Genetic Improvement of Routing Protocols for Delay Tolerant Networks

MICHELA LORANDI, LEONARDO LUCIO CUSTODE, and GIOVANNI IACCA,
Department of Information Engineering and Computer Science, University of Trento

Routing plays a fundamental role in network applications, but it is especially challenging in Delay Tolerant Networks (DTNs). These are a kind of mobile ad hoc networks made of e.g. (possibly, unmanned) vehicles and humans where, despite a lack of continuous connectivity, data must be transmitted while the network conditions change due to the nodes' mobility. In these contexts, routing is NP-hard and is usually solved by heuristic "store and forward" replication-based approaches, where multiple copies of the same message are moved and stored across nodes in the hope that at least one will reach its destination. Still, the existing routing protocols produce relatively low delivery probabilities. Here, we genetically improve two routing protocols widely adopted in DTNs, namely Epidemic and PRoPHET, in the attempt to optimize their delivery probability. First, we dissect them into their fundamental components, i.e., functionalities such as checking if a node can transfer data, or sending messages to all connections. Then, we apply Genetic Improvement (GI) to manipulate these components as terminal nodes of evolving trees. We apply this methodology, in silico, to six test cases of urban networks made of hundreds of nodes, and find that GI produces consistent gains in delivery probability in four cases. We then verify if this improvement entails a worsening of other relevant network metrics, such as latency and buffer time. Finally, we compare the logics of the best evolved protocols with those of the baseline protocols, and we discuss the generalizability of the results across test cases.

CCS Concepts: • **Networks** → *Routing protocols*; *Ad hoc networks*; • **Software and its engineering** → **Genetic programming**.

Additional Key Words and Phrases: ad hoc network, delay tolerant networks, epidemic routing, PRoPHET, genetic improvement, genetic programming

ACM Reference Format:
Michela Lorandi, Leonardo Lucio Custode, and Giovanni Iacca. 2020. Genetic Improvement of Routing Protocols for Delay Tolerant Networks. *ACM Trans. Evol. Learn.* 1, 1, Article xxx (March 2020), 39 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A fundamental element in many modern applications is the use of networked systems: be it environment monitoring, smart industries, smart cities, or distribution systems, networks of various scales and complexity are employed today practically everywhere. One of the most important aspects in networking is the concept of *network protocol*, i.e., a set of well-defined data format and rules that allow nodes in a network to communicate with each other [Holzmann, Gerard J 1991]. Typically, a physical network relies on multiple protocols, which are arranged as a *protocol stack* where protocols at lower layers provide basic functionalities which are progressively enriched by

Authors' address: Michela Lorandi, michela.lorandi@studenti.unitn.it; Leonardo Lucio Custode, leonardo.custode@unitn.it; Giovanni Iacca, giovanni.iacca@unitn.it,
Department of Information Engineering and Computer Science, University of Trento, Via Sommarive 9, Povo, Italy, 38123.

2688-3007/2020/3-ARTxxx $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the higher layer protocols. Among those basic functionalities, routing is a crucial one, as it allows data to flow across the network and reach their destination. While well-established efficient routing protocols exist for IP networks, routing in mobile ad hoc networks (MANETs) and other Internet of Things (IoT) instances, such as networks of cars, unmanned vehicles, or collectives of vehicles and humans, is still a very active area of research. One particularly challenging kind of MANETs is represented by the *delay tolerant networks* (DTNs), also known as *disruption tolerant networks*, *opportunistic networks* or *intermittently connected wireless networks*. Originally designed in the 1970s for space communications, but applied since the early 2000s also to terrestrial applications such as urban mobile networks, DTNs are (typically, heterogeneous) decentralized networks that lack continuous connectivity due to their node sparsity, limited wireless radio access, and limited energy resources.

In these contexts, routing is NP-hard [Balasubramanian, Aruna and Levine, Brian and Venkataramani, Arun 2007] and as such it is usually solved by heuristic (best-effort) "store and forward" replication-based approaches, where multiple copies of the same message are moved and stored across nodes in the hope that at least one will eventually reach its destination. Still, the sparsity and mobility of the nodes causes unpredictable meeting patterns and frequent disconnections [Alouf, Sara and Neglia, Giovanni and Carreras, Iacopo and Miorandi, Daniele and Fialho, Álvaro 2010], which result in relatively low data delivery probabilities (also called delivery rates, or delivery ratios), even with well-established protocols such as Epidemic [Amin Vahdat and David Becker 2000], PRoPHET [Lindgren, Anders and Doria, Avri and Schelén, Olov 2003], and their variants. As shown in [Abolhasan, Mehran and Wysocki, Tadeusz and Dutkiewicz, Eryk 2004; Boukerche, Azzedine and Turgut, Begumhan and Aydin, Nevin and Ahmad, Mohammad Z and Bölöni, Ladislau and Turgut, Damla 2011; Hong, Xiaoyan and Xu, Kaixin and Gerla, Mario 2002], the node density and their mobility affect the delivery probability in most kinds of MANETs (not only DTNs), although the problem is further exacerbated in DTNs: while in dense MANETs composed of slow-moving nodes (e.g. pedestrians) the delivery probability obtained by state-of-the-art routing protocols can be higher than 90% [Johansson, Per and Larsson, Tony and Hedman, Nicklas and Mielczarek, Bartosz and Degermark, Mikael 1999], in sparse MANETs made of fast-moving nodes (e.g. vehicles) the delivery probability can be as low as 15-18%, [Clausen, Thomas 2004; Saudi, Nur Amirah Mohd and Arshad, Mohamad Asrol and Buja, Alya Geogiana and Fadzil, Ahmad Firdaus Ahmad and Saidi, Raihana Md 2019]. Furthermore, other factors such as the number of sources (i.e., the nodes transmitting data) can also affect the delivery probability: the lower the number of sources, the higher the delivery probability [Clausen, Thomas 2004; Perkins, Charles E and Royer, Elizabeth M and Das, Samir R and Marina, Mahesh K 2001].

Traditionally, network protocols are modelled as a *reactive system*, i.e., a two-player game (agent vs environment) where an agent (a node in the network) *reacts*, by performing a certain action, to pre-defined conditions in the environment (the rest of the network): for instance, the agent retries a message transmission if it does not receive an acknowledgment. As such, a protocol can be described with an automaton, for which formal specifications can be logically expressed and verified. For that, one usually needs to have complete knowledge about (and strict assumptions on) the environment. This approach, rooted in the theory of Temporal Logic and infinite (Büchi) automata [Buchi, J Richard and Landweber, Lawrence H 1990], has been the gold standard in protocol design and verification for decades. However, there are limitations. First of all, this way of designing protocols assumes in general the environment, and all its states, to be known: this is usually not the case for DTNs, where the environment conditions can be unpredictable due to the nodes' mobility. Secondly, the numerical (time and space) complexity of these design methods makes them impractical when the number of states of protocol and environment grows [Vardi, Moshe Y 2018]. This is the case of DTNs, where the number of states of the environment can be very large, depending on the number of nodes and their state. For these reasons, finding a way to design better routing protocols for DTNs, or at least improving the performance of the existing ones, is still an open research question.

Here, we consider the *Genetic Improvement* (GI) [Langdon, William B 2015] of two of the main routing protocols used in DTNs, namely Epidemic and PRoPHET. Our methodology consists in the following: first, we dissect the two protocols into their fundamental components, i.e., basic network functionalities such as checking if a node can start transferring data, or sending messages to all connections; then, we apply Genetic Programming (GP) to rearrange these components into evolving trees, in the attempt to maximize the data delivery probability. It is worth stressing that, in principle, this methodology can be easily generalized to other protocols, also at different layers of the protocol stack, and to different kinds of networks.

To evaluate the proposed methodology, we perform a broad *in silico* experimentation where we improve Epidemic and PRoPHET on six test cases of urban networks made of three different kinds of mobile nodes (pedestrians, cars, and trams). Overall, we find that GI consistently produces a gain in data delivery probability. We then verify if this improvement in the delivery probability entails a worsening of other relevant network metrics, and find some counterintuitive results obtained as "byproducts" of GI: in fact, we find that the best evolved protocols not only increase the delivery probability, but also reduce the overall network overhead (i.e., the number of retransmissions). However, they trade these improvements for a higher latency. We also investigate the generalizability of the best evolved protocols across test cases, and find that apart from one specific map (Manhattan), the evolved protocols are able to generalize to unseen test cases, producing results that are still better than the baseline protocols. Finally, we compare the working logics of the best evolved protocols with those of the baseline protocols, and identify some common aspects which underlie their improved performance.

Novel aspects of this work. Compared to the existing works on GP applied to protocol evolution, our work presents various elements of novelty which are worth highlighting. In particular, among the literature we will briefly survey in the next Section, the closer works on the application of GP to the evolution of protocols act either on the application layer, focusing in particular on aggregation protocols (i.e., protocols that calculate an aggregation function of distributed data, such as their mean) [Weise, Thomas and Geihs, Kurt and Baer, Philipp A 2007; Weise, Thomas and Tang, Ke 2011; Weise, Thomas and Zapf, Michael and Geihs, Kurt 2008], protocol adaptors, i.e., interfaces between incompatible application protocols [Van Belle, Werner and Mens, Tom and D'Hondt, Theo 2003], or application logics in Wireless Sensor Networks [Johnson, Derek M. and Teredesai, Ankur M. and Saltarelli, Robert T. 2005; Valencia, Philip and Lindsay, Peter and Jurdak, Raja 2010]; or, they use GP to optimize only a very specific equation used in existing protocols, such as the formula used to variate the contention window size in IEEE802.11 DCF [Lewis, Tim and Fanning, Neil and Clemo, Gary 2006], or the update rule of the routing table [Roohitavaf, Mohammad and Zhu, Ling and Kulkarni, Sandeep and Biswas, Subir 2018] (the latter work being based on rather simplified network simulations). Besides these works, three other papers have addressed, although with various limitations, very related research questions, namely [Alouf, Sara and Neglia, Giovanni and Carreras, Iacopo and Miorandi, Daniele and Fialho, Álvaro 2010; Yamamoto, Lidia and Tschudin, Christian 2005a,b]. Among these, the first two works apply GP to perform the online distributed adaptation of an extremely simplified delivery protocol (which allows only direct message exchanges, i.e., without routing): in this case GP automatically selects different combinations of modules in order to adapt the resulting protocol to the network conditions. This approach is quite different from the one we propose here, which instead is based on offline genetic improvement of *realistic* routing protocols for DTNs. The last one, on the other hand, does focus on the same protocols we study in this paper, i.e., Epidemic and PRoPHET, however it uses a GA to adjust the protocols' parameters (such as the number of copies) in response to the network dynamics, in order optimize the overall delivery probability. Again, this approach is quite different from our proposal. Thus, to the best of our knowledge no prior work has used GP to genetically improve the inner logic of existing routing protocols, yet alone those used in DTNs.

Structure of the paper. The remaining of this paper is organized as follows. In the next Section, we present the related work. Then, in Section 3 we introduce the method, in particular the GP configuration and the simulation setup. In Section 4, we describe the details of the experimentation, while the numerical results are discussed in Section 5. Finally, in Section 6 we draw the conclusions and hint at possible extensions of this work.

2 RELATED WORK

In the following we briefly summarize the related works on Genetic Improvement and the applications of Evolutionary Learning to networked systems.

2.1 Genetic Improvement

While the application of GP for bug fixing, sometimes referred to as *automatic repair* of programs, dates back to the early 2000s [Le Goues, Claire and Dewey-Vogt, Michael and Forrest, Stephanie and Weimer, Westley 2012; Orlov, Michael and Sipper, Moshe 2011; Weimer, Westley and Forrest, Stephanie and Le Goues, Claire and Nguyen, ThanhVu 2010], the term "Genetic Improvement" was originally coined by Langdon and collaborators in some seminal works from 2014 [Langdon, William B 2014; Langdon, William B. and Harman, Mark 2014a,b; Langdon, William B. and Modat, Marc and Petke, Justyna and Harman, Mark 2014; Petke, Justyna and Harman, Mark and Langdon, William B. and Weimer, Westley 2014] and finally formalized in [Langdon, William B 2015]. The broad definition of GI is the application of optimization techniques, particularly evolutionary search algorithms such as GP [Woodward, John R and Johnson, Colin G and Brownlee, Alexander EI 2016], to improve existing software w.r.t. either functional requirements (and in particular bug fixing), or non-functional requirements, such as speed or memory.

In the past few years, several works have shown the potential of GI. Some of these works have focused on speed improvement, e.g. in the case of CUDA code [Langdon, William B. and Harman, Mark 2014a; Langdon, William B. and Modat, Marc and Petke, Justyna and Harman, Mark 2014] and CUDA-based sequencing tools such as BarraCUDA [Langdon, William B. and Lam, Brian Yee Hong 2017], complex C/C++ applications such as Bowtie2 [Langdon, William B. and Harman, Mark 2014b], the OpenCV library [Langdon, William B. and White, David R. and Harman, Mark and Jia, Yue and Petke, Justyna 2016], or the MiniSAT solver [Petke, Justyna and Harman, Mark and Langdon, William B. and Weimer, Westley 2014]. A particular case of GI aimed at speed improvement is discussed in [López-López, Víctor R. and Trujillo, Leonardo and Legrand, Pierrick 2019], where the authors applied GI to a C++ GP library and noted a speedup due to the deletion of some operators (crossover, point mutation and others). Another area of GI research is the accuracy improvement of low-level implementations of various mathematical functions, such as $sqrt$ [Langdon, William B 2019], log_2 [Langdon, WB 2018; Langdon, William B. and Petke, Justyna 2019] and other functions based on lookup tables [Krauss, Oliver and Langdon, William B 2020]. In other works, energy consumption has been considered as primary goal of GI [Bokhari, Mahmoud and Wagner, Markus 2016; Bruce, Bobby R. and Petke, Justyna and Harman, Mark 2015].

As for bug fixing, successful applications of GI have been reported in [Le Goues, Claire and Dewey-Vogt, Michael and Forrest, Stephanie and Weimer, Westley 2012], [Weimer, Westley and Forrest, Stephanie and Le Goues, Claire and Nguyen, ThanhVu 2010], and [Schulte, Eric M. and

Weimer, Westley and Forrest, Stephanie 2015]. This latter work is especially interesting since it uses GI to repair a bug in a MIPS binary without accessing the source code. Beyond bug fixing, automatic code generation has also shown promising results e.g. in the porting of existing C code to CUDA [Langdon, William B 2014], or in the automatic generation of equivalent Java bytecode starting from existing programs [Orlov, Michael and Sipper, Moshe 2011],

Finally, it is worth mentioning two recent surveys of the GI literature, [Petke, Justyna and Haraldsson, Saemundur O. and Harman, Mark and Langdon, William B. and White, David R. and Woodward, John R 2017] and [Langdon, William B. and Ochoa, Gabriela 2016]. The latter work, in particular, mentions the application to programming languages different from C/C++ (which has attracted so far most of the attention of the GI literature) as one of the key challenges for GI. We find that our work is loosely related to this issue.

2.2 Evolutionary Learning applied to networked systems

In the past two decades, various Machine Learning and bio-inspired technique, especially Evolutionary Algorithms (EAs), have been applied to network problems and particularly protocol optimization. For instance, some researchers have proposed various solutions based on collective intelligence [Wolpert, David and Tumer, Kagan and Frank, Jeremy 1999] and Reinforcement Learning (RL) [Peshkin, Leonid and Savova, Virginia 2002; Stampa, Giorgio and Arias, Marta and Sánchez-Charles, David and Muntés-Mulero, Victor and Cabellos, Albert 2017; Tao, Nigel and Baxter, Jonathan and Weaver, Lex 2001] to optimize routing protocols for Wireless Sensor Networks [Alsheikh, Mohammad Abu and Lin, Shaowei and Niyato, Dusit and Tan, Hwee-Pink 2014; Förster, Anna and Murphy, Amy L 2011; Kulkarni, Raghavendra V and Forster, Anna and Venayagamoorthy, Ganesh Kumar 2010]. Albeit quite powerful, the main limitation of most of these approaches is that they often require a large amount of data collected from the network, in order to train a model of the protocol, to be used later at runtime for further optimization.

