An Efficient Learning Assistant for a Contextual Road Navigation

Jean-Michel Ilie, Karim Lahiani, Ahmed-Chawki Chaouche, François Pêcheux

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2019) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

The 11th International Conference on Ambient Systems, Networks and Technologies (ANT)
April 6 - 9, 2020, Warsaw, Poland

An Efficient Learning Assistant for a Contextual Road Navigation

Jean-Michel Ilié[a,*], Karim Lahiani[a], Ahmed-Chawki Chaouche[b], François Pêcheux[a]

[a]*LIP6, UMR 7606 Sorbonne Université, 4 Place Jussieu, 75005 Paris, France*
[b]*MISC Laboratory, University Abdelhamid Mehri - Constantine 2, Campus Ali Mendjeli, 25000 Constantine, Algeria*

Abstract

Due to traffic conditions that are very context dependent, the computation of optimized or shortest paths is a very complex problem for both drivers and autonomous vehicles. In this paper, we introduce a learning mechanism that is able to efficiently evaluate path durations based on an abstraction of the available traffic information. We demonstrate that a cache data structure allows a permanent access to the results whereas a lazy politics taking new data into account is used to increase the viability of those results. Our measures highlight the performance of each mechanism, according to different learning strategies.

Keywords: Context awareness; Intelligent transportation systems; Learning systems; Cache memory; Path planning; Clustering

1. Introduction

One of today's most sensitive and hot topic of current industrial and academic research is smart and autonomous vehicles. This domain gave rise to numerous works and technological innovations, all concurring to assist the driver, relieve him from the stress of an environment that heavily depends on varying traffic conditions and improve his comfort. In this context, road map guidance applications like Google Maps or Waze, which compute in real-time an estimation of an optimized path to a given destination (e.g. [7]), are of paramount importance.

In this paper, we propose a method for estimating the time duration of a destination path with respect to a varying spatio-temporal context with partial information. As a use case of a rolling robot which must react rapidly to whatever road purpose event, it is about to approximate the computation time of the path estimations while ensuring the relevance. In fact, the most popular navigation engines use various traffic information to estimate the duration of a path [1, 7], however, in the case the access to some traffic information is only partial or missing, this analysis does not yield to a correct anticipation of future traffic situations. For instance, when day at school is over, the presence of a

* Corresponding author. Tel.: +33-678-139-432 ; fax: +33-144-277-495.
E-mail address: jean-michel.ilie@lip6.fr

traffic officer warranting the security of pupils and causing the vehicle traffic to slow down is for the moment hardly taken into account by Google and Waze algorithms.

It has been proven that traffic congestions are likely to occur and repeat themselves according to similar traffic and ambient conditions. In this paper, we propose a method to estimate path duration that takes advantage of past events and activities to model a pattern that can reliably used to represent future traffic and time delays. Thus, past experiences and their associated contexts are smoothly aggregated and jointly consolidate the model for a driving agent. The whole difficulty lies in a good use of accumulated experiences to estimate the durations and other performance indicators of the path to be taken. For instance, the accumulation of daily experiences for the previous school example will automatically make visible the impact of the presence of a traffic officer.

We are also interested in the problem of system responsiveness. In order to make the results available as quickly as possible, we propose to develop a highly competitive software architecture with pre-processing mechanisms.

This paper is organized as follows: Section 2 details the modular architecture if our approach. In Section 3, we develop a learning mechanism and discuss different strategies for estimating durations. In section 4, we evaluate the efficiency of our approach, from a qualitative and quantitative viewpoints. Section 5 concludes the paper and gives some perspectives.

2. Context Learning Architecture

Our software architecture is composed of several interactive and concurrent processes, that communicate altogether as clients or servers for other processes. The distributed processes communicate over TCP/IP and can run on separate hardware. The two essential components of the architecture are the Context process and the Learning process, which respectively allow to have access to the recorded experiments and to process them. So, any process of the architecture that wishes to obtain estimation on a given destination path sends a request the Learning process [2].

