Journal of Machine Learning Research 10 (2009) 1851-1880 Submitted 4/08; Revised 11/08; Published 8/09

CarpeDiem: Optimizing the Viterbi Algorithm and Applications to Supervised Sequential Learning

Roberto Esposito ESPOSITO@DI.UNITO.IT
Daniele P. Radicioni RADICION@DI.UNITO.IT
Department of Computer Science
University of Turin
Corso Svizzera, 185 - 10149 Turin - Italy

Editor: Michael Collins

Abstract

The growth of information available to learning systems and the increasing complexity of learning tasks determine the need for devising algorithms that scale well with respect to all learning parameters. In the context of supervised sequential learning, the Viterbi algorithm plays a fundamental role, by allowing the evaluation of the best (most probable) sequence of labels with a time complexity linear in the number of time events, and quadratic in the number of labels.

In this paper we propose CarpeDiem, a novel algorithm allowing the evaluation of the best possible sequence of labels with a sub-quadratic time complexity.[1] We provide theoretical grounding together with solid empirical results supporting two chief facts. CarpeDiem always finds the optimal solution requiring, in most cases, only a small fraction of the time taken by the Viterbi algorithm; meantime, CarpeDiem is never asymptotically worse than the Viterbi algorithm, thus confirming it as a sound replacement.

Keywords: Viterbi algorithm, sequence labeling, conditional models, classifiers optimization, exact inference

1. Introduction

In supervised learning systems, classifiers are learnt from sets of labeled examples and then used to predict the "correct" labeling for new objects. According to how relations between objects are exploited to build and evaluate the classifier, different categories of learning systems can be individuated. When the learning system deals with examples as isolated individuals, thus disregarding any relation among them, the system is said to work in a *propositional* setting. In this case classifiers can optimize the assignment of labels individually. Instead, in the setting of supervised sequential learning (SSL) the objects are assumed to be arranged in a sequence: relationships between previous and subsequent objects exist, and are used to improve the classification accuracy. SSL classifiers are then required to find the globally optimum sequence of labels, rather than the sequence of locally optimal labels. For instance, in the optical character recognition task, the labelling "*learning*" is probably better than "*learn1ng*", even though the description of the sixth character taken in isolation might suggest otherwise. A SSL classifier may deal with such ambiguities by exploiting the

1. The implementation of CarpeDiem and of several sequence learning algorithms can be downloaded at:
http://www.di.unito.it/~esposito/Software/seqlearning.tar.gz
a working GUI (Mac OS X only) for experimenting with the software can be downloaded at:
http://www.di.unito.it/~esposito/Software/SequenceLearningExperimenterBinaries.zip.

higher sequential correlation, in the English language, of the bigram *in* with respect to *1n*. Conceptually, given a sequence of T observations and K possible labels, K^T possible combinations of labels are to be considered by SSL classifiers. Most systems deal with such complexity by assuming that relations may span only over nearby objects and use the Viterbi algorithm (Viterbi, 1967) to find the globally optimal sequence of labels in $\Theta(TK^2)$ time.

In the last few years it has become increasingly important for supervised sequential learning algorithms to handle problems with large state spaces (Dietterich et al., 2008). Unfortunately, even the drastic reduction in complexity achieved by the Viterbi algorithm may be not sufficient in such domains. For instance, this is the case of web-logs related tasks (Felzenszwalb et al., 2003), music analysis (Radicioni and Esposito, 2007), and activity monitoring through body sensors (Siddiqi and Moore, 2005), where the number of possible labels is so large that the classification time can grow prohibitively high.

Some recent works propose techniques that under precise assumptions allow faster execution time of classifiers based on hidden Markov models (HMMs) (Rabiner, 1989). One feature shared by these approaches is the assumption that the transition matrix has a specific form allowing one to rule out most transitions. Such approaches are highly valuable when the problem naturally fits the assumption; *vice versa* they either lose the optimal solution or cannot be applied at all, when it does not. Moreover, they assume the transition matrix to be known beforehand and fixed over time. While this is a natural assumption in HMMs, recent algorithms based on the boolean features framework (McCallum et al., 2000) allow for more general settings where the transition matrix is itself a function of the observations around the object to be labelled. In such cases it is hard to figure out how the aforementioned approaches apply.

In this paper we introduce `CarpeDiem`. It is a parameter-free algorithm, sporting best case sub-quadratic complexity, devised as a replacement for the Viterbi algorithm. `CarpeDiem` avoids considering a transition whenever local observations make it impossible for the transition to be part of the optimal path. `CarpeDiem` preserves the optimality of the result, never being asymptotically worse than the Viterbi algorithm. Moreover, `CarpeDiem` automatically adapts to the sequence being evaluated, so that its complexity degrades to the Viterbi algorithm complexity in case the underlying assumption is not met. Interestingly, in contrast with alternative approaches, the assumption made by `CarpeDiem` needs not be "always" valid. On the contrary, the algorithm is able to take advantage of the assumption even when it holds for small portions of the sequence. This implies that the worst case complexity is hit only in the very unlikely situation where the assumption does not hold for the entire sequence. Finally, `CarpeDiem` can be directly applied in any sequential learning system based on the Viterbi algorithm, even in those where the transition matrix changes over time.

The present work is structured as follows: we briefly recall the Viterbi algorithm and state the problem (Section 2). After surveying related work (Section 3), we illustrate `CarpeDiem` in full detail, and an execution example on a toy problem is provided (Section 4). We then show how `CarpeDiem` can be embodied in the voted perceptron algorithm (Section 5) and, in Section 6, we report the experimental results and discuss the results as well as several related algorithms, and elaborate on future directions of research. The soundness of the algorithm as well as its complexity are formally proved in Appendices A and B, respectively.

2. Preliminaries

The problem of finding the best sequence of labels is often represented as a search for the optimal path in a layered and weighted graph (Figure 1).

Definition 1 Layered graph. *A layered graph is a connected graph where vertices are partitioned into a set of "layers" such that:* i*) edges connect only vertices in adjacent layers;* ii*) any vertex in a given layer is connected to all vertices of the successive layer.*

We adopt the convention of indicating the layer to which a vertex belongs as a subscript to the vertex name, so that y_t denotes a vertex in layer t. We associate to each vertex y_t a weight $S^0_{y_t}$, and to each edge (y_{t-1}, y_t) a weight $S^1_{y_t, y_{t-1}}$ (Figure 1). In the following we use the term "vertical" in referring to "per node" properties. For instance, we will use the expressions "vertical weight" of y_t and "vertical information" to refer to $S^0_{y_t}$ and to the information provided by evidence related to vertices, respectively. Similarly, we use the term "horizontal" in referring to "per edge" properties. For instance, we will use the expression "horizontal weight" in referring to the weight associated to a given transition. The distinction between vertical and horizontal information is important in the present work, the key idea in `CarpeDiem` is to exploit vertical information to avoid considering the horizontal one.

Given a layered and weighted graph with T layers and K vertices per layer, a *path* is a sequence of vertices $y_1, y_2, \ldots, y_t$ ($1 \leq t \leq T$). The reward for a path is the sum of the vertical and horizontal weights associated to the path:

$$\text{reward}(y_1, y_2, \ldots, y_t) = \left(\sum_{u=1}^{t-1} S^0_{y_u} + S^1_{y_{u+1}, y_u} \right) + S^0_{y_t}.$$

We define $\gamma(y_t)$ as the maximal reward associated to any path from any node in layer 1 to y_t:

$$\gamma(y_t) = \max_{y_1, y_2, \ldots, y_{t-1}} \text{reward}(y_1, y_2, \ldots, y_{t-1}, y_t).$$

We consider the problem of picking the maximal path from the leftmost layer to the rightmost layer. The *naive* solution considers all the K^T possible paths, and returns the maximal one. The Viterbi algorithm (Viterbi, 1967) solves the problem in $\Theta(TK^2)$ time by exploiting a dynamic programming strategy. The main idea stems from noticing that the reward of the best path to node y_t can be recursively computed as: *i)* the reward of the best path to the predecessor $\pi(y_t)$ on the optimal path to y_t; *ii)* plus the reward for transition $S^1_{y_t, \pi(y_t)}$; *iii)* plus the weight of node y_t. In formulae:

$$\gamma(y_t) = \begin{cases} S^0_{y_t} & \text{if } t = 1 \\ \gamma(\pi(y_t)) + S^1_{y_t, \pi(y_t)} + S^0_{y_t} & \text{otherwise.} \end{cases} \quad (1)$$

We will also make use of the equivalent formulation obtained by noticing that $\pi(y_t)$ is the best predecessor for y_t. That is, $\pi(y_t)$ is the vertex y_{t-1} (in layer $t-1$) that maximizes the quantity $\gamma(y_{t-1}) + S^1_{y_t, y_{t-1}}$. Then:

$$\gamma(y_t) = \begin{cases} S^0_{y_t} & \text{if } t = 1 \\ \max_{y_{t-1}} \left(\gamma(y_{t-1}) + S^1_{y_t, y_{t-1}} + S^0_{y_t} \right) & \text{otherwise.} \end{cases} \quad (2)$$

