A DOMAIN-SPECIFIC LANGUAGE FOR ROUTING PROBLEMS

Benjamin Hoffmann, Michael Guckert,
Thomas Farrenkopf
KITE - Kompetenzzentrum für
Informationstechnologie
Technische Hochschule Mittelhessen, Germany
{benjamin.hoffmann,michael.guckert,thomas.farrenkopf}
@mnd.thm.de

Kevin Chalmers,
Neil Urquhart
School of Computing
Edinburgh Napier University, Scotland
{k.chalmers,n.urquhart}
@napier.ac.uk

KEYWORDS

Domain-specific language, Agent-based modelling, Ant algorithms, Dynamic TSP.

ABSTRACT

Vehicle Routing Problems (VRPs) are commonly used as benchmark optimisation problems and they also have many applications in industry. Using agent-based approaches to solve VRPs allows the analysis of dynamic VRP instances that incorporate congestion effects. By using a domain-specific language as part of a model-driven approach, routing problems can be modelled in an abstract form that does not contain implementation and other technical details. With such a tool domain experts can concentrate on the actual modelling task without being distracted by low-level intricacies. We present the DSL Athos in which computational and platform independent routing problems can be defined. The DSL offers an efficient way to model problems with seamless integration of established optimisation methods. Generators create executable code for several agent based platforms. Proof of concept is given by applying the tools to the Oliver 30 TSP and an instance of a dynamic TSP.

INTRODUCTION

The planning and optimisation of logistics networks and other VRP problems have many industrial applications. Planning and optimisation of resources is necessary in a world in which sustainability and efficiency are important for organisations that wish to remain competitive. A typical example is where algorithms and models have to find optimal, or near-optimal, solutions for vehicles in delivery processes. Creating a schedule in which all customers are visited in an efficient order means solving a Travelling Salesman Problem (TSP) (see Schwab, Guckert, and Willems 2017).
TSP is NP-hard, and thus heuristic approaches are commonly used to find solutions. Nature inspired techniques are such an approach (Afaq and Saini 2011). Schwab et. al also describe how a solution for TSPs with additional time constraints (see Savelsberg 1985) based on the ant algorithm was integrated into an off-the-shelf transportation management system (Schwab, Guckert, and Willems 2017).
In real-world applications, solutions should consider the current traffic situation and respect congestion effects. Adding a dynamic element increases the complexity of the problem. Such problems belong to a special class of TSPs known as Dynamic TSP (DTSP) (Cheong and White 2012). By modelling the problem by means of an agent based approach in which the behaviour of the agents can be dynamically adjusted to reflect the current level of traffic congestion, it is possible to analyse and optimise instances of the DTSP.
In this paper, we present the DSL Athos that allows domain experts to specify traffic and transport related optimisation problems in a declarative and concise way. In a program written in Athos, agents move through a network of roads. The agents in the network mutually influence the speed with which they can travel the routes of the given network. Agent behaviour can be defined in various ways. While it is possible to let agents travel a pre-defined list of nodes, they can also be assigned the task of travelling an optimised route. For finding an optimised tour (measured by distance) that visits each node of a given set of nodes the agents must solve a static TSP instance. Agents could also be instructed to optimise for the *fastest* route that visits each node in their list. Since agents increase the time it takes to pass a given road in the network by using it themselves, finding the fastest tour for a given set of nodes requires to solve a *dynamic* version of the TSP. Pillac et al. refer to this as the *evolution of information* (Victor Pillac et al. 2013): agents can only plan their tour based on the information they have at the moment the plan is created. The best they can hope for is that this information represents the current traffic situation in the network. However, a single time step later, the traffic situation has changed, and the more time passes, the more likely it is that the created tour is no longer the optimal one. Agents therefore have to recalculate and adjust their plan regularly.
We will describe Athos and then, as a proof of concept, apply it to the published Oliver30 TSP based on the problem published by Hopfield and Tank (Hopfield and Tank 1987) and on an instance of a dynamic TSP which will be solved by using *Ant Colony System (ACS)* (Dorigo and Gambardella 1997).

RELATED WORK

The TSP is an NP-hard benchmark combinatorial optimisation problem. The problem requires a route to be determined for a

Proceedings 32nd European Conference on Modelling and Simulation ©ECMS Lars Nolle, Alexandra Burger, Christoph Tholen, Jens Werner, Jens Wellhausen (Editors)
ISBN: 978-0-9932440-6-3/ ISBN: 978-0-9932440-7-0 (CD)

salesman who must find the shortest tour to visit a number of cities. Each city must be visited once and once only within a tour so that the solution is a Hamiltonian circuit that starts form and ends in a designated starting point. The number of possible tours is calculated as $(n-1)!$. However if the distances between the cities are bi-directional then the number of possible tours reduces to $(n-1)!/2$.