2.1. The Context Process

The Context process that manages the agent database makes extensive use of a Neo4j server[1], a NoSQL Database Management System (DBMS) allowing one a fast extraction of graph data. The provided data represents the road map (Map) and the past driving experiences, which are attached to two major models of roads:

- **Track:** typically represents a part of a road or an avenue in an urban context, independently of its global shape (straight line, curves, ...),
- **Intersection:** represents a crossroad, allowing tracks to cross each other.

In the rest of the paper, we generically note *vertex*, a track or an intersection of the Map. Moreover, each experience is situated, linked to an entity of the Map in the database. The experiences are not only spatially specified but also temporally, thus defining a global spatio-temporal context for each experience. An experience has the following global shape:

```
"actionType": "move",
"beginTime" : "16/10/2019 09:13:00",
"duration"  : 20
```

where,

- `actionType` is the action type that generated the experience,
- `beginTime` is the starting date of this experience,
- `duration` means the effective duration of the action in minute.

[1] https://neo4j.com/

2.2. The Learning process

The Learning process achieves the computation of duration and other performance indicators, with regards to some situated Map elements (*vertex*). Further, different strategies are highlighted in Section 3.2.

2.3. The Client Processes

Any client process can ask the Learning process to get information about some road path [3]. Here, we concentrate on the driving request message `GetPathMessage`, which has the following parameters:

```
messageParams = {
   "messageType": "GetPathMessage",
   "actionType" : "move"
   "at"         : "09/08/2019 10:25:00",
   "source"     : "0-0",
   "target"     : "2-2",
}
```

`actionType` represents the kind of action to be realized. `source` and `target` respectively represent the starting and ending points of a apth, and `at` corresponds to the selected date and time at which the path duration must be estimated. For instance, the message described above illustrates a `move` path request, starting from the intersection `"0-0"` and ending in `"2-2"`, knowing that the path duration should be estimated at `10h`, on the `09/08/2019`.

3. Learning Mechanism for Delay Estimations

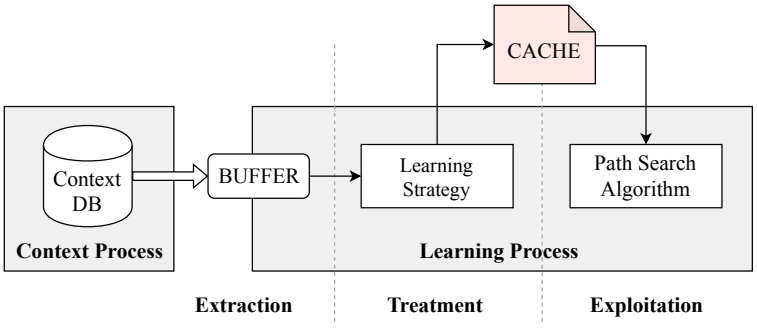

Fig. 1. Phases of the learning mechanism

Figure 1 represents the different processing phases performed by the Learning process, such that:

- **Extraction:** concerns the Map which can evolve over time as well as the located experiences which are also dynamically added in a *BUFFER*,
- **Process:** consists in estimating durations according to a learning strategy on the managed experiences,
- **Exploitation:** uses the computed duration values that have been put in a *CACHE*, to evaluate global path durations on the Map.

For sake of responsiveness and temporal efficiency, these three phases are each executed by a dedicated threads, thus offering a real parallelism of execution on a multicore hardware architecture.

3.1. Extraction Phase

In this part, we show how to manage data in order to extract contextually useful experiences. In addition, we propose a buffering mechanism to achieve efficiency and yield the results:

Data extraction. Our approach consists in estimating the quality of the experiences according to their *beginTime*. This allows us to introduce a time interval from a given time (further denoted $TIME$). As they are created, the experiences of a given action are firstly sorted spatially according to their location *vertex*, then sorted temporally with respect to the date *beginTime* of the experience. As a result, a queue of sorted experiences is defined for each vertex such that the stored experiences are organized according to a given periodic classification C (like daily, monthly, yearly, ...).