As for EAs, there is a very large body of research in the general area of networked systems, as surveyed for instance in [Dressler, Falko and Akan, Ozgur B 2010; Nakano, Tadashi 2010][1], with several works focusing especially on protocol optimization (not only for routing). In this context, a seminal paper is represented by the work from late 1990s by El-Fakih et al. [Khaled El-fakih and Hirozumi Yamaguchi and Gregor Bochmann 1999], who formulated the protocol design problem in the form of a 0-1 integer programming message exchange model, optimized by means of a Genetic Algorithm in order to minimize the number of messages to be exchanged while meeting a given specification of network services. As for GP, it has been successfully used to optimize protocol adaptors [Van Belle, Werner and Mens, Tom and D'Hondt, Theo 2003], aggregation protocols [Weise, Thomas and Geihs, Kurt and Baer, Philipp A 2007; Weise, Thomas and Tang, Ke 2011; Weise, Thomas and Zapf, Michael and Geihs, Kurt 2008], or MAC access protocols [Lewis, Tim and Fanning, Neil and Clemo, Gary 2006; Roohitavaf, Mohammad and Zhu, Ling and Kulkarni, Sandeep and Biswas, Subir 2018]. The latter have been synthesized also by means of probabilistic Finite State Machines (FSMs) whose transition probabilities are optimized by means of a Genetic Algorithm [Hajiaghajani, Faezeh and Biswas, Subir 2015a,b; Sharples, Nicholas and Wakeman, Ian 2000].

While the methods above are meant for centralized offline protocol optimization, some works have also considered distributed online protocol optimization, although these usually focus on higher network layers (e.g. application) rather than routing. For instance, a bio-inspired distributed learning approach was introduced in [Su, Yi and Van Der Schaar, Mihaela 2010], where each

[1]Another link between network engineering and evolutionary theory also exists: recent evidence has shown that the typical hourglass-shaped protocol stacks are the result of an implicit evolutionary process that led to a minimal complexity, maximal robustness architecture [Dovrolis, Constantine 2008; Siyari, Payam and Dilkina, Bistra and Dovrolis, Constantine 2017].

node observes the other nodes' behavior and forms internal conjectures on how they would react to its actions, in order to choose the action that maximizes a local utility function: the authors demonstrated, analytically and through numerical simulations, that this method reaches Nash equilibria corresponding to optimal traffic fairness and throughput. Other works have investigated distributed EAs [Iacca, Giovanni 2013] and distributed GP [Johnson, Derek M. and Teredesai, Ankur M. and Saltarelli, Robert T. 2005; Valencia, Philip and Lindsay, Peter and Jurdak, Raja 2010] to evolve the nodes' parameters and functioning logics (at the application layer) of Wireless Sensor Networks. Finally, two notable online methods are STEM-Net [Aloi, Gianluca and Bedogni, Luca and Felice, Marco Di and Loscri, Valeria and Molinaro, Antonella and Natalizio, Enrico and Pace, Pasquale and Ruggeri, Giuseppe and Trotta, Angelo and Zema, Nicola Roberto 2014] and Fraglets [Tschudin, Christian 2003; Yamamoto, Lidia and Schreckling, Daniel and Meyer, Thomas 2007]. The first one is a wireless network where each node uses an EA "to reconfigure itself at multiple layers of the protocol stack, depending on environmental conditions, on the required service and on the interaction with other analogous device". The latter is based on the concept of "autocatalytic software" [Tschudin, Chr and Yamamoto, Lidia 2005], or chemical computing [Miorandi, Daniele and Yamamoto, Lidia 2008]: essentially, protocols emerge automatically as collections of "fraglets", i.e., combinations of code segments and parameters which are evolved, respectively, by distributed GP [Yamamoto, Lidia and Tschudin, Christian 2005a,b] and distributed EAs [Alouf, Sara and Neglia, Giovanni and Carreras, Iacopo and Miorandi, Daniele and Fialho, Álvaro 2010], and spread over the network through opportunistic (epidemic) propagation [Alouf, Sara and Carreras, Iacopo and Miorandi, Daniele and Neglia, Giovanni 2007] regulated by interactions with the environment. On top of this, another EA optimizes the combination of protocols, i.e., the protocol stack [Baude, Françoise and Legrand, Virginie and Henrio, Ludovic and Naoumenko, Paul and Pfeffer, Heiko and Bassbouss, Louay and Linner, David 2010; Imai, Pierre and Tschudin, Christian 2010; Miorandi, Daniele and Yamamoto, Lidia and Dini, Paolo 2006].

Finally, it is worth mentioning recent works on the application of GA to DTNs which are somehow related to this paper in a broader sense, namely [Bucur, Doina and Iacca, Giovanni and Squillero, Giovanni and Tonda, Alberto 2015], where a GA is used to find specific DTN conditions characterized by abnormally low delivery rates, and [Bucur, Doina and Iacca, Giovanni 2017; Bucur, Doina and Iacca, Giovanni and Gaudesi, Marco and Squillero, Giovanni and Tonda, Alberto 2016], where the parameters of groups of heterogeneous malicious nodes attacking the network are optimized in the attempt to *reduce* the delivery probability. Indeed, security is a major concern in this kind of networks: their inherent intermittent (and open) nature makes it difficult to apply encryption and authentication techniques which are common in other kinds of networks [Farrell, Stephen and Cahill, Vinny 2006; Kate, Aniket and Zaverucha, Gregory M and Hengartner, Urs 2007], although some possible countermeasures have been proposed.

3 METHODS

For the numerical experiments, we used **The ONE** (Opportunistic Network Environment) [Keränen, Ari and Ott, Jörg and Kärkkäinen, Teemu 2009], which allows us to simulate a given urban environment composed of a map and a number of moving agents of different types, producing a network traffic with user-defined characteristics. As for the GP algorithm, we used the implementation of the strongly typed GP [Montana, David J 1995] provided by the **Jenetics** library [Wilhelmstötter, Franz 2017]. At each generation, the GP algorithm generates a set of Java classes that implement candidate instances of the routing protocol, differing only for the instructions contained in the `update()` method defined in the abstract Java class `ActiveRouter` implemented in The ONE. We also include in our scheme a validity check with repair mechanism, in order to ensure the validity of the code generated by GP. The dynamically generated classes are then compiled at runtime and fed to the simulator, in order to extract the relevant information needed for the fitness evaluation. In the following, we detail the GP configuration, the validity check and repair mechanism, and the way the code generated by GP is evaluated in The ONE.

3.1 Genetic Programming configuration

The candidate GP individuals, represented as tree structures, are obtained by composing the elements in the non-terminal and terminal sets specified in Table 1 and Table 2 respectively (see Section 3.2). The non-terminals include basic Boolean operators as well as the inequality test, the `if` and the `sequence` operators. Note that `if` is considered as a 2-argument operator (i.e., without `else`) since `if-else` statements can be obtained as a concatenation of `if` and `sequence`. The terminals, instead, are obtained by "dissecting" the `update()` method of the baseline protocols into its *main functional components*, which are then rearranged by GP. This is an important aspect of our proposal: rather than evolving from scratch the entire protocol's logic, which would entail an excessively large, hard-to-explore protocol space, we use available knowledge in the form of protocol basic components, for which we then try to identify a better rearrangement by means of GP. We believe that this form of Genetic Improvement is particularly interesting in that it couples the advantages of using pre-existing code (in this case, in the form of Java functions), representing necessary functional elements, with the power of the genetic search. In our case, we identify as basic components six functions, including the superclass' `update()` method, that implement basic operations which are at the base of the two selected protocols considered in our experimentation. The terminals also include the `return` keyword to exit from the `update()` method.

The implementation of the GP algorithm follows the logics of Jenetics [Wilhelmstötter, Franz 2017] and the genetic operators available therein. At the first generation, the individuals are randomly generated using the aforementioned non-terminals and terminals (note that the baseline protocols are not used as seeds in the initial GP population), with given maximum tree depth and maximum number of nodes ("grow" initialization method). Then, the evolutionary loop starts, where at each cycle:

(1) The individuals in the current population are verified and, if needed, repaired (see Section 3.2), before being evaluated into The ONE (see Section 3.3).
(2) The new population is obtained by applying tournament selection to the current population. A predefined offspring fraction is set, which determines how many selected individuals (the offspring) will be altered by crossover and mutation; the remaining individuals (the survivors) are kept unaltered in the new population.
(3) The offspring are altered by applying *single-node crossover*: with a given crossover probability, every individual among the offspring is selected to undergo crossover; the selected individuals are randomly paired (for a total number of pairs equal to half the number of offspring), and for every pair two new individuals are obtained by swapping two nodes (randomly chosen from the two original individuals in the pair), with the corresponding subtrees. The two new individuals replace the two original ones in the pair.
(4) The individuals obtained by crossover are then selected to undergo also mutation, with a given individual mutation probability (otherwise individuals are not mutated). Each individual selected to undergo mutation is altered by *swap-mutation*: with a given node mutation probability, every node in the GP individual is selected and swapped with another randomly chosen node from the same individual.

This loop (steps 1 to 4) goes on until one of two stop criteria is met: a) the maximum number of generations is reached, or b) the maximum number of generations with steady fitness is reached. The latter criterion refers to the maximum number of generations in which the best fitness in the population does not improve. No specific anti-bloat mechanism (apart from the limited depth and number of nodes in the initial GP population) or history of previously evaluated solutions is used. See Table 3 for the detailed parameters (default values in Jenetics) used in the GP algorithm.

Table 1. Non-terminals used, their corresponding Java code, argument types and return types.

Non-terminal	Java code	Argument types	Return type
`or`	`( arg1 || arg2 )`	*condition*, *condition*	*condition*
`not`	`!arg1`	*condition*	*condition*
`notEqual`	`( arg1 != arg2 )`	*condition*, *condition*	*condition*
`if`	`if ( arg1 ){ arg2 };`	*condition*, *body*	*body*
`sequence`	`arg1; arg2;`	*body*, *body*	*body*

Table 2. Terminals used, their corresponding Java code and type. The terminal `tryOtherMessages` is used only for improving the PRoPHET protocol. All the other terminals are used for both Epidemic and PRoPHET. Note that the method `tryAllMessagesToAllConnections()` is called on the child class (`this`) in order to override the method of the parent class `ActiveRouter`.

Terminal	Java code	Type
`isTransferring`	`isTransferring()`	*condition*
`canStartTransfer`	`canStartTransfer()`	*condition*
`update`	`super.update();`	*body*
`exchangeDeliverableMessages`	`exchangeDeliverableMessages();`	*body*
`tryAllMessagesToAllConnections`	`this.tryAllMessagesToAllConnections();`	*body*
`tryOtherMessages`	`tryOtherMessages();`	*body*
`return`	`return;`	*body*

Table 3. Parameter setting (Koza-style tableau) of the Genetic Programming algorithm.

Parameter	Value
Objective	Delivery probability
Non-terminal set	See Table 1
Terminal set	See Table 2
Population size	150
Initialization	Grow, max depth 5, max nodes 50
Offspring fraction	0.6
Crossover	Single-node crossover, prob. 0.1
Mutation	Swap-mutation, individual prob. $0.1^{2/3}$, node prob. $0.1^{1/3}$
Selection	Tournament selection, size 3
Steady fitness generations	50
Max number of generations	100

3.2 Validity check and repair method

Not all the elements in the non-terminal and terminal sets defined in Tables 1-2 can be an argument of every non-terminal: for example, a `return` instruction cannot be inside an `if` condition and an `or` condition cannot be inside an `if` body, as shown in Figure 1 (top). To address this issue, we have developed two methods: the first one checks the validity of each individual tree structure, the latter tries to repair it, if needed. In order to check the validity of the trees and repair them, an additional structure is built specifying, for each non-terminal, the number of arguments and their types. In our case, we defined two possible types, namely *body* and *condition*.

The validity check starts from the root node and checks if it has the correct number and types of arguments. For example, if the root node is an `if` node, it must have two arguments, respectively of type *condition* and *body*, as shown in Table 1. If this check fails, the current tree is considered not correct. Otherwise, the method recursively checks the arguments, until it reaches the terminals, whose type must comply with those of the arguments of their corresponding parent node. If this recursive check does not fail, the tree is correct.

The repair method is invoked whenever the validity check fails and, following a similar logic, it visits recursively all the nodes in the tree (see Figure 1 for an example). More specifically: 1) in the case of the non-terminals, if it finds that a node does not have the correct number of arguments, or these are not of the correct type, the method tries to randomly replace that node with another non-terminal that has that number of arguments of those types; 2) in the case of terminals, these are replaced, if needed, to comply with the type of the arguments of their parent node. The method also ensures that the tree contains at least one `return` terminal. If the tree cannot be repaired, the tree is not evaluated in the simulator and it is assigned a delivery probability of zero.

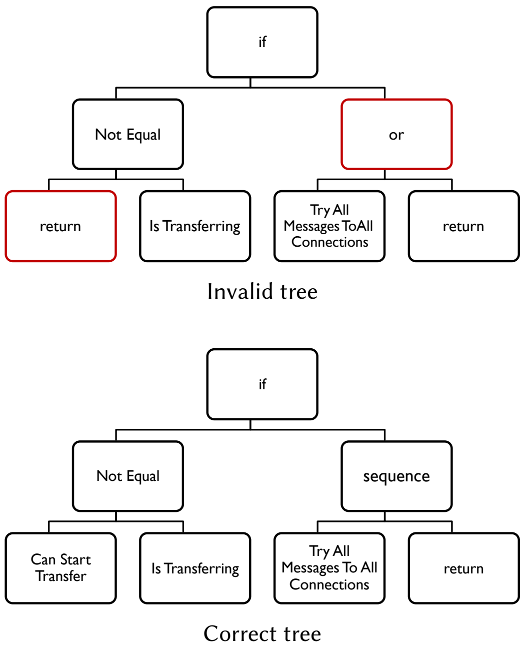
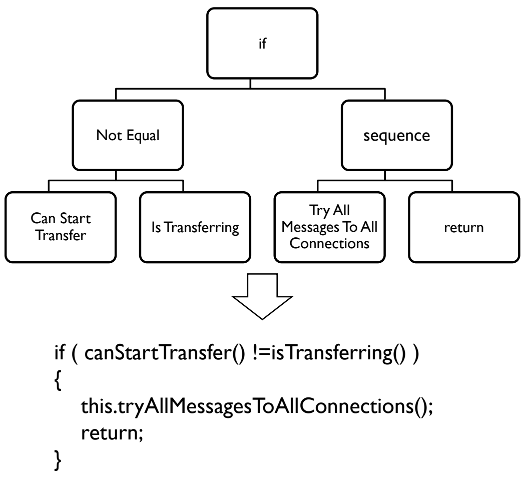


Fig. 1. Example of the repair mechanism on an invalid tree in which the `if` condition contains a `return` instruction and the `if` body contains an `or` condition.



```
if ( canStartTransfer() !=isTransferring() )
{
    this.tryAllMessagesToAllConnections();
    return;
}
```

Fig. 2. Example of translation from tree to Java code.

3.3 Individual evaluation

Each individual (i.e., a candidate implementation of the `update()` method, and its corresponding Java class that is dynamically generated and compiled) is evaluated by measuring the delivery probability in The ONE simulations. This is calculated as the fraction of messages that reach their destination over the total number of messages generated during the simulation. In particular, the individual evaluation proceeds according to the following steps:

(1) Class name generation: A *unique id* is appended to the class name of each GP individual, in order to avoid having GP classes with the same name (the dynamically generated classes are placed in the same folder, and each class is loaded by The ONE at the beginning of a simulation).
(2) Node evaluation: Each node composing the current individual tree is translated into the corresponding Java instructions (see Tables 1-2), generating the code that will be placed in the `update()` method of the dynamically generated Java class. In Figure 2, we show an example of translation from a GP tree to the corresponding Java code inserted into the `update()` of the Java class implementing the routing protocol to be evaluated.
(3) Class generation: The generation of the Java class is done by using a template class, which contains the structure of a routing protocol that extends the `ActiveRouter` class present in The ONE simulator. The templates of the Epidemic and PRoPHET routing protocols are presented in Appendix D, where it can be seen that each template contains the necessary imports, the class definition, its constructors, public and private fields, the `update()` method and other additional methods contained in the baseline protocol. It should be noted that apart from the `update()` method, all the other parts of the class are not modified by GP. For each individual, the template is modified by replacing the code of the baseline protocol inside the `update()` method with the code generated at the previous step.
(4) Class compiling: The dynamically generated class is compiled by calling `javac`, and placed in the correct classpath, in order to allow the simulator to execute it. If the class fails to compile, the corresponding tree is assigned a delivery probability of zero.
(5) Settings file configuration: The ONE uses a settings file to store all the relevant parameters of the simulation. An example of this file is given in Appendix E, where we highlight the parameters that are relevant to our experimentation. The main parameters of interest (see Table 4) are:
    - `Scenario.name`: name of the simulation (i.e., the Java class name generated in the first step);
    - `Group.router`: name of the routing protocol (same as `Scenario.name`);
    - `Group.nrofHosts`: number of nodes (a.k.a. hosts) per group (pedestrians and cars only);
    - `Report.nrofReports`: number of reports created as output of the simulation;
    - `Report.report1`: type of report created as output of the simulation;
    - `Events1.hosts`: range of message source/destination addresses;
    - `MovementModel.worldSize`: size of the map;
    - `MapBasedMovement.nrofMapFiles`: number of map files used in the simulation;
    - `MapBasedMovement.mapFile[1-4]`: maps used in the simulation;
    - `Group[4-5-6].routeFile`: routes followed by the first/second/third group of trams;
    - `ProphetRouter.secondsInTimeUnit`: specific setting for the PRoPHET routing protocol.

