evaporates only on links used at least once by an ant in the current iteration as in (40);
- **Rho** (0.5): value of $\rho \in (0,1]$, the pheromone trail decay coefficient;
- **Total_rho** (1): value of $\check{\rho} \in (0,1]$, the trail decay coefficient of the total released pheromone. Giving it a value of 1, means that no memory is kept on the previous iteration pheromone values when assigning the new flow.

Flow assignment

The parameters listed here are used during the flow assignment, described in chapter 3.4.4.

- **Flow_chooser_type** (PheromoneDistribution):
  - ***PheromoneDistribution*:** assigns the flow using the released pheromone as in (43);
  - ***RoutesTreeDistribution:*** assigns the flow enumerating all the paths as in (44);
  - ***AntDistribution:*** assigns the flow using ants as in (45) and (46)

- **Flow_distribution_iterations** (1000): value of *$k_{tot,}$*, which is the number of ants used in *AntDistribution*;

Rho update

The parameters listed here are used during the ρ update, described in chapter 3.4.5.

- **Rho_updater_type** (NoUpdate):
  - ***NoUpdate*:** the value of ρ remains always the same during the algorithm execution;
  - ***CostIntegralAverage:*** the value of ρ varies if the cost integral does not change much, as in (48);

- **Average_length** (10): the value of I;
- **Average_distance** (0.1): the value of d;
- **Rho_division** (2): the value of q;
- **Rho_min** (0.01): the value of ρ_{min}
- **Rho** (0.5): the initial value of ρ ;

Stop condition

The parameters listed here are used during the stop condition, described in chapter 3.4.6.

- **Stop_type** (*optional*):
  - ***CostVariance*:** uses the highest variance of the links cost to determine if stop, as in (51);
  - ***FlowVariance***: uses the highest variance of the flow in links to determine if stop, as in (54);

- **Min_iteration** (1000): minimum number of iterations, regardless of the selected *Stop_type* used;
- **Max_iteration** (1000): maximum number of iterations, regardless of the selected *Stop_type* used;
- **Epsilon** (0.01): value of ε used in variance check;
- **Variance_length** (20): value of K, the iterations number used in the variance calculation.

Java Virtual Machine parameters

Using the right JVM parameters when lunching the software can have a huge impact on the application performance. To avoid dynamical memory assignment, which takes some time and blocks the application, it is better to assign the same value for the heap minimum and maximum value, using the parameters *-Xms* and *-Xmx*. It is best to assign most of the available memory, leaving at least 1GB free for the operative system, to avoid frequent use of garbage collector, which can increase the computation time of the algorithm.

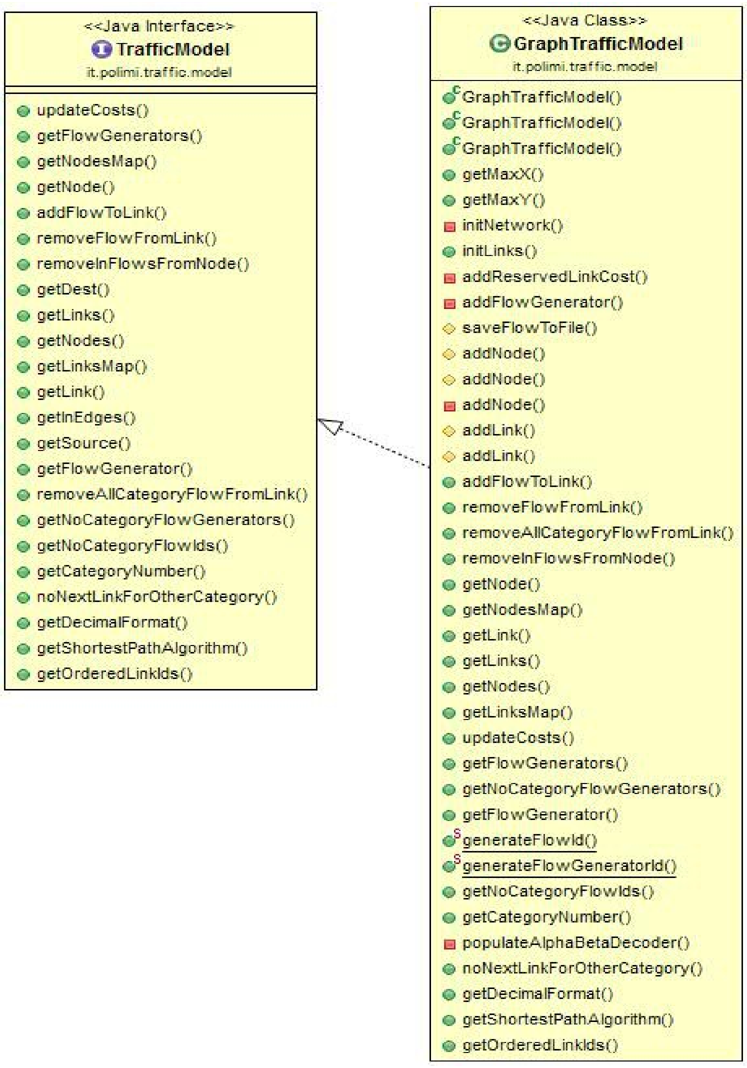
Another important parameter is the heap dimension assigned to the "Young generation objects". Every allocated object in java starts as a Young generation, and only if it survives to several garbage collections because it is addressed, it becomes a "Tenured Generation". For efficiency reasons, only Young Generation objects are checked and released during the normal garbage collection. The ideal scenario is to have all the objects used by the model in the Tenured Generation, while the frequently used and deleted objects remain in the Young Generation to be released after the use. If the software try to promote Young Generation objects into Tenured Generation and does not have enough memory, it will trigger a Global Garbage Collector, which blocks all the threads and executes for several seconds to remove unused objects in the Tenured Generation. Global Garbage Collector can severally increase the algorithm execution time, or completely block the program, if used too many times. To avoid this, it is important to assign the right amount of memory for the Tenured Generation and Young Generation. It is possible to choose the Young Generation heap space with the parameters *-XX:NewSize* and *-XX:MaxNewSize*. The memory usable by the Tenured Generation is the difference between the total allocated space and the Young Generation heap space. When running a new network, it is best to select the Tenured Generation heap space so that it is at least 20% more than the memory used by the network model. To know how much memory is used, it is possible to run a Global Garbage Collector during the algorithm execution using a profiling software for Java, like VisualVM[1], and check how much memory remains used after its execution. The default garbage collector used by java is the “Parallel garbage collector”, which is the most efficient when the software runs on multi-core systems and generates lot of Young Generation object like in our case. For a better understanding of the Java garbage collection, it is possible to read the official documentation [43] and [44].