The selection of experiences from a queue can rely on the thorough study of the experiences temporally close to $TIME$ with respect to C. For instance, according to the daily classification, we can extract the closest experiences around 10 AM. We now introduce the function $mod : C \times \mathbb{R}^+ \rightarrow \mathbb{R}^+$, such that $mod_C(date)$ is the current date modulo the period C. A concept of proximity is introduced with the last parameter $Delta$ setting a time bound around $TIME$, hence the formula to select the appropriate experiences (exp) of a queue is: $|TIME - mod_C(exp.beginTime)| < Delta$

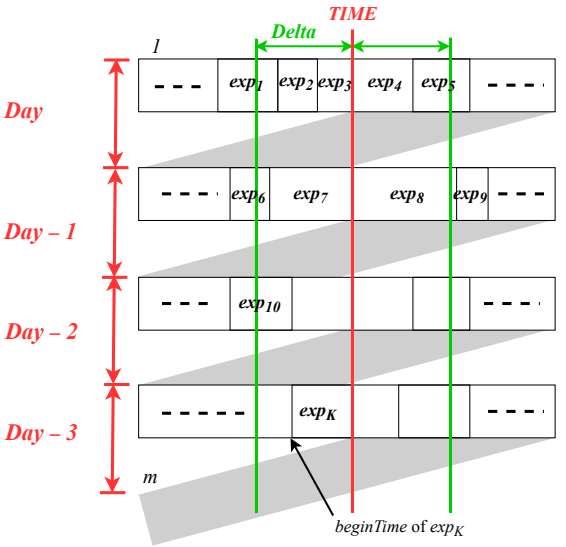

Fig. 2. Graphical view of past-experiences

As illustrated by Figure 2, the application of the modulo operation on an experiences queue seems as a "spiral ribbon" such that the ribbon rings correspond to successive periods. In a daily consideration and giving $Delta$, the selected experiences are $\{exp_2, exp_3, exp_4, exp_5, exp_7, exp_8, exp_K\}$, while the experiences $\{exp_1, exp_6, exp_9, exp_{10}\}$ are not considered because their *beginTime* is beyond the selectable area.

Input Buffering. The input buffer illustrated in Figure 1 plays an important role to ensure the existing computed data (the vertex transition durations) are always usable. This consists in postponing the use of the (too recent) newly created experiences so that to decrease the moments where the $CACHE$ is render unreachable for writing duration values purposes.

In order to maintain the soundness of the results, this also requires to concentrate on the computations which bring significant changes on the results. This involves the analysis of data of the newly experiences against the existing stored data [8]. However, for the sake of simplicity, we define a trigger effect ratio based on the respective numbers of newly created experiences $NEXP$ and the already selected by the extraction mechanism EXP, as follows:

$$\frac{|NEXP|}{|EXP|} > TRIG, \quad where\ TRIG \in \mathbb{R}^+$$

So, a new computation of vertex durations is triggered for a value $NEXP$ bigger than $TRIG * |EXP|$. Moreover when EXP is empty, we force EXP to the values in $NEXP$.

3.2. Treatment Phase

To be able to estimate the duration of a path, we must estimate first the transit duration of each *vertex* contained in that path. We have investigated four different computation strategies to reach this objective. All these strategies have the same inputs: a set of experiences named EXP coming from the data extraction concerning some *vertex* at a giving time ($TIME$) and with a daily periodic classification.

Mean Learning Strategy. The first strategy consists in a single average over the *duration* of EXP and results in a mean duration:

$$RESULT(vertex,time) = \frac{\sum_{i=0}^{n} EXP.duration_i}{|EXP|}$$

This simplistic strategy has some advantages, in particular a polynomial complexity on EXP size and a rapid computation.

Nevertheless, it should be noticed that this approach may present an overestimated bias of the actual vertex duration, because the relevance of one experience over another is not considered. Typically, an experience added 10 years ago has the same relevance as an experience recorded two days ago, which is not sound.