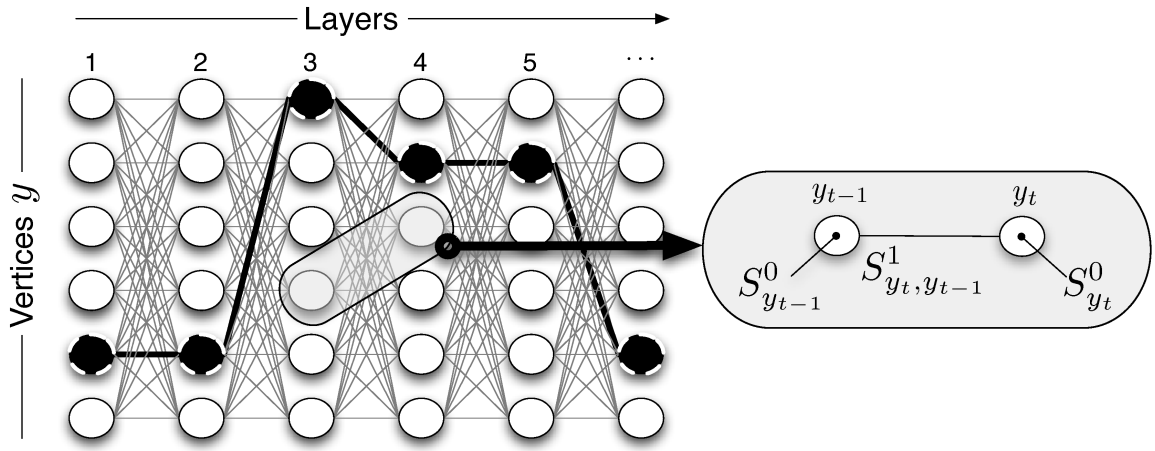


Figure 1: $S^0_{y_t}$ and $S^0_{y_{t-1}}$ denote per vertex (vertical) weights. $S^1_{y_t,y_{t-1}}$ denotes per edge (horizontal) weights.

The Viterbi algorithm proceeds from left to right storing the values of γ into an array $\mathbb{G}$ as soon as such values are computed. Assuming that $\forall y_{t-1} : \mathbb{G}(y_{t-1}) = \gamma(y_{t-1})$, then $\mathbb{G}(y_t)$ is computed as:

$$\mathbb{G}(y_t) = \max_{y_{t-1}} \left(\mathbb{G}(y_{t-1}) + S^1_{y_t,y_{t-1}} + S^0_{yt} \right).$$

The pseudo code for the algorithm is reported in Algorithm 1. Since the maximization at the line marked with label number 1 (henceforth simply "line 1 ") requires $\Theta(K)$ time, the time needed for processing each layer is in the order of $\Theta(K^2)$. The total time required by Viterbi is then $\Theta(K^2T)$. The standard formulation of the Viterbi algorithm would also store the optimal path information as it becomes available. Since this can be done using standard techniques (Cormen et al., 1990, page 520) without affecting the complexity of the algorithm, we do not explicitly report that in the pseudo-code.

Let us now consider how the above definitions instantiate in the context of a learning environment. The symbol $S^0_{y_t}$ conveys information about label y_t provided by observing the data at time t. In hidden Markov models terminology, $S^0_{y_t}$ corresponds to probability $b_{y_t}(x_t)$ of observing symbol x_t in state y_t (Rabiner, 1989, definition of $b_j(k)$ pag. 261, Eq. 8). More generally, $S^0_{y_t}$ is a quantity that depends on both the label y_t predicted for time (layer) t and the observations at and *around* time t. Likewise, in HMMs terminology, $S^1_{y',y}$ corresponds to the probability $a_{yy'}$ of transiting from state y to state y' (Rabiner, 1989, definition of a_{ij} pag. 260, Eq. 7). More in general, $S^1_{y',y}$ may depend on both the labels (y',y) *and* on the observations at and around the current layer. We note that since $S^1_{y',y}$ may vary over time (which motivates the notation $S^1_{y_t,y_{t-1}}$), the setup considered here is more general than the one of HMMs, where the transition matrix does not depend on the time instant.

```
begin
    forall y_1 do
        𝔾(y_1) ← S^0_{y_1};
    end
    for t = 2 to T do
        forall y_t do
1           𝔾(y_t) ← max_{y_{t-1}} (𝔾(y_{t-1}) + S^1_{y_t,y_{t-1}} + S^0_{y_t});
        end
    end
    y*_T ← arg max_{y_T} 𝔾(y_T);
    return y*_T;
end
```

Algorithm 1: The Viterbi algorithm.

3. Related Work

As we illustrated in Section 1, in cases where hundreds or thousands of labels are to be handled, the quadratic dependence on the number of labels is still a high burden that limits the applicability of sequential learning techniques.

In other fields (e.g., telecommunications) there exist *ad hoc* solutions that allow one to tame the complexity of the Viterbi algorithm by means of hardware implementations (Austin et al., 1990) or methods for approximating the optimum path (Fano, 1963). For instance, in the research field of speech recognition, the Viterbi algorithm is routinely applied to huge problems. This is a typical case where approximate solutions really pay off: suboptimal paths could be tolerated (to some extent) and tight time constraints prevent exhaustive search. A popular approach in this field is the *Viterbi beam search* (VBS) (Lowerre and Reddy, 1980; Spohrer et al., 1980; Bridle et al., 1982): essentially, VBS performs a breadth-first suboptimal search in which only the most promising solutions are retained at each step. Many improvements over this basic strategy have been proposed to refine either the computational performance or the accuracy of the solution (e.g., Ney et al., 1992). In most cases domain-based knowledge (such as language constraints) is used to restrict the search efforts to some relevant regions of the search space (Ney et al., 1987). Also, in recent years, several algorithms have been proposed that overcome the difficulties inherent in heuristic ranking strategies by learning ranking functions specifically optimized for the problem at hand (Xu and Fern, 2007).

Although promising, the VBS approach does not come without difficulties. For instance, Collins and Roark (2004) propose Viterbi beam search to improve the performances of the perceptron algorithm on the particular problem of natural language parsing. Interestingly, the authors note how the sub-optimality of the beam search can negatively affect the learning performances. The problem arises when a sub-optimal sequence is used instead of the optimal one to update the weights of the features (please refer to Section 5). In order to alleviate this issue, the authors stop the search—during learning—as soon as the beam does not contain the optimal solution. In such case only the partial sequence, up to when the stopping occurred, is used to update the weights. This prevents from training the perceptron using "bad" predictions, but it still has the drawback of exploiting only partially the training sequences. In such system, then, the sub-optimality of Viterbi beam search

has *two* drawbacks: at learning time, it hinders the process of finding better classifiers (or at least it slows the process down); at testing time, it yields sub-optimal classifications.

In recent years, despite a widespread usage of the Viterbi algorithm within the sequential learning field, only few works addressed the problem of reducing its time complexity and at the same time retaining the optimal result. In Felzenszwalb et al. (2003), linear (and near linear) algorithms are proposed to compute the optimal labels sequence. Their algorithms work under the assumption that the reward for the transition between states number i and j is a "simple" function of $|i-j|$.[2]

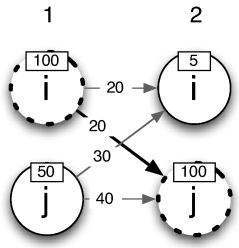
In Siddiqi and Moore (2005) it is assumed that the transition matrix is well approximated by a particular one where, for each vertex, the probability mass is concentrated on the k highest transitions leaving it. Then, the weights associated to the other transitions are approximated by a constant, and the optimal path is evaluated with $\Theta(kKT)$ time complexity. Clearly, the smaller k, the faster the algorithm.

Both techniques provide significant time savings with respect to the Viterbi algorithm. However, they are both based on assumptions about the entries in the transition matrix that are not guaranteed to hold in practice. More in particular, the assumption by Felzenszwalb et al. (2003) does not seem to easily fit general cases. Also, the investigation needed to devise the correct parameter space may require knowledge and efforts that are not always at disposal of the average practitioner. The assumption underlying the work of Siddiqi and Moore (2005) is, in our opinion, simpler to be fulfilled in practice. However, the extent to which it holds (which determines the magnitude of k) cannot be easily forecasted. Again, the extra efforts needed to assess the applicability of the approach may be detrimental to its application. Moreover, both approaches require *homogeneous* transition matrices: that is, transition matrices that do not vary over time. This is a common assumption, but unfortunately it cannot be guaranteed in some recently developed approaches, as those based on the boolean feature framework (McCallum et al., 2000). `CarpeDiem` can be safely applied even in this more complex scenario. In the experimentation, we successfully apply `CarpeDiem` in both settings, the one where the transition matrix is not constant (Section 6.1), as well as the one where it is (Sections 6.2 and 6.3).

In a recent paper Mozes et al. (2007) propose an *exact* compression-based technique to speed up the Viterbi algorithm. The authors propose to use three well known compression schemes achieving significant speed-ups whose magnitude depends on which compression algorithm is adopted. Interestingly the cited approach is not a search scheme, rather it is a preprocessing step. As such, it qualifies as an orthogonal technique amenable to be used together with `CarpeDiem` obtaining the advantages of both techniques.