    Before each individual simulation, a specific settings file is created starting from the template given in Appendix E, where the relevant settings are set accordingly to the specific test case.
(6) Simulation execution: The DTN simulation uses the automatically generated Java class, producing a report containing the delivery probability of the messages sent in the network.
(7) Result retrieval: The report file `MessageStatsReport` generated by The ONE is parsed in order to extract the delivery probability, which is then used as fitness of the GP individual.

In order to automate these steps, we have implemented a procedure that creates a script to automatically compile the generated Java class starting from the code generated by GP, prepare the settings file, and run The ONE simulation (in batch mode). This script is executed by using the Java Process Builder. When the simulation is terminated, the procedure accesses the report file, and extracts the delivery probability. If the report does not exist, e.g. because the generated class is invalid or the simulation fails, the current individual is assigned a delivery probability of zero.

It should be noted that the bottleneck in the process described above is the simulation execution (step 6): in our experiments, the wall-clock time of a single simulation ranged between 1 minute in the simplest test case to 10 minutes in the most complex one. On the other hand, the steps 1 to 5 and 7 are executed overall in approximately 3 seconds per GP individual.

Table 4. The ONE settings used in the different test cases.

Common settings	Scenario.name:	⟨Name of the Java class generated by Jenetics⟩
	Group.router:	⟨Name of the Java class generated by Jenetics⟩
	Group.nrofHosts:	40 or 100
	Report.nrofReports:	1
	Report.report1:	MessageStatsReport
	Events1.hosts:	0,126 or 0,306
Default settings	MovementModel.worldSize:	4500 × 3400 meters
	MapBasedMovement.nrofMapFiles:	4
	MapBasedMovement.mapFile[1-4]:	Roads, Main Roads, Pedestrian Paths, Shops
	Group[4-5-6].routeFileGroups:	Tram 3-4-10
Helsinki settings	MovementModel.worldSize:	100000 × 100000 meters
	MapBasedMovement.nrofMapFiles:	1
	MapBasedMovement.mapFile1:	Helsinki Medium - Roads
	Group[4-5-6].routeFileGroups :	Helsinki Medium - Bus A. Bus B, Bus C
Manhattan settings	MovementModel.worldSize:	100000 × 100000 meters
	MapBasedMovement.nrofMapFiles:	1
	MapBasedMovement.mapFile1:	Manhattan - roads
	Group[4-5-6].routeFileGroups:	Manhattan - Bus
PRoPHET settings	ProphetRouter.secondsInTimeUnit:	30

4 EXPERIMENTAL SETUP

We considered six different test cases, based on three different maps and two different numbers of agents. The maps used are those available in The ONE, namely the city center of Helsinki (that in The ONE is identified as the “default” map), the metropolitan area of Helsinki (a.k.a. Greater Helsinki), and a Manhattan-like map, see Figure 3. In the following, we will refer to these three maps as **Default**, **Helsinki** and **Manhattan** respectively.

4.1 Simulation configuration

The simulation time of each DTN simulation is 12 hours (with an update interval of 0.1 seconds), starting after a warm-up period of 1000 seconds of simulation time needed to allow the node mobility to reach steady state conditions. The simulated DTNs are open networks (i.e., without authentication) made of three types of mobile nodes (referred to as *hosts*): pedestrians, cars and trams. These heterogeneous networks represent realistic scenarios of hosts moving at different velocity so that establishing a cabled network infrastructure is either obviously impossible (for pedestrian and cars) or too costly to set up and maintain (for trams). In our experiments, hosts are further divided into 6 groups: two groups of pedestrians, one group of cars and three groups of trams. The groups of pedestrians and cars are composed of 40 or 100 hosts each (depending on the experiments), while in all the experiments the groups of trams are composed of 2 hosts each. Thus, the simulations with 40 hosts per group have a total number of $40 \times 2 + 40 + 3 \times 2 = 126$ hosts. The simulations with 100 hosts per group have a total number of $100 \times 2 + 100 + 3 \times 2 = 306$ hosts. During the simulations, a new message of size 500KB-1MB is generated every 25-35 seconds (both the message size and the interval are uniformly sampled in these ranges), with source and destination randomly chosen among all the hosts in the network.

Each group has different networking parameters and mobility behaviors, which are specified in the settings file and have been set according to the default parameters of The ONE (see Appendix E). Concerning the networking parameters, we considered two network interfaces, namely: low-speed short-range Bluetooth (transmit speed: 250kBps, message buffer: 5MB, range: 10m), and a high-speed connection (transmit: 10MBps, message buffer: 10MB, range: 1km). Bluetooth is available to all groups, while the high-speed connection is available only to the first group of trams.

As for the mobility behavior, the hosts are randomly placed on the map at the beginning of the simulation, and the destination of each host is chosen randomly between a set of available target points. The hosts then move according to one of the following mobility models:

- Pedestrians and cars use the `ShortestPathMapBasedMovement`, in which each host moves towards its destination following the shortest valid path on the map. Once the destination is reached, the host waits for a given period of time before choosing (randomly) the next destination. Pedestrians move at 0.5-1.5 m/s, while cars move at 2.7-13.9 m/s. Both kinds of hosts have a waiting time of 0-120 seconds.
- Trams use the `MapRouteMovement`, in which the hosts move between adjacent points on a predefined set of routes of 10-100 points. Once a point is reached, each host waits for 10-30 seconds and then moves to the next randomly selected adjacent point, trying to avoid the point where it came from. Trams move at 7-10 m/s.

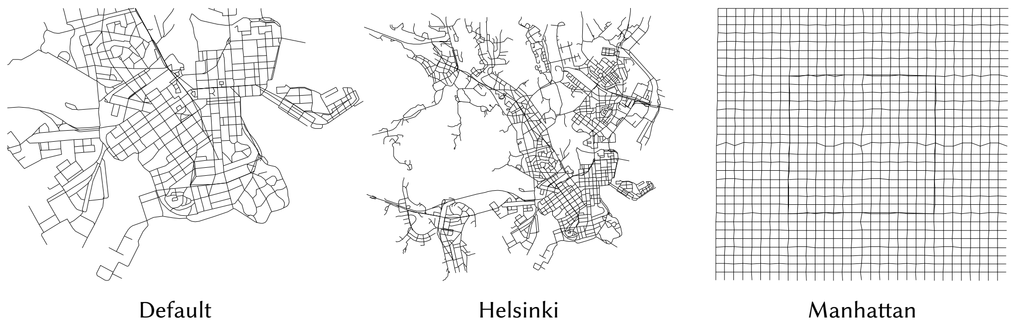


Fig. 3. City maps used in the experimentation.

4.2 Routing protocols

As discussed in Section 1, routing in DTNs is challenging (it is a NP-hard problem [Balasubramanian, Aruna and Levine, Brian and Venkataramani, Arun 2007]) due to their highly dynamic conditions caused by the sparsity and mobility of the nodes. In fact, while in other kinds of ad hoc networks (denser and/or less mobile) nodes can build explicit routes and forward data via established paths (this is the case, for instance, of routing protocols like Ad hoc On-demand Distance Vector (AODV) [Perkins, Charles E and Royer, Elizabeth M 1999] or Dynamic Source Routing (DSR) [Johnson, David B and Maltz, David A 1996]) this is not possible in DTNs due to their lack of continuous connectivity and the consequent absence of instantaneous (and stable) end-to-end paths. Rather than building explicit routes, routing in these cases must take a "store and forward" approach, leveraging the nodes' mobility itself to allow nodes to exchange and carry messages as they move, hoping that this *epidemic-like* propagation will eventually allow messages to reach their intended destination. In the simplest form (called *forwarding-based*) of this kind of routing, only one copy of each message exists at a time in the network: however, this usually does not provide sufficiently high delivery rates. On the other hand, *replication-based* epidemic protocols allow multiple copies of each message to exist in the network at the same time. While this introduces some obvious overhead and hinders scalability, it is the only way to ensure a satisfactory (yet, sub-optimal) delivery probability. This second class of protocols is the focus of our work. In particular, we consider two of the main replication-based protocols, namely Epidemic and PRoPHET, whose functioning can be summarized as below:

- The Epidemic routing protocol [Amin Vahdat and David Becker 2000] is a flooding-based protocol in which each node transmits its messages to every other node met that does not have a copy of them. The only limitation is the maximum number of hops for each message, or alternatively its time-to-live (TTL), i.e. the predefined maximum lifetime of a message over the network.
- The PRoPHET (Probabilistic Routing Protocol using the History of Encounters and Transitivity) protocol [Lindgren, Anders and Doria, Avri and Schelén, Olov 2003] is a variant of the Epidemic routing protocol. This protocol defines a delivery predictability between any two nodes based on the history of contacts between them. A high delivery predictability means a high probability of future contacts between the two nodes. Instead of copying all messages, a message is copied only if the destination node's delivery predictability is higher than the transmitting node's delivery predictability. With respect to Epidemic, this mechanism allows PRoPHET to obtain comparable delivery probability yet with a lower overhead.

In all the experimental test cases, we used the original implementation of Epidemic and PRoPHET available in The ONE, see Listings 1 and 2 reported in Appendix D.

4.3 Computing environment

The experiments have been performed on the High Performance Computing (HPC) facility available at our host institution. We used the JDK v1.8.0_201 64bit, with Jenetics v5.2.0 and The ONE v1.6.0, with thread parallelization at the level of GP individuals (i.e., each thread handles the code generation of a single GP individual and its related simulation in The ONE)[2]. The total HPC runtime of our experiments changed based on the size of the map and the number of hosts, ranging from ∼3-5 hours (Default map, 40 hosts per group) to ∼9-11 hours (Helsinki map or Manhattan map, 40 hosts per group), ∼13-14 hours (Default map, 100 hosts per group), and ∼1 day (Helsinki map or Manhattan map, 100 hosts per group). Each simulation has been executed with 12 cores.

5 EXPERIMENTAL RESULTS

In order to test the proposed method for genetically improving routing protocols, we considered the three urban maps discussed above, with two numbers of hosts per group (40 or 100), thus for a total of six test cases. In each test case, we used as baseline Epidemic and PRoPHET. For reference, we show in Figures 4-5 the functioning logic (in the form of GP tree) corresponding to the `update()` method of the two baseline protocols. We conducted our experimental campaign as follows:

- First, we compared the delivery probability obtained by each of the two baseline protocols against the corresponding genetically improved protocols obtained by GP, in each test case. We

[2]Our code is publicly available at https://github.com/michiL96/evolution_routing_protocol.

considered all the combinations of ⟨protocol, map, number of hosts per group⟩ in: {Epidemic, PRoPHET} × {Default, Helsinki, Manhattan} × {40 hosts, 100 hosts}. For each test case, we executed the baseline protocol for 10 simulations (with different seeds). Similarly, we executed 10 runs (with different seeds) of the GP algorithm on each test case. We then compared the delivery probability of baseline vs best evolved protocols, and performed a statistical analysis based on the Wilcoxon rank-sum test. To further validate our results, we compared our improved PRoPHET protocols against three PRoPHET variants recently proposed in the literature (see Section 5.1).
- Then, we performed a trade-off analysis aimed at understanding if optimizing for the delivery probability produces a degradation of other relevant network metrics (see Section 5.2).

In addition to this, we assessed the generalizability of the evolved protocols between one test case and another, in order to understand if the improved protocols are optimized for a specific test case or rather they can be used in different test cases (see Appendix A). Furthermore, we analyzed the functioning logics of the evolved test cases and identified some common patterns exploited by evolution to improve the existing protocols, and compared these logics with the ones underlying the baseline protocols (see Appendix B).

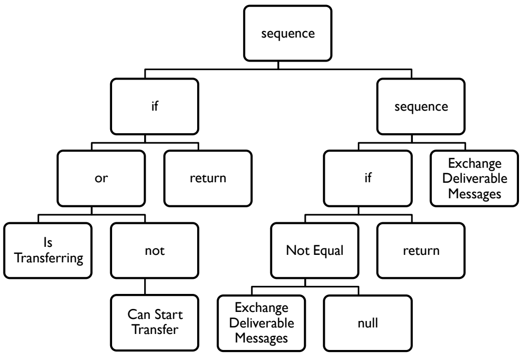


Fig. 4. Tree of the baseline Epidemic routing protocol (note that this tree is not produced by GP).

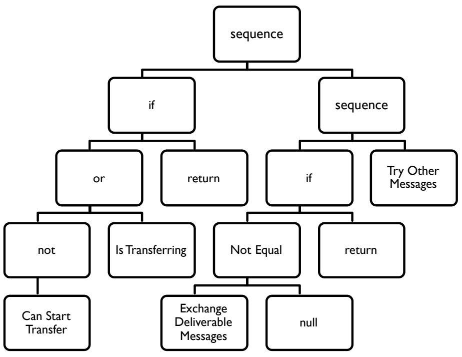


Fig. 5. Tree of the baseline PRoPHET routing protocol (note that this tree is not produced by GP).

5.1 Evolved vs baseline protocols: comparison on the data delivery probability

In Table 6, we report the comparative results (median across 10 simulations) of the delivery probability obtained by the baseline Epidemic protocol vs that obtained by the *best evolved protocol* on each test case. We consider as best evolved protocol the one showing the highest delivery probability across 10 runs of GP, and the lowest number of nodes in case of equal delivery probability. For each pairwise comparison, we report also the p-value of the Wilcoxon rank-sum test ($N = 10, \alpha = 0.05$). From the table, it can can observed that GP is able to obtain statistically significant improvements of delivery probability (p-value $\leq \alpha$) in the Default and Helsinki cases. On the other hand, in the Manhattan test cases the Null Hypothesis on the statistical equivalence between the delivery probability of the baseline protocol and that of the best evolved one cannot be rejected (p-value $> \alpha$). For the sake of completeness, the delivery probability distribution across 10 simulations of the baseline Epidemic protocol and the best evolved ones are shown, in the form of violin plots, in Figure 6. Finally, the analysis of the fitness trends shown in Figure 7 (mean ± std. dev. of the best delivery probability found at each generation across 10 runs of GP) reveals that in 5 out of 6 test cases the initial GP population shows an average delivery probability lower than the corresponding baseline (median across 10 simulations, shown as a dashed blue line). In the remaining test case, i.e, the Default map with 100 hosts per group, the initial GP population is even better than the baseline. In all cases, the average delivery probability quickly increases during the evolutionary process, to stabilize after 20-40 generations on average. It can also be noted that the GP algorithm is quite robust, since the std. dev. across runs (indicated by the shaded area) decreases over time, reaching an almost-zero value towards the end of the available budget.

The same analysis has been performed comparing the PRoPHET routing protocol, as baseline, and the best evolved protocol for each test case, see Table 6. The corresponding violin plots and fitness trends are shown in Figure 8 and 9 respectively. The results reveal that also in this case GP is able to obtain statistically significant improvements of delivery probability (p-value $\leq \alpha$) in all cases except the two Manhattan test cases. As for the fitness trends, it can be noted that in the two Default test cases the average delivery probability of the initial GP population is approximately equal to that of the baseline, while in the remaining cases it is quite lower.

Table 5. Delivery probability (median across 10 simulations) and p-value of the Wilcoxon rank-sum test ($N = 10, \alpha = 0.05$) of the Epidemic routing protocol vs the corresponding best evolved protocol. The evolved protocols perform statistically better in the Default and Helsinki test cases.