Problems related to the TSP were discussed by William Hamilton in the 1800s, but the first published work proposing a method for solving the TSP appeared in 1954 (Dantzig, Fulkerson, and Johnson 1954). Subsequently, Chvátal (Chvátal et al. 2010) proposed the cutting plane method based on linear equations and solved a 49 city TSP instance. Subsequent notable methods for solving TSP instances include the 2-opt (Croes 1958), 3-opt (Shen Lin 1965) and Lin-Kernighan (S. Lin and Kernighan 1973) heuristics. Stochastic and nature inspired methods applied to the TSP include genetic algorithms Grefenstette et al. 1985 and ant colony optimisation (Dorigo and Gambardella 1997).

There exists a number of variants of the basic TSP. These include TSP with time windows (TSPW), multiple TSP (MTSP) and dynamic TSP (DTSP). TSPW (Baker 1983) allocates a time window to each city. Cities may only be visited if the salesman arrives within the respective time window. In the MTSP (Laporte and Nobert 1980), with multiple start/end points, the solution has to contain multiple Hamiltonian circuits. The DTSP adds and removes cities at run time (Gharehchopogh, Maleki, and Khaze 2013). The challenge is not to produce one solution, but to produce a series of updated solutions in response to cities being added or deleted from the problem. Beyond that, our definition of a DTSP allows dynamic changes of the weights (i.e. length) of the edges in the network (compare Tinos 2015).

A DSL named Turn was developed by Steil et al. (Steil et al. 2011) in order to specify how vehicles should be routed with a specific real-world VRP instance. A reference to a DSL is made by Pigden et. al. in (Pigden et al. 2012), but no details are supplied. It becomes apparent that whilst some work has been undertaken in relation to the application of DSLs to VRP type problems, there remains a significant requirement of a DSL that can be adapted to a range of VRP instances.

A DSL FOR ROUTING PROBLEMS

Athos is a DSL that allows researchers to define models for traffic-related optimisation problems at a computational and platform independent level. Domain experts can take a declarative approach, instead of imperatively coding agent behaviours. A convention over configuration approach is taken that allows, but does not require, users to control certain aspects of the modelled simulation. For many aspects of the simulation, Athos assumes reasonable default values and leaves it open to the language users to override these defaults with their desired values.

The problems that we wish to simulate comprise agents that must try to optimise routes within a network of roads. Each road (edge) in the network has a capacity attribute that determines the extent to which the road is affected by congestion. These congestion effects are also dependent on the number and types of agents on the road. Some agents have a greater congestion effect on roads than others. For example, an agent that represents a slow-moving large tractor is far more likely to congest a low-capacity road than an agent that represents a small motorcycle. Agents enter the network from arbitrary nodes, meaning that any type of vehicle may potentially enter from any node. It is possible to define distribution functions that represent the probability with which a given place in the network is the origin of a given type of agent. These distributions can be calibrated with data taken from empirical observations. Agents may be specified to differ in their travelling behaviour, e.g. some agents start at a given node and seek to reach a given destination node, visiting a given list of nodes en route. Other agents exhibit a circling behaviour. They repeatedly travel along a given route within the network. Similar to circling agents are shuttle agents which shuttle along a given route of points. The agents with circling and shuttling behaviour are used to create noise and traffic in the network thus simulating high rates of traffic.

Architecture of Athos' generator

The architecture of the generator used to transform Platform-Independent Models (PIMs) into Platform-Specific Models (PSMs) is depicted in Figure 1. The Athos-generator comes in the form of an Eclipse plug-in. Once the generator is given a program written in Athos, it uses a set of templates to generate a traffic model for the NetLogo platform. At this point, it is important to note that Athos is not a mere interface to facilitate the creation of NetLogo models. It is also possible to generate models for other ABM platforms like Repast (a version with different and reduced functionality for Repast Simphony has already been implemented) or Jadex. Together with the generator and its templates, we have developed a NetLogo extension that can be accessed from the generated NetLogo models via the NetLogo-Extension-API. This extension contains several optimisation algorithms for graph structures (e.g. Dijkstra's algorithm) by using available frameworks (e.g. JUNG http://jung.sourceforge.net/).

The extension also features a meta-heuristic known as *Ant Colony System (ACS)* (Dorigo and Gambardella 1997). We have implemented ACS as a NetLogo extension so that agents in the network can find solutions of sufficient quality for the given TSP. ACS is one out of many other possible approaches to solve TSPs heuristically. It was chosen to demonstrate how the philosophy of our architecture can integrate optimisation algorithms. As Athos matures we will possibly implement other algorithms if that widens the applicability. According to Athos' philosophy of providing sensible defaults, if not specified differently, ACS parameter values are those obtained from Dorigo and Gambardella 1997 (exception: $t_0 = 10$).