In facts, to the best of our knowledge, the algorithms `CarpeDiem` and (Mozes et al., 2007) are the only exact ones, capable of speeding up the Viterbi algorithm when the assumption of homogeneous matrices is dropped. We would also argue that the other approaches presented above are not easily adapted to work in this, more complex, scenario. In Siddiqi and Moore (2005) the transition matrix needs to be traversed in advance in order to obtain the highest ranking frequencies. If those frequencies change over time, this operation needs to be repeated for each t, and the algorithm would require $O(TK^2)$ only to compute this preprocessing step. In Felzenszwalb et al. (2003) it is necessary to express the weights of the transition from label i to label j in terms of a function of $|i-j|$. The effectiveness of the approach depends on particular properties of this function. It could be argued that, in very particular situations, those properties could be shown to hold even when

2. One whose maximum can be calculated in (nearly) constant time.



Vertical weights	$S^0_{i_1}$	100	$S^0_{j_1}$	50	$S^0_{i_2}$	5	$S^0_{j_2}$	100
Horizontal weights	$S^1_{i_1,i_2}$	20	$S^1_{i_1,j_2}$	20	$S^1_{j_1,i_2}$	30	$S^1_{j_1,j_2}$	40

Figure 2: Sometimes horizontal weights can be disregarded without loosing the optimal path.

the transition matrix varies with t. However, it is hard to figure out a general way to enforce this property without inspecting the whole transition matrix at each time step.

`CarpeDiem` enjoys the desirable property of smoothly scaling to the Viterbi algorithm complexity when the underlying assumptions soften (as argued in Section 4.5). This has two consequences: 1) the algorithm adapts to the problem *and* to the sequence at hand, and 2) the algorithm can be universally applied (even when one is unsure about whether the problem fits `CarpeDiem` assumptions or not). In contrast with other state-of-the-art algorithms, no domain knowledge is to be given, nor any parameter needs to be set. This makes `CarpeDiem` well suited to be used on a regular basis as a *drop-in* replacement of the Viterbi algorithm: in the worst case, with no time saving.

4. The `CarpeDiem` Algorithm

In the general case, in order to determine the end point of the best path to a given layer, one can avoid inspecting all vertices in that layer. In particular, after sorting the vertices in layer t according to their vertical weight, the search can be stopped when the difference in vertical weight of the best node so far and the next vertex in the ordering is big enough to counterbalance any advantage that can be possibly derived from exploiting a better transition and/or a better ancestor.

To clarify this point, it is interesting to consider the minimal example reported in Figure 2. Let us assume that the reward for the maximal weight for any transition is 60. Our objective is to find the endpoint of the best path to each layer. For layer 1 we have no incoming paths, the best endpoint is simply the vertex with the maximal vertical weight: in the example, i_1. In our approach, we consider vertices with highest vertical weight first. Hence we start by calculating the reward of the optimal path to node j_2: in the example, the path i_1, j_2 (with score $S^0_{i_1} + S^1_{j_2,i_1} + S^0_{j_2} = 100 + 20 + 100 = 220$). We notice that the reward attainable by reaching i_2 cannot be higher than 165, computed as the sum of the reward for the best path to layer 1 (i.e., 100), plus the maximal weight for any transition (i.e., 60), plus the vertical weight of i_2 (i.e., 5). Therefore the endpoint of the best path to layer 2 must be j_2 and it is not necessary to calculate the reward for reaching i_2. In the course of the algorithm (hopefully) many vertices will be left unexplored by means of the above strategy. We note, however, that this does not prevent from the need of exploring those vertices in the following steps. When necessity arises `CarpeDiem` goes back through the previous layers gathering the required information. This is why `CarpeDiem` makes use of two procedures: one that finds the best vertex in

each layer (Algorithm 3), and one that finds the reward for reaching a given node by traversing the graph from right to left (Algorithm 4).

We say that a vertex is *open* if the reward of its best incoming path has been computed; otherwise the vertex is said to be *closed*. `CarpeDiem` finds the best vertex for each layer by calling Algorithm 3 (also referred to as *forward search strategy*), which leaves closed as many vertices as possible. The *backward search strategy* is called to open vertices whenever necessary.

The main procedure of `CarpeDiem` is presented in Algorithm 2; the forward and the backward search strategies are presented in Algorithms 3 and 4, respectively. Before detailing the algorithm, we need to introduce several definitions.

Definition 2 *Let us define:*

S^{1*} *: an upper bound to the maximal transition weight in the current graph*

$$S^{1*} \geq \max_{y_t, y_{t-1}} S^1_{y_t, y_{t-1}}; \tag{3}$$

γ^*_t *: the reward of the best path to any vertex in layer t (including the vertical weight of the ending vertex)*

$$\gamma^*_t = \max_{y_t} \gamma(y_t);$$

β_t *: an upper bound to the reward that can be obtained in reaching layer t* ($2 \leq t \leq T$)

$$\beta_t = \gamma^*_{t-1} + S^{1*}; \tag{4}$$

$\sqsupseteq_t$ *: a total ordering—based on vertical weights—of vertices at layer t.*

$$\sqsupseteq_t \equiv \{(y_t, y'_t) | S^0_{y_t} \geqslant S^0_{y'_t}\}. \tag{5}$$

Also, we say that vertex y_t is more promising *than vertex y'_t iff $y_t \sqsupseteq_t y'_t$.*

During execution, `CarpeDiem` calculates several values that are strictly connected to the definitions above. In particular $\mathbb{G}$ is a vector of $K \times T$ elements. $\mathbb{G}(y_t)$ contains the value of $\gamma(y_t)$ as calculated by `CarpeDiem`. Also, $\mathbb{B}$ is a vector of T elements. $\mathbb{B}_t$ contains the value of β_t as calculated by `CarpeDiem`. To a good extent, proving `CarpeDiem` correct will involve proving that, indeed, $\mathbb{G}(y_t) = \gamma(y_t)$ and $\mathbb{B}_t = \beta_t$.

4.1 Algorithm 2 – Main Procedure

Algorithm 2 initializes $\mathbb{G}(y_1)$ and $\mathbb{B}_2$ values and calls Algorithm 3 on all layers $2 \ldots T$. More specifically, $\mathbb{G}(y_1)$ is set to S^0_1 (as required by Equation 1). Also $\mathbb{B}_2$ is set to the maximal vertical weight found plus S^{1*} (as required by Equation 4).

4.2 Algorithm 3 – Forward search strategy

The forward strategy searches for the best vertex for layer t stopping as soon as this vertex can be determined unambiguously.

At the beginning of the analysis of each layer all vertices in the layer are *closed*. The algorithm scans vertices in the order given by $\sqsupseteq_t$. As mentioned at the beginning of Section 4, in the general

```
begin
    foreach y_1 { Initialization Step } do
2       G(y_1) ← S^0_{y_1}; { Opens vertex y_1 }
    end
3   y*_1 ← arg max_{y_1}(G(y_1));
    B_2 ← G(y*_1) + S^{1*};
    foreach layer t ∈ 2...T do
        y*_t ← result of Algorithm 3 on layer t;
    end
    return path to y*_T;
end
```

Algorithm 2: `CarpeDiem`.

```
begin
    y*_t ← most promising vertex;
    y'_t ← next vertex in the ⊒_t ordering;
    Open vertex y*_t {call Algorithm 4};
    while G(y*_t) < B_t + S^0_{y'_t} do
4       Open vertex y'_t {call Algorithm 4};
5       y*_t ← arg max_{y'' ∈ {y*_t, y'_t}} [G(y'')];
        y'_t ← next vertex in the ⊒_t ordering;
    end
6   B_{t+1} ← G(y*_t) + S^{1*};
    return y*_t;
end
```

Algorithm 3: Forward search strategy.

case, the algorithm can avoid opening all vertices in every layer. The search is stopped when the difference in vertical weight of y_t^* and y_t' is big enough to counterbalance any advantage that can be possibly derived from exploiting a better transition and/or a better ancestor. In formulae, let $\pi(y_t^*)$ be the best predecessor for y_t^*, the (forward) search is stopped when the currently best vertex y_t^* and the next vertex y_t' in the $\sqsupseteq_t$ ordering satisfy:

$$\overbrace{S^0_{y_t^*} - S^0_{y_t'}}^{\text{accounts for a better vertical weight of } y_t^* \text{ w.r.t. } y_t} \geq \overbrace{\left(\gamma^*_{t-1} - \gamma(\pi(y_t^*))\right)}^{\text{accounts for a possibly better predecessor of } y_t' \text{ w.r.t. } y_t^*} + \overbrace{\left(S^{1*} - S^1_{y_t^*,\pi(y_t^*)}\right)}^{\text{accounts for a possibly better transition from } y_t' \text{ predecessor}} . \quad (6)$$

The above formula is a direct consequence of the exit condition of the *while* loop of Algorithm 3, and it can be obtained by substituting[3] $\mathbb{B}$ and $\mathbb{G}$ with β and γ, and then expanding β and γ using their definitions (we repeat the relevant definitions in Table 1-*a* and *b*).