1 http://visualvm.java.net/

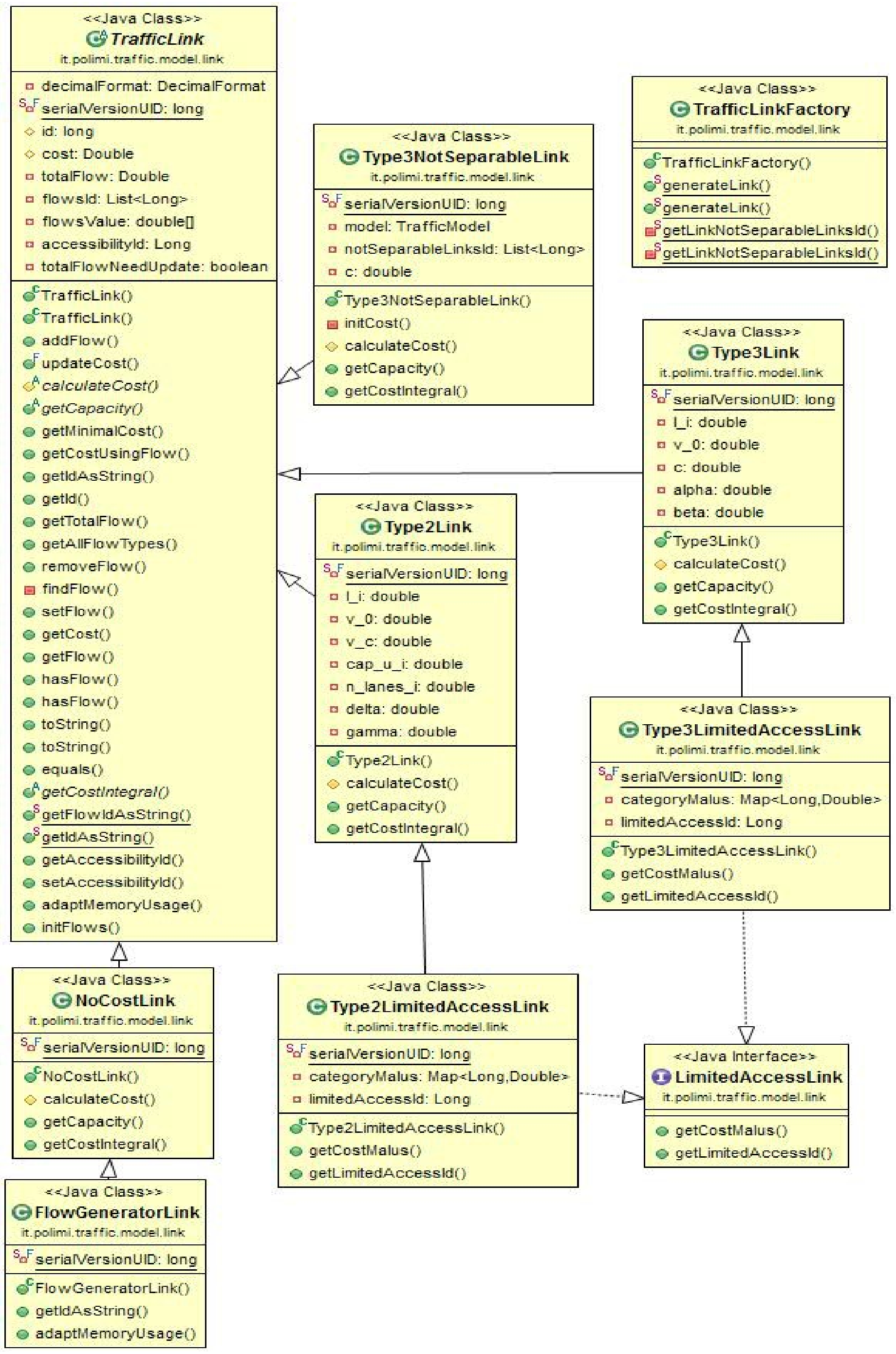
Appendix B

Class diagrams

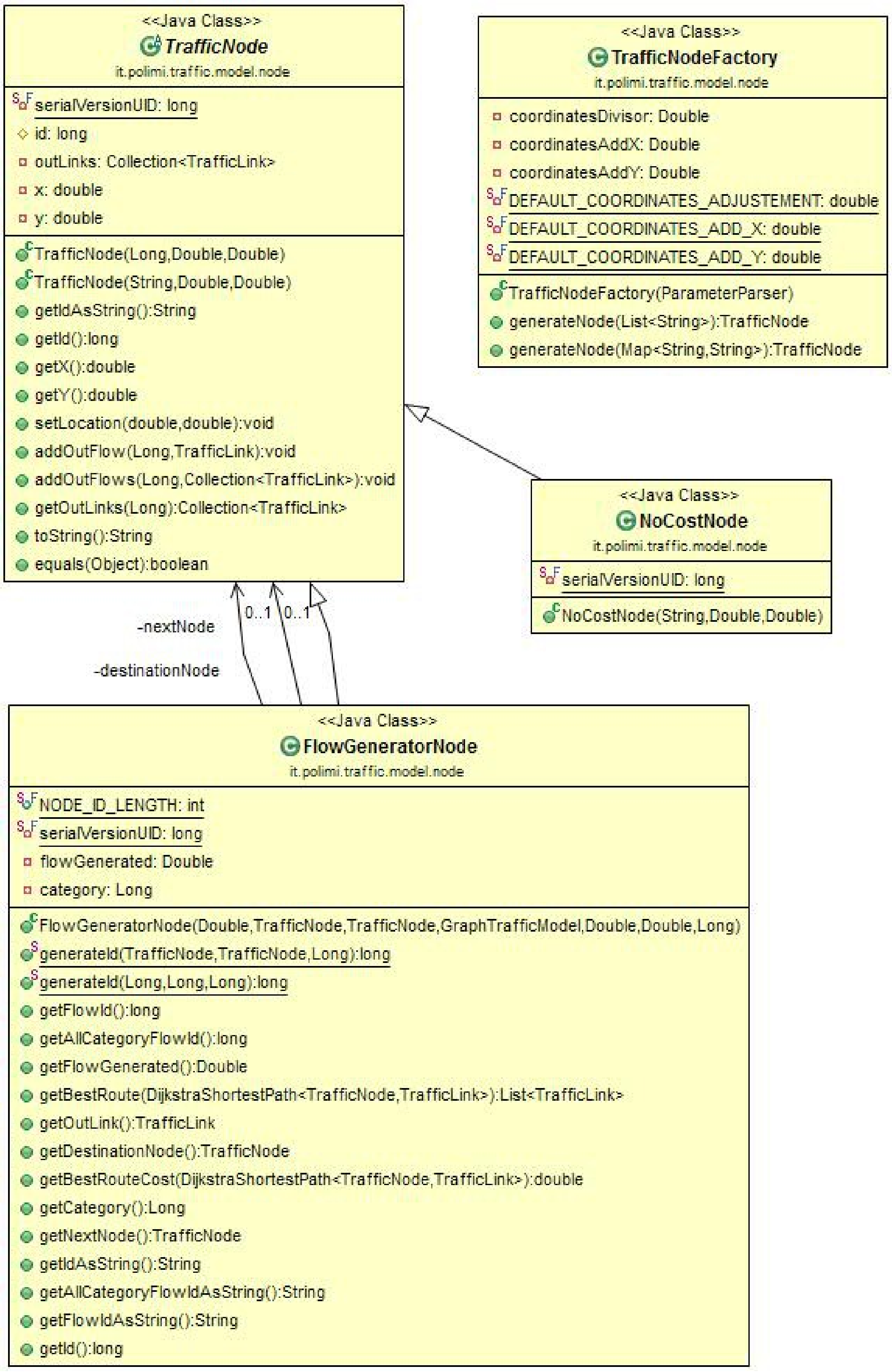
Package it.polimi.traffic.model



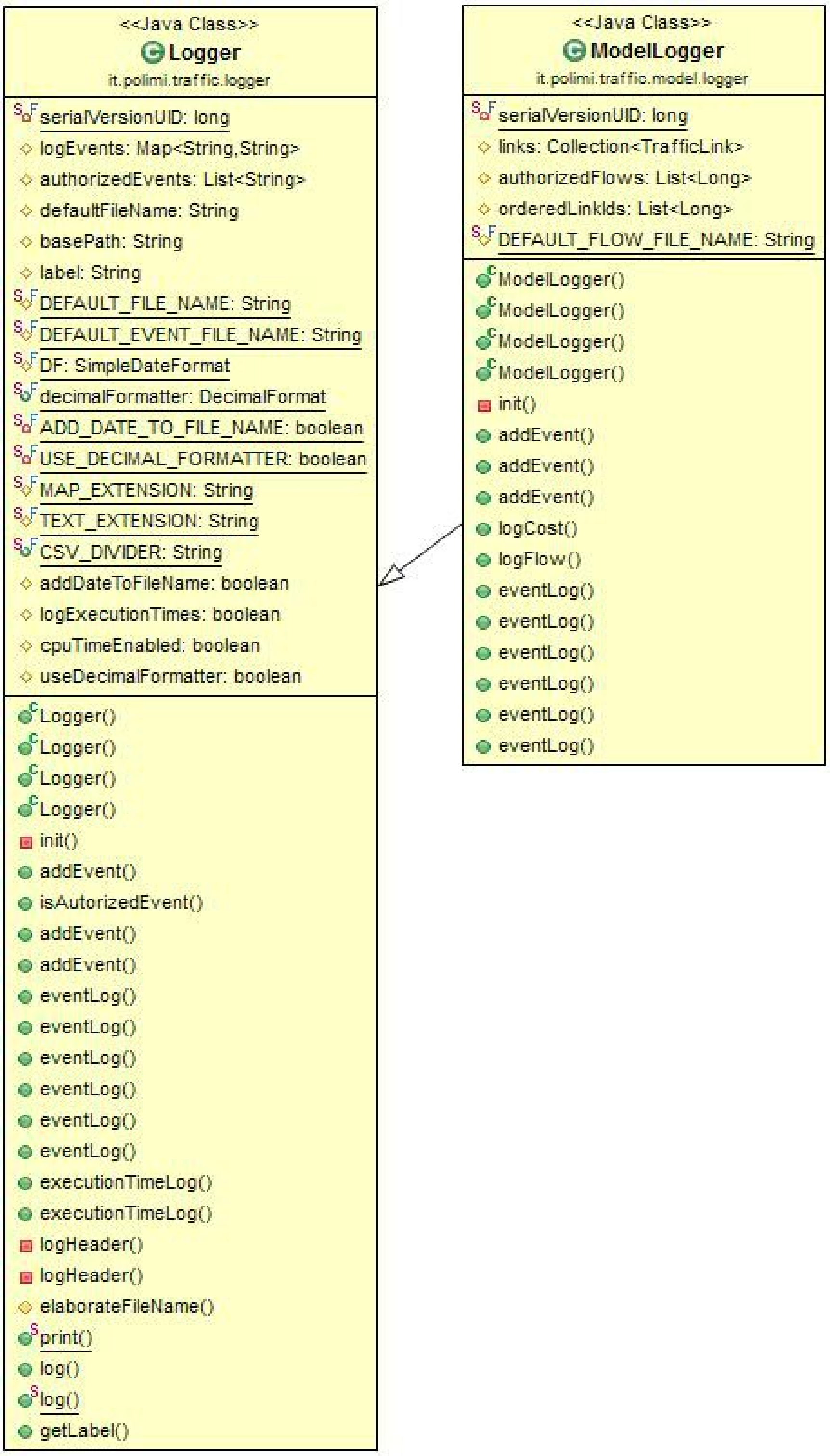
Package it.polimi.traffic.model.link



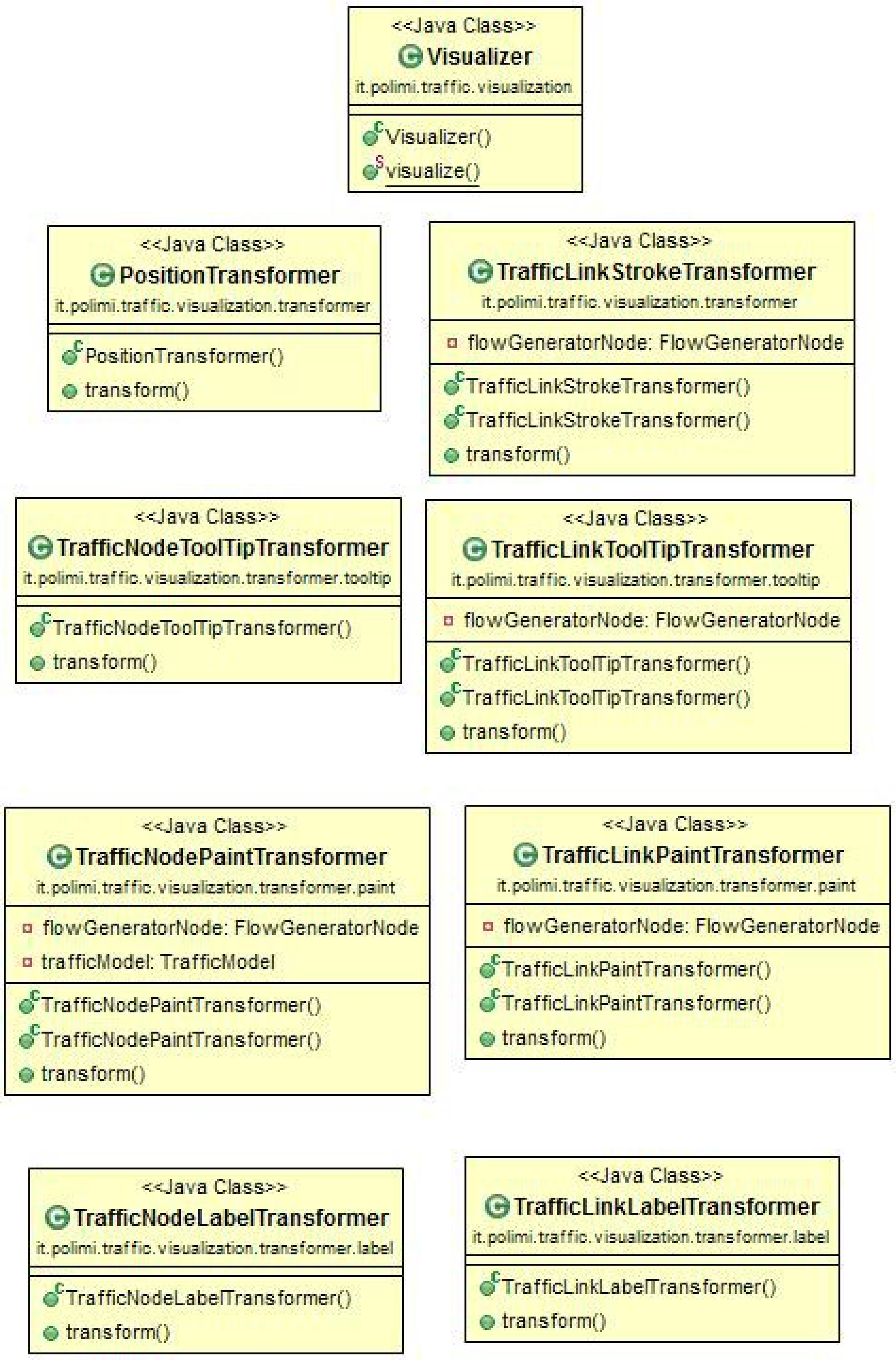
Package it.polimi.traffic.model.node


<<Java Class>>
TrafficNode
it.polimi.traffic.model.node
serialVersionUID: long
id: long
outLinks: Collection<TrafficLink>
x: double
y: double
TrafficNode(Long,Double,Double)
TrafficNode(String,Double,Double)
getIdAsString():String
getId():long
getX():double
getY():double
setLocation(double,double):void
addOutFlow(Long,TrafficLink):void
addOutFlows(Long,Collection<TrafficLink>):void
getOutLinks(Long):Collection<TrafficLink>
toString():String
equals(Object):boolean
<<Java Class>>
TrafficNodeFactory
it.polimi.traffic.model.node
coordinatesDivisor: Double
coordinatesAddX: Double
coordinatesAddY: Double
DEFAULT_COORDINATES_ADJUSTEMENT: double
DEFAULT_COORDINATES_ADD_X: double
DEFAULT_COORDINATES_ADD_Y: double
TrafficNodeFactory(ParameterParser)
generateNode(List<String>):TrafficNode
generateNode(Map<String,String>):TrafficNode
<<Java Class>>
NoCostNode
it.polimi.traffic.model.node
serialVersionUID: long
NoCostNode(String,Double,Double)
-nextNode
-destinationNode
0..1
0..1
<<Java Class>>
FlowGeneratorNode
it.polimi.traffic.model.node
NODE_ID_LENGTH: int
serialVersionUID: long
flowGenerated: Double
category: Long
FlowGeneratorNode(Double,TrafficNode,TrafficNode,GraphTrafficModel,Double,Double,Long)
generateId(TrafficNode,TrafficNode,Long):long
generateId(Long,Long,Long):long
getFlowId():long
getAllCategoryFlowId():long
getFlowGenerated():Double
getBestRoute(DijkstraShortestPath<TrafficNode,TrafficLink>):List<TrafficLink>
getOutLink():TrafficLink
getDestinationNode():TrafficNode
getBestRouteCost(DijkstraShortestPath<TrafficNode,TrafficLink>):double
getCategory():Long
getNextNode():TrafficNode
getIdAsString():String
getAllCategoryFlowIdAsString():String
getFlowIdAsString():String
getId():long