The ACS implementation

From a theoretical point of view, solving a TSP means finding a Hamiltonian cycle of shortest length (see e.g. Laporte and Osman 1995). As is noted by Laporte and Nobert (Laporte and Nobert 1987), this definition does not cause problems in

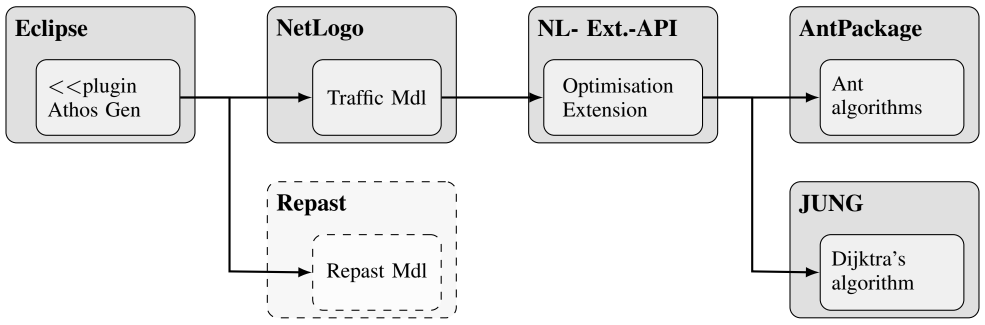


Fig. 1. Athos' modelling approach

complete graphs in which the triangle inequality holds (i.e. in which there is no shorter path between any two nodes in the network than their direct link). Athos does not require the underlying graph to be complete, because real world traffic networks are often incomplete. Therefore, for any two nodes in the network their distance is defined as the length of the shortest path from one to the other. Furthermore, the constraint that each node is to be visited exactly once is relaxed by distinguishing between *service* visits and *crossing* visits. Each node then has to be serviced exactly once but may be crossed an arbitrary number of times.

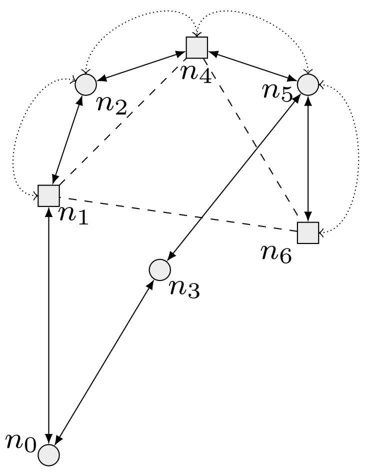


Fig. 2. Behaviour of the salesman-agent in an Athos-generated simulation. Solid lines represent edges, dashed lines represent calculated distances, dotted arcs represent the salesman's moving pattern.

In Figure 2, the graph consists of nodes $N = \{n_0, \ldots, n_6\}$. However, the actual tour of the salesman consists only of $T = \{n_1, n_4, n_6\}$ with $T \subset N$. Following Laporte and Norbert, in our ACS implementation, we first calculate the minimal distance between any two nodes of the tour by means of Dijkstra's algorithm. Take $\overline{n_1 n_6}$ as an example. In our implementation, we would use Dijkstra's to introduce an artificial edge and define $\overline{n_1 n_6} := \overline{n_1 n_2 n_4 n_5 n_6}$.

The arcs in Figure 2 then show the solution for the incomplete graph, given that the salesman starts in n_1. The salesman will first service n_4 by going from n_1 to n_2 and then to n_4. Next, he will go to n_5 and then service n_6. From there, he will cross the nodes n_5, n_4, n_2 to return to node n_1. Note that another feasible solution would have the salesman to first service node n_6 and then service node n_4 on the way back to node n_1.

Our ACS implementation works on two-dimensional arrays that contain the mutual distances between any two tour nodes and the pheromone values assigned to the edges. The implemented ACS closely follows the description described in (Dorigo and Gambardella 1997).

The ants work with two lists: one in which the indices of the tour are stored and one in which the indices of the cities yet to be visited are kept. By default, ten ants are used. The algorithm performs a given number of iterations in which the ants construct their tour incrementally beginning with an empty list for the tour. The ants are placed randomly at one of the nodes of the tour. In sequence, the ants are then asked to perform a *state transition*, i.e. add a node to their tour. In order to determine which node to service next, ants execute a *pseudo-random-proportional rule* which is a centrepiece of the ACS algorithm. Depending on the outcome the ant either chooses to *exploit* current length and pheromone information or to *explore* new links by application of a *probabilistic state transition* rule. The pheromone-value for the edge that connects an ant's current and next city is then updated *locally*.

When the list of the yet to be visited cities is empty, the other list in which the tour is stored must be complete. It is important to note that to this point the tour list of an ant only stores each node index exactly once. This means that the node in which the ant is supposed to end the tour must be added in a final step to have a complete tour. In a Hamiltonian circle, the ant must finish its tour at its starting node. However, in order to allow re-optimisation during tour execution, it must be possible to explicitly define a node where the tour is supposed to end. As an example, consider a salesman that found the tour of nodes a, b, c, d to be optimal. When the salesman enters node b, the remaining nodes consist of nodes b, c, d. If the salesman wants to re-optimise this tour, the tour still must be finished in a – and not in b which is the current location of the salesman. For this reason, we modified the algorithm in a way that allows to explicitly define a node where the tour is supposed to end. It is still possible to define an end node that is also the starting node, in which case the outcome will be a Hamiltonian cycle.

The nodes are permuted so that they start at the current location of the salesman (and not at the current location of the ant). Then, the final city is added to the list.