Test case	**Epidemic**		**GP**
	Deliv. prob.	**p-value**	
Default (40 hosts)	0.2542	0.005	0.3342
Default (100 hosts)	0.2041	0.005	0.3764
Helsinki (40 hosts)	0.1910	0.005	0.2467
Helsinki (100 hosts)	0.1798	0.005	0.2887
Manhattan (40 hosts)	0.1685	0.574	0.1654
Manhattan (100 hosts)	0.1774	0.139	0.1664

Table 6. Delivery probability (median across 10 simulations) and p-value of the Wilcoxon rank-sum test ($N = 10, \alpha = 0.05$) of the PRoPHET routing protocol vs the corresponding best evolved protocol. The evolved protocols perform statistically better in the Default and Helsinki test cases.

Test case	**PRoPHET**		**GP**
	Deliv. prob.	**p-value**	
Default (40 hosts)	0.2673	0.005	0.3281
Default (100 hosts)	0.2307	0.005	0.3829
Helsinki (40 hosts)	0.2047	0.005	0.2447
Helsinki (100 hosts)	0.2078	0.005	0.2887
Manhattan (40 hosts)	0.1719	0.333	0.1647
Manhattan (100 hosts)	0.2092	0.646	0.2109

Comparison vs other PRoPHET variants. To further validate our results, we compared the best evolved protocols in the case of PRoPHET against three recent PRoPHET variants, namely:

- **PRoPHET+**, proposed in [Huang, Ting-Kai and Lee, Chia-Keng and Chen, Ling-Jyh 2010]. It extends PRoPHET by adding a *deliverability* measure, used to forward messages based on buffer size and availability, energy consumption, bandwidth, popularity and predictability.
- **PRoPHETv2**, proposed in [Lindgren, Anders and Doria, Avri and Davies, Elwyn and Grasic, Samo 2011]. It introduces a dependency (for the delivery predictability) based on the predictability of the encounters of any two nodes. This way, the risk of increasing the delivery predictability of a node due to repeated connections in a short interval of time is reduced.
- **Evict PRoPHET**, proposed in [Sati, Salem and Ahmad, Khaleel 2020]. In this variant, an efficient eviction policy is introduced to decide which message should be removed from the buffer. The policy makes use of an utility function that accounts for time-to-live (elapsed and remaining), hop count, time spent into the buffer, and re-transmissions.

The comparative analysis, reported in Table 7 (median across 10 simulations), shows that the best evolved protocols statistically outperform (p-value $\leq \alpha$) the delivery probability obtained by Evict PRoPHET and PRoPHET+ in all the Default and Helsinki test cases, while the three protocols result equivalent in the Manhattan cases. Compared to PRoPHETv2, the best evolved protocol results statistically superior in the Default and Manhattan test cases with 100 hosts, and equivalent in the others. Overall, this comparison confirms the effectiveness of the proposed technique at finding efficient protocols which are at least comparable to, or even better than, the state-of-the-art protocol implementations.

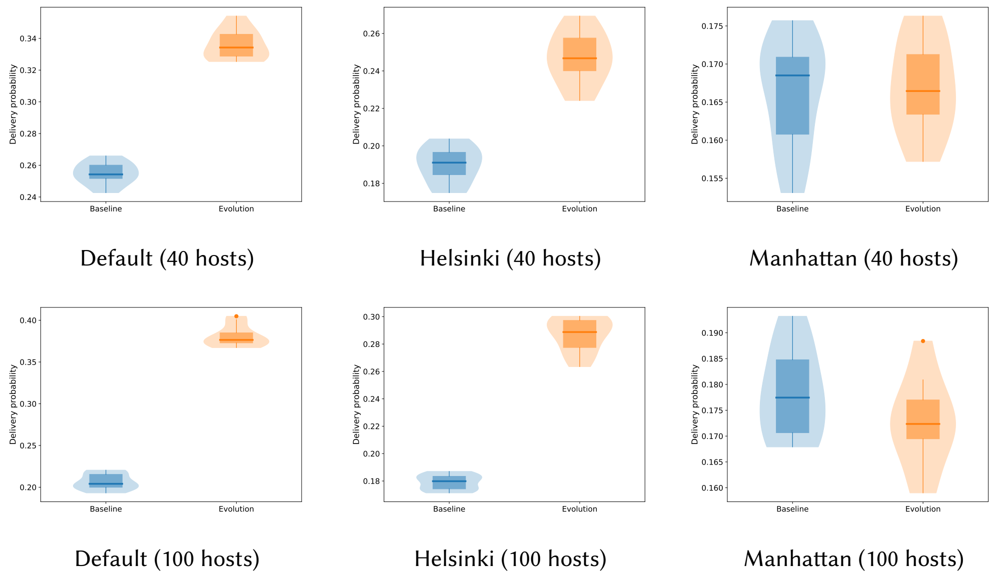


Fig. 6. Distribution of the delivery probability obtained by the Epidemic routing protocol (*Baseline*) and the best evolved protocols (*Evolution*), values from 10 simulations.

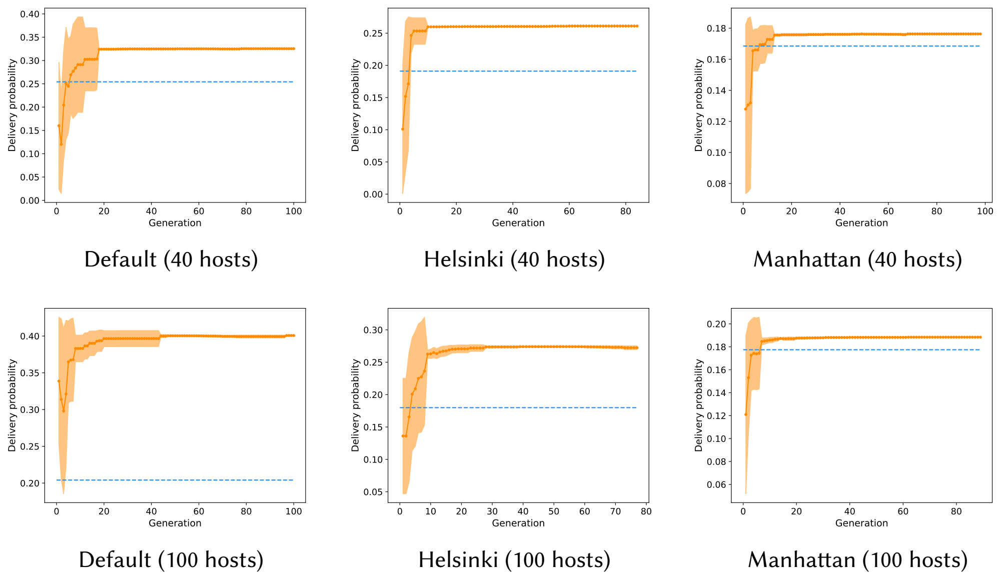


Fig. 7. Best delivery probability at each generation (mean ± std. dev. across 10 runs) obtained by GP vs the Epidemic routing protocol as baseline (median across 10 simulations, dashed blue line). Note that the trends stop at different generations due to the steady fitness stop criterion (50 generations without improvement).

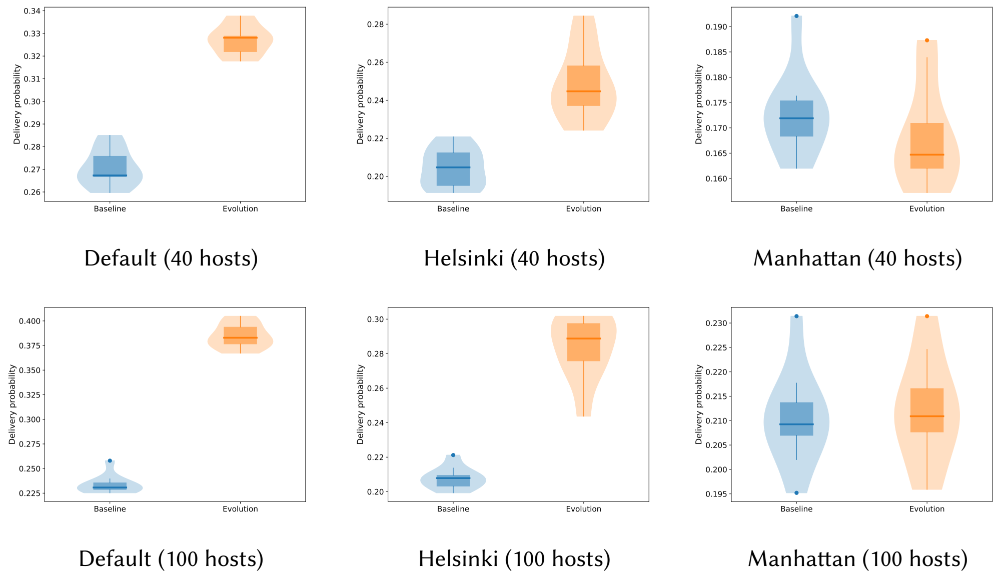


Fig. 8. Distribution of the delivery probability obtained by the PRoPHET routing protocol (*Baseline*) and the best evolved protocols (*Evolution*), values from 10 simulations.

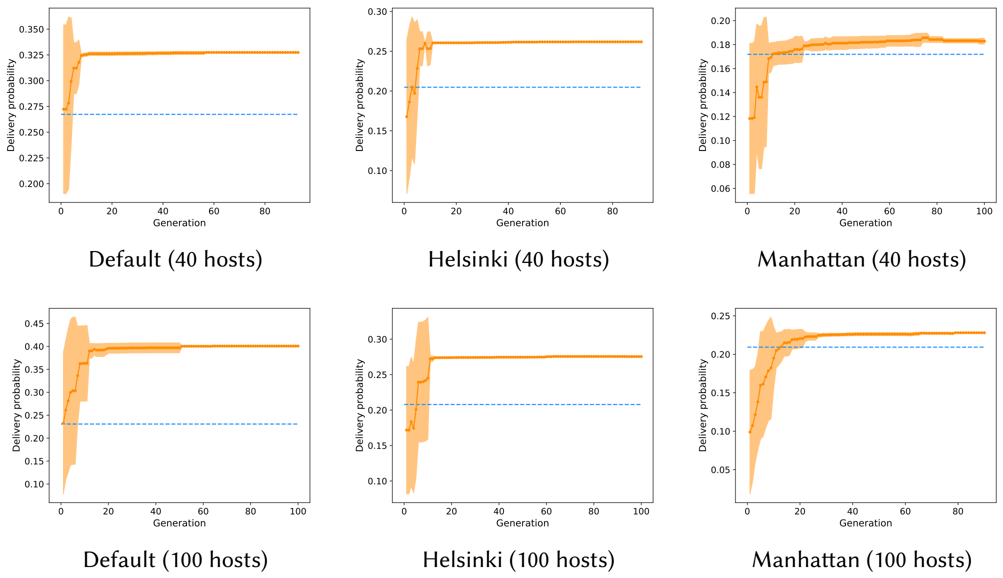


Fig. 9. Best delivery probability at each generation (mean ± std. dev. across 10 runs) obtained by GP vs the PRoPHET routing protocol as baseline (median across 10 simulations, dashed blue line). Note that the trends stop at different generations due to the steady fitness stop criterion (50 generations without improvement).

Table 7. Delivery probability (median across 10 simulations) and p-value of the Wilcoxon rank-sum test ($N = 10, \alpha = 0.05$) of the PRoPHET variants vs the corresponding best evolved protocol. The evolved protocols perform statistically better than Evict PRoPHET and PRoPHET+ in the Default and Helsinki test cases, and better than PRoPHETv2 in the Default and Manhattan test cases with 100 hosts.

Test case	**PRoPHETv2**		**Evict PRoPHET**		**PRoPHET+**		**GP**
	Deliv. prob.	**p-value**	**Deliv. prob.**	**p-value**	**Deliv. prob.**	**p-value**	
Default (40 hosts)	0.3185	0.028	0.2710	0.005	0.2724	0.005	0.3281
Default (100 hosts)	0.3201	0.005	0.2328	0.005	0.2348	0.005	0.3829
Helsinki (40 hosts)	0.2320	0.153	0.2047	0.005	0.2016	0.005	0.2447
Helsinki (100 hosts)	0.2546	0.007	0.2112	0.005	0.2116	0.005	0.2887
Manhattan (40 hosts)	0.1822	0.059	0.1743	0.093	0.1709	0.284	0.1647
Manhattan (100 hosts)	0.2419	0.005	0.2071	0.507	0.2044	0.414	0.2109

5.2 Evolved vs baseline protocols: trade-off between data delivery probability and other network metrics

After the positive assessment of the evolved protocols in terms of improved delivery probability, we have analyzed if there is a trade-off between the delivery probability and other network metrics of interest, in the attempt to understand if an improvement of the delivery probability entails a worsening of other network aspects. In particular, we took into account four different metrics provided in the `MessageStatsReport` log output file generated The ONE, defined below.

Overhead ratio. This metric is calculated according to the following formula:

$$\text{Overhead ratio} = \frac{n_{relayed} - n_{delivered}}{n_{delivered}} \qquad \text{if } n_{delivered} > 0$$

where $n_{relayed}$ indicates how many messages are transmitted over the network (including duplicates), while $n_{delivered}$ indicates how many messages reach their destination.

Latency (avg). This metric is calculated according to the following formula:

$$\text{Latency (avg)} = \frac{1}{n_{delivered}} \sum_{i=1}^{n_{delivered}} \left(T^i_{delivery} - T^i_{creation} \right)$$

where $T^i_{delivery}$ and $T^i_{creation}$ indicate respectively the timestep at which each delivered message reaches its destination and the timestep at which that message was created, $T^i_{delivery} > T^i_{creation}$.

Hop count (avg). This metric is calculated according to the following formula:

$$\text{Hop count (avg)} = \frac{1}{n_{delivered}} \sum_{i=1}^{n_{delivered}} \left(HC^i \right)$$

where HC^i is the hop count of the i-th message, i.e., how many nodes each delivered message passed through before reaching its destination.

Buffer time (avg). Every node in the network keeps a limited-size buffer to store the received messages waiting to be forwarded to other nodes. The different routing protocols implemented in The ONE have different ways to handle when a message is deleted from the buffer, implemented in the `deleteMessage()` of the classes contained in the `routing` package. The `EpidemicRouter` and `ProphetRouter` classes, inheriting the behavior of `ActiveRouter`, delete a message from the buffer either when a copy of that message has already reached its destination, in which case the deleted message is marked as *removed* (however, this condition never occurs in the Epidemic and

PRoPHET nor in the evolved protocols, see Figures 11-12 in Appendix B) or when the node needs to make room in its buffer for a new incoming message (deleting oldest messages first), in which case the deleted message is marked as *dropped*. A message can also be deleted when it reaches its time-to-live. The TTL is defined in the simulation settings file (in our experiments we set a TTL of 5 hours, out of a total simulation duration of 12). When the TTL is exceeded, the message is also marked as *dropped* (see below for a detailed analysis of the message status). Regardless the deletion reason, The ONE keeps track of the average waiting time of the messages deleted from the buffers:

$$\text{Buffer time (avg)} = \frac{1}{n_{deleted}} \sum_{i=1}^{n_{deleted}} \left(T^i_{deletion} - T^i_{receive}\right)$$

where $T^i_{deletion}$ and $T^i_{receive}$ indicate respectively the timestep at which a message was deleted from one node's buffer and the timestep at which it was received at that node, $T^i_{deletion} > T^i_{receive}$.

From an application point of view, it is desirable to increase the delivery probability (how many messages reach their destination) and decrease the latency (how long they take to do so). On the other hand, for a better network resource management, i.e., to optimize the amount of data transmitted over the network and the amount of memory in use in each node, it would be desirable to reduce the overhead and, possibly, the buffer time. This is especially crucial in DTNs, and MANETs in general, where the nodes can be battery-powered (and sending a message is the most energy-consuming event) and memory-limited, see e.g. [Iacca, Giovanni 2013] for a discussion of these two aspects in the context of Wireless Sensor Networks. Therefore, it is important to assess if improving the delivery probability produces a worsening of these other metrics.

We report the results of the aforementioned metrics in Tables 8 and 9, respectively for the Epidemic and PRoPHET protocols. A graphical representation of these values, also w.r.t. the corresponding data delivery probability, is reported in Figure 10 in the form of a matrix of 2D scatter plots. Our analysis reveals some interesting, and in some cases counterintuitive, findings. On the one hand, there are two positive "byproducts" of the evolution: 1) the overhead of the evolved protocols is dramatically lower than that of the baseline protocols (in some cases, even 3 orders of magnitude smaller), which is different from what we would have expected to observe (i.e., a larger overhead corresponding to an increased delivery probability: the more duplicates, the higher the chance to reach a destination); 2) apart from the Manhattan test cases, the hop count of the evolved protocols is in most cases very close to 1, compared to values between 2 and 8 observed with the baseline protocols. On the other hand, it can be noted that the evolved protocols are characterized by a higher latency w.r.t. the baseline protocols (in the worst case, i.e., the Default test case with both Epidemic and PRoPHET, it is twice as big), as well as a higher buffer time that in the worst case (again, the Default test cases with both baseline protocols) is even 20 times bigger.