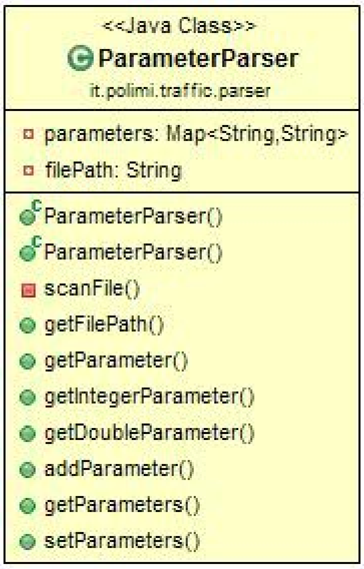
Package it.polimi.traffic.model.logger

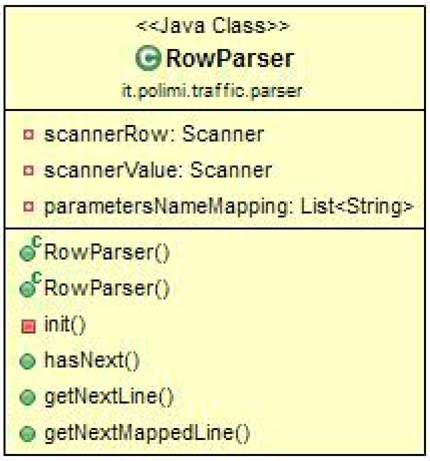


Package it.polimi.traffic.visualization

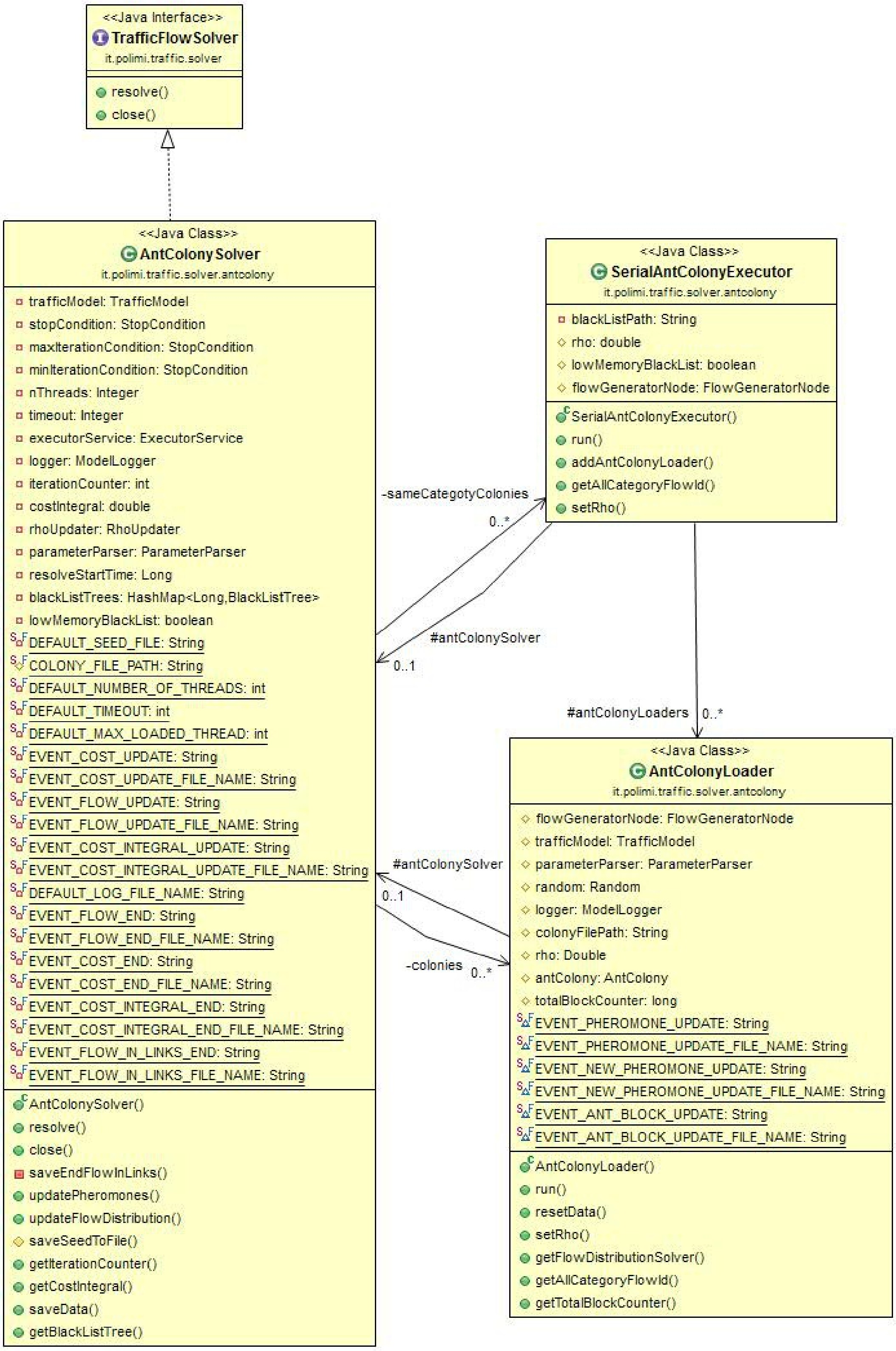

<<Java Class>>
Visualizer
it.polimi.traffic.visualization
Visualizer()
visualize()
<<Java Class>>
PositionTransformer
it.polimi.traffic.visualization.transformer
PositionTransformer()
transform()
<<Java Class>>
TrafficLinkStrokeTransformer
it.polimi.traffic.visualization.transformer
flowGeneratorNode: FlowGeneratorNode
TrafficLinkStrokeTransformer()
TrafficLinkStrokeTransformer()
transform()
<<Java Class>>
TrafficNodeToolTipTransformer
it.polimi.traffic.visualization.transformer.tooltip
TrafficNodeToolTipTransformer()
transform()
<<Java Class>>
TrafficLinkToolTipTransformer
it.polimi.traffic.visualization.transformer.tooltip
flowGeneratorNode: FlowGeneratorNode
TrafficLinkToolTipTransformer()
TrafficLinkToolTipTransformer()
transform()
<<Java Class>>
TrafficNodePaintTransformer
it.polimi.traffic.visualization.transformer.paint
flowGeneratorNode: FlowGeneratorNode
trafficModel: TrafficModel
TrafficNodePaintTransformer()
TrafficNodePaintTransformer()
transform()
<<Java Class>>
TrafficLinkPaintTransformer
it.polimi.traffic.visualization.transformer.paint
flowGeneratorNode: FlowGeneratorNode
TrafficLinkPaintTransformer()
TrafficLinkPaintTransformer()
transform()
<<Java Class>>
TrafficNodeLabelTransformer
it.polimi.traffic.visualization.transformer.label
TrafficNodeLabelTransformer()
transform()
<<Java Class>>
TrafficLinkLabelTransformer
it.polimi.traffic.visualization.transformer.label
TrafficLinkLabelTransformer()
transform()