In our implementation, ants possess a `tourCost` attribute. A `reset` method is executed at the beginning of each iteration, and the `tourCost` of each ant is set to a negative value. In this way an ant has to calculate the cost for its tour only when the value of the `tourCost` attribute is negative, otherwise it can simply return the value of the `tourCost` attribute. In a final iteration step, the ants are sorted according to the length of their tour. Over all iterations a `globalBestTourIndices` list is stored, and at the end of each iteration a *global pheromone update* on all edges that connect nodes of these list is performed.

EXAMPLES

Oliver 30 TSP

In this section we show how Athos can be applied to the Oliver 30 TSP instance (http://stevedower.id.au/blog/research/oliver-30/).

```
model oliver30 world xmax 100 xmin 0 ymax 100 ymin 0
functions
durationFunction normal  length  default
complete network
nodes
node n0 (54.0, 67.0)  node n1 (54.0, 62.0)  node n2 (37.0, 84.0)
node n3 (41.0, 94.0)  node n4 (2.0, 99.0)   node n5 (7.0, 64.0)
node n6 (25.0, 62.0)  node n7 (22.0, 60.0)  node n8 (18.0, 54.0)
node n9 (4.0, 50.0)   node n10 (13.0, 40.0) node n11 (18.0, 40.0)
node n12 (24.0, 42.0) node n13 (25.0, 38.0) node n14 (44.0, 35.0)
node n15 (41.0, 26.0) node n16 (45.0, 21.0) node n17 (58.0, 35.0)
node n18 (62.0, 32.0) node n19 (82.0, 7.0)  node n20 (91.0, 38.0)
node n21 (83.0, 46.0) node n22 (71.0, 44.0) node n23 (64.0, 60.0)
node n24 (68.0, 58.0) node n25 (83.0, 69.0) node n26 (87.0, 76.0)
node n27 (74.0, 78.0) node n28 (71.0, 71.0) node n29 (58.0, 69.0)
edges
sources
    n0 sprouts ( congestionFactor 2.0 route (n0,n1,n2,n3,n4,n5,n6,n7,n8,n9,n10,n11,
          n12,n13,n14,n15,n16,n17,
n18,n19,n20,n21,n22,n23,n24,n25,n26,n27,n28,n29) optimise )frequency 1.0 every 1
       until 1
```

Athos is purely declarative and neither contains information about the execution platform nor the method with which the problem is going to be solved. Therefore it is computationally independent and platform independent. The Athos generator generates an executable NetLogo program, which will use Ant Colony System (ACS) to solve the problem. By modifying the generator the optimisation heuristic can be changed.

ACS is a stochastic algorithm which can produce varying results with each execution. The generated solutions tend to have a length in the range of 430 to 480. This is achieved by a default of 30 iterations, for which our implementation requires less than one second on a machine equipped with an Intel i7-6820HQ CPU running at 2.70GHz. One example route generated by our implementation has a length of 448.834. The optimal solution published on the Oliver30 website is 423.741. It is not our aim to find optimal solutions or to outperform current published solutions. Our goal is the development of a tool that allows efficient definition of problems and integrates appropriate solving methods.

Dynamic vs. static agents

Our second example shows how Athos can be used to study the implications of dynamic decision making based on complete information within in a simple road network. For this purpose, we define a network that consists of nine nodes and eleven bidirectional edges between selected nodes. Within this network, there are five agents, two of which travel pre-defined paths and three who aim to find an optimal tour for a given set of nodes. In this example, the optimisation function is not defined in terms of the travelled distance but in the amount of time required to complete a tour. More precisely, the value of interest is the time it takes for the three agents to complete a total of 100 tours.

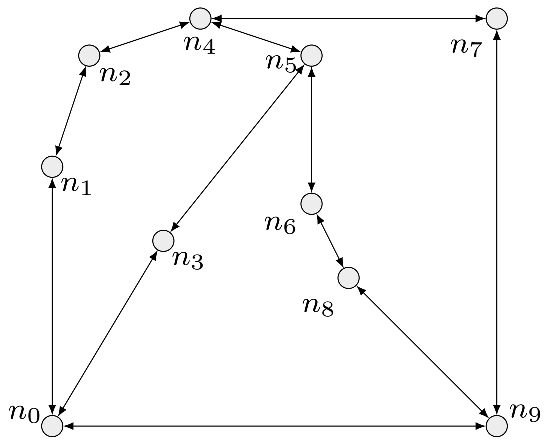


Fig. 3. Network for dynamic problem.

Figure 3 illustrates the network. Agent 1 is assigned a tour that consists of the nodes n_1, n_4, n_6. Agent 2 is to service nodes n_2, n_6, n_7, n_9 and Agent 3 services nodes n_8, n_3, n_1. The three agents aim at tour completion in a minimum amount of time. In order to intensify congestion effects, Agent 4 and Agent 5 travel the predefined route $n_4, n_5, n_3, n_0, n_1, n_2$ and $n_3, n_0, n_1, n_2, n_4, n_5$, respectively. Depending on the simulation configuration, the tour-optimising agents either exhibit a static or dynamic optimising behaviour.