3. The soundness of the substitution is guaranteed by Theorems 1 and 2.

a)	Definition of β	$\beta_t = \gamma^*_{t-1} + S^{1*}$
b)	Definition of γ (see Eq. 1)	$\gamma(y^*_t) = \gamma(\pi(y^*_t)) + S^1_{y^*_t,\pi(y^*_t)} + S^0_{y^*_t}$

Table 1: Summary of few useful quantities

In case the stop criterion is not met, the algorithm calls Algorithm 4 (also referred to as the *backward strategy*) which sets $\mathbb{G}(y'_t) = \gamma(y'_t)$. If necessary, the "maximal" vertex y^*_t (the vertex that, so far, has associated maximal reward) is updated. Before exiting, $\mathbb{B}_{t+1}$ is readied for later use, and the best vertex is returned.

4.3 Algorithm 4 – Backward search strategy

The backward search strategy opens a vertex y_t by finding its best ancestor and setting $\mathbb{G}(y_t)$ accordingly. In much the same spirit as in the forward strategy, the algorithm saves some computation *i*) by exploiting $\sqsupseteq_{t-1}$ in order to inspect first the most promising vertices, and *ii*) by taking advantage of β_{t-1} in order to stop the search as soon as possible.

Data: A vertex y_t to be opened
begin
  $y^*_{t-1} \leftarrow$ most promising vertex;
  $y'_{t-1} \leftarrow$ next vertex in the $\sqsupseteq_{t-1}$ ordering;
  while y'_{t-1} *is open* **do**
    $y^*_{t-1} \leftarrow \arg\max_{y'' \in \{y'_{t-1}, y^*_{t-1}\}} \left[\mathbb{G}(y'') + S^1_{y_t,y''}\right]$;
    $y'_{t-1} \leftarrow$ next vertex in the $\sqsupseteq_{t-1}$ ordering;
  end
  while $\left(\mathbb{G}(y^*_{t-1}) + S^1_{y_t,y^*_{t-1}} < \mathbb{B}_{t-1} + S^0_{y'_{t-1}} + S^{1*}\right)$ **do**
    Open y'_{t-1} {call Algorithm 4};
    $y^*_{t-1} \leftarrow \arg\max_{y'' \in \{y'_{t-1}, y^*_{t-1}\}} \left[\mathbb{G}(y'') + S^1_{y_t,y''}\right]$;
    $y'_{t-1} \leftarrow$ next vertex in the $\sqsupseteq_{t-1}$ ordering;
  end
7 $\mathbb{G}(y_t) \leftarrow \mathbb{G}(y^*_{t-1}) + S^1_{y_t,y^*_{t-1}} + S^0_{y_t}$;
end

Algorithm 4: Backward search strategy to open y_t.

The first loop finds the best predecessor among the open vertices of layer $t-1$. In the second loop, we exploit the same idea behind the forward strategy. Let us inspect the exit condition of the second loop:

$$\mathbb{G}(y^*_{t-1}) + \mathbf{S^1_{y_t,y^*_{t-1}}} < \mathbb{B}_{t-1} + S^0_{y'_{t-1}} + \mathbf{S^{1*}}.$$

With the exception of the symbols in bold font, the formula is the same as the one in the exit condition of the while loop in Algorithm 3. The bold symbols take into account the transition to the target vertex. Namely, $\mathbf{S^1_{y_t,y^*_{t-1}}}$ takes into account the transition from the current best vertex (y^*_{t-1})

to the target vertex y_t and $\mathbf{S^{1*}}$ accounts for the maximal reward that a transition from y'_{t-1} to y_t can possibly obtain.

Also the internal working of the second loop is very similar to the one in the forward strategy. After opening (through a recursive call) y'_{t-1}, the current best vertex is set to the best of y'_{t-1} and y^*_{t-1}.

4.4 Example

In the following we provide a description of an execution of `CarpeDiem` over a toy problem. The problem consists of labeling a sequence containing four events and two labels (named *i* and *j*). The example is reported in Figure 3. The weight shown on the edge between labels y_{t-1} and y_t corresponds to $S^1_{y_t,y_{t-1}}$. The bound S^{1*} on the maximum horizontal reward is 60. Two further quantities are reported in the figure, and shown graphically by means of boxes placed on vertices: within rectangular boxes, we report the vertical weight of the vertex. Within rounded boxes, we report:

- $\mathbb{G}(y_t)$, if y_t is open;
- $\mathbb{B}_t + S^0_{y_t}$, if y_t is closed and $\mathbb{B}_t$ has already been computed;
- 0, otherwise.

Here we give a detailed description of the algorithm execution over the given graph.

step (a) At the beginning of the execution, all vertices are closed. The initialization steps in Algorithm 2 open all vertices in layer 1. Clearly, there is no reward for arriving at vertices in layer 0 and no incoming transitions to be taken into account. The best vertex in layer 1 is thus the vertex having the maximum vertical weight.

step (b) The analysis of layer 2 starts by opening the most promising vertex in that layer (vertex j). Since all vertices at layer 1 are open, the backward strategy already has complete information at disposal, and it does not need to enter the second loop to open j_2. Once $\mathbb{G}(j_2)$ has been computed, the algorithm compares this value to the bound on the weight of the best path to i_2. Since $S^0_{i_2} + \mathbb{B}_2 = 165$ cannot outperform $\mathbb{G}(j_2) = 220$, there is no need to open vertex i_2.

step (c) To open i_3, the backward strategy goes back to layer 2 and searches for the best path to that vertex. Again, vertex i_2 can be left closed, since there is no chance that the best path to i_3 traverses it. In fact,

$$\mathbb{B}_2 + S^0_{i_2} + S^{1*} = (100+60)+5+60 = 225$$

cannot outperform the reward

$$\mathbb{G}(j_2) + S^1_{i_3,j_2} = 220+15 = 235$$

obtained by passing through j_2. Then $\mathbb{G}(i_3)$ is set to $235+100 = 335$.

Unfortunately, this does not allow to make a definitive decision about whether this is the best vertex of layer 3, since $\mathbb{B}_3 + S^0_{j_3}$ is $(220+60)+70 = 350$. Next step will thereby settle the question by opening vertex j_3.

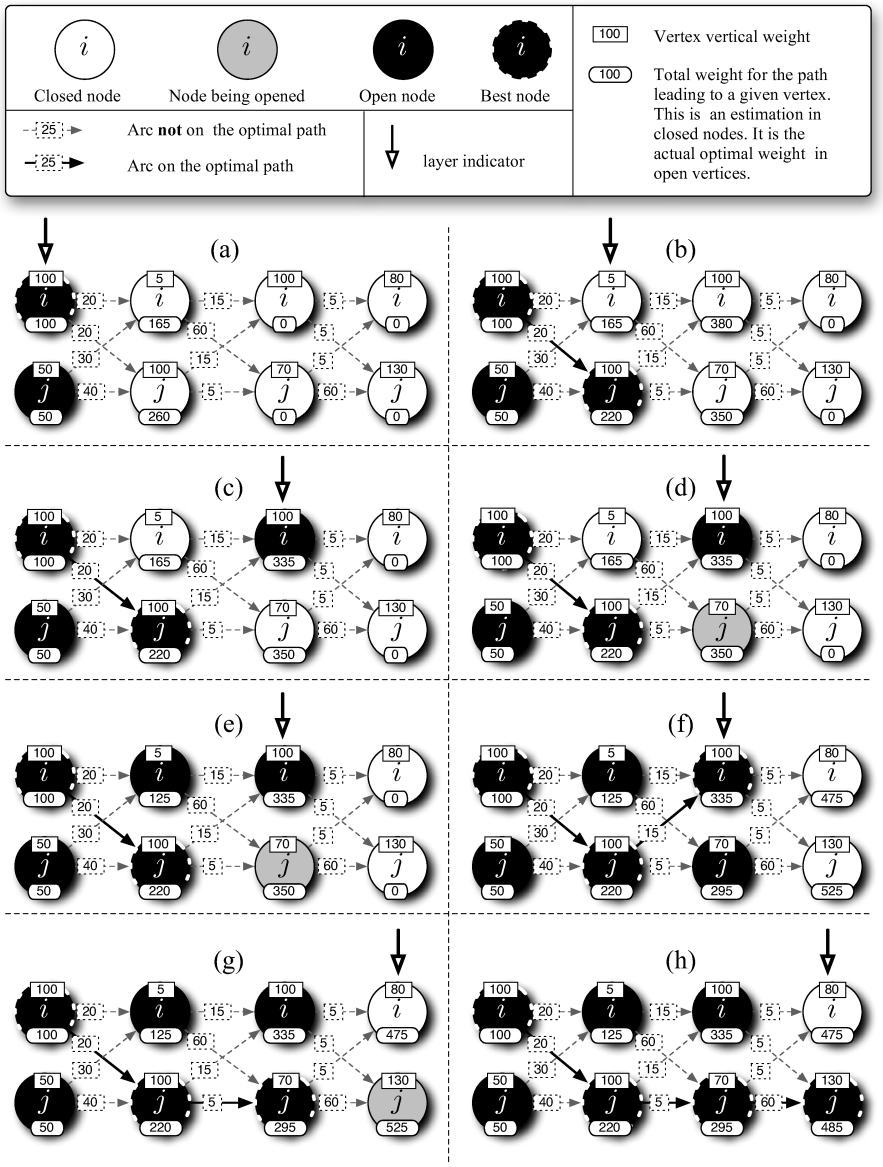


Figure 3: `CarpeDiem` in action on a toy problem.