Overall, these results suggest that the evolved protocols are very efficient in terms of delivery probability as well as hop count and overhead, but there exists a trade-off between these three metrics and latency and buffer time. This trade-off is particularly evident in Figure 10, see e.g. the ⟨Buffer time (avg)⟩ vs ⟨Overhead ratio⟩ and the ⟨Buffer time (avg)⟩ vs ⟨Hop count (avg)⟩ subplots, where a Pareto front can be clearly identified: in both cases, the evolved protocols appear on one corner of the Pareto front (characterized by low overheads and hop counts but high buffer times), while the baseline protocols seem to be designed to have a low buffer time but a high overhead and hop count. Concerning the correlation between the delivery probability, i.e., the goal of the optimization performed by GP, and the other metrics, a somehow less "sharp" (but, still evident) Pareto front can be identified in the ⟨Overhead ratio⟩ vs ⟨Delivery probability⟩ and ⟨Hop count (avg)⟩ vs ⟨Delivery probability⟩ subplots. The ⟨Latency⟩ vs ⟨Delivery probability⟩ and the ⟨Buffer time (avg)⟩ vs ⟨Delivery probability⟩ reveal instead two well-defined clusters separating the results of the evolved protocols from those of the baseline protocols. Finally, considering the other combinations of metrics, it appears that the latency and hop count correlate negatively and positively, respectively, with the overhead. Furthermore, the hop count and buffer time correlate negatively and positively, respectively, with the latency. As for this last aspect, while the negative correlation between hop count and latency might be counterintuitive, it should be considered that latency does not depend only on the number of hops, but also on the time spent by each message in the local buffer of each node it goes through.

All in all, this analysis indicates that a specific characteristic of the evolved protocols is that they tend to generate less duplicates than their baseline counterparts, thus avoiding filling the buffers too frequently. This makes it possible to keep messages longer in the buffers (higher buffer times), and eventually transmit them before they are deleted. Furthermore, apart for the Manhattan cases, the messages reach their destination with an average hop count close to 1, i.e., they are delivered as soon as they are within reach of the source node. This, in turn, contributes to having less duplicates, thus a lower overhead, as shown by the positive correlation between hop count and overhead. The resulting effect of this behavior is then an increase of the delivery probability.

Table 8. Network metrics of the evolved protocols vs the Epidemic routing protocol (median across 10 simulations). The evolved protocols show lower overhead and hop count but higher latency and buffer time.

Test case	**Overhead ratio**		**Latency (avg)**		**Hop count (avg)**		**Buffer time (avg)**	
	Epidemic	**GP**	**Epidemic**	**GP**	**Epidemic**	**GP**	**Epidemic**	**GP**
Default (40 hosts)	85	0.4	4473	6676	4.4	1.1	1364	13836
Default (100 hosts)	621	1	3681	7343	7.4	1.1	553	14354
Helsinki (40 hosts)	66	0.1	5189	6908	4.0	1.0	1847	15341
Helsinki (100 hosts)	466	0.7	4881	7706	7.6	1.0	770	15851
Manhattan (40 hosts)	42	42	6569	6427	3.3	3.2	3051	2998
Manhattan (100 hosts)	251	0.3	6152	8076	6.2	1.0	1342	17231

Table 9. Network metrics of the evolved protocols vs the PRoPHET routing protocol (median across 10 simulations). The evolved protocols show lower overhead and hop count but higher latency and buffer time.

Test case	**Overhead ratio**		**Latency (avg)**		**Hop count (avg)**		**Buffer time (avg)**	
	PRoPHET	**GP**	**PRoPHET**	**GP**	**PRoPHET**	**GP**	**PRoPHET**	**GP**
Default (40 hosts)	67	0.2	4751	6781	3.4	1.0	1486	14731
Default (100 hosts)	415	1.0	3891	7254	5.0	1.1	629	14426
Helsinki (40 hosts)	48	0.1	5472	6996	2.7	1.0	2054	15154
Helsinki (100 hosts)	290	0.7	5137	7708	4.2	1.1	888	15891
Manhattan (40 hosts)	32	42	6927	6585	2.5	3.4	3373	3006
Manhattan (100 hosts)	153	152	6710	6809	3.8	3.8	1503	1493

6 CONCLUSIONS

Main findings. We applied Genetic Programming to *genetically improve* two replication-based routing protocols widely adopted in intermittently connected networks, namely Epidemic and PRoPHET. In four out of six test cases, GP was able to find protocol implementations that produced significantly better delivery probabilities w.r.t. the two baseline protocols. In the two Manhattan-like test cases, on the other hand, no significant improvement was obtained. A similar difference in

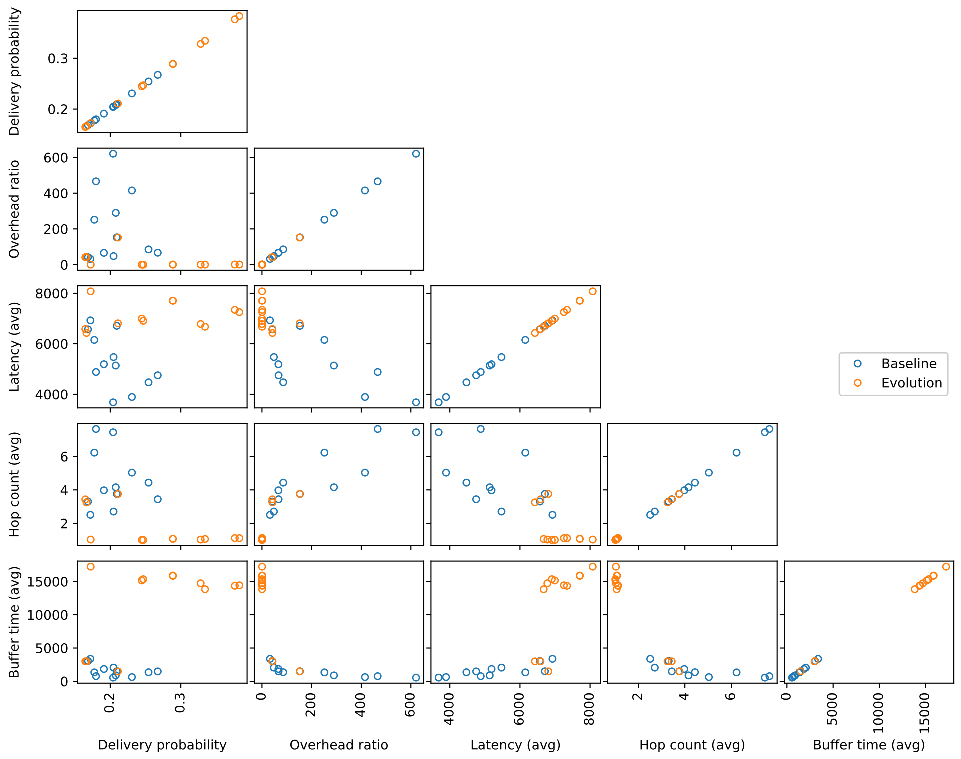


Fig. 10. Trade-off between the delivery probability and the different metrics reported for the different test cases in Table 8 and Table 9.

performance was also observed when comparing our improved PRoPHET protocols against three variants of PRoPHET proposed in the recent literature. We also observed that the evolved protocols are in general characterized by a reduced overhead yet larger latencies w.r.t. the baseline protocols. Furthermore, we found that apart from the Manhattan cases the evolved protocols could generalize across test cases (results reported in Appendix A). Finally, it is worth noticing that the genetically improved protocols are not expensive to implement (as they use the same components of the baseline protocols) and they can cope with the typical hardware/computational constraints of the devices for which the existing protocols are usually intended.

Limitations. This work represents a first attempt to establish a more general methodology to evolve protocols, and stacks thereof. As such, the present methodology has some clear limitations that it would be worth to overcome: for instance, it would be useful to implement a mechanism to automatically extract the fundamental components of the existing protocols (currently this is done manually), or to evolve also other parts of the routing protocol (i.e., not only the `update()` method). Another improvement might be the introduction of an anti-bloat mechanism as well as a history of the evaluated solutions based on syntactic or semantic similarity, in order to avoid fitness re-evaluation and improve diversity during the search.

Future works. A straightforward extension of this work would be to test the proposed methodology to other DTN protocols, such as RAPID [Balasubramanian, Aruna and Levine, Brian and Venkataramani, Arun 2007], MaxProp [Burgess, John and Gallagher, Brian and Jensen, David D and Levine, Brian Neil and others 2006], or Spray & Wait [Spyropoulos, Thrasyvoulos and Psounis, Konstantinos and Raghavendra, Cauligi S 2005]. It would be possible to extend it also to routing protocols used in other kinds of MANETs (i.e., non-DTN), such as table-driven protocols, e.g. Optimized Link State Routing Protocol (OLSR) [Clausen, Thomas and Jacquet, Philippe and Adjih, Cédric and Laouiti, Anis and Minet, Pascale and Muhlethaler, Paul and Qayyum, Amir and Viennot, Laurent 2003] and Destination Sequence Distance Vector (DSDV) [Perkins, Charles E and Bhagwat, Pravin 1994], or on-demand (reactive) protocols, e.g. Ad hoc On-demand Distance Vector (AODV) [Perkins, Charles E and Royer, Elizabeth M 1999] and Dynamic Source Routing (DSR) [Johnson, David B and Maltz, David A 1996]. However, since these protocols are in general more complex than Epidemic and PRoPHET, the search space projected by their components is potentially much larger. Thus, more computational resources and/or specific search operators might be needed in order to find improved protocols. Going beyond routing, it would be possible to apply the proposed methodology to other network layers, for instance to improve existing MAC or congestion protocols. As we have discussed in Section 2, albeit some previous research has applied GP to other network layers, the existing works either evolve *from scratch* a protocol, starting from its specifications, or optimize specific elements of the protocols, such as e.g. the formulas used to vary the congestion window, rather than performing an actual Genetic Improvement in terms of software code. Another possibility would be to use not only existing high-level components (obtained, as we have seen, from the decomposition of the existing protocols), but also components that are evolved *ex novo*. While the search space would be tremendously larger, the additional degrees of freedom offered might provide important opportunities to improve the existing protocols. An intriguing possibility would also be to implement forms of transfer learning across different networks, and online adaptation in response to network changes. A further point of attention is the trade-off between different network metrics: rather than improving a protocol w.r.t. only one metric of interest, it would possible be to apply multi-objective GP and look explicitly for the different trade-offs existing between two or more metrics of interest, so to eventually define *a posteriori* the right protocol to use. Finally, it would be valuable to assess in hardware the performance of the evolved protocols.

REFERENCES

Abolhasan, Mehran and Wysocki, Tadeusz and Dutkiewicz, Eryk. 2004. A review of routing protocols for mobile ad hoc networks. *Ad hoc networks* 2, 1 (2004), 1–22.

Aloi, Gianluca and Bedogni, Luca and Felice, Marco Di and Loscri, Valeria and Molinaro, Antonella and Natalizio, Enrico and Pace, Pasquale and Ruggeri, Giuseppe and Trotta, Angelo and Zema, Nicola Roberto. 2014. STEM-Net: an evolutionary network architecture for smart and sustainable cities. *Transactions on Emerging Telecommunications Technologies* 25, 1 (2014), 21–40.

Alouf, Sara and Carreras, Iacopo and Miorandi, Daniele and Neglia, Giovanni. 2007. Embedding evolution in epidemic-style forwarding. In *International Conference on Mobile Adhoc and Sensor Systems*. IEEE, New York, NY, 1–6.

Alouf, Sara and Neglia, Giovanni and Carreras, Iacopo and Miorandi, Daniele and Fialho, Álvaro. 2010. Fitting genetic algorithms to distributed on-line evolution of network protocols. *Computer Networks* 54, 18 (2010), 3402–3420.

Alsheikh, Mohammad Abu and Lin, Shaowei and Niyato, Dusit and Tan, Hwee-Pink. 2014. Machine learning in wireless sensor networks: Algorithms, strategies, and applications. *Communications Surveys & Tutorials* 16, 4 (2014), 1996–2018.

Amin Vahdat and David Becker. 2000. Epidemic routing for partially connected ad hoc networks. Technical Report CS-2000-06.

Balasubramanian, Aruna and Levine, Brian and Venkataramani, Arun. 2007. DTN routing as a resource allocation problem. In *SIGCOMM*. ACM, New York, NY, 373–384.

Baude, Françoise and Legrand, Virginie and Henrio, Ludovic and Naoumenko, Paul and Pfeffer, Heiko and Bassbouss, Louay and Linner, David. 2010. Mixing workflows and components to support evolving services. *International Journal of Adaptive, Resilient and Autonomic Systems* 1, 4 (2010), 60–84.

Bokhari, Mahmoud and Wagner, Markus. 2016. Optimising energy consumption heuristically on android mobile phones. In *Genetic and Evolutionary Computation Conference Companion*. ACM, New York, NY, 1139–1140.

Boukerche, Azzedine and Turgut, Begumhan and Aydin, Nevin and Ahmad, Mohammad Z and Bölöni, Ladislau and Turgut, Damla. 2011. Routing protocols in ad hoc networks: A survey. *Computer networks* 55, 13 (2011), 3032–3080.

Bruce, Bobby R. and Petke, Justyna and Harman, Mark. 2015. Reducing energy consumption using genetic improvement. In

Genetic and Evolutionary Computation Conference. ACM, New York, NY, 1327–1334.

Buchi, J Richard and Landweber, Lawrence H. 1990. Solving sequential conditions by finite-state strategies. In *The Collected Works of J. Richard Büchi*. Springer, Berlin, Heidelberg, 525–541.

Bucur, Doina and Iacca, Giovanni. 2017. Improved search methods for assessing Delay-Tolerant Networks vulnerability to colluding strong heterogeneous attacks. *Expert systems with applications* 80 (2017), 311–322.

Bucur, Doina and Iacca, Giovanni and Gaudesi, Marco and Squillero, Giovanni and Tonda, Alberto. 2016. Optimizing groups of colluding strong attackers in mobile urban communication networks with evolutionary algorithms. *Applied Soft Computing* 40 (2016), 416–426.

Bucur, Doina and Iacca, Giovanni and Squillero, Giovanni and Tonda, Alberto. 2015. Black holes and revelations: using evolutionary algorithms to uncover vulnerabilities in disruption-tolerant networks. In *European Conference on the Applications of Evolutionary Computation*. Springer, Berlin, Heidelberg, 29–41.

Burgess, John and Gallagher, Brian and Jensen, David D and Levine, Brian Neil and others. 2006. MaxProp: Routing for Vehicle-Based Disruption-Tolerant Networks. In *Infocom*, Vol. 6. IEEE, New York, NY, 1–11.

Clausen, Thomas. 2004. *Comparative Study of Routing Protocols for Mobile Ad-hoc networks*. Research Report RR-5135. INRIA.

Clausen, Thomas and Jacquet, Philippe and Adjih, Cédric and Laouiti, Anis and Minet, Pascale and Muhlethaler, Paul and Qayyum, Amir and Viennot, Laurent. 2003. Optimized link state routing protocol (OLSR). inria-00471712.

Dovrolis, Constantine. 2008. What would Darwin think about clean-slate architectures? *SIGCOMM Computer Communication Review* 38, 1 (2008), 29–34.

Dressler, Falko and Akan, Ozgur B. 2010. A survey on bio-inspired networking. *Computer Networks* 54, 6 (2010), 881–900.

Farrell, Stephen and Cahill, Vinny. 2006. Security considerations in space and delay tolerant networks. In *International Conference on Space Mission Challenges for Information Technology*. IEEE, New York, NY, 8.

Förster, Anna and Murphy, Amy L. 2011. Machine Learning across the WSN Layers. In *Emerging Communications for Wireless Sensor Networks*, Anna Foerster and Alexander Foerster (Ed.). IntechOpen, London.

Hajiaghajani, Faezeh and Biswas, Subir. 2015a. Feasibility of Evolutionary Design for Multi-Access MAC Protocols. In *Global Communications Conference*. IEEE, New York, NY, 1–7.