In this context, static behaviour allows the agent to calculate the optimal route at the beginning of the simulation, based on the traffic situation at the moment the calculation is performed. Once a route is calculated, the agent sticks to this route throughout the entire simulation. Dynamic optimisation implies that each time an agent reaches a node within its tour it calculates a new optimised tour that comprises the unserviced nodes. Agents in this dynamic scenario only recalculate their tour when they have serviced a node. They do not recalculate when travelling through a node. If an agent has just recalculated its tour, and the next node to be serviced is not directly connected to the current node, the path to the next node to be serviced is calculated based on the data used when recalculating the tour. The agent will then follow the path and not perform any recalculation until it arrives at the node to be serviced.

TABLE I. RESULTS OF DYNAMIC TSP EXPERIMENT IN THE NETWORK. ALL EDGES IN THE NETWORK WHERE ASSIGNED THE TRAVEL DURATION FUNCTION $t = l + 2AC$ (l : LENGTH OF EDGE, AC : ACCUMULATED CONGESTION FACTOR). OPTIMAL TOURS WERE CALCULATED BY MEANS OF THE ACS ALGORITHM WITH THE FOLLOWING PARAMETERS $\alpha = 0.1, \rho = 0.1, q_0 = 0.9, \beta = 2.0$, 10 ANTS WERE USED IN 30 ITERATIONS.

Agent	Agent 1		Agent 2		Agent 3		Agent 4		Agent 5		
Mode	Optimising		Optimising		Optimising		Static		Static		
Tour	n_1, n_4, n_6		n_2, n_6, n_7, n_9		n_8, n_3, n_1		$n_4, n_5, n_3, n_0, n_1, n_2$		$n_3, n_0, n_1, n_2, n_4, n_5$		
Config	dyn?	CF	dyn?	CF	dyn?	CF	dyn?	CF	dyn?	CF	Avg. time 100 tours
C_1	no	50	no	50	no	50	no	50	no	50	117,783.4
C_2	yes	50	yes	50	yes	50	no	50	no	50	109,593.0
C_3	no	250	no	250	no	250	no	50	no	50	589,651.4
C_4	yes	250	yes	250	yes	250	no	50	no	50	555,630.9
C_5	no	1250	no	1250	no	1250	no	50	no	50	2,706,275.7
C_6	yes	1250	yes	1250	yes	1250	no	50	no	50	2,691,348.8

The listing below demonstrates how the intended scenario is modelled in Athos. For Agents 1 – 3 the model only uses the keyword `optimise` – it does not go into any computational detail as to *how* this optimisation is to be achieved. Also note how each `edge` is associated with a `durationFunction` that defines the time it takes to travel an edge in terms of its `length`, congestion factor (`cfactor`) and its current traffic (accumulated congestion factor of all agents on the respective road `accCongestionFactor`). If the `optimise` keyword is directly followed by the `dynamic` keyword, the respective agent behaves in the dynamic manner described above.

```
model incomplete world xmax 30 xmin 0 ymax 30 ymin 0
functions
durationFunction normal  length + cfactor * accCongestionFactor default
network
nodes
node n0 (1.0, 1.0) node n1 (1.0, 8.0) node n2 (2.0, 11.0)
node n3 (4.0, 6.0) node n4 (5.0, 12.0) node n5 (8.0, 11.0)
node n6 (8.0, 7.0) node n7 (13.0, 12.0) node n8 (9.0, 5.0)
node n9 (13.0, 1.0)
edges
edge undirected e0 from n0 to n1 length 0.0 cfactor 2.0 function normal
edge undirected e1 from n1 to n2 length 0.0 cfactor 2.0 function normal
edge undirected e2 from n2 to n4 length 0.0 cfactor 2.0  function normal
edge undirected e3 from n4 to n5 length 0.0 cfactor 2.0  function normal
edge undirected e4 from n5 to n3 length 0.0 cfactor 4.0 function normal
edge undirected e5 from n6 to n5 length 0.0 cfactor 2.0 function normal
edge undirected e6 from n3 to n0 length 0.0 cfactor 4.0 function normal
edge undirected e7 from n7 to n4 length 0.0 cfactor 2.0 function normal
edge undirected e8 from n7 to n9 length 0.0 cfactor 2.0 function normal
edge undirected e9 from n9 to n0 length 0.0 cfactor 2.0 function normal
edge undirected e10 from n9 to n8 length 0.0 cfactor 2.0 function normal
edge undirected e11 from n8 to n6 length 0.0 cfactor 2.0 function normal
sources
   n1 sprouts ( congestionFactor 250.0 route (n1, n4, n6) optimise dynamic)
        frequency 1.0 every 1 until 1 // Agent 1
   n2 sprouts ( congestionFactor 250.0 route (n2, n6, n7, n9) optimise dynamic)
        frequency 1.0 every 1 until 1 // Agent 2
   n8 sprouts ( congestionFactor 250.0 route (n8, n3, n1) optimise dynamic)
        frequency 1.0 every 1 until 1  // Agent 3
   n4 sprouts ( congestionFactor 250.0 route (n4, n5, n3, n0, n1, n2) mode 1 )
        frequency 1.0 every 1 until 1    //Agent 4
   n3 sprouts ( congestionFactor 250.0 route (n3, n0, n1, n2, n4, n5) mode 1)
        frequency 1.0 every 1 until 1    //Agent 5
```