step (d) The goal is, at this point, to find the best path to j_3. Even though j_2 has a clear advantage over i_2, this does not suffice to exclude that the latter one is on the optimal path to j_3 (since $\mathbb{G}(j_2) + S^1_{j_3,j_2} \not< \mathbb{B}_2 + S^0_{i_2} + S^{1*}$): the backward strategy is then forced to recursively call itself to open i_2.

step (e) By opening i_2, the algorithm sets $\mathbb{G}(i_2)$ to 125 (the best path being $i_1 \rightarrow i_2$), thus ruling it out as a candidate for being on the optimal path to j_3.

step (f) In returning to consider layer 3, we are back to the path $i_1 \rightarrow j_2 \rightarrow i_3$. To open vertices i_2 and j_3 has been wasteful, though unavoidable.

step (g) The first vertex to be opened in its layer is j_4. Interestingly, the best path to j_4 is not through the best vertex in layer 3. In fact, while the highest reward for a three steps walk is on vertex i_3, it is more convenient to go through vertex j_3 to reach vertex j_4.

step (h) Since $\mathbb{G}(j_4) = 485$ is larger than $\mathbb{B}_4 + S^0_{i_4}$, the algorithm terminates leaving i_4 closed.

In Section 6 real world problems are considered, and much larger optimizations obtained.

4.5 Algorithm Properties

An intuitive way of characterizing the algorithm complexity is to consider Formula 6, where the exit condition of Algorithm 3 is rewritten in order to point out why in some cases it is safe to stop inspecting the current layer. Clearly, the sooner the loop exit condition is satisfied, the faster the algorithm.

Arguably, the worst case happens when vertical rewards, being equal for each label, do not provide any discriminative power. In such a case, the left term in Formula 6 is zero, the inequality is never satisfied, and Algorithm 3 calls Algorithm 4 over all K vertices in every layer. In this case, for each one of the k vertices to be opened in a new layer, the first loop of Algorithm 4 iterates over all K predecessors. However, no recursive call takes place. Overall, in the worst case hypothesis, `CarpeDiem` has order of $O(TKK + TK\log(K)) = O(TK^2)$ time complexity.[4] `CarpeDiem` is never asymptotically worse than the Viterbi algorithm.

The best case happens when horizontal rewards, being equal for each transition, do not provide any discriminative power. In such a case the right hand side of the inequality in Formula 6 is zero and the inequality is guaranteed to be satisfied immediately. Moreover, being the backward strategy based on a bound similar to the one that leads to Formula 6, it will never open any other vertex. Then, a single vertex per layer is opened and `CarpeDiem` has order of $O(T + TK\log(K)) = O(TK\log(K))$ time complexity. A more formal argument about `CarpeDiem` complexity is stated by Theorem 3 and proved in Appendix B.

Theorem 3 `CarpeDiem` *has* $O(TK^2)$ *worst case time complexity and* $O(TK\log K)$ *best case time complexity.*

`CarpeDiem` finds the optimal sequence of labels. By using standard book-keeping techniques, the optimal sequence of labels can be tracked back by starting from the optimal end point. Then, the optimality of `CarpeDiem` can be proved by showing that the vertex returned by the forward strategy at the end of the algorithm is the end-point of the optimal path through the graph. This property, stated by Theorem 1, is formally proved in Appendix A.

Theorem 1 *Let us consider a sequence of calls to Algorithm 3 on layers* $2,3,\ldots,t$ $(t \leq T)$. *When Algorithm 3 terminates on layer* t*, the returned vertex* y^*_t *is the endpoint of the optimal path to layer* t*. Formally,*

$$\forall y_t : \gamma(y^*_t) \geq \gamma(y_t).$$

Beside the theoretical properties of the algorithm, it is important for the practitioner to consider its actual performances over real world problems. In the general case the algorithm will open some, but not all vertices: the exact number of the vertices that will be inspected depends on the particular

4. The $O(TK\log(K))$ term in the formula accounts for the time needed to sort vertices according to $\sqsupseteq_t$.

application and on how the features have been engineered. Empirical evidence (Section 6) suggests that many problems are closer to the best case than to the worst. Before introducing the experimentation, we show how `CarpeDiem` can be instantiated in the context of a supervised sequential learning system and, more in particular, in a system based on the voted perceptron algorithm.

5. Grounding the Voted Perceptron Algorithm on `CarpeDiem`

The supervised sequential learning problem can be formulated as follows (Dietterich, 2002).

> Let $\{(\vec{x}_i,\vec{y}_i)\}_{i=1}^{N}$ be a set of N training examples. Each example is a pair of sequences $(\vec{x}_i,\vec{y}_i)$, where $\vec{x}_i = \langle x_{i,1}, x_{i,2}, \ldots, x_{i,T_i}\rangle$ and $\vec{y}_i = \langle y_{i,1}, y_{i,2}, \ldots, y_{i,T_i}\rangle$. The goal is to construct a classifier H that can correctly predict a new label sequence $\vec{y} = H(\vec{x})$ given an input sequence $\vec{x}$.

The SSL problem has been approached with many different techniques. Among others, we recall Sliding Windows (Dietterich, 2002), hidden Markov models (Rabiner, 1989), Maximum Entropy Markov Models (McCallum et al., 2000), Conditional Random Fields (Lafferty et al., 2001), Dynamic Conditional Random Fields (Sutton et al., 2007), and the voted perceptron algorithm (Collins, 2002).

The voted perceptron uses the Viterbi algorithm at both learning and classification time. It is then particularly appropriate for the application of our technique. Moreover, it relies on the *boolean features framework* (McCallum et al., 2000) which is more general than the HMMs model with respect to representing the graph. In this framework, depending on how features are implemented, both static (homogeneous) and *dynamic* transition matrices can be modeled. We use the term *dynamic* transition matrix to indicate that weights associated to edges may change from time point to time point, depending on the observations.

In the boolean features framework the learnt classifier is built in terms of a set of boolean features. Each feature ϕ reports about a salient aspect of the sequence to be labelled in a given time instant. More formally, given a time point t, a boolean feature is a $1/0$-valued function of the whole sequence of feature vectors $\vec{x}$, and of a restricted neighborhood of y_t. The function is meant to return 1 if the characteristics of the sequence $\vec{x}$ around time step t support the classifications given at and around y_t. Under a first order Markov assumption, each ϕ depends only on y_t and y_{t-1}. Let us denote with w_ϕ the weight associated to feature ϕ. The classifier learnt by the voted perceptron algorithm has the form

$$H(\vec{x}) = \arg\max_{\vec{y}} \sum_{t=1}^{T} \sum_{\phi} w_\phi \cdot \phi(\vec{x}, y_t, y_{t-1}, t)$$

and is suitable to be evaluated using the Viterbi algorithm.

In practice, not all boolean features depend on both y_t and y_{t-1}. Let us distinguish features depending on both y_t and y_{t-1} from those depending only on y_t. We denote with Φ^0 the set of features that depends only on y_t and thus models per vertex (vertical) information. Analogously, we denote with Φ^1 the set of features that depend on both y_t and y_{t-1} modeling, thus, per edge (horizontal) information. The vertical and horizontal weights can be then calculated as:

$$S^0_{y_t} = \sum_{\phi \in \Phi^0} w_\phi \phi(\vec{x}, y_t, t)$$

and

$$S^1_{y_t,y_{t-1}} = \sum_{\phi \in \Phi^1} w_\phi \phi(\vec{x}, y_t, y_{t-1}, t).$$

In general, the bound on the maximal transition weight S^{1*} can be set to the sum of all positive horizontal weights:

$$S^{1*} = \sum_{\phi \in \Phi^1} J(w_\phi)$$

where $J(x)$ is x if $x > 0$, and 0 otherwise. It is noteworthy that this quantity can be computed without any extra—domain specific—knowledge.

Often, however, better bounds can be given based on specific domain knowledge. An example of such improvements (one that we exploit throughout our experimentation) consists in partitioning the horizontal features into sets of mutually exclusive features. Then the bound can be computed as the sum of the maximal weight of each partition. In case the partitions degenerate to a single set (i.e., all horizontal features are mutually exclusive), the maximal horizontal weight can be used. For instance, in many domains where HMMs are routinely applied, horizontal features are used only to check the last two predicted labels. In such domains, if a horizontal feature is asserted, no other feature can and we can appropriately set

$$S^{1*} = \max_{\phi \in \Phi^1} J(w_\phi). \tag{7}$$

6. Experimentation

To figure out whether and how `CarpeDiem` can be applied to actual tasks, we tested it on three different problems: the problem of music harmony analysis (Radicioni and Esposito, 2007), the frequently asked questions (FAQs) segmentation problem (McCallum et al., 2000), and a text recognition problem built starting from the "letter recognition" data set from the UCI machine learning repository (Frey and Slate, 1991).