Weighted Mean Learning Strategy. With this strategy, a weight is associated to each experience so that the experiences closest to $TIME$ are more relevant than the others. The weight is computed based on the distance between *beginTime* of the experience and $TIME$. This distance is normalized and multiplied by the *duration* of the experience:

$$distance(vertex,time) = |TIME - exp(vertex,time).beginTime|, \quad \forall exp \in EXP$$

$$weights(vertex,time) = \frac{distance(vertex,time)}{\sum distance(vertex,time)}$$

$$RESULT(vertex,time) = \frac{\sum_{i=0}^{n} EXP.duration_i * weights(vertex,time)_i}{n}$$

Compared to the first strategy, results are more reliable while keeping a polynomial complexity. However, this strategy does not take the time distribution of the experiences into account. For example, some isolated experiences, with long durations (produced by a temporary traffic jam) and close to the beginning of the experiences queue, so with high weights, indeed hide other more relevant experiences.

Least Square Polynomial Fit Learning Strategy. From [4], we know that a linear regression can be computed over some set of distributed points, by applying the least squares method. For this strategy, we specify a linear regression over the experiences in order to better describe the distribution of the transit durations for any given *vertex*. Then, We can generate a formula having a 3rd degree polynomial, that is used to estimate the transit duration starting from $TIME$ on *vertex*:

$$f(vertex,time) = W0 + W1 * time + W2 * time^2 + W3 * time^3$$

$$e = \sum_{i=0}^{n} (f(vertex,EXP.beginTime_i) - EXP.duration_i)^2$$

$$argmin_e = [W0^*, W1^*, W2^*, W3^*]$$

$$RESULT(vertex,TIME) = f^*(vertex,TIME)$$

This strategy seems efficient for future estimates, moreover, uncontrollable isolated events are no longer a problem because the polynom strategy and its least squares method lets them ignored. The complexity is quadratic in the num-

ber of experiences. Nevertheless, we have noticed from our experiments that a set of experiences is not systematically correctly approximated by a linear model.

Trust Learning Strategy. Since the distribution of data may be arbitrary, we propose in this strategy to use a generic model based on an unsupervised learning method, the K-means algorithm which consists in partitioning the data into a number of k clusters. Combined with the "average silhouette" method we can evaluate the optimal k [6, 5].

To estimate the duration of a *vertex*, the centroid of each cluster is weighted by its size. Normalized then summed, this gives us an estimate of the duration at $TIME$. This method is relatively more complex than the previous ones , but as far as accuracy is concerned it is the more representative of the considered data in the general case.

3.3. Cached Duration Mechanism

The cached duration mechanism stores the duration of a given action and for each vertex of the Map over several time slots. The use of slots is necessary due to the impossibility of considering a continuous set of time values. Next, the stored structure ($CACHE$) is defined by the bijective function $Dur : ACT \times LOC \times CRE \rightarrow \mathbb{R}^+$ where ACT is a finite set of possible actions, LOC is a finite set of locations and CRE is a finite set of $\langle date, Delta \rangle$ pairs of $TIME \times TIME$, which respectively represent a date (with respect to a periodic classification C) and a time offset (like t and $Delta$ in Figure 2). Dur function gives the duration estimate as a result.

For each entry $\langle a, v, c \rangle$ of the Dur function, we apply a durations computation strategy (among the strategies discussed in Section 3.2), to the subset of experiences EXP (defined by the extraction method presented in Section 3.1) for classification C. The $CACHE$ structure allows us to quickly find these results hence to efficiently estimate the duration of a path in the Map.

3.4. Exploitation Phase

Since the durations of each *vertex* are now evaluated, we can compute the duration of the shortest path from a source position to a target destination. This evaluation is done on the Map where each *vertex* is tagged by the set of durations returned by the Dur function of the cached duration mechanism.