Table I shows the results of the simulation experiments. The columns of the table show the agents that were used in the simulations and their general behaviour as well as their respective route. For the experiment, six different configurations, with differing agent configurations, were executed. Ten simulations were executed for each configuration and the amount of time in simulation ticks was recorded. The tour-optimising agents had to complete a total of 100 tours. Three different congestion factor (CF) values were used for all agents: 50, 250 and 1250. For each of those values, 10 simulations were executed, in which all tour-optimisation agents behaved in a static way and ten simulations were executed in which they utilised dynamic behaviours. The right column of the table presents the mean time (measured in simulation ticks) the tour-optimising agents required to perform 100 tours.

Table I shows that in the underlying network, the agents performed better when they acted in a dynamic way. With a congestion factor of 50, the dynamic agents required approx. 6.9% less time than the static agents. Surprisingly, this value decreased to only approx. 5.7%, when we intended to increase congestion effects by altering the agents' congestion factor to 250. When the congestion factor was increased to 1,250, this saw the advantage of the dynamic agents decline to 0.55%.

The reason for the diminishing performance advantage of the dynamic agents with increased congestion factors requires further investigation. However, we decided to modify the underlying network in a way that it contained one road of increased length so that dynamic agents could avoid this road when they identified congestion on it. To this end, we created the network depicted in Figure 4. The network features an additional edge between nodes n_5 and n_7 and the undirected edges $n_0, n_3, n_3, n_5, n5, n7$ form a coherent path. The congestion factor of an agent that is on an edge of such a coherent path, is also considered in the calculations of all other edges that belong to the same coherent path. Athos allows the specification of such a coherent path of edges simply by adding the `path` keyword followed by the name of the path to any edge in a network. For this scenario, we also wanted some more influence on the underlying ant algorithm and thus chose to leave the computationally independent level. Due to this step, we were able to explicitly define the parameters of the ACS algorithm. Note how the Athos model below increases the iterations to 60.

```
model incomplete world xmax 30 xmin 0 ymax 30 ymin 0
functions
durationFunction normal  length + cfactor * accCongestionFactor default
network nodes
node n0 (1.0, 1.0) node n1 (1.0, 8.0) node n2 (2.0, 11.0)
node n3 (4.0, 6.0) node n4 (5.0, 12.0) node n5 (8.0, 11.0)
node n6 (8.0, 7.0) node n7 (13.0, 12.0) node n8 (9.0, 5.0)
node n9 (13.0, 1.0)
edges
edge undirected e0 from n0 to n1 length 0.0 cfactor 2.0 function normal
edge undirected e1 from n1 to n2 length 0.0 cfactor 2.0 function normal
edge undirected e2 from n2 to n4 length 0.0 cfactor 2.0  function normal
edge undirected e3 from n4 to n5 length 0.0 cfactor 2.0  function normal
edge undirected e4 from n5 to n3 length 0.0 cfactor 4.0 path "ab" function normal
edge undirected e5 from n6 to n5 length 0.0 cfactor 2.0 function normal
edge undirected e6 from n3 to n0 length 0.0 cfactor 4.0 path "ab" function normal
edge undirected e7 from n7 to n4 length 0.0 cfactor 2.0 function normal
edge undirected e8 from n7 to n9 length 0.0 cfactor 2.0 function normal
edge undirected e9 from n9 to n0 length 0.0 cfactor 2.0 function normal
edge undirected e10 from n9 to n8 length 0.0 cfactor 2.0 function normal
edge undirected e11 from n8 to n6 length 0.0 cfactor 2.0 function normal
edge undirected e12 from n5 to n7 length 0.0 cfactor 2.0 path "ab" function normal
sources
   n0 sprouts (congestionFactor 50.0 route (n0, n8, n4) ant dynamic ants 10 alpha
        0.1 rho 0.1 iterations 120 q0 0.9 beta 2.0) frequency 1.0 every 1 until 1
   n9 sprouts (congestionFactor 50.0 route (n9, n4, n0) ant dynamic ants 10 alpha
        0.1 rho 0.1 iterations 120 q0 0.9 beta 2.0) frequency 1.0 every 1 until 1
   n7 sprouts (congestionFactor 50.0 route (n7, n0) ant dynamic ants 10 alpha 0.1
        rho 0.1 iterations 120 q0 0.9 beta 2.0) frequency 1.0 every 1 until 1
   n4 sprouts ( congestionFactor 50.0 route (n4, n5, n3, n0, n1, n2) mode 1 )
        frequency 1.0 every 1 until 1
   n3 sprouts ( congestionFactor 50.0 route (n3, n0, n1, n2, n4, n5) mode 1)
        frequency 1.0 every 1 until 1
```