The running time of an execution of `CarpeDiem` depends on how the weights of vertical and horizontal features compare: the more discriminative are vertical features with respect to horizontal features, the larger is the edge `CarpeDiem` has over the Viterbi algorithm.

Overall the three experiments cover three situations that are likely to occur in practice. The music analys problem represents a situation where S^{1*} has been selected by exploiting detailed domain knowledge: horizontal features have been divided into sets of non trivial partitions and the bound has been set accordingly (see end of Section 5). Features of the FAQs segmentation problem have been developed by McCallum et al. (2000) on a different system, and then imported into ours without modifications. Features used in the text recognition task didn't go through a real engineering process; on the contrary, they can be seen as a first, to some extent *naive*, attempt to tackle the problem. In these last two cases, we have set S^{1*} using Formula 7.

6.1 Tonal Harmony Analysis

Given a musical flow, the task of music harmony analysis consists in associating a label to each time point (Temperley, 2001; Pardo and Birmingham, 2002). Such labels reveal the underlying harmony by indicating a fundamental note (*root*) and a *mode*, using chord names such as 'C minor'.

Music analysis task can be naturally represented as a supervised sequential learning problem. In fact, by considering only the "vertical" aspects of musical structure, one would hardly produce reasonable analyses. Experimental evidences about human cognition reveal that in order to disambiguate unclear cases, composers and listeners refer to "horizontal" features of music as well: in these cases, context plays a fundamental role, and contextual cues can be useful to the analysis system.

The system relies on 39 features. They have been engineered so that they take into account the prescriptions from music harmony theory, a field where vertical and horizontal features naturally arise. *Vertical* features report about simultaneous sounds and their correlation with the currently predicted chord. *Horizontal* features capture metric patterns and chordal successions. This is a case where not all horizontal features are mutually exclusive (i.e., S^{1*} is not the maximal of positive horizontal weights) and where horizontal weights may change over time. For instance, the same transition between two chords can receive different weights according to whether it falls on accented/unaccented beats.

The training set is composed of 30 chorales ($3,020$ events) by J.S. Bach (1675-1750). The classifiers have been tested on 42 separate chorales ($3,487$ events) from the same author.

6.2 FAQs Segmentation

We experimented on the FAQs segmentation problem as introduced by McCallum et al. (2000). It basically consists of segmenting Usenet FAQs into four distinct sections: 'head', 'question', 'answer', and 'tail'.

In this data set, events correspond to text lines and sequences correspond to FAQs. McCallum et al. define 24 boolean features. Each one is coupled with each possible label for a total of 96 features. Additionally, 16 features are used to take into account the possible transitions between labels.

The data set consists of a learning set containing 26 sequences (29,406 events) and a test set containing 22 sequences (33,091 events).

6.3 Text Recognition

Our third experiment deals with the problem of recognizing printed text. We trained the classifiers on the "The Frog King" tale (122 sequences, $6,931$ events) by Grimm brothers, and tested over the "Cinderella" tale (240 sequences, $13,354$ events) by the same authors. The classifier is called to recognize each letter composing the tale. The data set has been built as follows. Each letter (corresponding to an individual event) in the tales has been encoded by picking at random one of its possible descriptions as provided by the *letters* UCI data set[5] (Frey and Slate, 1991). Each sentence corresponds to a distinct sequence.

We briefly recall here the characteristics of the letters data set as originally proposed by the authors. The data set contains $20,000$ letters described using 16 integer valued features. Such attributes capture highly heterogeneous facets of the scanned raw image such as: horizontal and vertical position, the width and height, the mean number of edges per pixel row. The images have been obtained by randomly distorting 16 fonts taken from the US National Bureau of Standards. The features used by the learning system are:

5. It can be found at `ftp://ftp.ics.uci.edu/pub/machine-learning-databases/letter-recognition`.

Experiment	Viterbi	CarpeDiem	Time Saved (%)
music Analysis	13,582	1,507	88.90%
FAQs segmentation	537	144	73.15%
letter recognition	961,969	34,244	96.44%

Table 2: CPU Time (expressed in seconds) and percentage of time saved by CarpeDiem.

1. $26 \times (16 \times 16) = 6,656$ vertical features, obtained by the original attributes devised by Frey and Slate (1991). The reported figure is explained as follows. We consider each of the 16 values of the 16 attributes in the original data set, thus resulting in 256 possible combinations. Each such combination is still to be coupled with the 26 letters of the English alphabet.

2. the $27 \times 27 = 729$ horizontal features, obtained by considering $\mathcal{A} \times \mathcal{A}$, where $\mathcal{A}$ is the set of letters in the english alphabet, plus a sign for blanks.

6.4 Procedure

We compare the performances of CarpeDiem against those provided by the Viterbi algorithm. To this aim, we embedded CarpeDiem in a SSL system implementing the voted perceptron learning algorithm (Collins, 2002). The learnt weights have been then used to build two classifiers: one based on the Viterbi algorithm, the other one based on CarpeDiem.

For each one of the three problems we divided the data into a learning set and a test set. Each learning set has been further divided into ten data sets of increasing sizes (the first one contains 10% of the data, the second one 20% of the data, and so forth). Also, each experiment has been repeated by varying the number of learning iterations from 1 to 10, for a grand total of 100 classifiers per problem. We tested each learnt classifier on the appropriate test set recording the classification time obtained by using first CarpeDiem and then Viterbi.

In the following we will indicate each one of the 100 classifiers by using two numbers separated by a colon: the former number corresponds to the size of the training set (1 standing for 10%, 2 for 20%, ..., 10 for 100%), the latter one indicates the number of iterations. For instance, the classifier 8:1 has been acquired by iterating once on 80% of the learning set.

6.5 Results

As earlier mentioned (and implied by Theorem 1), CarpeDiem performs exact inference: classifiers built on CarpeDiem provide the same answers as those built on the Viterbi algorithm.

We measured the total classification time spent by the algorithms as well as the average time *saved* by CarpeDiem with respect to the Viterbi algorithm. They are provided in Table 2: average time savings range from about 73% (FAQs segmentation) to over 96% (letter recognition). Figures 4, 5 and 6 graphically report a detailed account of each experiment. Classification times for each problem were obtained using a fixed size test set. By observing the profiles reported in the figures, it is apparent that, while the Viterbi algorithm runs in approximately constant time, CarpeDiem performances depend on the particular classifier used.

In all trials of all experiments CarpeDiem runs in a small fraction of the time needed by Viterbi.

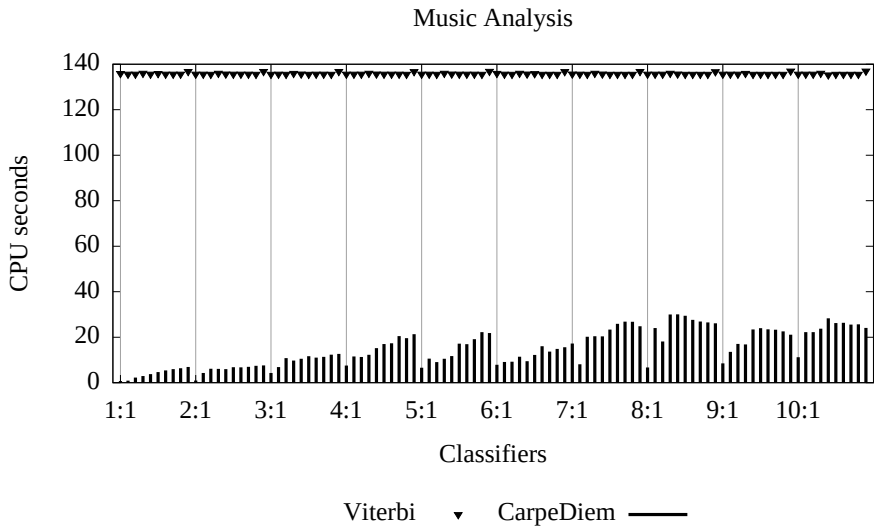


Figure 4: Results on the music analysis problem. We report on the abscissas the learnt classifiers indicating $i{:}j$ for classifier acquired using $(10 \cdot i)\%$ of the training set and j iterations. Labels on the abscissas refer only to the first classifier for each block of experiments; the following nine bars refer to the remaining ones. For instance, label 3:1 is positioned below the bar corresponding to the classifier trained on the 30% of the training set and using 1 iteration. The following nine bars refer to classifiers 3:2, 3:3, ..., 3:10. We report on the ordinates the CPU seconds needed for the classification of the test set. Vertical bars refer to the time spent by `CarpeDiem`. Triangles refer to the time spent by Viterbi.

One interesting fact is unveiled by the profile of the classification time. Since—in each one of the three experiments—classification is performed on a data set of fixed size, one would expect roughly constant classification time. By converse, at least in the first two experiments (Figures 4 and 5), the emerging patterns are similar to those usually observed at learning time. We remark that the hundred runs of each experiment differ only in the set of weights used by the classifier. Then it is apparent that, as the voted perceptron learns, it somehow modifies the weights in a way that proves to be detrimental to the work of `CarpeDiem`.