Package it.polimi.traffic.parser



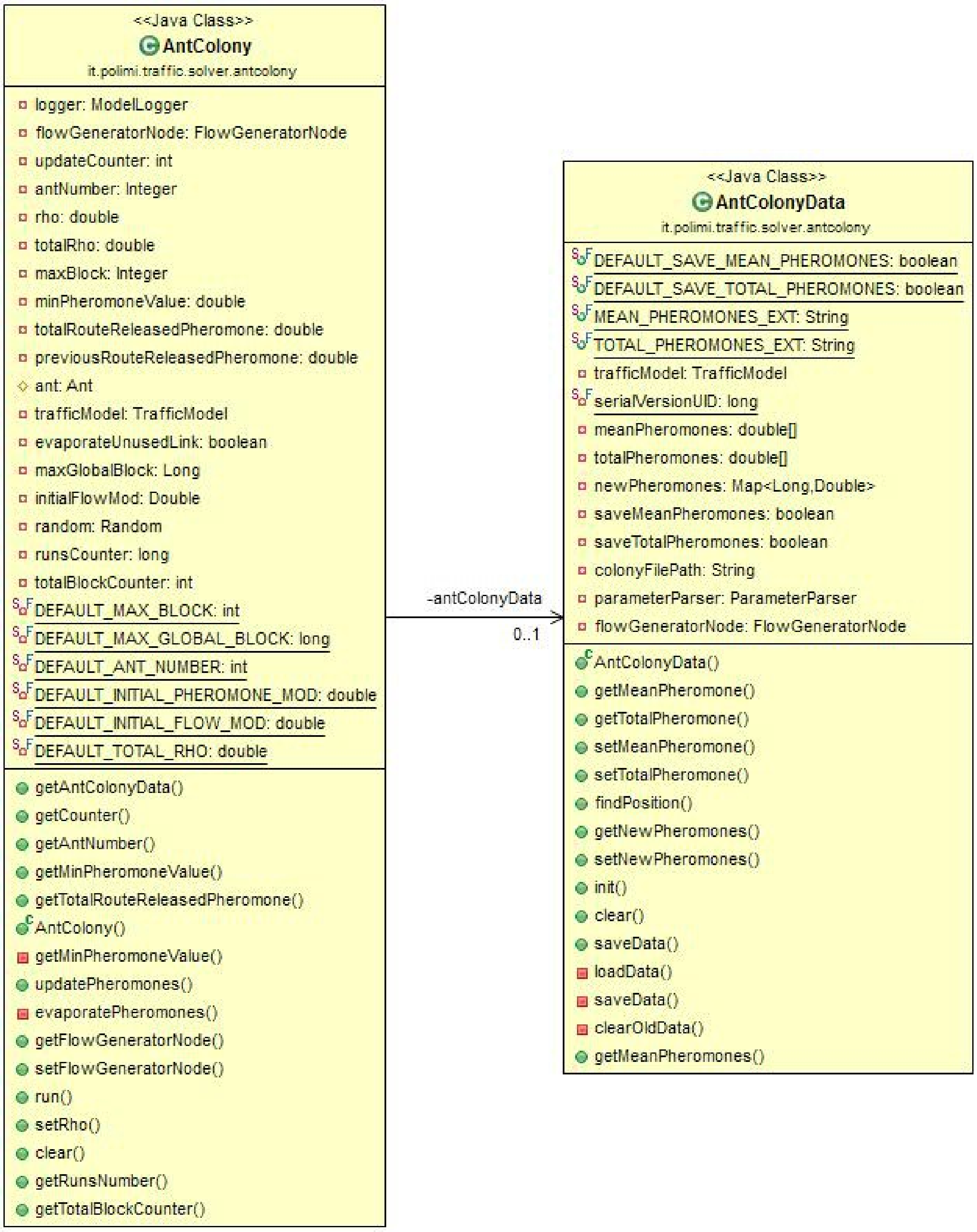


Package it.polimi.traffic.solver – Part 1

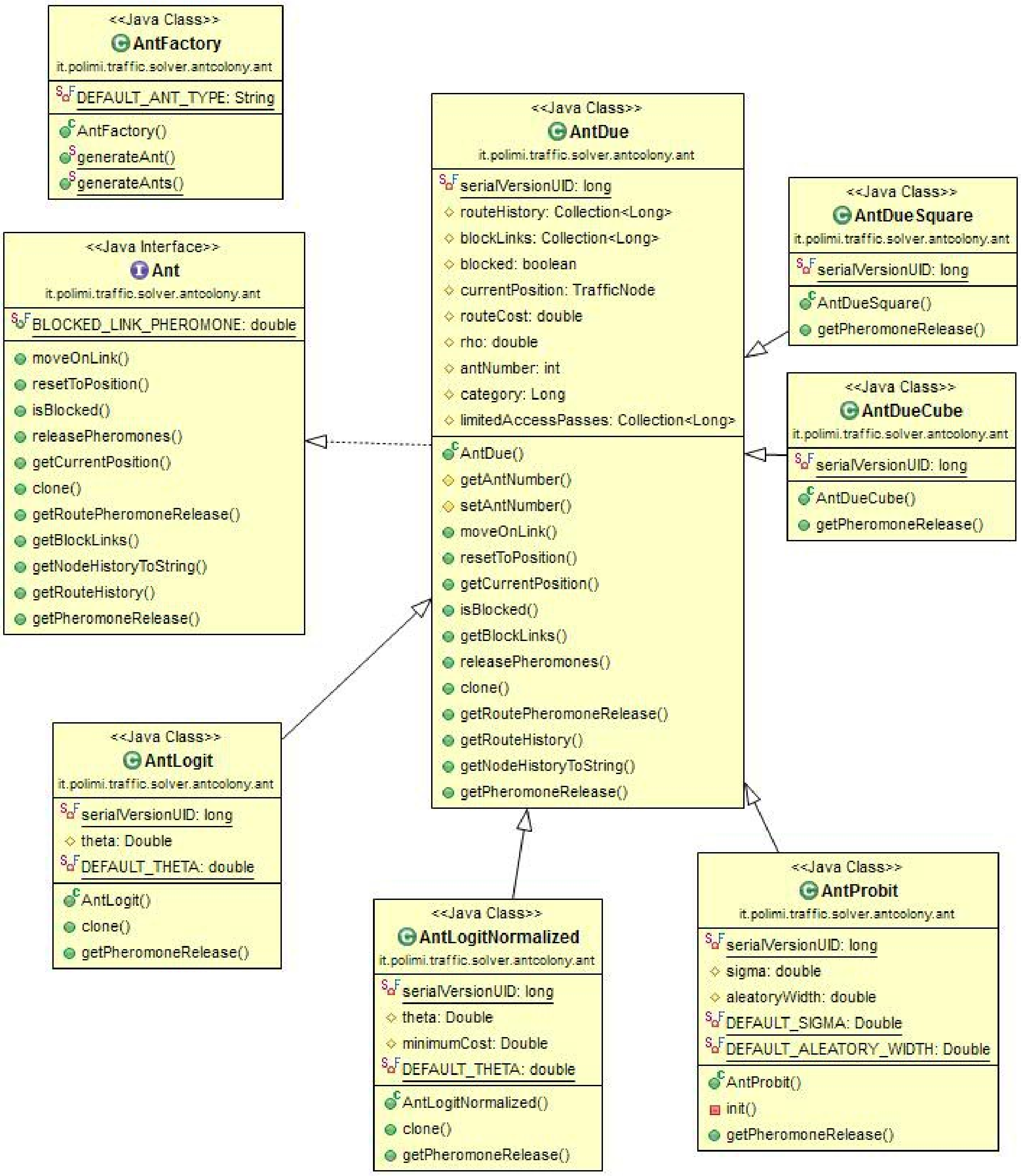


<<Java Interface>>
TrafficFlowSolver
it.polimi.traffic.solver
resolve()
close()
<<Java Class>>
AntColonySolver
it.polimi.traffic.solver.antcolony
trafficModel: TrafficModel
stopCondition: StopCondition
maxIterationCondition: StopCondition
minIterationCondition: StopCondition
nThreads: Integer
timeout: Integer
executorService: ExecutorService
logger: ModelLogger
iterationCounter: int
costIntegral: double
rhoUpdater: RhoUpdater
parameterParser: ParameterParser
resolveStartTime: Long
blackListTrees: HashMap<Long,BlackListTree>
lowMemoryBlackList: boolean
DEFAULT_SEED_FILE: String
COLONY_FILE_PATH: String
DEFAULT_NUMBER_OF_THREADS: int
DEFAULT_TIMEOUT: int
DEFAULT_MAX_LOADED_THREAD: int
EVENT_COST_UPDATE: String
EVENT_COST_UPDATE_FILE_NAME: String
EVENT_FLOW_UPDATE: String
EVENT_FLOW_UPDATE_FILE_NAME: String
EVENT_COST_INTEGRAL_UPDATE: String
EVENT_COST_INTEGRAL_UPDATE_FILE_NAME: String
DEFAULT_LOG_FILE_NAME: String
EVENT_FLOW_END: String
EVENT_FLOW_END_FILE_NAME: String
EVENT_COST_END: String
EVENT_COST_END_FILE_NAME: String
EVENT_COST_INTEGRAL_END: String
EVENT_COST_INTEGRAL_END_FILE_NAME: String
EVENT_FLOW_IN_LINKS_END: String
EVENT_FLOW_IN_LINKS_FILE_NAME: String
AntColonySolver()
resolve()
close()
saveEndFlowInLinks()
updatePheromones()
updateFlowDistribution()
saveSeedToFile()
getIterationCounter()
getCostIntegral()
saveData()
getBlackListTree()
<<Java Class>>
SerialAntColonyExecutor
it.polimi.traffic.solver.antcolony
blackListPath: String
rho: double
lowMemoryBlackList: boolean
flowGeneratorNode: FlowGeneratorNode
SerialAntColonyExecutor()
run()
addAntColonyLoader()
getAllCategoryFlowId()
setRho()
-sameCategotyColonies
0..*
#antColonySolver
0..1
#antColonyLoaders
0..*
<<Java Class>>
AntColonyLoader
it.polimi.traffic.solver.antcolony
flowGeneratorNode: FlowGeneratorNode
trafficModel: TrafficModel
parameterParser: ParameterParser
random: Random
logger: ModelLogger
colonyFilePath: String
rho: Double
antColony: AntColony
totalBlockCounter: long
EVENT_PHEROMONE_UPDATE: String
EVENT_PHEROMONE_UPDATE_FILE_NAME: String
EVENT_NEW_PHEROMONE_UPDATE: String
EVENT_NEW_PHEROMONE_UPDATE_FILE_NAME: String
EVENT_ANT_BLOCK_UPDATE: String
EVENT_ANT_BLOCK_UPDATE_FILE_NAME: String
AntColonyLoader()
run()
resetData()
setRho()
getFlowDistributionSolver()
getAllCategoryFlowId()
getTotalBlockCounter()
#antColonySolver
0..1
-colonies
0..*

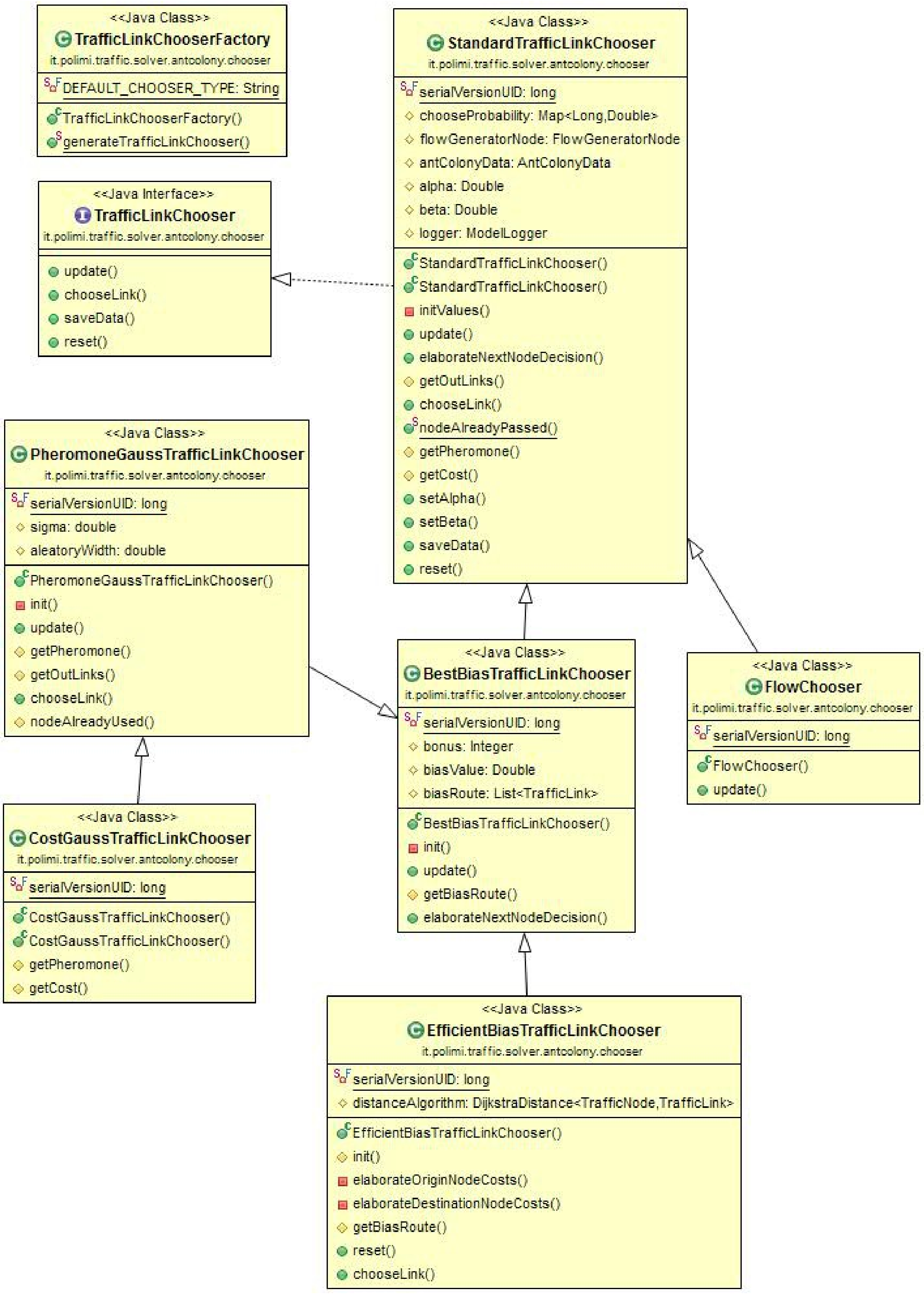
Package it.polimi.traffic.solver – Part 2