The algorithm 1 returns the shortest path of the Map, considering it as a graph described by the surrounding neighbor nodes:

$$Graph = \langle V, Neighbour(v \in V) \subset 2^V \rangle$$

It also takes into account the values that have been evaluated in the $CACHE$. The latter is a three-dimensional dictionary contextualizing the durations:

$$Action \times TimePeriod \times Location$$

The other inputs of $(beginTime,\ source,\ target,\ actionType)$ come from the $GetPathMessage$ request parameters and $explore$ parameter is an exploration probability. This algorithm is a variant of the classic Dijkstra one, consisting in the incremental development of a path on-the-fly, from *source* to the destination *target*. It is characterized by the three following points:

- The knowledge of the destination makes it possible to stop the course of the graph as soon as it is processed. The Dijkstra properties guarantee that at the end of processing with the *target* node, the computed path will be the shortest one,
- The weights of the graph nodes are contextually dynamic. Indeed, the weight of each node in the path is temporally relative to the date of departure from his father node and the duration to reach it,
- An exploration mechanism is included in the case where a *vertex* does not need any experience; if the duration of a *vertex* in the $CACHE$ has a zero value, this *vertex* can be explored according to the *explore* parameter.

Algorithm 1 Compute-path

Require:
$Closed := [\,]$
$Open := [source]$
$Dist := [(0, null, source)]$
$min := Integer.min$
while $target \notin Closed$ **do**
  for all $node \in Open$ **do**
    if $min > Dist(node)$ **then**
      $min := Dist(node)$
      $select := node$
    end if
  end for
  for all $node \in Neighbour(select)$ **do**
    if $cache[actionType, select, beginTime + Dist(select)] < 0$ **then**
      if $random.normal(0,1) > explore$ **then**
        $cache[actionType, select, beginTime + Dist(select)] := 0$
      else
        continue
      end if
    end if
    if $Dist(node) > Dist(select) + cache[actionType, select, beginTime + Dist(select)]$ **then**
      $Dist(node) := (Dist(select) + cache[actionType, select, beginTime + Dist(select)], node)$
      $Open := Open + node$
    end if
  end for
  $Closed := Closed + select$
  $Open := Open - select$
end while

4. Experiments and Results

We evaluate the specific advantages of the four mechanisms proposed in this paper. Plots of Figure 3 compare the calculation times of the four learning strategies, tested independently of the other structures, according to the size of data. For this purpose, new experiments are added iteratively (1000 packets at a time). The Trust strategy (K-means) is the most reliable but it has a cost compared to the other strategies, the calculation time of which seems negligible.

Figure 4 represents the response time of Learning process to 300 messages sent by some client process[2]. It is clear that the response time is negligible (less than 10ms) even if the Trust Learning Strategy is used. Actually, the three phases extraction-process-exploitation are supported by threads allowing a concurrent management of data. In particular, the exploitation by the algorithm 1 of some durations values in the *CACHE* is only constrained when the durations are actually written in the *CACHE*. As this write operations are limited, the *CACHE* is almost always available, hence the response to a client depends only on the exploration phase.

The two peaks that appear on the 100th and 200th messages each correspond to the computation and writing of the duration values in the *CACHE*. For this, 1000 experiments are added at the 100th message so as to saturate the buffer and restart the cached duration computation mechanism, then 1000 others experiences are added at the 200th message. When the buffer is saturated, the response time to a client message does not only depend on the exploration phase but also on the process one due to the fact the *CACHE* is not accessible during its update.

[2] Our approach is evaluated on a computer whose configuration is: CPU: Intel i7-6600U @ 2.60Ghz, RAM: DDR4-2133Mhz 16GB, GPU: Intel HD 520, Disk: SSD 512GB M.2 SATA - Max. Read/Write Transfer Rate 510MB/s