To explain the observed patterns, let us consider an informative vertical feature $\phi_\bullet$ and examine the first iteration of the voted perceptron on a sequence of length $T = 100$. Also, we assume that $\phi_\bullet$ is asserted 60 times, and that it votes for the correct label 50 times out of 60. Even though this example may seem unrealistic, it is not.[6] The first iteration on the first sequence the voted perceptron chooses labels at random. Then, the vast majority of them will be incorrectly assigned, thus implying a large number of feature weights updates. If all the labels for which $\phi_\bullet$ is asserted are actually mislabelled, due to the way the update rule acts, the weight associated to $\phi_\bullet$ will be increased[7] by 40. This large increase occurs all at once at the end of the first iteration on the first sequence, and it is likely to overestimate the weight of $\phi_\bullet$. The voted perceptron will spend the rest of learning trying to compensate for this overestimation. However, subsequent updates will be

6. For instance, in the music analysis problem, this could be the case for the feature that votes for the chord that has exactly 3 notes asserted in the current event.

7. That is, $50 - (60 - 50) = 40$.

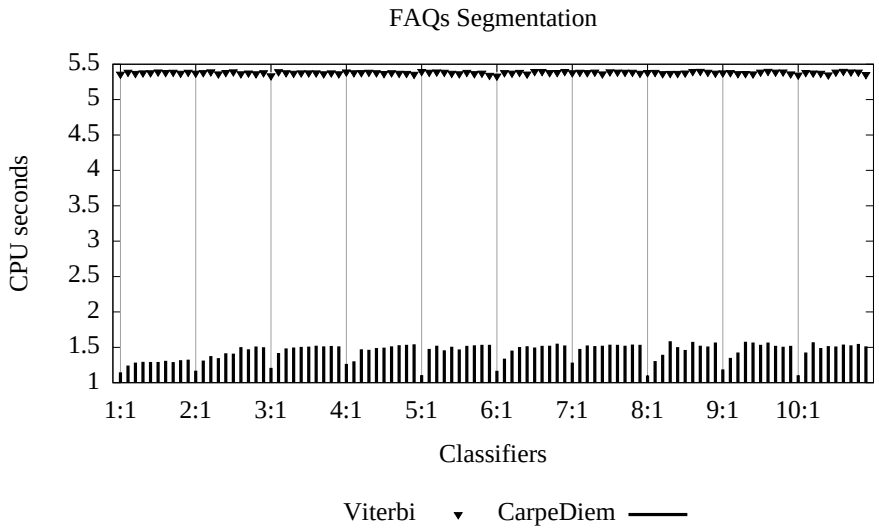


Figure 5: Results on the FAQs Segmentation problem. The conventions adopted are the same as for Figure 4.

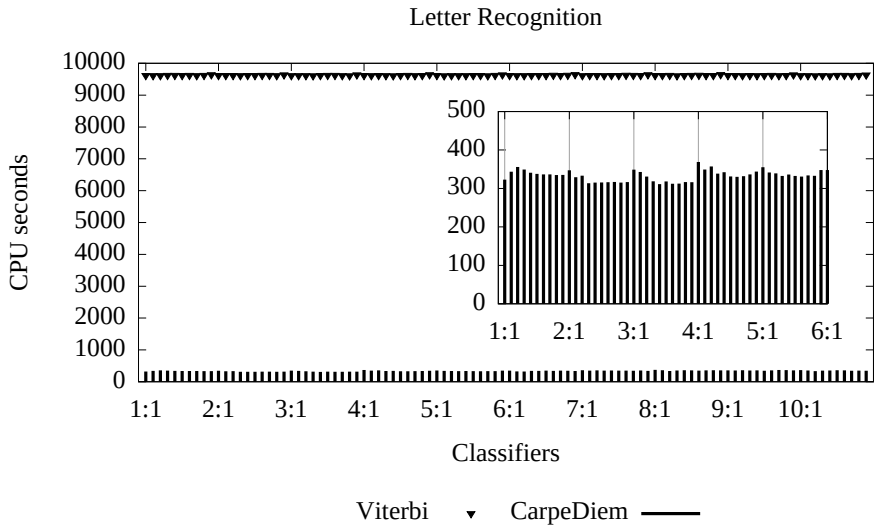


Figure 6: Results on the letter recognition problem. In the inner frame we detail the time spent by `CarpeDiem` using the first fifty classifiers. The conventions adopted are the same as for Figure 4.

of smaller magnitude. In fact, the following predicted labeling will not be randomly guessed, thus implying a reduced number of updates.

By summarizing, highly predictive features have their weights initially set to very large values; such weights slowly decrease in the following. The behavior described clearly emerges in Figure 7, where the individual weights of vertical features (for the music analysis problem) are plotted as the updates occur. Then, since `CarpeDiem` is efficient when vertical features are discriminative compared to horizontal ones, the algorithm is particularly well-suited to be used during the early

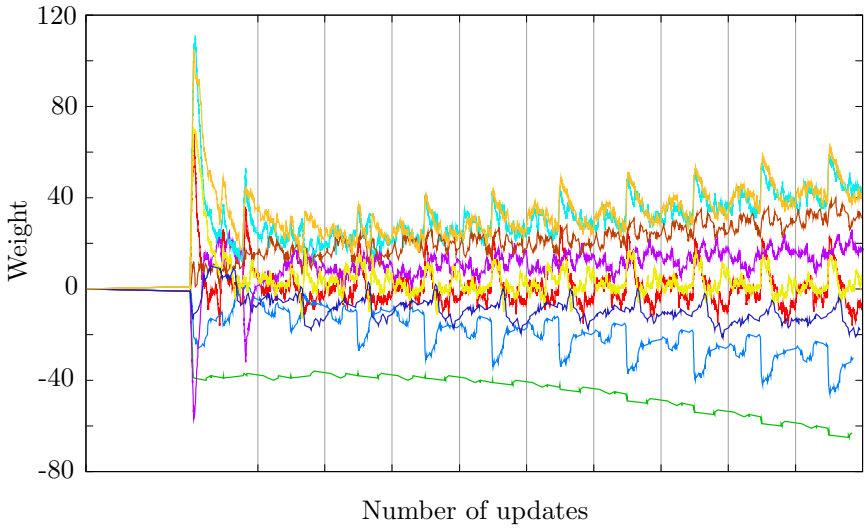


Figure 7: Evolution of vertical weights throughout learning. Each line corresponds to an individual vertical feature; vertical lines correspond to the beginning of new iterations. The plot refers to the music analysis data set.

learning steps of the voted perceptron where the time saved by `CarpeDiem` reaches peaks of 99.47% (music analysis, classifier 1:1) and 78.60% (FAQs segmentation, classifier 1:1).

Lastly, one might wonder why the observed pattern wasn't seen in the text recognition problem. Notwithstanding that the problem itself is a sequential one, to model only bigrams probabilities proved—against our expectations—to be not enough to provide sufficient discrimination power to horizontal features. In other words, the horizontal features we devised contribute to the correct classification only in a marginal way. This is a setting where `CarpeDiem` can, and actually does, attain exceptional time savings (Figure 6).

Prompted by such time savings, we re-ran the same experiments using no horizontal features: the accuracy does not drop down as significantly as in the other two problems. This fact explains why the pattern observed in Figures 4 and 5 is not observed in Figure 6: since vertical features remain predictive with respect to horizontal ones throughout the learning process the performances of `CarpeDiem` do not change over time. Here we note one important property of `CarpeDiem`: `CarpeDiem` time performances reflect the degree of sequentiality inherent to the problem at hand. Actually, by tracking the number of vertices opened in each layer, one can even measure how important sequential information is in different parts of the same sequence.

6.6 Comparison with Related Algorithms

In the following we present a brief discussion about two technologies that, if not directly comparable with `CarpeDiem`, are much related to the algorithm. We start by reporting the results of implementing the `CarpeDiem` heuristic for the A^* algorithm. Then, we report about a performance comparison with non-optimal search algorithms based on the Viterbi beam search approach.

6.6.1 RELATIONS WITH A^*

It could be argued that the `CarpeDiem` algorithm looks interestingly similar to the A^* algorithm (Hart et al., 1968). In order to investigate this similarity let us consider the following heuristic, based on the same ideas underlying `CarpeDiem`:

$$h(y_t) = \sum_{t'>t} \left[\max_{y_{t'}} \left(S^0_{y_{t'}} \right) + S^{1*} \right] = (T-t)S^{1*} + \sum_{t'>t} \left[\max_{y_{t'}} \left(S^0_{y_{t'}} \right) \right].$$

The heuristic estimates the distance to the goal by summing the best vertical weight in each layer and the best possible transition between any two layers.

Even though the heuristic is intuitively connected to the strategy implemented in `CarpeDiem`, the differences are indeed remarkable. A first important difference is in the choice criteria implemented in Algorithms 3 and 4 (we will focus only on Algorithm 3 for the sake of exposition). The criterion in Algorithm 3 states: *do not consider any further node in this layer if*

$$\mathbb{G}(y^*_t) >= \mathbb{B}_t + S^0_{y'_t}.$$

Since $\mathbb{B}_t = \mathbb{G}(y^*_{t-1}) + S^{1*}$, the above criterion approximates the reward of the optimal path by means of S^{1*} only once rather than the $T-t$ times taken by $h(y_t)$. Hence, A^* incurs an high risk of opening vertices in layers far from the goal only because of the cumulation of these approximations.