Package it.polimi.traffic.solver.ant

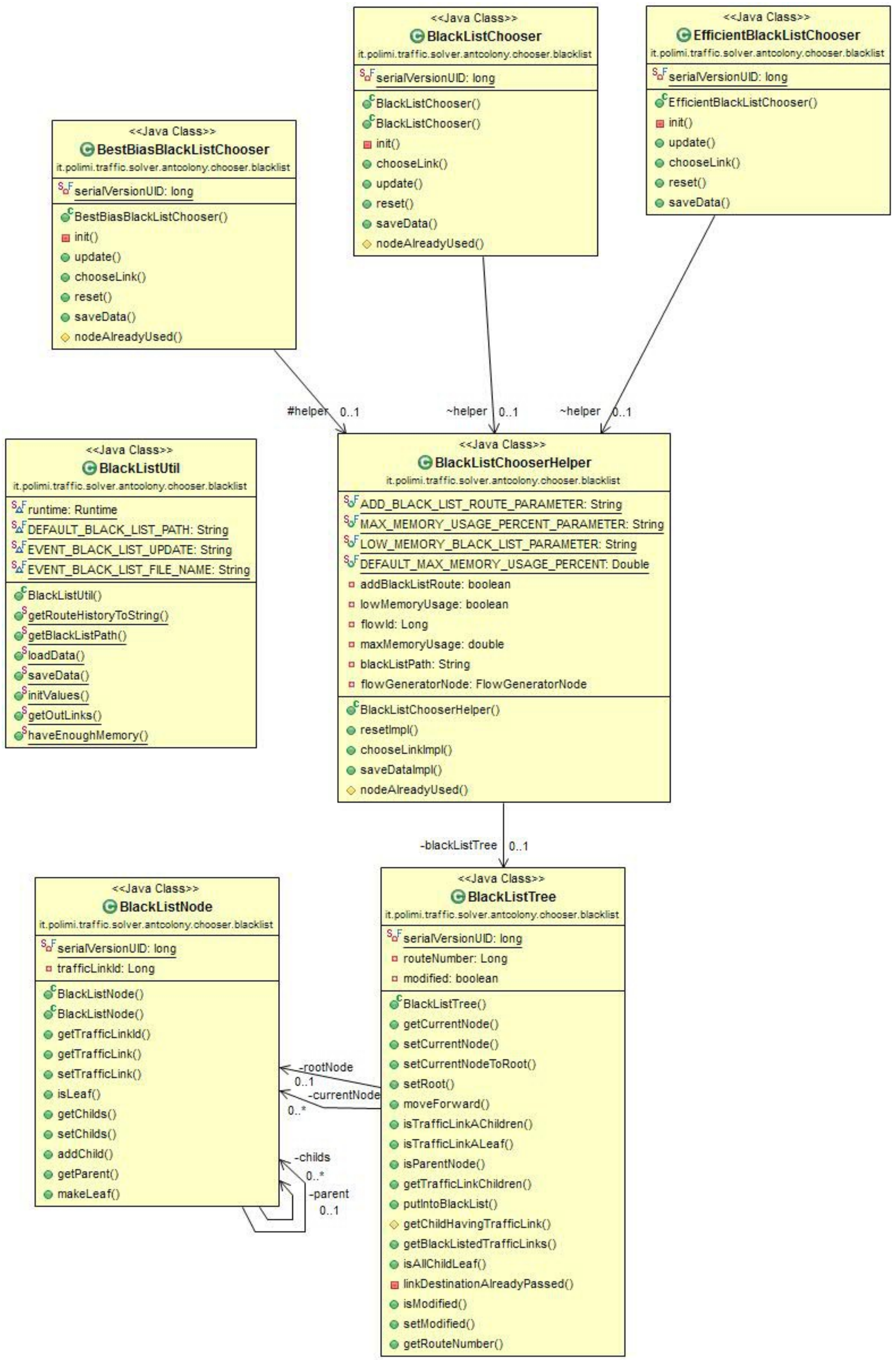


Package it.polimi.traffic.solver.chooser

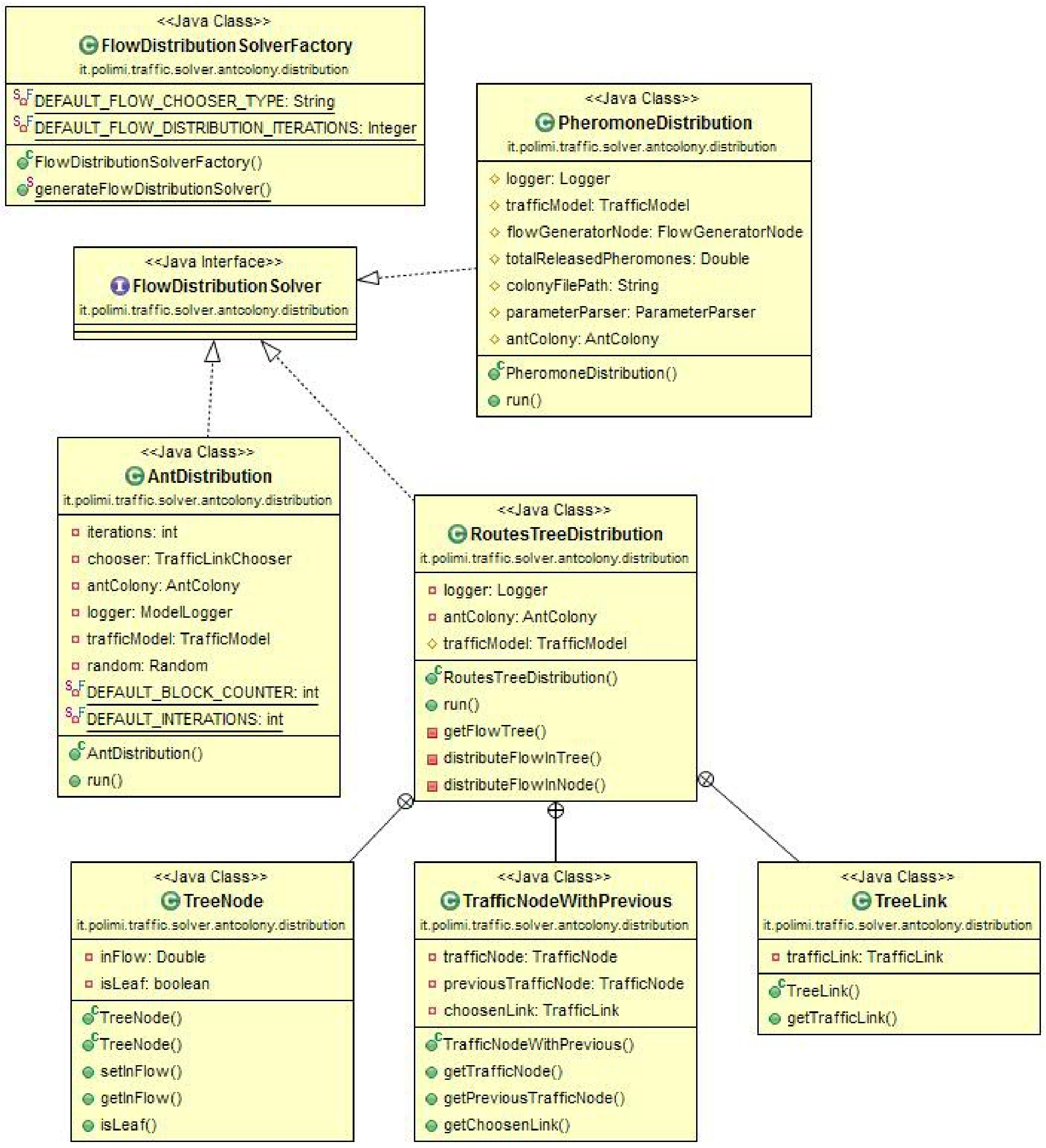


<<Java Class>>
TrafficLinkChooserFactory
it.polimi.traffic.solver.antcolony.chooser
DEFAULT_CHOOSER_TYPE: String
TrafficLinkChooserFactory()
generateTrafficLinkChooser()
<<Java Interface>>
TrafficLinkChooser
it.polimi.traffic.solver.antcolony.chooser
update()
chooseLink()
saveData()
reset()
<<Java Class>>
StandardTrafficLinkChooser
it.polimi.traffic.solver.antcolony.chooser
serialVersionUID: long
chooseProbability: Map<Long,Double>
flowGeneratorNode: FlowGeneratorNode
antColonyData: AntColonyData
alpha: Double
beta: Double
logger: ModelLogger
StandardTrafficLinkChooser()
StandardTrafficLinkChooser()
initValues()
update()
elaborateNextNodeDecision()
getOutLinks()
chooseLink()
nodeAlreadyPassed()
getPheromone()
getCost()
setAlpha()
setBeta()
saveData()
reset()
<<Java Class>>
PheromoneGaussTrafficLinkChooser
it.polimi.traffic.solver.antcolony.chooser
serialVersionUID: long
sigma: double
aleatoryWidth: double
PheromoneGaussTrafficLinkChooser()
init()
update()
getPheromone()
getOutLinks()
chooseLink()
nodeAlreadyUsed()
<<Java Class>>
BestBiasTrafficLinkChooser
it.polimi.traffic.solver.antcolony.chooser
serialVersionUID: long
bonus: Integer
biasValue: Double
biasRoute: List<TrafficLink>
BestBiasTrafficLinkChooser()
init()
update()
getBiasRoute()
elaborateNextNodeDecision()
<<Java Class>>
FlowChooser
it.polimi.traffic.solver.antcolony.chooser
serialVersionUID: long
FlowChooser()
update()
<<Java Class>>
CostGaussTrafficLinkChooser
it.polimi.traffic.solver.antcolony.chooser
serialVersionUID: long
CostGaussTrafficLinkChooser()
CostGaussTrafficLinkChooser()
getPheromone()
getCost()
<<Java Class>>
EfficientBiasTrafficLinkChooser
it.polimi.traffic.solver.antcolony.chooser
serialVersionUID: long
distanceAlgorithm: DijkstraDistance<TrafficNode,TrafficLink>
EfficientBiasTrafficLinkChooser()
init()
elaborateOriginNodeCosts()
elaborateDestinationNodeCosts()
getBiasRoute()
reset()
chooseLink()

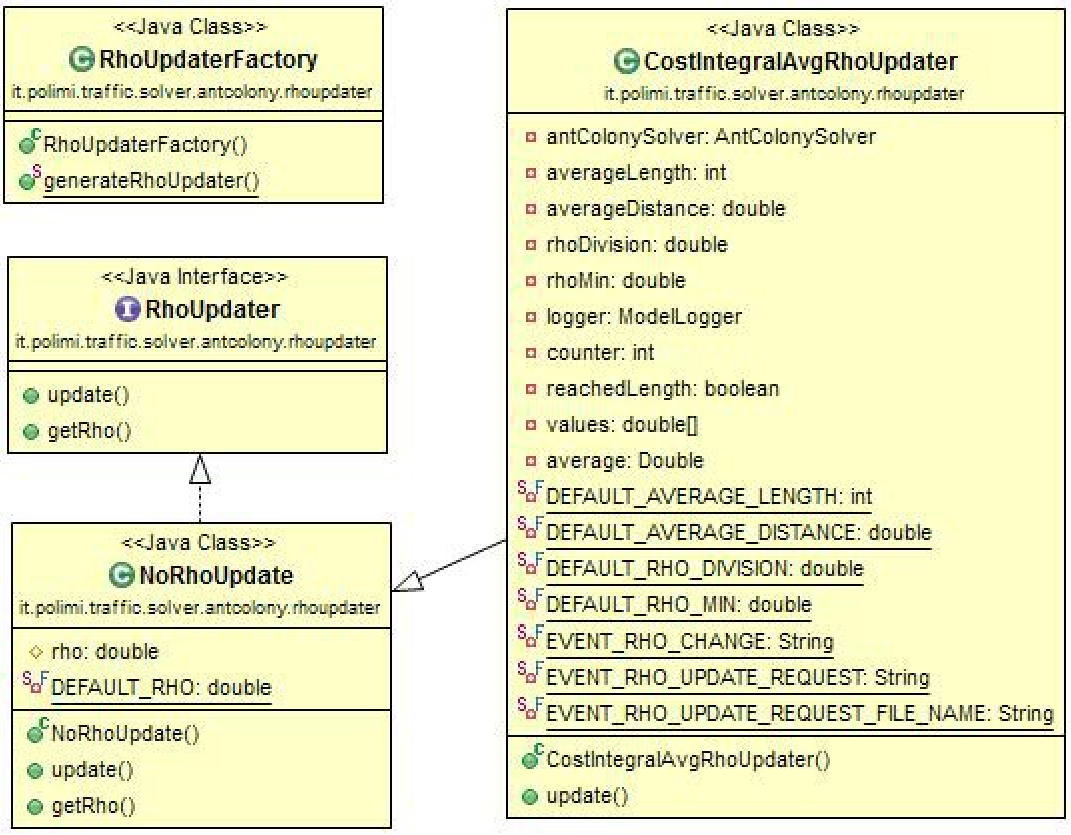
Package it.polimi.traffic.solver.chooser.blacklist