Hajiaghajani, Faezeh and Biswas, Subir. 2015b. MAC protocol design using evolvable state-machines. In *International Conference on Computer Communication and Networks*. IEEE, New York, NY, 1–6.

Holzmann, Gerard J. 1991. *Design and validation of computer protocols*. Vol. 512. Prentice Hall, Upper Saddle River, NJ.

Hong, Xiaoyan and Xu, Kaixin and Gerla, Mario. 2002. Scalable routing protocols for mobile ad hoc networks. *IEEE Network* 16, 4 (2002), 11–21.

Huang, Ting-Kai and Lee, Chia-Keng and Chen, Ling-Jyh. 2010. PRoPHET+: An adaptive prophet-based routing protocol for opportunistic network. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*. IEEE, New York, NY, 112–119.

Iacca, Giovanni. 2013. Distributed optimization in wireless sensor networks: an island-model framework. *Soft Computing* 17, 12 (2013), 2257–2277.

Imai, Pierre and Tschudin, Christian. 2010. Practical online network stack evolution. In *International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE, New York, NY, 34–41.

Johansson, Per and Larsson, Tony and Hedman, Nicklas and Mielczarek, Bartosz and Degermark, Mikael. 1999. Scenario-based performance analysis of routing protocols for mobile ad-hoc networks. In *International Conference on Mobile Computing and Networking*. ACM, New York, NY, 195–206.

Johnson, David B and Maltz, David A. 1996. Dynamic source routing in ad hoc wireless networks. In *Mobile computing*. Springer, Berlin, Heidelberg, 153–181.

Johnson, Derek M. and Teredesai, Ankur M. and Saltarelli, Robert T. 2005. Genetic programming in wireless sensor networks. In *European Conference on Genetic Programming*. Springer, Berlin, Heidelberg, 96–107.

Kate, Aniket and Zaverucha, Gregory M and Hengartner, Urs. 2007. Anonymity and security in delay tolerant networks. In *International Conference on Security and Privacy in Communications Networks and the Workshops*. IEEE, New York, NY, 504–513.

Keränen, Ari and Ott, Jörg and Kärkkäinen, Teemu. 2009. The ONE simulator for DTN protocol evaluation. In *International Conference on Simulation Tools and Techniques*. ACM, New York, NY, 1–10.

Khaled El-fakih and Hirozumi Yamaguchi and Gregor Bochmann. 1999. A Method and a Genetic Algorithm for Deriving Protocols for Distributed Applications with Minimum Communication Cost. In *International Conference on Parallel and Distributed Computing and Systems*. International Association of Science and Technology for Development, Calgary, 1–6.

Krauss, Oliver and Langdon, William B. 2020. Automatically Evolving Lookup Tables for Function Approximation. In *European Conference on Genetic Programming (Part of EvoStar)*. Springer, Berlin, Heidelberg, 84–100.

Kulkarni, Raghavendra V and Forster, Anna and Venayagamoorthy, Ganesh Kumar. 2010. Computational intelligence in wireless sensor networks: A survey. *Communications Surveys & Tutorials* 13, 1 (2010), 68–96.

Langdon, WB. 2018. Evolving square root into binary logarithm. *RN* 18 (2018), 05.

Langdon, William B. 2014. Genetic improvement of programs. In *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, New York, NY, 14–19.

Langdon, William B. 2015. Genetically improved software. In *Handbook of Genetic Programming Applications*. Springer, Berlin, Heidelberg, 181–220.

Langdon, William B. 2019. Genetic Improvement of Data gives double precision invsqrt. In *Genetic and Evolutionary Computation Conference Companion*. ACM, New York, NY, 1709–1714.

Langdon, William B. and Harman, Mark. 2014a. Genetically improved CUDA C++ software. In *European Conference on Genetic Programming*. Springer, Berlin, Heidelberg, 87–99.

Langdon, William B. and Harman, Mark. 2014b. Optimizing existing software with genetic programming. *Transactions on Evolutionary Computation* 19, 1 (2014), 118–135.

Langdon, William B. and Lam, Brian Yee Hong. 2017. Genetically improved barraCUDA. *BioData Mining* 10, 1 (2017), 28.

Langdon, William B. and Modat, Marc and Petke, Justyna and Harman, Mark. 2014. Improving 3D medical image registration CUDA software with genetic programming. In *Genetic and Evolutionary Computation Conference*. ACM, New York, NY, 951–958.

Langdon, William B. and Ochoa, Gabriela. 2016. Genetic improvement: A key challenge for evolutionary computation. In *Congress on Evolutionary Computation*. IEEE, New York, NY, 3068–3075.

Langdon, William B. and Petke, Justyna. 2019. Genetic improvement of data gives binary logarithm from sqrt. In *Genetic and Evolutionary Computation Conference Companion*. ACM, New York, NY, 413–414.

Langdon, William B. and White, David R. and Harman, Mark and Jia, Yue and Petke, Justyna. 2016. API-constrained genetic improvement. In *International Symposium on Search Based Software Engineering*. Springer, Berlin, Heidelberg, 224–230.

Le Goues, Claire and Dewey-Vogt, Michael and Forrest, Stephanie and Weimer, Westley. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, Zurich, 3–13.

Lewis, Tim and Fanning, Neil and Clemo, Gary. 2006. Enhancing IEEE802.11 DCF using genetic programming. In *Vehicular Technology Conference*, Vol. 3. IEEE, New York, NY, 1261–1265.

Lindgren, Anders and Doria, Avri and Davies, Elwyn and Grasic, Samo. 2011. Probabilistic Routing Protocol for Intermittently Connected Networks. https://tools.ietf.org/html/draft-irtf-dtnrg-prophet-09. draft-irtf-dtnrg-prophet-09.

Lindgren, Anders and Doria, Avri and Schelén, Olov. 2003. Probabilistic routing in intermittently connected networks. *ACM SIGMOBILE mobile computing and communications review* 7, 3 (2003), 19–20.

López-López, Víctor R. and Trujillo, Leonardo and Legrand, Pierrick. 2019. Applying genetic improvement to a genetic programming library in C++. *Soft Computing* 23, 22 (2019), 11593–11609.

Miorandi, Daniele and Yamamoto, Lidia. 2008. Evolutionary and embryogenic approaches to autonomic systems. In *International Conference on Performance Evaluation Methodologies and Tools*. Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, Brussels, 1–12.

Miorandi, Daniele and Yamamoto, Lidia and Dini, Paolo. 2006. Service Evolution in Bio-Inspired Communication Systems. *International Transactions on Systems Science and Applications* 2, 1 (2006), 51–60.

Montana, David J. 1995. Strongly typed genetic programming. *Evolutionary Computation* 3, 2 (1995), 199–230.

Nakano, Tadashi. 2010. Biologically inspired network systems: A review and future prospects. *Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 41, 5 (2010), 630–643.

Orlov, Michael and Sipper, Moshe. 2011. Flight of the FINCH through the Java wilderness. *Transactions on Evolutionary Computation* 15, 2 (2011), 166–182.

Perkins, Charles E and Bhagwat, Pravin. 1994. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. *SIGCOMM computer communication review* 24, 4 (1994), 234–244.

Perkins, Charles E and Royer, Elizabeth M. 1999. Ad-hoc on-demand distance vector routing. In *Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications*. IEEE, New York, NY, 90–100.

Perkins, Charles E and Royer, Elizabeth M and Das, Samir R and Marina, Mahesh K. 2001. Performance comparison of two on-demand routing protocols for ad hoc networks. *IEEE Personal communications* 8, 1 (2001), 16–28.

Peshkin, Leonid and Savova, Virginia. 2002. Reinforcement learning for adaptive routing. In *International Joint Conference on Neural Networks*, Vol. 2. IEEE, New York, NY, 1825–1830.

Petke, Justyna and Haraldsson, Saemundur O. and Harman, Mark and Langdon, William B. and White, David R. and Woodward, John R. 2017. Genetic improvement of software: a comprehensive survey. *Transactions on Evolutionary Computation* 22, 3 (2017), 415–432.

Petke, Justyna and Harman, Mark and Langdon, William B. and Weimer, Westley. 2014. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In *European Conference on Genetic Programming*. Springer, Berlin, Heidelberg, 137–149.

Roohitavaf, Mohammad and Zhu, Ling and Kulkarni, Sandeep and Biswas, Subir. 2018. Synthesizing customized network protocols using genetic programming. In *Genetic and Evolutionary Computation Conference Companion*. ACM, New York, NY, 1616–1623.

Sati, Salem and Ahmad, Khaleel. 2020. Efficient evict policy for PRoPHET. *International Journal of Information Technology* 12, 1 (2020), 251–260.

Saudi, Nur Amirah Mohd and Arshad, Mohamad Asrol and Buja, Alya Geogiana and Fadzil, Ahmad Firdaus Ahmad and Saidi, Raihana Md. 2019. Mobile ad-hoc network (MANET) routing protocols: A performance assessment. In *International Conference on Computing, Mathematics and Statistics*. Springer, Berlin, Heidelberg, 53–59.

Schulte, Eric M. and Weimer, Westley and Forrest, Stephanie. 2015. Repairing COTS router firmware without access to source code or test suites: A case study in evolutionary software repair. In *Genetic and Evolutionary Computation Conference Companion*. ACM, New York, NY, 847–854.

Sharples, Nicholas and Wakeman, Ian. 2000. Protocol construction using genetic search techniques. In *Workshops on Real-World Applications of Evolutionary Computation*. Springer, Berlin, Heidelberg, 235–246.

Siyari, Payam and Dilkina, Bistra and Dovrolis, Constantine. 2017. Emergence and evolution of hierarchical structure in complex systems. In *Dynamics on and of Complex Networks*. Springer, Berlin, Heidelberg, 23–62.

Spyropoulos, Thrasyvoulos and Psounis, Konstantinos and Raghavendra, Cauligi S. 2005. Spray and wait: an efficient routing scheme for intermittently connected mobile networks. In *SIGCOMM workshop on Delay-tolerant networking*. ACM, New York, NY, 252–259.

Stampa, Giorgio and Arias, Marta and Sánchez-Charles, David and Muntés-Mulero, Victor and Cabellos, Albert. 2017. A deep-reinforcement learning approach for software-defined networking routing optimization. arXiv preprint arXiv:1709.07080.

Su, Yi and Van Der Schaar, Mihaela. 2010. Dynamic conjectures in random access networks using bio-inspired learning. *Journal on Selected Areas in Communications* 28, 4 (2010), 587–601.

Tao, Nigel and Baxter, Jonathan and Weaver, Lex. 2001. A multi-agent, policy-gradient approach to network routing. In *International Conference on Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, 553–560.

Thong, Lee K. 2004. Performance analysis of mobile ad hoc networking routing protocols.

Tschudin, Chr and Yamamoto, Lidia. 2005. Self-evolving network software. *Praxis der Informationsverarbeitung und Kommunikation* 28, 4 (2005), 206–210.

Tschudin, Christian. 2003. Fraglets-a metabolistic execution model for communication protocols. In *Symposium on Autonomous Intelligent Networks and Systems*. IEEE, New York, NY, 1–3.

Valencia, Philip and Lindsay, Peter and Jurdak, Raja. 2010. Distributed genetic evolution in WSN. In *International Conference on Information Processing in Sensor Networks*. ACM/IEEE, New York, NY, 13–23.

Van Belle, Werner and Mens, Tom and D'Hondt, Theo. 2003. Using genetic programming to generate protocol adaptors for interprocess communication. In *International Conference on Evolvable Systems*. Springer, Berlin, Heidelberg, 422–433.

Vardi, Moshe Y. 2018. The Siren Song of Temporal Synthesis. In *International Conference on Concurrency Theory*. Leibniz Center, Leibniz, 39.

Weimer, Westley and Forrest, Stephanie and Le Goues, Claire and Nguyen, ThanhVu. 2010. Automatic program repair with evolutionary computation. *Communications of the ACM* 53, 5 (2010), 109–116.

Weise, Thomas and Geihs, Kurt and Baer, Philipp A. 2007. Genetic programming for proactive aggregation protocols. In *International Conference on Adaptive and Natural Computing Algorithms*. Springer, Berlin, Heidelberg, 167–173.

Weise, Thomas and Tang, Ke. 2011. Evolving distributed algorithms with genetic programming. *Transactions on Evolutionary Computation* 16, 2 (2011), 242–265.

Weise, Thomas and Zapf, Michael and Geihs, Kurt. 2008. Evolving proactive aggregation protocols. In *European Conference on Genetic Programming*. Springer, Berlin, Heidelberg, 254–265.

Wilhelmstötter, Franz. 2017. Jenetics: Java genetic algorithm library (2017). http://jenetics.io

Wolpert, David and Tumer, Kagan and Frank, Jeremy. 1999. Using collective intelligence to route internet traffic. In *Advances in Neural Information Processing Systems*. MIT Press, Cambridge, MA, 952–960.

Woodward, John R and Johnson, Colin G and Brownlee, Alexander EI. 2016. GP vs GI: if you can't beat them, join them. In *Genetic and Evolutionary Computation Conference Companion*. ACM, New York, NY, 1155–1156.

Yamamoto, Lidia and Schreckling, Daniel and Meyer, Thomas. 2007. Self-replicating and self-modifying programs in fraglets. In *Workshop on Bio-Inspired Models of Network, Information and Computing Systems*. IEEE, New York, NY, 159–167.

Yamamoto, Lidia and Tschudin, Christian. 2005a. Experiments on the automatic evolution of protocols using genetic programming. In *Workshop on Autonomic Communication*. Springer, Berlin, Heidelberg, 13–28.

Yamamoto, Lidia and Tschudin, Christian. 2005b. Genetic evolution of protocol implementations and configurations. In *International Workshop on Self-Managed Systems and Services*. IFIP/IEEE, New York, NY, 34–2007070218786.

A GENERALIZATION OF THE EVOLVED PROTOCOLS

We performed additional experiments aimed at assessing the generalizability of the evolved protocols between different test cases, i.e., testing how a protocol evolved on a specific test case performs on another one. Due to the large number of combinations of test cases, we limited this analysis to six selected cases that we considered representative of the generalization capabilities of the evolved protocols, i.e., covering at least one case for each protocol, map, and number of hosts per group. Also in this case, each test case was simulated 10 times.

Table 10 shows the results of the test cases considered in this part of the experimentation. From the table, it can be seen that the evolved protocols are able to generalize in the first four cases, where the performance (*Baseline*) of the protocol evolved on the *Target test case* is statistically equivalent (p-value $> \alpha$) to the performance (*Tested*) of the protocol evolved on the *Source test case*. In the remaining two cases, the *Tested* performance is statistically lower (p-value $\leq \alpha$) than the *Baseline* one, meaning that the protocol specifically evolved on the *Target test case* shows a higher delivery probability than that of the protocol evolved on the *Source test case*. In particular, it appears that the Manhattan cases are harder to generalize (both from and to), see the last two rows in the table. This is likely due to the lower node density that characterizes the Manhattan cases (on the effect of density on the delivery probability in Manhattan-like test cases, see e.g. [Thong, Lee K 2004]), that is comparably lower than in the two other maps: observing Table 4 and Figure 3 in the main text, it is worth highlighting the fact that the world size (i.e, the size of the map) in the Manhattan case is much larger than in the Default map, while it is the same as in the Helsinki map, although the latter presents way less roads and thus nodes therein have a much more confined mobility. This is likely the same reason that makes it harder to improve the performance of Epidemic and PRoPHET protocols on the Manhattan map.

Finally, it should be noted that apart from the Manhattan cases, in all the other cases the *Tested* performances are still much higher than those of the baseline Epidemic and PRoPHET protocols. In fact, comparing the delivery probability shown in the *Tested* column in the first four rows in Table 10 with the corresponding results shown in the main text (Tables 5-6), it results: 0.2874 vs 0.1798, 0.3260 vs 0.2542, 0.2519 vs 0.2047, 0.3641 vs 0.2307. In the Manhattan cases, the comparison results instead in 0.2075 vs 0.2041 and 0.1493 vs 0.1685, respectively for the last two rows of Table 10.

Table 10. Results of the generalizability experiments (median across 10 simulations). The *Baseline* column indicates the delivery probability obtained by the best routing protocol evolved on the *Target test case*, while the *Tested* column indicates the delivery probability obtained by the best routing protocol evolved on the *Source test case* when this is applied to the *Target test case*. For each experiment, we report the p-value of the Wilcoxon rank-sum test ($N = 10, \alpha = 0.05$). The evolved protocols are able to generalize in the first four cases (no statistical difference between *Baseline* and *Tested*), while they are not in the remaining ones.