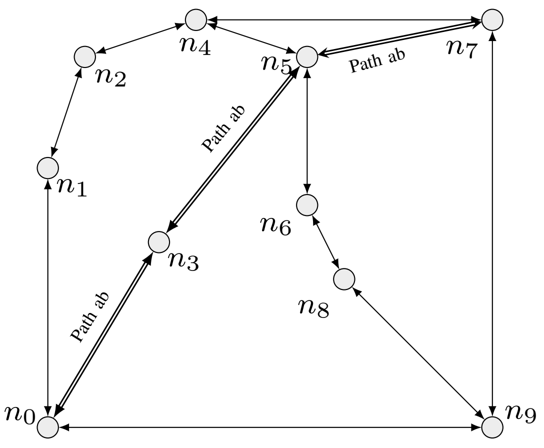


Fig. 4. Network for dynamic problem.

The right column of Table II shows the average time required by the agents for the new network. When assigned a congestion factor of 50, the static agents exhibited better performance. While the dynamic agents required an average of 99,027 ticks (over ten simulation runs), their static counterparts only required 92,928.3 ticks. However, when the congestion factor was increased to 250, the dynamic agents outperformed the static agents by 4.9%. With a congestion factor of 1.250, the performance advantage dropped to 2.6%.

The numbers indicate that dynamism alone is no guarantee for better performance. Other factors like the topology of the underlying network have to be considered. As the presented problems are highly dynamic, even an updated tour is outdated very quickly. This example showed how Athos allows for the convenient definition of dynamic traffic simulations. From these simulations further data can be derived that allow to analyse traffic and routing related problems.

CONCLUSIONS AND FUTURE WORK

We have presented the DSL Athos with which computational and platform independent descriptions of traffic simulations and routing problems can be created. These models are free of implementation details and do not explicitly specify the method to be applied to solve the problem. All necessary details are generated into the code to be executed in the target environment. We have shown how Athos can be applied to different traffic-related optimisation problems. In a first example, we have demonstrated how Athos describes and solves a popular TSP benchmark problem. In a second example, we applied Athos to analyse a traffic scenario where multiple dynamic TSP problems occurred.

Our goal is to develop a language in which models can be described more efficiently than with conventional methods and not to improve algorithms and solutions. In the current version, we generate code that implements the ACS heuristic but this can and will be extended to other approaches. There is the potential to create a more intelligent solution generator which can analyse given models and chose appropriate methods and heuristics. Doing this in the generator, rather than in a more interpretative way at runtime, results in lean best-practice implementations which can scale in size and usability by choosing the best platform and algorithmic approach for a problem. The next steps in the development of Athos are to extend the expressiveness of the language (to encompass more problem types) and intelligent features of the generator.

REFERENCES

Afaq, Hifza and Janjay Saini (2011). “On the solutions to the Travelling Salesman Problem using Nature Inspired Computing Techniques”. In: *IJCSI International Journal of Computer Science Issues* 8.2, pp. 326–334.

Baker, E K (1983). “An exact algorithm for the time-constrained travelling salesman problem”. In: *Operations Research* 31.5, pp. 938–945.

Cheong, T and C.C. White (2012). “Dynamic Traveling Salesman Problem: Value of Real-Time Traffic Information”. In: *IEEE Transactions on Intelligent Transportation Systems* 13.2, pp. 619–630.

Chvátal, Vašek et al. (2010). “Solution of a Large-Scale Traveling-Salesman Problem”. In: *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*. Ed. by Michael Jünger et al. Springer Berlin Heidelberg, pp. 7–28.

Croes, G. A. (1958). “A method for solving traveling salesman problems”. In: *Operations Research* 6, pp. 791–812.

Dantzig, George B., Delbert R. Fulkerson, and Selmer M. Johnson (1954). “Solution of a large-scale travelling-salesman problem”. In: *Technical Report P-510, RAND Corporation, Santa Monica, California, USA*.

Dorigo, M. and L. M. Gambardella (1997). “Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem”. In: *Trans. Evol. Comp* 1.1, pp. 53–66.

Gharehchopogh, F. S., I. Maleki, and S.R Khaze (2013). “A New Optimization Method for Dynamic Travelling Salesman Problem with Hybrid Ant Colony Optimization Algorithm and Particle Swarm Optimization”. In: *International Journal of Advanced Research in Computer Engineering and Technology (IJARCET)* 2.2, pp. 352–358.

TABLE II. Results of dynamic TSP experiment in the network. All edges in the network where assigned the travel duration function $t = l + 2AC$ (l : length of edge, AC : accumulated congestion factor). Optimal tours were calculated by means of the ACS algorithm with the following parameters $\alpha = 0.1, \rho = 0.1, q_0 = 0.9, \beta = 2.0$, 10 ants were used in 60 iterations.