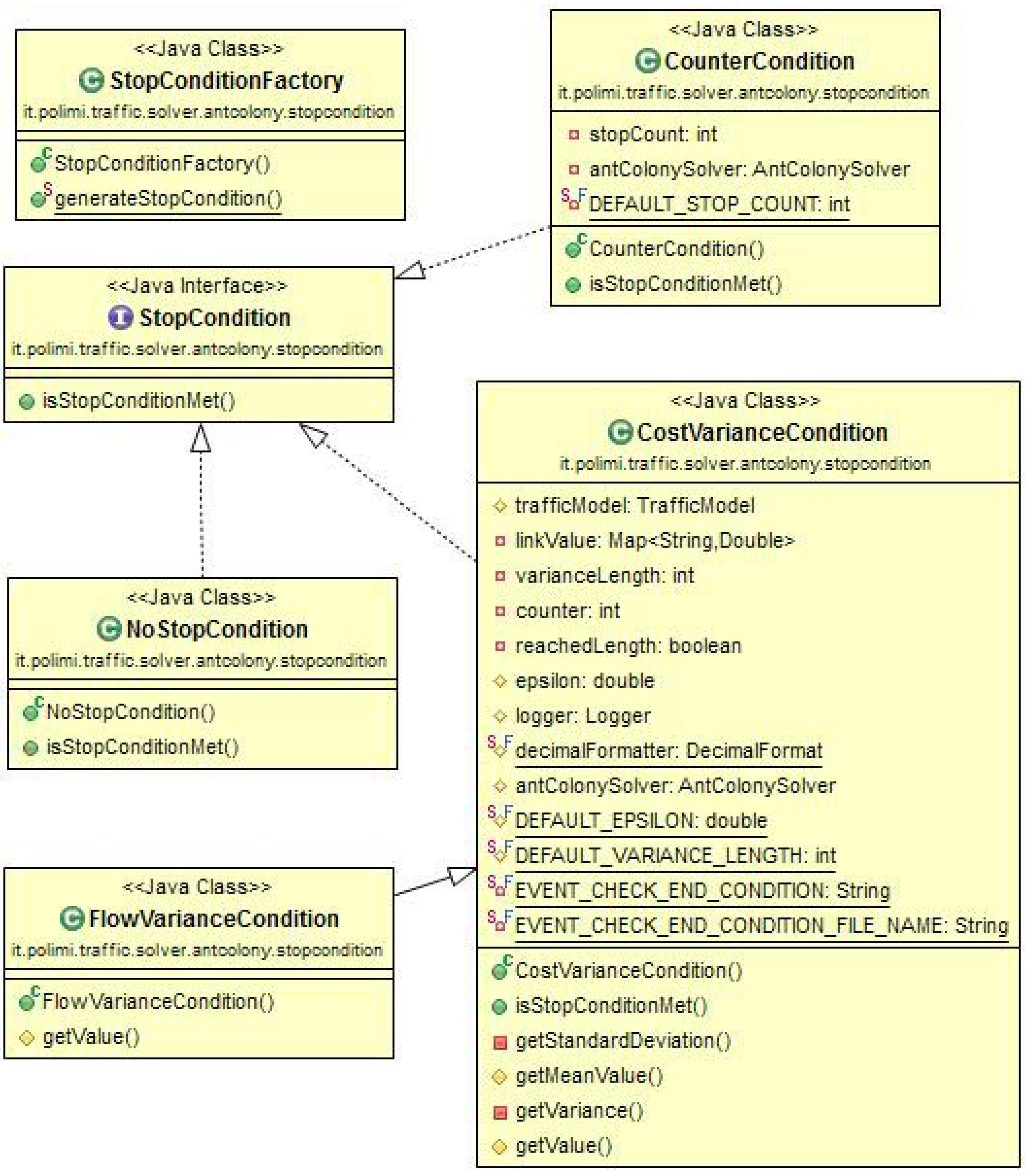
Package it.polimi.traffic.solver.distribution


<<Java Class>>
FlowDistributionSolverFactory
it.polimi.traffic.solver.antcolony.distribution
DEFAULT_FLOW_CHOOSER_TYPE: String
DEFAULT_FLOW_DISTRIBUTION_ITERATIONS: Integer
FlowDistributionSolverFactory()
generateFlowDistributionSolver()
<<Java Class>>
PheromoneDistribution
it.polimi.traffic.solver.antcolony.distribution
logger: Logger
trafficModel: TrafficModel
flowGeneratorNode: FlowGeneratorNode
totalReleasedPheromones: Double
colonyFilePath: String
parameterParser: ParameterParser
antColony: AntColony
PheromoneDistribution()
run()
<<Java Interface>>
FlowDistributionSolver
it.polimi.traffic.solver.antcolony.distribution
<<Java Class>>
AntDistribution
it.polimi.traffic.solver.antcolony.distribution
iterations: int
chooser: TrafficLinkChooser
antColony: AntColony
logger: ModelLogger
trafficModel: TrafficModel
random: Random
DEFAULT_BLOCK_COUNTER: int
DEFAULT_INTERATIONS: int
AntDistribution()
run()
<<Java Class>>
RoutesTreeDistribution
it.polimi.traffic.solver.antcolony.distribution
logger: Logger
antColony: AntColony
trafficModel: TrafficModel
RoutesTreeDistribution()
run()
getFlowTree()
distributeFlowInTree()
distributeFlowInNode()
<<Java Class>>
TreeNode
it.polimi.traffic.solver.antcolony.distribution
inFlow: Double
isLeaf: boolean
TreeNode()
TreeNode()
setInFlow()
getInFlow()
isLeaf()
<<Java Class>>
TrafficNodeWithPrevious
it.polimi.traffic.solver.antcolony.distribution
trafficNode: TrafficNode
previousTrafficNode: TrafficNode
choosenLink: TrafficLink
TrafficNodeWithPrevious()
getTrafficNode()
getPreviousTrafficNode()
getChoosenLink()
<<Java Class>>
TreeLink
it.polimi.traffic.solver.antcolony.distribution
trafficLink: TrafficLink
TreeLink()
getTrafficLink()

Package it.polimi.traffic.solver.rhoupdater


<<Java Class>>
RhoUpdaterFactory
it.polimi.traffic.solver.antcolony.rhoupdater
RhoUpdaterFactory()
generateRhoUpdater()
<<Java Interface>>
RhoUpdater
it.polimi.traffic.solver.antcolony.rhoupdater
update()
getRho()
<<Java Class>>
NoRhoUpdate
it.polimi.traffic.solver.antcolony.rhoupdater
rho: double
DEFAULT_RHO: double
NoRhoUpdate()
update()
getRho()
<<Java Class>>
CostIntegralAvgRhoUpdater
it.polimi.traffic.solver.antcolony.rhoupdater
antColonySolver: AntColonySolver
averageLength: int
averageDistance: double
rhoDivision: double
rhoMin: double
logger: ModelLogger
counter: int
reachedLength: boolean
values: double[]
average: Double
DEFAULT_AVERAGE_LENGTH: int
DEFAULT_AVERAGE_DISTANCE: double
DEFAULT_RHO_DIVISION: double
DEFAULT_RHO_MIN: double
EVENT_RHO_CHANGE: String
EVENT_RHO_UPDATE_REQUEST: String
EVENT_RHO_UPDATE_REQUEST_FILE_NAME: String
CostIntegralAvgRhoUpdater()
update()

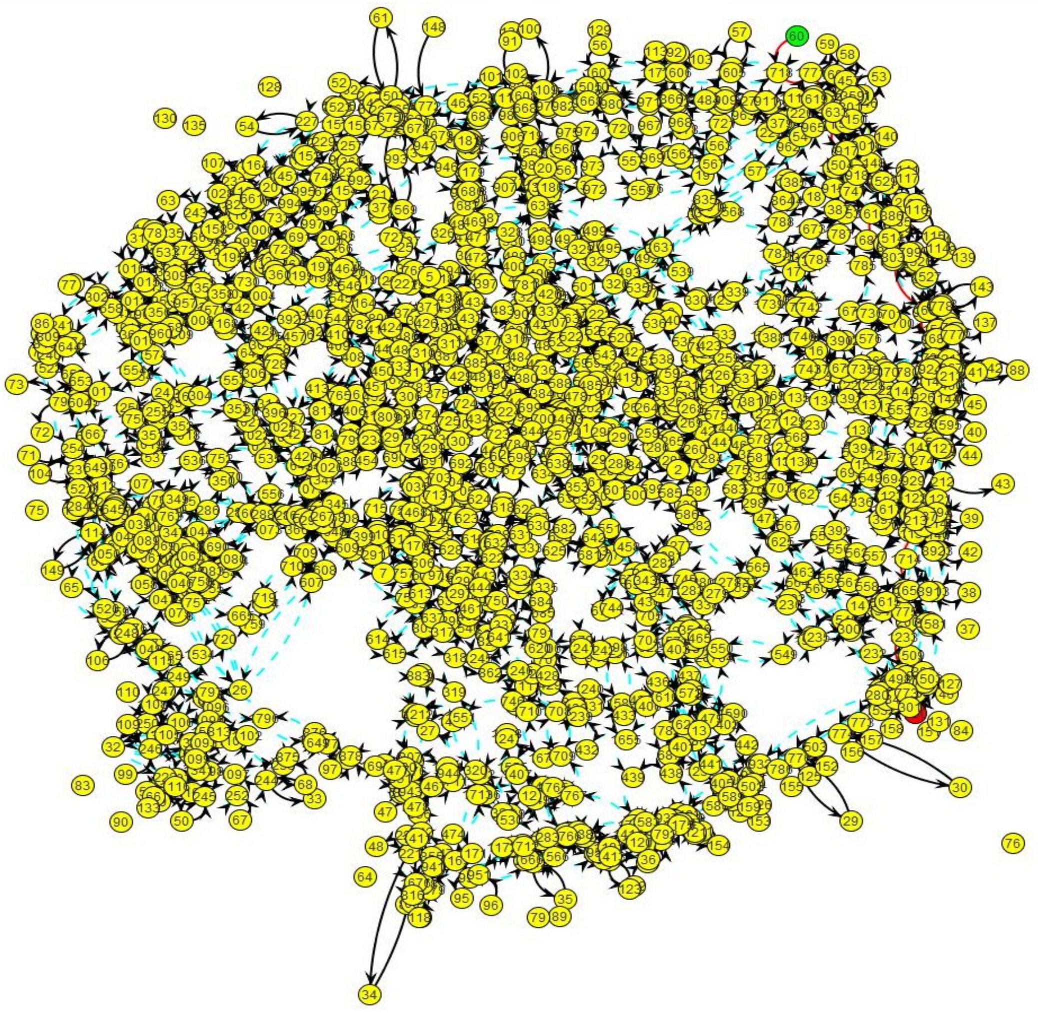
Package it.polimi.traffic.solver.stopcondition