Source test case	**Target test case**	**Baseline**	**Tested**	**p-value**
Epidemic, Default (40 hosts)	Epidemic, Helsinki (100 hosts)	0.2887	0.2874	0.726
Epidemic, Helsinki (100 hosts)	Epidemic, Default (40 hosts)	0.3342	0.3260	0.093
PRoPHET, Default (100 hosts)	PRoPHET, Helsinki (40 hosts)	0.2447	0.2519	0.878
PRoPHET, Helsinki (40 hosts)	PRoPHET, Default (100 hosts)	0.3829	0.3641	0.022
Epidemic, Manhattan (40 hosts)	Epidemic, Default (100 hosts)	0.3764	0.2075	0.005
Epidemic, Default (100 hosts)	Epidemic, Manhattan (40 hosts)	0.1654	0.1493	0.005

B ANALYSIS OF THE EVOLVED PROTOCOLS

Comparing the logics (in the form of GP tree) of the baseline protocols, shown in Figures 4-5 in the main text, with that of the best evolved protocols, shown in Figures in 13-24 in Appendix C, it can be noted that the evolved protocols are quite different from the baselines. In particular, apart from the case of Epidemic on the Default map with 40 hosts per group (Figure 13), the evolved protocols appear more complex, in terms of number of nodes and depth, than their corresponding baseline. The main difference in the routing logic consists in the condition in which the evolved protocols can send messages. In the baseline protocols, `isTransferring()` is a condition that does not allow the transmission of messages. Instead, in most of the evolved protocols, when `isTransferring()` is true, the protocol still tries to transmit its messages (i.e., puts them in the sending buffer). A second important difference is the number of attempts of transmissions that the evolved protocols perform at each update. In fact, the baseline protocols perform at most two attempts per update, while the evolved protocols perform up to 4 attempts.

To gain further insight into the comparison between the behavior of the baseline and the best evolved protocols, we also analyzed the number of messages and their status during the simulations. Also this information is provided as an output of The ONE simulations[3]. Figures 11-12 display the mean ± std. dev. (across 10 simulations of the baseline vs best evolved protocol, log scale), of the number of messages marked as created, started, relayed, aborted, dropped, removed and delivered, respectively for Epidemic and PRoPHET. It can be seen that the number of removed messages is zero, by construction of the baseline protocols. As for the other values, while the number of created messages is obviously the same in all simulations, the most interesting thing to note is that, apart from the Manhattan simulations, the number of started/relayed/aborted/dropped messages is consistently (up to two orders of magnitude) lower for the evolved protocols than for the baseline ones, which is consistent with the dramatically reduced overhead and average hop count observed in the Default and Helsinki test cases. Furthermore, the reduced number of dropped messages associated with the higher buffer time observed before, seems to suggest that in practice the evolved protocols do drop less messages, but these are kept much longer in the buffers.

Combining these observations with the trade-off analysis discussed in the main text (Section 5.2), we can try to identify the reason for the improved delivery probability of the evolved protocols, at least in the Default and Helsinki test cases. On the one hand, the higher number of transmission attempts per update intuitively leads to a higher delivery probability: the more retransmissions a node tries, the higher the chance the messages in its local buffer will be eventually delivered. Furthermore, allowing the protocol to transmit also when `isTransferring()` is true makes it even more likely to transmit messages to nodes that are one hop away from their destination, which is consistent with the close to 1 average hop count measured with the evolved protocols. In practice, since nodes attempt more frequent transmissions than the baseline protocols (and this happens *while the nodes are moving*), messages are more likely to be delivered with just one hop. On the other hand, contrarily to what we would have expected, this increased number of transmissions per update does not reflect in a higher number of started/relayed messages (and thus an increased

[3]In particular, the simulator defines seven possible message status: *created* (i.e., when new message is created at its source node), *started* (i.e., when a message transmission starts, either from the source or from an intermediate hop), *relayed* (i.e., when a message transmission to any receiving node, either the destination or an intermediate node, is completed; note that this check is done on a receiving node), *aborted* (i.e., when a message transmission fails, i.e., not all the bytes are received correctly; also this check is done on a receiving node), *dropped* (i.e., when a message, oldest first, is deleted from a node's buffer to make room for a new incoming message), *removed* (i.e., when a message is deleted from a node's buffer since it has already reached its destination; note however that this information is not available in the Epidemic and PRoPHET protocols), and *delivered* (i.e., the message is relayed and reaches its destination; the ratio *delivered*/*created* is the delivery probability considered throughout this study).

overhead), or even less so aborted ones, but rather in a lower number of messages that are kept in the buffer for a longer time (as shown by the increased buffer time), before they get dropped due to new incoming messages. This longer buffer time is the cause of the observed increased latency.

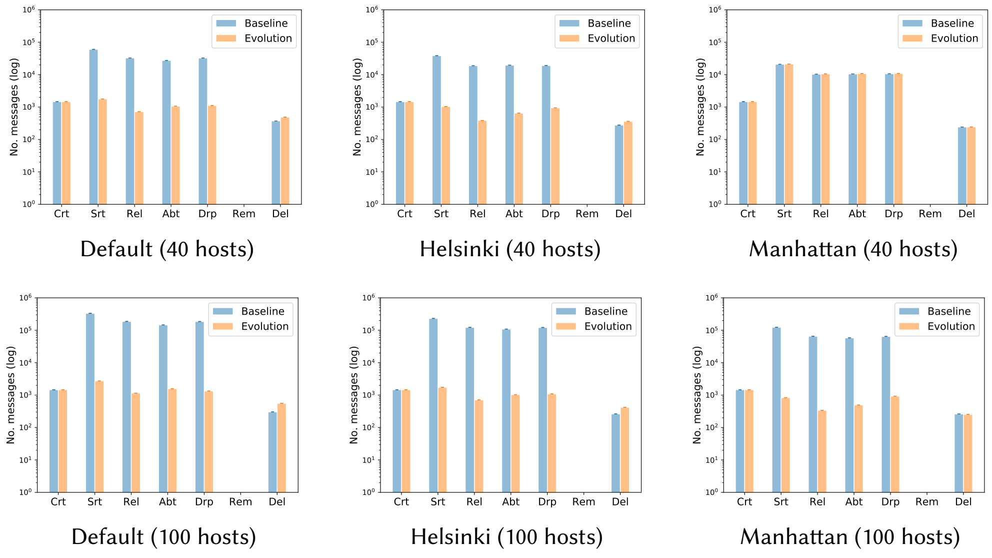


Fig. 11. Number of created (Crt), started (Srt), relayed (Rel), aborted (Abt), dropped (Drp), removed (Rem) and delivered (Del) messages with the Epidemic routing protocol (*Baseline*) and the best evolved protocols (*Evolution*), mean ± std. dev. (log scale) across 10 simulations.

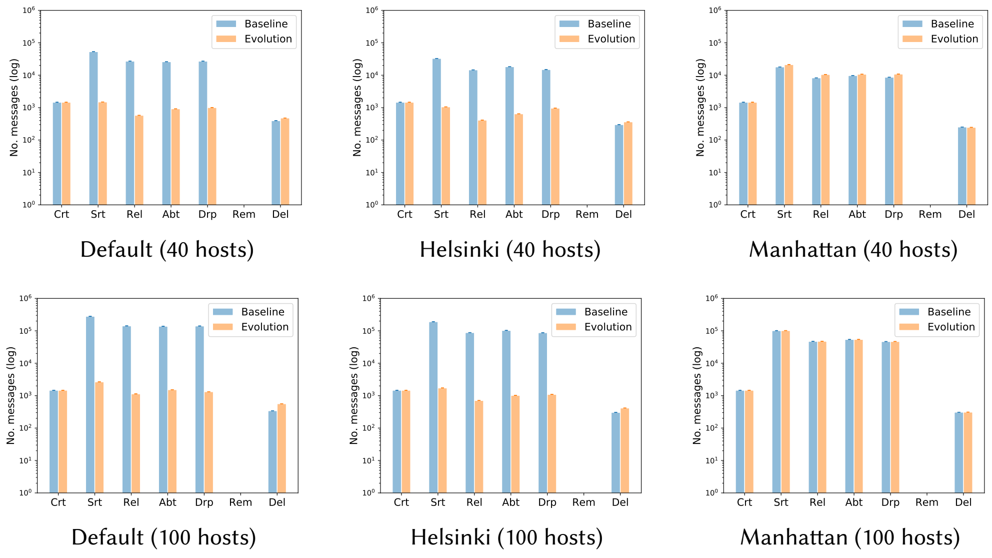


Fig. 12. Number of created (Crt), started (Srt), relayed (Rel), aborted (Abt), dropped (Drp), removed (Rem) and delivered (Del) messages with the PRoPHET routing protocol (*Baseline*) and the best evolved protocols (*Evolution*), mean ± std. dev. (log scale) across 10 simulations.

C BEST EVOLVED TREES

We report below the best performing trees evolved by Genetic Programming in all the test cases considered in our experimentation, namely all the combinations of ⟨protocol, map, number of hosts per group⟩ in: {Epidemic, PRoPHET} × {Default, Helsinki, Manhattan} × {40 hosts, 100 hosts}.

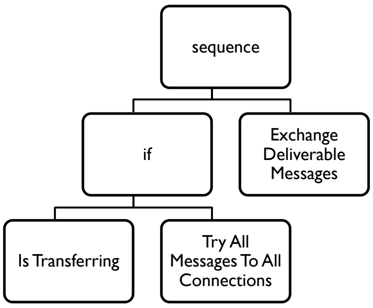

sequence
if
Exchange Deliverable Messages
Is Transferring
Try All Messages To All Connections

Fig. 13. Best evolved tree in the test case: Epidemic, Default map and 40 hosts per group.

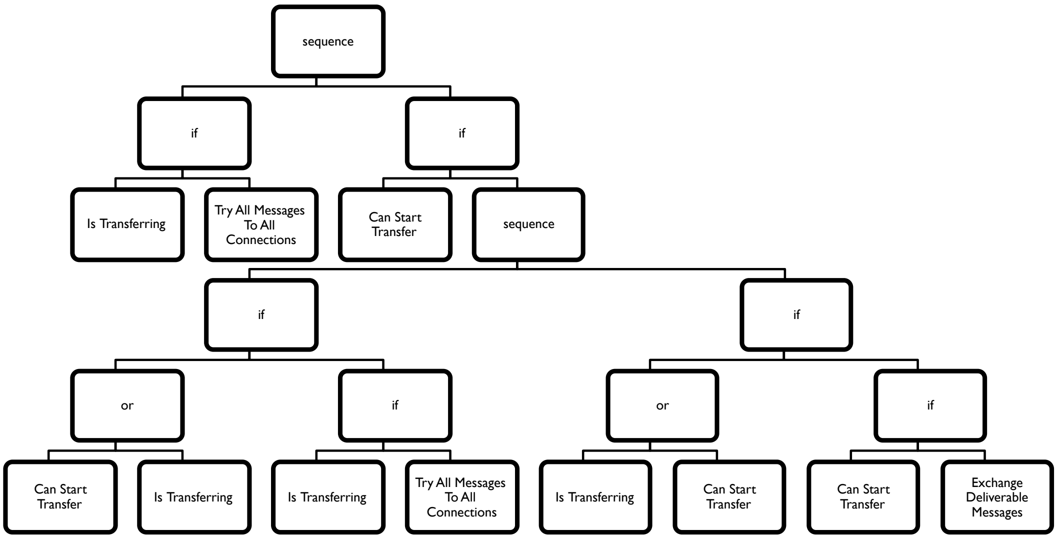


Fig. 14. Best evolved tree in the test case: Epidemic, Default map and 100 hosts per group.

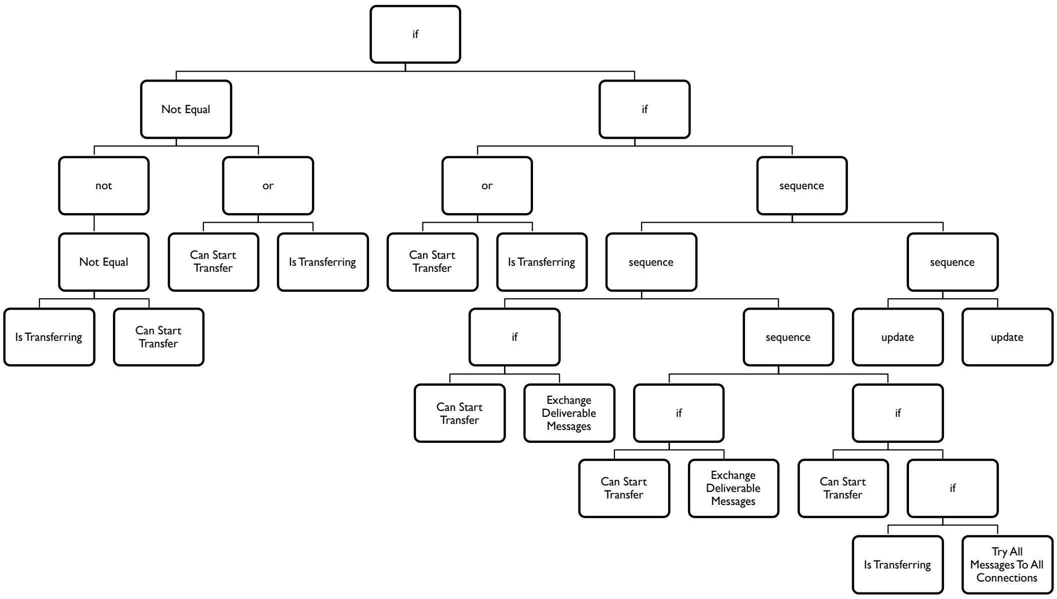


Fig. 15. Best evolved tree in the test case: Epidemic, Helsinki map and 40 hosts per group.

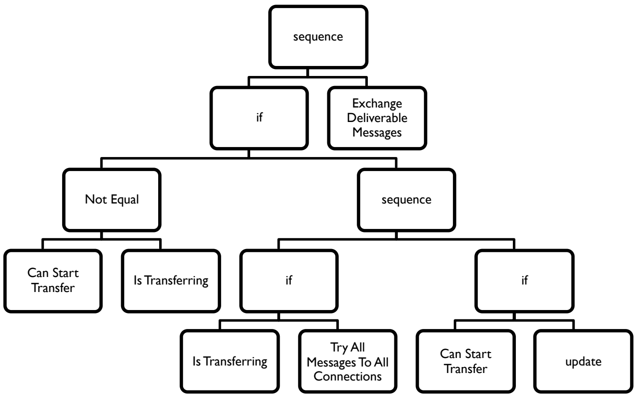


Fig. 16. Best evolved tree in the test case: Epidemic, Helsinki map and 100 hosts per group.

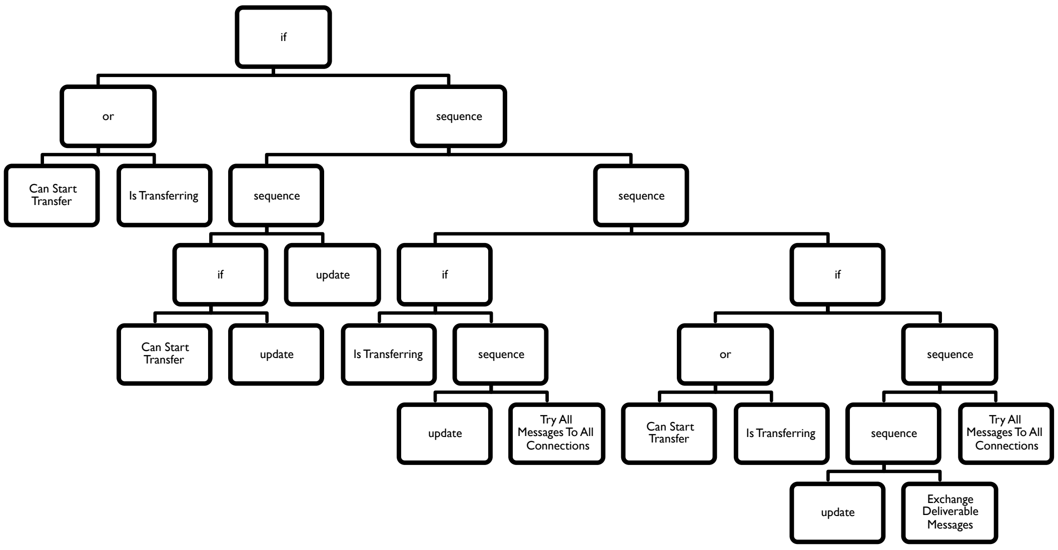


Fig. 17. Best evolved tree in the test case: Epidemic, Manhattan map and 40 hosts per group.