Agent	Agent 1		Agent 2		Agent 3		Agent 4		Agent 5		
Mode	Optimising		Optimising		Optimising		Static		Static		
Tour	n_0, n_8, n_4		n_9, n_4, n_0		n_7, n_0		$n_4, n_5, n_3, n_0, n_1, n_2$		$n_3, n_0, n_1, n_2, n_4, n_5$		
Config	dyn?	CF	dyn?	CF	dyn?	CF	dyn?	CF	dyn?	CF	Avg. time 100 tours
C_1	no	50	no	50	no	50	no	50	no	50	92,928.3
C_2	yes	50	yes	50	yes	50	no	50	no	50	99,027.0
C_3	no	250	no	250	no	250	no	50	no	50	491,478.8
C_4	yes	250	yes	250	yes	250	no	50	no	50	467,292.3
C_5	no	1250	no	1250	no	1250	no	50	no	50	2,356,096.0
C_6	yes	1250	yes	1250	yes	1250	no	50	no	50	2,294,698.8

Grefenstette, John J. et al. (1985). “Genetic Algorithms for the Traveling Salesman Problem”. In: *Proceedings of the 1st International Conference on Genetic Algorithms*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., pp. 160–168.

Hopfield, J.J. and D.W. Tank (1987). “A study of permutation crossover operators on the travelling salesman problem”. In: *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pp. 224–230.

Laporte, Gilbert and Yves Nobert (1980). “A Cutting Planes Algorithm for the m-Salesmen Problem”. In: *Journal of the Operational Research Society* 31.11, pp. 1017–1023.

Laporte, Gilbert and Yves Nobert (1987). “Exact algorithms for the vehicle routing problem”. In: *North-Holland Mathematics Studies* 132, pp. 147–184.

Laporte, Gilbert and Ibrahim H. Osman (1995). “Routing problems: A bibliography”. In: *Annals of Operations Research* 61.1, pp. 227–262.

Lin, S. and B. W. Kernighan (1973). “An Effective Heuristic Algorithm for the Traveling-Salesman Problem”. In: *Operations Research* 21.2, pp. 498–516.

Lin, Shen (1965). “Computer Solutions of the Traveling Salesman Problem”. In: *Bell System Technical Journal* 44.10, pp. 2245–2269.

Pigden, Tim et al. (2012). “VeRoLog (Vehicle Routing and Logistics Conference)”. In: *EURO Working Group on Vehicle Routing and Logistics Optimisation.*

Savelsberg, M.W.P. (1985). “Local search in routing problems with time windows”. In: *Annals of Operations-Research* 4.1, pp. 285–305.

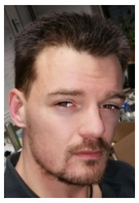
Schwab, J., M. Guckert, and M. Willems (2017). “Tourenoptimierung”. In: *Prozesse, Technologie, Anwendungen, Systeme und Management 2017: Angewandte Forschung in der Wirtschaftsinformatik*. Ed. by Thomas Barton. Heide, Deutschland: Mana Buch, pp. 1–33.

Steil, D. A. et al. (2011). “Patrol Routing Expression, Execution, Evaluation, and Engagement”. In: *IEEE Transactions on Intelligent Transportation Systems* 12.1, pp. 58–72.

Tinos, R. (2015). “Analysis of the dynamic traveling salesman problem with weight changes”. In: *Proceedings of the 2015 Latin America Congress on Computational Intelligence (LA-CCI).*

Victor Pillac et al. (2013). “A review of dynamic vehicle routing problems”. In: *European Journal of Operational Research* 225.1, pp. 1–11.

AUTHOR BIOGRAPHIES



BENJAMIN HOFFMANN is a research assistant at Technische Hochschule Mittelhessen in Friedberg from which he also received his master’s degree. He is also a PhD student at Edinburgh Napier University. His research activities are in domain-specific languages, model-driven software development and optimisation problems.

MICHAEL GUCKERT is a Professor of Applied Informatics at Technische Hochschule Mittelhessen and head of KITE - AACC (Kompetenzzentrum für Informationstechnologie - Advanced Analytics Cognitive Computing) . He received a degree in Mathematics from Justus Liebig University Giessen and a PhD in Computer Science from Philipps University Marburg. His research areas are multi agent systems, model driven software development and applications of artificial intelligence.

THOMAS FARRENKOPF is a lecturer at Technische Hochschule Mittelhessen, Friederg. He completed a PhD in the use of software agents and ontologies for business simulation at School of Computing, Edinburgh Napier University. Alongside his research activities, he currently teaches modules on software engineering and algorithms.

KEVIN CHALMERS is a senior lecturer at Edinburgh Napier University where he leads the Computer Science and Software Engineering subject area. He gained his PhD from Edinburgh Napier University in 2009, examining the application of mobile concurrency models to ubiquitous computing. His research is focused primarily on concurrency and parallelism and how different technologies can support this.

NEIL URQUHART is a lecturer in Computing Science at Edinburgh Napier University where he is Programme Leader for the Computing Science. He gained is PhD from Edinburgh Napier University in 2002, writing a thesis examining the use of Software Agents and Evolutionary Algorithms to solve a real-world routing optimisation problem. His research interests include Evolutionary Computation and Agent-based Systems and their application to real-world optimisation problems