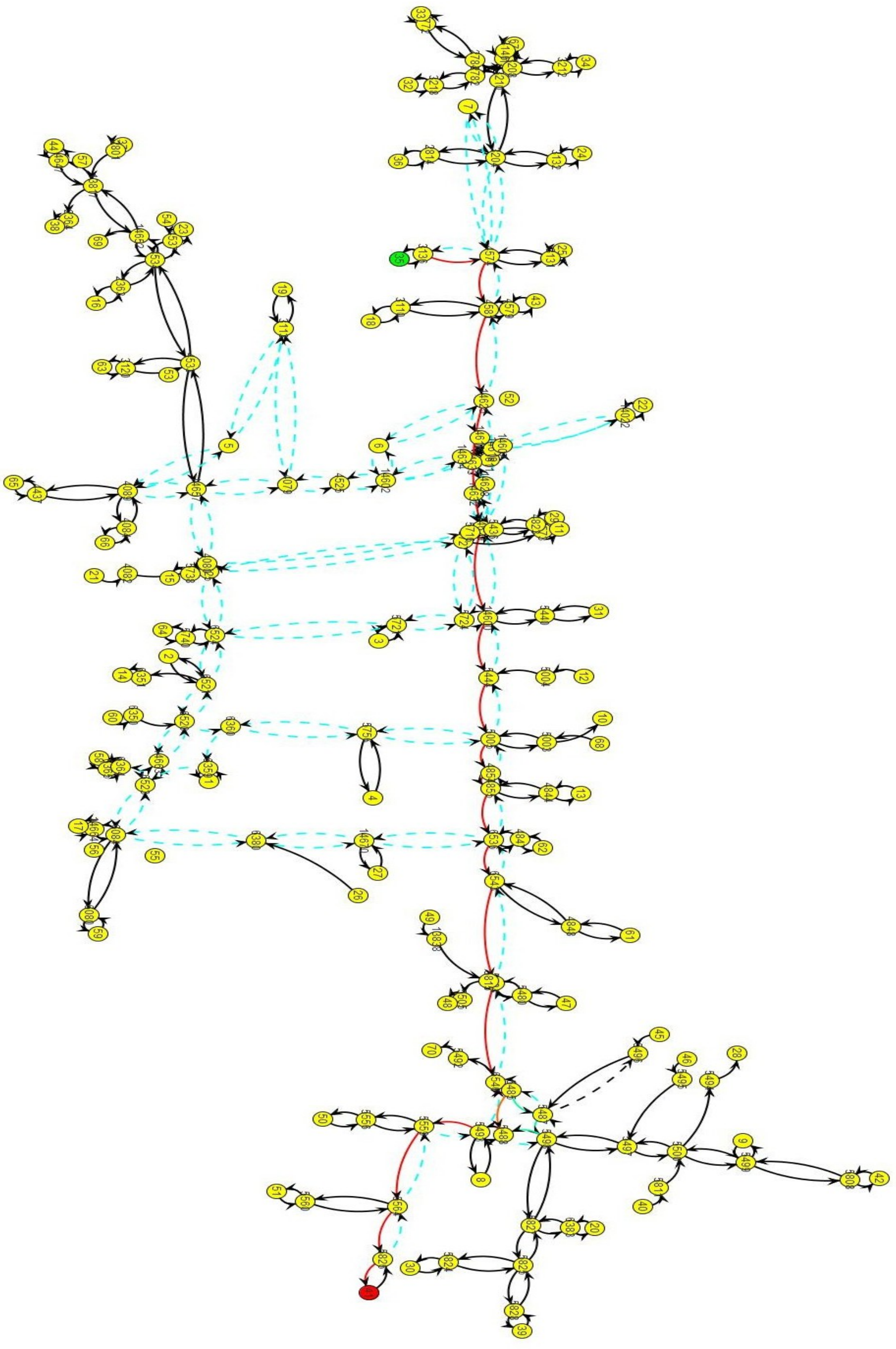
Appendix C

Tested networks images

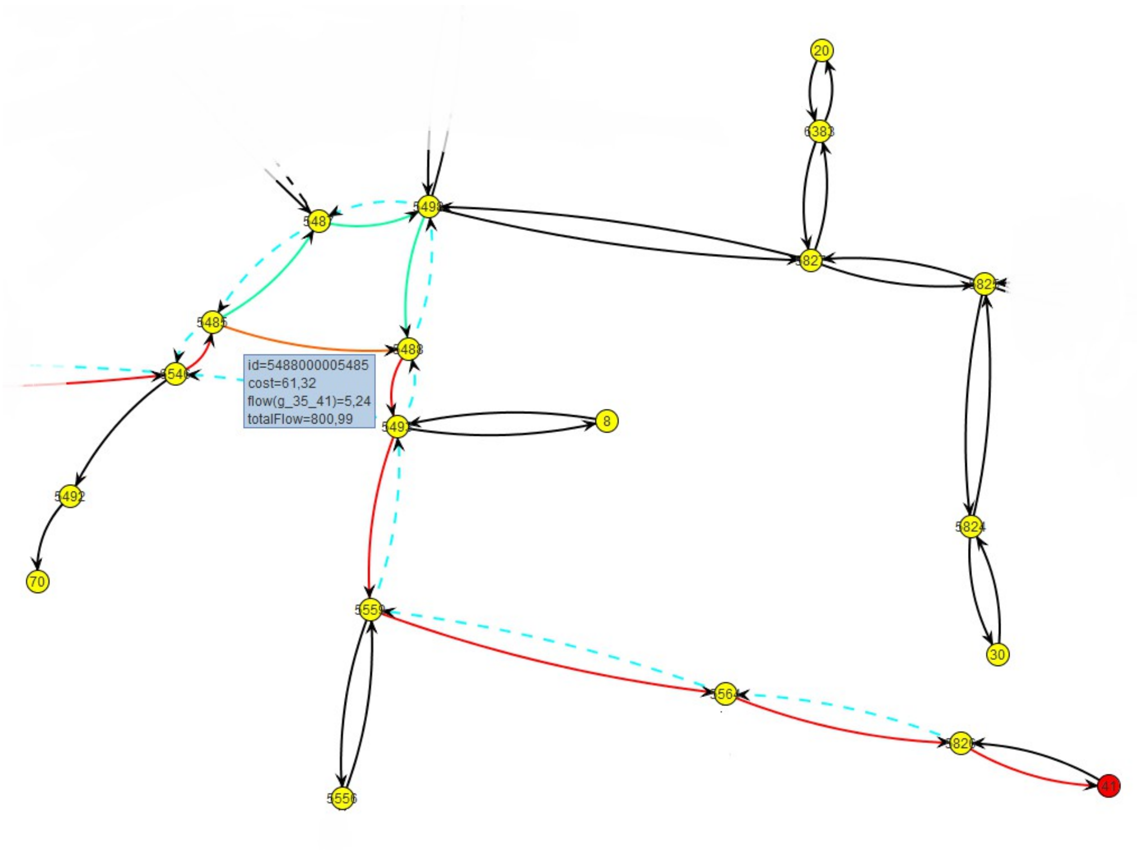
Transportation network of area Bastioni in Milan



Transportation network of area Maggi in Milan

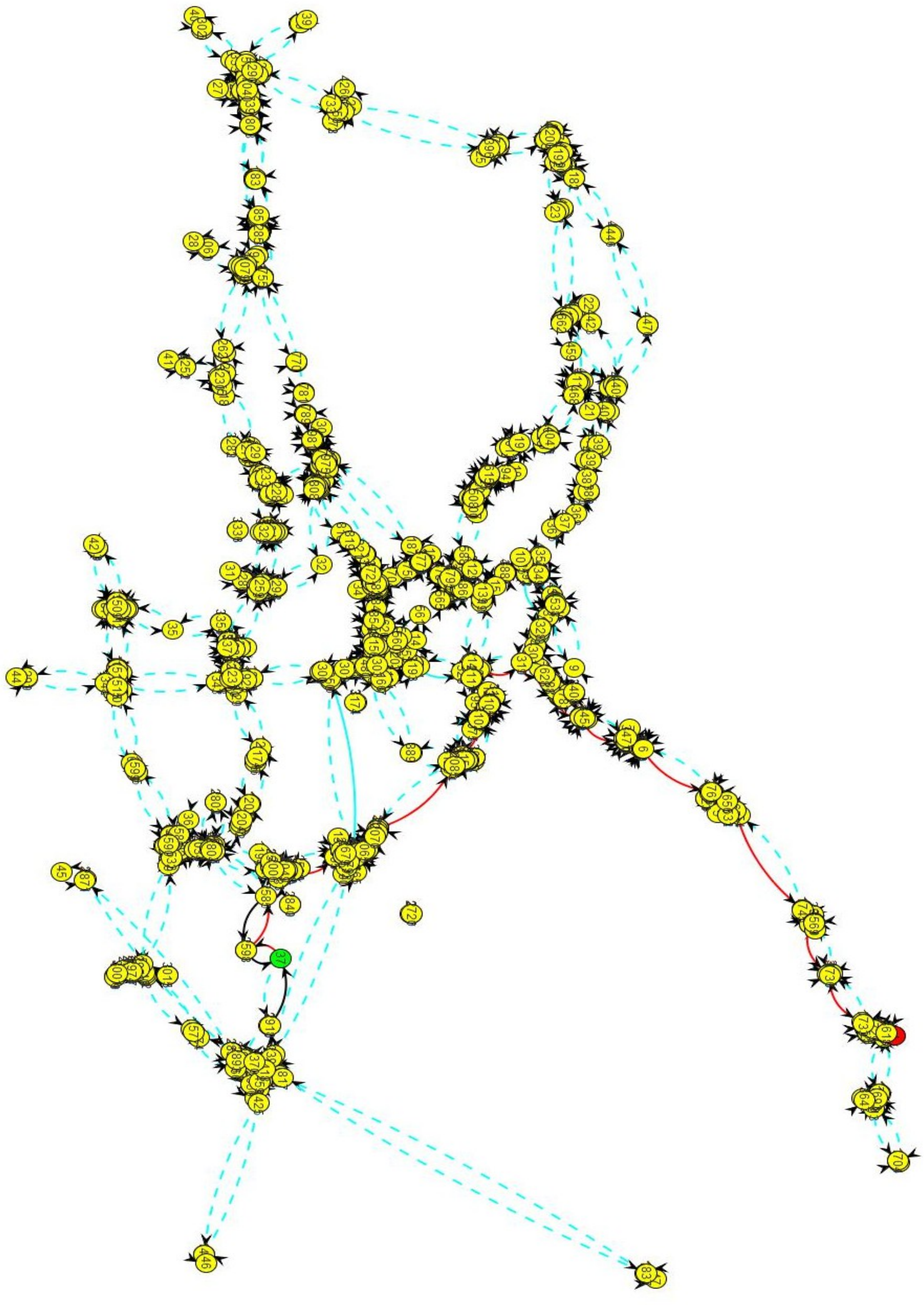


A detail of the same network, showing how part of the traffic flow takes a detour to avoid passing on a congested link with a high cost.

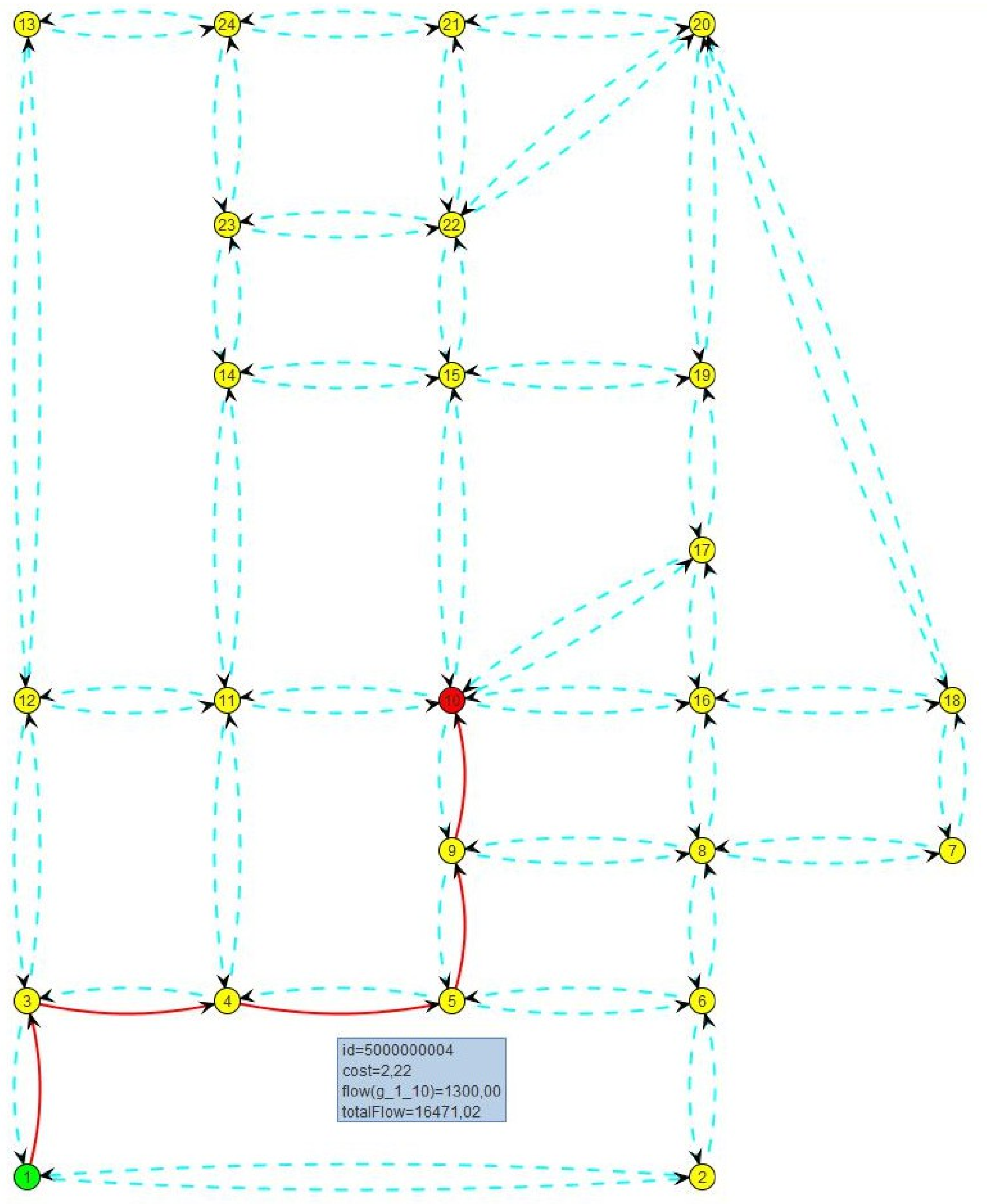


```
id=5488000005485
cost=61,32
flow(g_35_41)=5,24
totalFlow=800,99
```