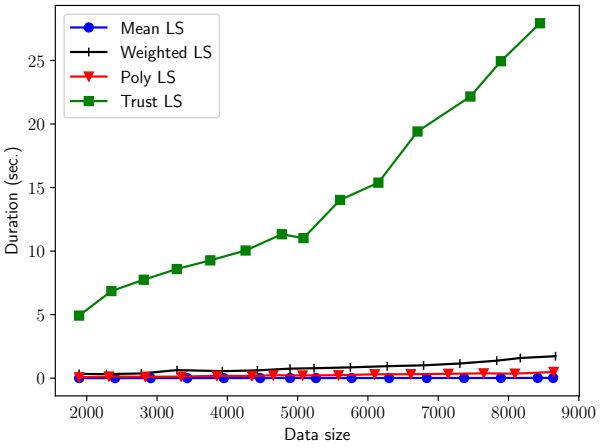

Fig. 3. Calculation times comparison of the 4 learning strategies

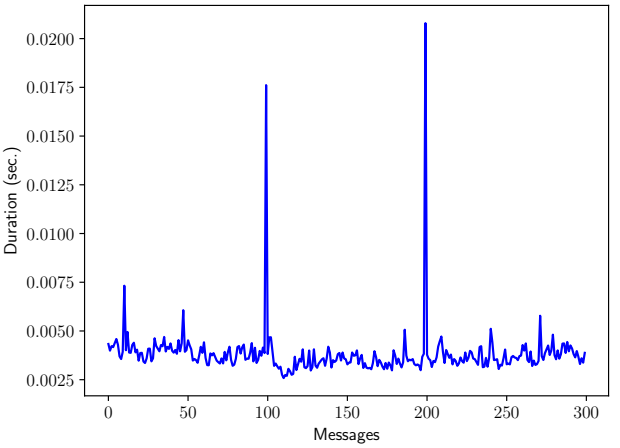

Fig. 4. Response time of Learning process for a *getPathMsg* using the Trust learning strategy

The interest of having an input buffer is to make the occurrences of such peaks as scarce as possible by postponing the inputs of new experiences. This has the effect of reducing the rate of launching the cached duration mechanism. From a qualitative point of view, it makes it possible to lengthen the durability and the viability of the calculated results. On the contrary, without buffer, the addition of an experience would restart the mechanism, without necessarily changing the estimate durations. It is worth noticing that the addition of many new experiences at the 100th and 200th messages, causes a new calculation of durations but the impact on the response time of the Learning process to the client remains very localized comparing to the strategy used.

5. Discussion and Conclusion

The search of a best path over a road map submitted to an ambient traffic is an interesting challenge that we have investigated according to a learning approach. The proposed mechanism exploits the past experiences durations of the driver agents in order to abstract the traffic fluctuation and estimate the overall duration of such paths.

By focusing on the move aspects over the road tracks, the corner point of our approach is not only to benefit from the parallelism of execution, used to reduce the response time of our learning mechanism impacted by the computation of the section durations, but also to qualify these durations according to a spatio-temporal classification directly usable in several time slots in the day.

Among the immediate perspectives, we will test different trigger functions for the input buffer in order to optimize the relevance of results. Moreover, it would be worth comparing our approach against existing road map guidance applications, typically Google Maps and Waze which exploit both traffic and user information. In case of long distance travels where such information cannot be end-to-end guaranteed, we guess we can obtain better duration accuracy.

References

[1] Blamont, J., 2018. Networks!: The Bet on Collective Intelligence. Springer.
[2] Chaouche, A.C., El Fallah Seghrouchni, A., Ilié, J.M., Saïdouni, D., 2015. Improving the Contextual Selection of BDI Plans by Incorporating Situated Experiments, in: AIAI. Springer. volume 458 of *IFIP Advances in Information and Communication Technology*, pp. 266–281.
[3] Chaouche, A.C., El Fallah Seghrouchni, A., Ilié, J.M., Saïdouni, E.D., 2016. Learning from situated Experiences for a Contextual Planning Guidance. Journal of Ambient Intelligence and Humanized Computing 7, 555–566.
[4] Howard, M., Workman, J., 2018. Chapter 112 - Choosing the Best Regression Model, in: Chemometrics in Spectroscopy, pp. 851 – 861.
[5] Likas, A., Vlassis, N., Verbeek, J.J., 2003. The Global K-Means Clustering Algorithm. Pattern Recognition 36-2, 451 – 461.
[6] Pedregosa, F.e.a., 2011. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research 12, 2825–2830.
[7] Silva, T.H., de Melo, P.O.S.V., Viana, A.C., Almeida, J.M., Salles, J., Loureiro, A.A.F., 2013. Traffic condition is more than colored lines on a map: Characterization of waze alerts, in: Social Informatics, Springer. pp. 309–318.
[8] Webb, G.I., 2010. Lazy Learning. Springer US, Boston, MA. pp. 571–572.