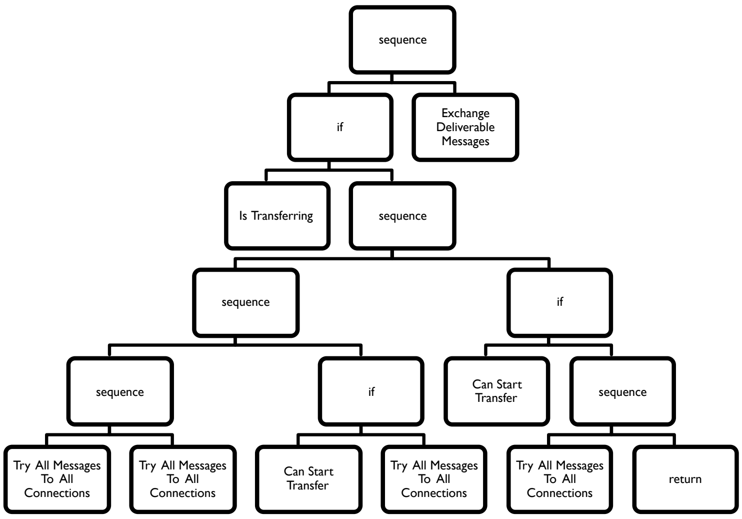

sequence
if
Exchange Deliverable Messages
Is Transferring
sequence
sequence
if
sequence
if
Can Start Transfer
sequence
Try All Messages To All Connections
Try All Messages To All Connections
Can Start Transfer
Try All Messages To All Connections
Try All Messages To All Connections
return

Fig. 18. Best evolved tree in the test case: Epidemic, Manhattan map and 100 hosts per group.

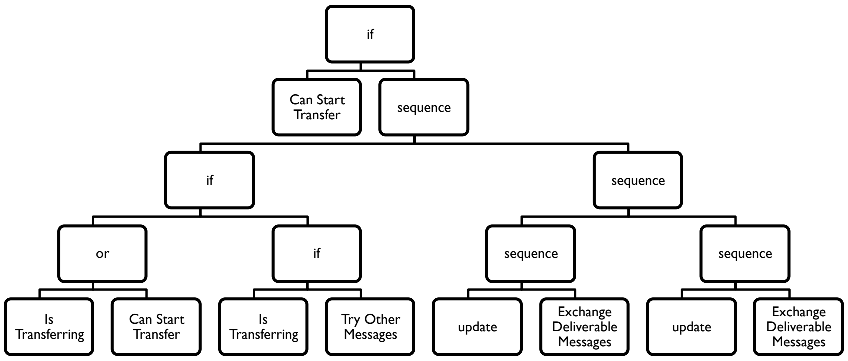


Fig. 19. Best evolved tree in the test case: PRoPHET, Default map and 40 hosts per group.

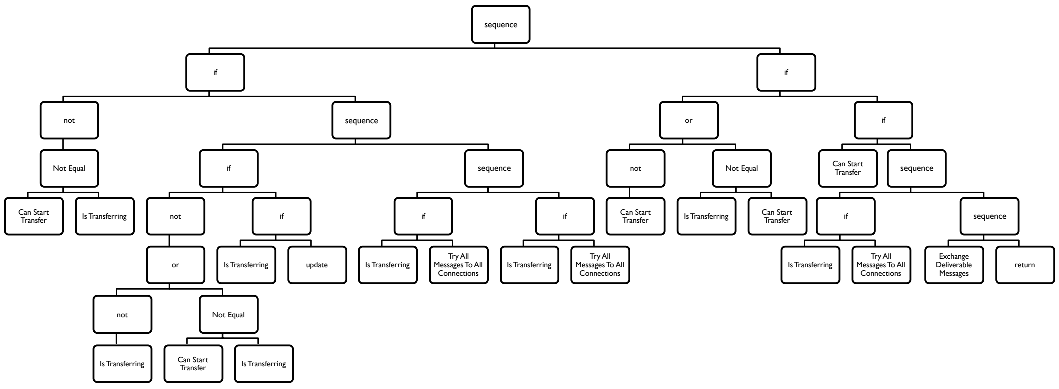


Fig. 20. Best evolved tree in the test case: PRoPHET, Default map and 100 hosts per group.

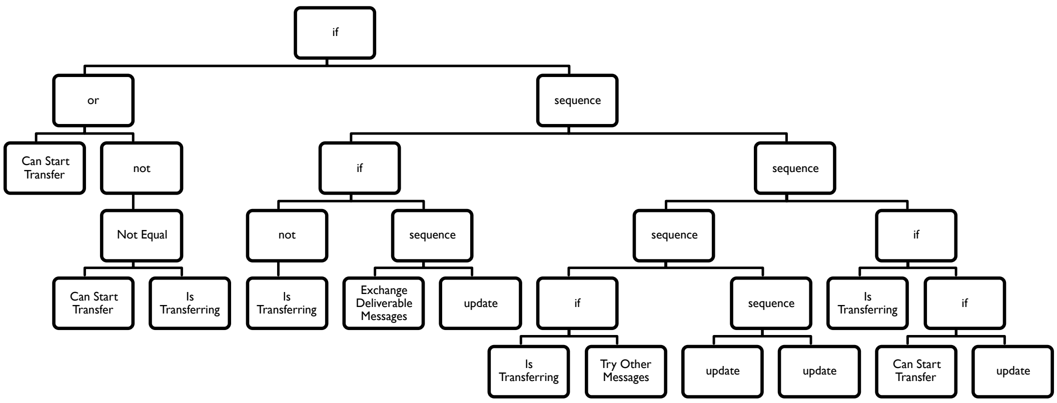


Fig. 21. Best evolved tree in the test case: PRoPHET, Helsinki map and 40 hosts per group.

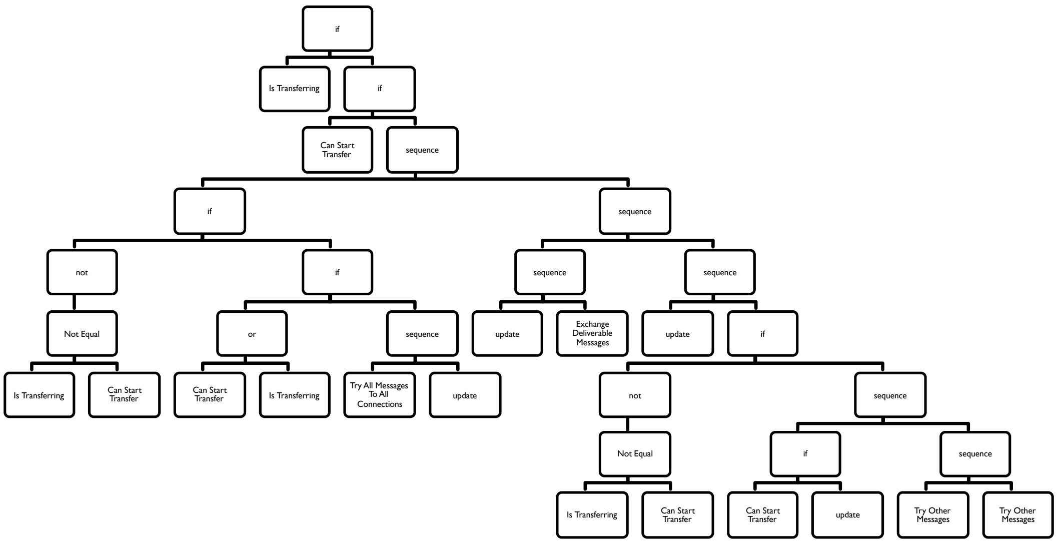


Fig. 22. Best evolved tree in the test case: PRoPHET, Helsinki map and 100 hosts per group.

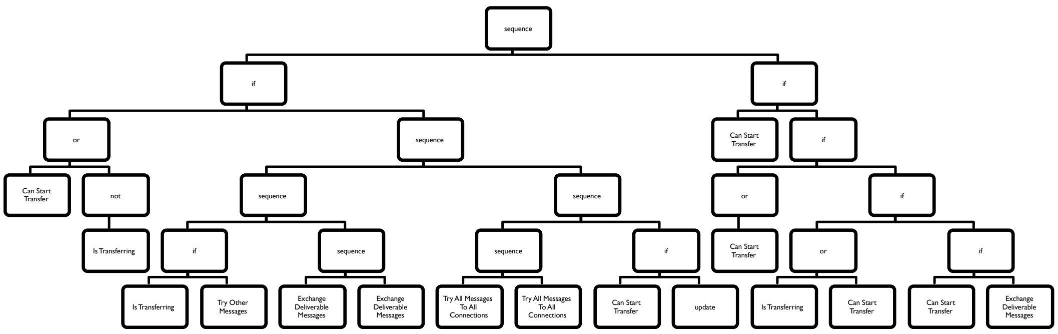


Fig. 23. Best evolved tree in the test case: PRoPHET, Manhattan map and 40 hosts per group.

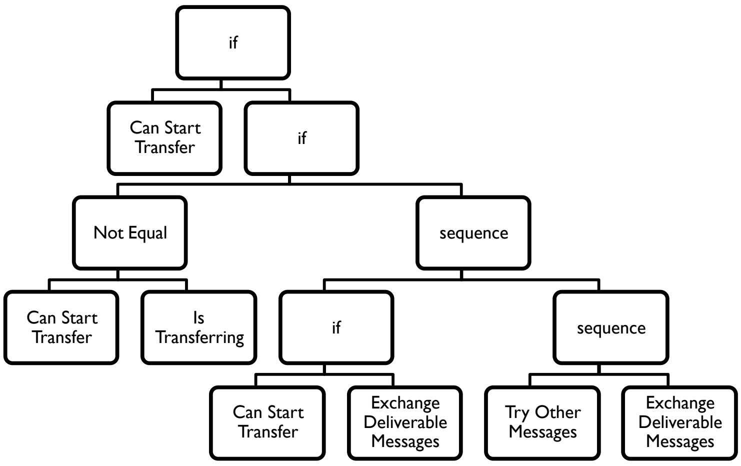


Fig. 24. Best evolved tree in the test case: PRoPHET, Manhattan map and 100 hosts per group.

D ROUTING PROTOCOL TEMPLATES

We report below the (simplified) class templates used for the Epidemic and PRoPHET routing protocols implemented in The ONE. We omit for brevity most comments and function implementations[4]. The `update()` method, highlighted in green, is the fragment of code improved by means of Genetic Programming, see Section 3 in the main text for details.

```java
/*
 * Copyright 2010 Aalto University, ComNet
 * Released under GPLv3. See LICENSE.txt for details.
 */
package routing;

import core.Settings;

/**
 * Epidemic message router with drop-oldest buffer and only single transferring
 * connections at a time.
 */
public class EpidemicRouter extends ActiveRouter {
    public EpidemicRouter(Settings s) {
        super(s);
    }

    protected EpidemicRouter(EpidemicRouter r) {
        super(r);
    }

    @Override
    public void update() {
        super.update();
        if (isTransferring() || !canStartTransfer()) {
            return; // transferring, don't try other connections yet
        }
        // Try first the messages that can be delivered to final recipient
        if (exchangeDeliverableMessages() != null) {
            return; // started a transfer, don't try others (yet)
        }
        // then try any/all message to any/all connection
        this.tryAllMessagesToAllConnections();
    }

    @Override
    public EpidemicRouter replicate() {
        return new EpidemicRouter(this);
    }
}
```

Listing 1. EpidemicRouter class template

[4]The complete code implementation is available at https://github.com/akeranen/the-one/tree/master/src/routing.

```java
/*
 * Copyright 2010 Aalto University, ComNet
 * Released under GPLv3. See LICENSE.txt for details.
 */
package routing;

import java.util.ArrayList;
// other imports
// ...

/**
 * Implementation of PRoPHET router as described in <I>Probabilistic routing
 * in intermittently connected networks</I> by Anders Lindgren et al.
 */
public class ProphetRouter extends ActiveRouter {
    // public and private fields
    // ...

    public ProphetRouter(Settings s) {
        super(s);
        //...
    }

    protected ProphetRouter(ProphetRouter r) {
        super(r);
        //...
    }

    private void initPreds() { /* ... */ }

    @Override
    public void changedConnection(Connection con) { /* ... */ }

    private void updateDeliveryPredFor(DTNHost host) { /* ... */ }

    public double getPredFor(DTNHost host) { /* ... */ }

    private void updateTransitivePreds(DTNHost host) { /* ... */ }

    private void ageDeliveryPreds() { /* ... */ }

    private Map<DTNHost, Double> getDeliveryPreds() { /* ... */ }

    @Override
    public void update() {
        super.update();
        if (!canStartTransfer() ||isTransferring()) {
            return; // nothing to transfer or is currently transferring
        }
        // try messages that could be delivered to final recipient
        if (exchangeDeliverableMessages() != null) {
            return;
        }
        tryOtherMessages();
    }

    private Tuple<Message, Connection> tryOtherMessages() { /* ... */ }

    private class TupleComparator implements Comparator <Tuple<Message, Connection>> { /*
     ... */ }

    @Override
    public RoutingInfo getRoutingInfo() { /* ... */ }

    @Override
    public MessageRouter replicate() { /* ... */ }
}
```

Listing 2. ProphetRouter class template

E THE ONE CONFIGURATION FILE

We report below the configuration files used in The ONE simulations. We highlight in green the settings that are modified (depending on the specific test case) during the evolutionary runs. We omit the parameters related to The ONE GUI, which are not relevant for our experimentation.

```
Scenario.name = default_scenario
Scenario.simulateConnections = true
Scenario.updateInterval = 0.1
Scenario.endTime = 43200

btInterface.type = SimpleBroadcastInterface
btInterface.transmitSpeed = 250k
btInterface.transmitRange = 10

highspeedInterface.type = SimpleBroadcastInterface
highspeedInterface.transmitSpeed = 10M
highspeedInterface.transmitRange = 1000

Scenario.nrofHostGroups = 6

Group.movementModel = ShortestPathMapBasedMovement
Group.router = EpidemicRouter
Group.bufferSize = 5M
Group.waitTime = 0, 120
Group.nrofInterfaces = 1
Group.interface1 = btInterface
Group.speed = 0.5, 1.5
Group.msgTtl = 300

Group.nrofHosts = 40

Group1.groupID = p

Group2.groupID = c
Group2.okMaps = 1
Group2.speed = 2.7, 13.9

Group3.groupID = w

Group4.groupID = t
Group4.bufferSize = 50M
Group4.movementModel = MapRouteMovement
Group4.routeFile = data/tram3.wkt
Group4.routeType = 1
Group4.waitTime = 10, 30
Group4.speed = 7, 10
Group4.nrofHosts = 2
Group4.nrofInterfaces = 2
Group4.interface1 = btInterface
Group4.interface2 = highspeedInterface

Group5.groupID = t
Group5.bufferSize = 50M
Group5.movementModel = MapRouteMovement
Group5.routeFile = data/tram4.wkt
Group5.routeType = 2
Group5.waitTime = 10, 30
Group5.speed = 7, 10
Group5.nrofHosts = 2

Group6.groupID = t
Group6.bufferSize = 50M
Group6.movementModel = MapRouteMovement
Group6.routeFile = data/tram10.wkt
Group6.routeType = 2
Group6.waitTime = 10, 30
Group6.speed = 7, 10
Group6.nrofHosts = 2

```

```
Events.nrof = 1
Events1.class = MessageEventGenerator
Events1.interval = 25,35
Events1.size = 500k,1M
Events1.hosts = 0,126
Events1.prefix = M

MovementModel.rngSeed = 1
MovementModel.worldSize = 4500, 3400
MovementModel.warmup = 1000

MapBasedMovement.nrofMapFiles = 4

MapBasedMovement.mapFile1 = data/roads.wkt
MapBasedMovement.mapFile2 = data/main_roads.wkt
MapBasedMovement.mapFile3 = data/pedestrian_paths.wkt
MapBasedMovement.mapFile4 = data/shops.wkt

Report.nrofReports = 2
Report.warmup = 0
Report.reportDir = reports/
Report.report1 = MessageStatsReport
Report.report2 = ContactTimesReport

ProphetRouter.secondsInTimeUnit = 30

Optimization.cellSizeMult = 5
Optimization.randomizeUpdateOrder = true
```

Listing 3. Default The ONE settings file