Transportation network of Naples interurban area

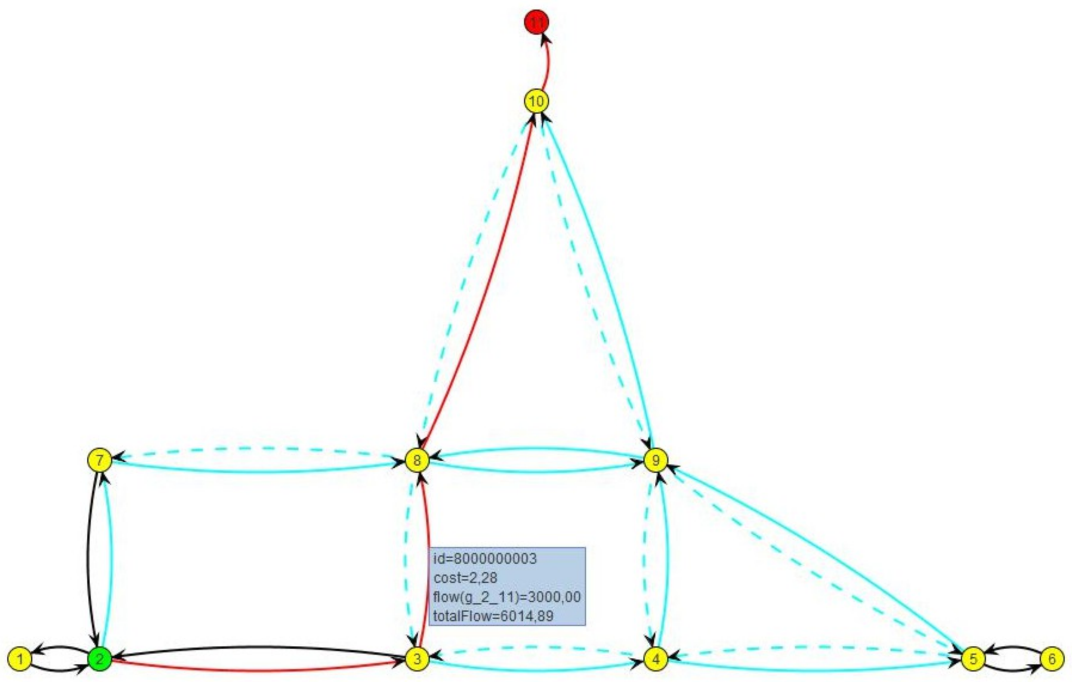


Transportation network of Sioux-falls



id=5000000004
cost=2,22
flow(g_1_10)=1300,00
totalFlow=16471,02

“Not separable costs” trial network



id=8000000003
cost=2,28
flow(g_2_11)=3000,00
totalFlow=6014,89

Bibliography and links

[1] Frank Knight. *Some Fallacies in the Interpretation of Social Cost*, 1924.
[2] John Glen Wardrop et al. *Correspondence. Some Theoretical Aspects of Road Traffic Research*, 1952.
[3] Beckman et. al. *Studies in the economics of transportation*, 1968.
[4] Dietrich Braess. *Uber ein Paradoxon aus der Verkehrsplanung*, 1968.
[5] Springer and Verlag. *Graphentheoretische Methoden und ihre Anwendungen*, 1969.
[6] New York Times. «What if They Closed 42d Street and Nobody Noticed?», 1990. http://www.nytimes.com/1990/12/25/health/what-if-they-closed-42d-street-and-nobody-noticed.html.
[7] McFadden et al. *The urban travel demand forecasting project*, 1979.
[8] Cascetta. *Modello di interazione domanda/offerta*, 2007.
[9] Frank Marguerite and Wolfe Philip. *An algorithm for quadratic programming*, 1956.
[10] Gerardo Beni. *From Swarm Intelligence to Swarm Robotics*, 2005.
[11] J. G. Kohl. *Der verkehr und die ansiedelungen der menschen in ihrer abh¨angigkeit von der gestaltung der erdoberfl¨ache*, 1841.
[12] T. Larsson and M. Patriksson. «Side constrained traffic equilibrium models—analysis, computation and applications», 1999.
[13] M. J. Beckmann, C. B. McGuire, and C. B. Winsten. *Studies in the Economics of Transportation.* 1956.
[14] M. Florian. «Untangling traffic congestion: Application of network equilibrium models in transportation planning» (1999).
[15] Y. Sheffi. *Urban Transportation Networks.* Prentice-Hall, Englewood, NJ, 1985.
[16] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* 1993.
[17] A. Charnes and W. W. Cooper. *Multicopy traffic network models*, 1959.
[18] A. Haurie and P. Marcotte. *On the relationship between Nash-Cournot and Wardrop equilibria*, 1985.
[19] A. de Palma and Y. Nesterov. *Optimization formulations and static equilibrium in congested transportation networks.* CORE Discussion Paper 9861, Universit´e Catholique de Louvain, Louvain-la-Neuve, Belgium, 1998.
[20] P. Marcotte and M. Patriksson. *Traffic equilibrium.* In C. Barnhart and G. Laporte, editors, Transportation, volume 14 of Handbooks in Operations Research and Management Science, chapter 10, pages 623–713. Elsevier, New York, NY, 2007.
[21] E. Koutsoupias and C. H. Papadimitriou. *Worst-case equilibria.* In C. Meinel and S. Tison, editors, Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science (STACS), volume 1563 of Lecture Notes in Computer Science, pages 404–413, Trier, Germany, March 1999. Springer, Heidelberg.

[22] C. H. Papadimitriou. *Algorithms, games, and the Internet.* In Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC), pages 749–753, Hersonissos, Greece, 2001. ACM Press, New York, NY.
[23] M. Dorigo, *Optimization, Learning and Natural Algorithms*, PhD thesis, Politecnico di Milano, Italy, 1992.
[24] R. B. Dial. *A probabilistic multi-path traffic assignment algorithm which obviates path enumeration.* Transportation Research, 5(2):83–111, 1971.
[25] Y. Sheffi and W. Powell. *An algorithm for the traffic assignment problem with random link costs.* Networks, 12:191–207, 1982.
[26] M. Trahan. *Probabilistic assignment: An algorithm*. Transportation Science, 8(4):311–320, 1974.
[27] C. Fisk. *Some developments in equilibrium traffic assignment.* Transportation Research, 14B(3):243–255, 1980.
[28] C. Daganzo and Y. Sheffi. *On stochastic models of traffic assignment.* Transportation Science, 11(3):253–274, 1977.
[29] S. Daferemos. *Traffic equilibrium and variational inequalities.* Transport. Sci 14, 1980.
[30] R. Dial. *T2: Another multipath probabilistic traffic assignment model that obviates path enumeration.* Transportation Research B, 1995.
[31] B. N. Jason. *Dynamic traffic assignment for urban road networks.* Transportation Research, 1991.
[32] Fisk, C. S. *Some developments in equilibrium traffic assignment.* Transportation Research B, 14B 243-255, 1980.
[33] H. Bar-Gera. *Origin-based algorithm for the traffic assignment problem.* Transportation Science, 36(4):398–417, 2002.
[34] L. J. LeBlanc, R. V. Helgason, and D. E. Boyce. *Improved efficiency of the Frank-Wolfe algorithm for convex network programs*. Transportation Science, 19:445–462, 1985.
[35] A. Colorni, M. Dorigo, and V. Maniezzo, *Distributed optimization by ant colonies.* Proceedings of ECAL'91, European Conference on Artificial Life, Elsevier Publishing, Amsterdam, 1991.
[36] M. Dorigo, V. Maniezzo, and A. Colorni, *The ant system: an autocatalytic optimizing process.* Technical Report TR91-016, Politecnico di Milano, 1991.
[37] Marsh, L. & Onof, C. *Stigmergic epistemology, stigmergic cognition.* Cognitive Systems Research, 2007.
[38] M. Matteucci, L. Mussone., M. Ponzi. *Ant colony optimization for stochastic user equilibrium*. In Bifulco G (ed) I sistemi stradali di trasporto nella società dell'informazione: monitoraggio, simulazione e predisposizione di basi informative dinamiche, Aracne, Rome, pp 175–199, 2005.
[39] M. Matteucci, L. Mussone. *Ant colony optimization technique for equilibrium assignment in congested transportation networks.* In Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation. ACM Press, New York, NY, pp 87–88, 2006.

[40] L. D'Acierno, B. Montella, F. De Lucia. *A stochastic traffic assignment algorithm based on ant colony optimisation*. In: Lecture Notes in Computer Science, vol 4150, Springer, pp 25–36, 2006.
[41] E. Cascetta. *Transportation Systems Engineering: Theory and Methods.* No. 49 in Applied Optimization. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001.
[42] G. Cantarella. *A general fixed-point approach to multimodal multi-user equilibrium assignment with elastic demand*. Transport Science 31:107–128, 1997.
[43] «Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine». Last viewed on 30 March 2013. http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html.
[44] «Java SE 6 HotSpot[tm] Virtual Machine Garbage Collection Tuning». Accessed March 30, 2013. http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html.
[45] «Aimsun». Accessed March 30, 2013. http://www.aimsun.com/wp/.
[46] «Cube | Citilabs». Accessed March 30, 2013. http://www.citilabs.com/products/cube.
[47] Taylor, NicholasB. «The CONTRAM Dynamic Traffic Assignment Model». *Networks and Spatial Economics* 3, n. 3 (1 September 2003): 297–322.
[48] Ben-Akiva, Moshe, HarisN. Koutsopoulos, Constantinos Antoniou, and Ramachandran Balakrishna. "Traffic Simulation with DynaMIT." In *Fundamentals of Traffic Simulation*, edited by Jaume Barceló, 145:363–398. International Series in Operations Research & Management Science. Springer New York, 2010.
[49] "DYNASMART." Accessed March 30, 2013. http://www.its.uci.edu/~paramics/sim_models/dynasmart.html.
[50] "Emme Transportation Forecasting Software." Accessed March 30, 2013. http://www.inrosoftware.com/en/products/emme/index.php.
[51] "Quadstone Paramics | Traffic and Pedestrian Simulation, Analysis and Design Software." Accessed March 30, 2013. http://www.paramics-online.com/.
[52] "TransCAD Transportation Planning Software." Accessed March 30, 2013. http://www.caliper.com/tcovu.htm.
[53] "Transims - an Open Source Transportation Modeling and Simulation Toolbox - Google Project Hosting." Accessed March 30, 2013. http://code.google.com/p/transims/.
[54] "TSIS: Traffic Software Integrated System." Accessed March 30, 2013. http://www-mctrans.ce.ufl.edu/featured/TSIS/Version6/.
[55] "Atkins-ITS SATURN." Accessed March 30, 2013. http://www.saturnsoftware.co.uk/.
[56] "Vision Traffic - PTV Group." Accessed March 30, 2013. http://vision-traffic.ptvgroup.com/en-uk/.
[57] "Traffic Assignment Vista Transport Group." Accessed March 30, 2013. http://vistatransport.com/products/traffic-assignment/.
[58] LeBlanc LJ, Morlok EK, Pierskalla WP, *An efficient approach to solving the road network equilibrium traffic assignment problem."*Transportation Research 9(1):309–318, 1975

[59] G. Bifulco. *I sistemi stradali di trasporto nella societ`a dell'informazione: monitoraggio, simulazione e predisposizione di basi informative dinamiche.* Aracne, Rome, Italy (2005)