Another important difference between the two algorithms is in the fact that `CarpeDiem` implements two different heuristics: one is used by Algorithm 3 and one is used by Algorithm 4. On the contrary A^* uses the same criterion throughout the computation. Finally, the data structures needed by `CarpeDiem` are simpler (and faster) than those needed to implement A^*.

In order to empirically assess whether A^* could be competitive with respect to `CarpeDiem`, we implemented the above heuristic and paid particular attention to tuning the data structures. The resulting algorithm turned out to be even less efficient than the Viterbi algorithm.

Of course, this does not mean that the same principles implemented by `CarpeDiem` could not be plugged into A^* by means of a carefully chosen heuristic. Rather, it shows that the problem is not trivial, and deserves *ad-hoc* research efforts.

6.6.2 VITERBI BEAM SEARCH

To compare `CarpeDiem` with algorithms based on the Viterbi beam search (VBS) strategy presents several difficulties. On the one hand, we have an algorithm that guarantees optimality, but not computational performances; on the other hand, we have VBS approaches that guarantee computational performances abdicating optimality.

In VBS approaches the width of the beam (hereafter b) is particularly important in that it affects the tradeoff between computational gain and result optimality. For very small b, VBS would most probably lead to crude approximations; in this case VBS is likely to run faster than CarpeDiem just because it disregards most of the possibly-optimal paths. By converse, if we were to choose the beam size almost equal to the number of possible labels K (i.e., $b \simeq K$), then VBS would run in almost the same time as the Viterbi algorithm. In between there are all other options. Also, the quality of the solution found by VBS approaches highly depends on the heuristic adopted. In the following we will disregard any accuracy concern, so that the two approaches could be compared in terms of execution times solely.

Let ξ be the ratio between the time spent by `CarpeDiem` and the time spent by the Viterbi algorithm for solving a given problem: that is, the time savings reported in Table 2 are computed as $[(1-\xi)\times 100]\%$. Given that the time spent by a Viterbi beam search algorithm is in the order of b^2T, the time saved by a VBS algorithm w.r.t. the Viterbi algorithm is in the order of:

$$\frac{(K^2-b^2)T}{K^2T} = 1 - \frac{b^2}{K^2}\ .$$

By equating the time saved by `CarpeDiem` (i.e., $1-\xi$) and the time saved via the VBS approach, we have that by setting:

$$b = \sqrt{\xi K^2} \tag{8}$$

an algorithm based on the VBS approach will run in about the same time as `CarpeDiem`.

We implemented a basic VBS algorithm and measured its running time over the data set described in Section 6 and experimented with different settings of b. The results show that to obtain the same running times as `CarpeDiem`, the beam size needs to be set as follows:

- $b = 25$ for the Tonal Harmony Analysis data set. Being $K = 78$, this amounts to consider 32% of the possible labels;
- $b = 2$ for the FAQ Segmentation data set. Being $K = 4$, this amounts to consider 50% of the possible labels;
- $b = 5$ for the text recognition data set. Being $K = 27$, this amounts to consider 18.5% of the possible labels;

Two aspects are remarkable in the above results: *i)* the reported figures have been observed empirically rather than determined using the above formula. It is immediate to verify that they are very close to the ones returned by using Formula 8; *ii)* we omitted to implement a heuristic to guide the VBS search. Since computing the heuristic would require additional efforts, the timings used to derive such numbers are optimistic approximations of the actual time needed by a full fledged VBS algorithm.

In summary, `CarpeDiem` runs at least as fast as a VBS approach when a small to medium beam size is used while providing some additional benefits. Again, investigating the conditions that make appropriate one approach over the other one, deserves further deepening the research.

7. Conclusions

In this paper we have proposed `CarpeDiem`, a replacement of the Viterbi algorithm. On average, `CarpeDiem` allows evaluating the best path in a layered and weighted graph in a fraction of the time needed by the Viterbi algorithm. `CarpeDiem` is based on a property exhibited by most tasks commonly tackled by means of sequential learning techniques: the observations at and around a time instant t are very relevant for determining the t-th label, while information about the succession of labels is mainly useful in order to disambiguate unclear cases. The extent to which the property holds determines the performances of the algorithm. We formally proved that `CarpeDiem` finds the best possible sequence of labels *and* that the algorithm complexity ranges between $O(TK\log K)$ in the best case, and $O(TK^2)$ in the worst case. The fact that the worst case complexity is the same as the Viterbi algorithm complexity suggests that, by and large, `CarpeDiem` is suitable for substituting the Viterbi algorithm.

In Section 3, we reviewed recently proposed alternatives and pointed out the following advantages of CarpeDiem with respect to the competitors:

- it is parameter free;
- it does not require any prior knowledge or tuning;
- it never compromises on optimality.

At the present time, the main strength of other approaches with respect to ours is that they have been devised to improve the forward-backward algorithm as well. We defer the extension of our approach in that direction to future work.

In addition to the theoretical grounding of CarpeDiem complexity, we provided an experimentation on three real world problems, and compared its execution time with that of the Viterbi algorithm. The experiments show that large time savings can be obtained. The reported figures show time savings ranging from 73% to 96% with respect to the Viterbi algorithm.

A further interesting facet of CarpeDiem is that its execution trace provides clues about the problems at hand. It can then be used to understand how salient sequential information is. Also, within the same sequence, it is often interesting to investigate the time steps where sequential information gets crucial thus implying higher classification time. For instance, in Radicioni and Esposito (2007) we used this fact to find where musical excerpts get more difficult.

In a world where the quantity of information as well as its complexity is ever increasing, there is the need for sophisticated tools to analyse it in reasonable time. Our perception is that, in most problems, long chains of dependencies are useful to reach top results, but their influence on the correct labelling decreases with the length of the chain. In a probabilistic setting, the length of the chain of dependencies would be called the 'order of the Markov assumption'. In such a context, it would be appropriate to say that what we call 'vertical' features are actually features making a zero order Markov assumption (no dependency at all), while horizontal features are features making a first order Markov assumption. The higher the order of the Markov assumption we want to plug into the model, the slower the algorithm that evaluates the sequential classifier. If our guess is correct, however, the higher the Markov assumption, the less informative are the features that use it. Should this be the case, CarpeDiem strategy could be extended to efficiently handle higher order Markov assumptions, thereby allowing to use sequential classifiers to tackle a larger set of problems.

Acknowledgments

We wish to thank Luca Anselma, Guido Boella, Leonardo Lesmo, and Lorenza Saitta for their precious advices on preliminary versions of the work. We also want to thank the anonymous reviewers for their helpful comments and suggestions.

Appendix A. Soundness

A proof of soundness for CarpeDiem consists in showing that y_T^* is the endpoint of the optimal path through the graph of interest. Although the whole algorithm is concerned with finding out the optimal path, we presently restrict ourselves to find the optimal endpoint, since standard book

symbol	description
$S^0_{y_t}$	vertical weight of vertex y_t
$S^1_{y_t,y_{t-1}}$	horizontal weight for transition from y_{t-1} and y_t
S^{1*}	maximal transition reward (fixed for the whole graph)
$\gamma(y_t)$	the weight of the best path to y_t
γ^*_t	the weight of the best path to the best vertex in level t
β_t	$\gamma^*_{t-1} + S^{1*}$
$\mathbb{G}(y_t)$	$\gamma(y_t)$ as calculated by `CarpeDiem`
$\mathbb{B}_t$	β_t as calculated by `CarpeDiem`

Table 3: Summary of the notation adopted.

keeping techniques can be used during the search to store path information. The path can be then be retrieved in $O(T)$ time.

The proof consists in two Theorems and one Lemma; in particular, Theorem 1 directly implies the soundness of `CarpeDiem`, while Theorem 2 and Lemma 1 are necessary in the proof of Theorem 1.

Before entering the core of the proof, let us summarize the notation adopted. We distinguish between values that are calculated by `CarpeDiem`, and those representing properties of the graph. We will use blackboard characters ($\mathbb{G}$ and $\mathbb{B}$) to denote the former ones and greek letters (γ and β) for the latter ones. A summary of important definitions is reported in Table 3.

We start by stating and proving Lemma 1, which ensures the soundness of the main bound used by `CarpeDiem`.

Lemma 1 *If $\mathbb{B}_t = \beta_t$ then the bound exploited by CarpeDiem does not underestimate the reward of the optimal path to any vertex. Formally,*

$$\mathbb{B}_t = \beta_t \Rightarrow \mathbb{B}_t + S^0_{y_t} \geq \gamma(y_t).$$

Proof Let us consider the optimal path to y_t and denote with $\pi(y_t)$ the predecessor of y_t. Then, by definition we have: $\gamma^*_{t-1} \geq \gamma(\pi(y_t))$, and $S^{1*} \geq S^1_{y_t,\pi(y_t)}$. It immediately follows that:

$$\gamma^*_{t-1} + S^{1*} + S^0_{y_t} \geq \gamma(\pi(y_t)) + S^1_{y_t,\pi(y_t)} + S^0_{y_t}.$$

By definition $\beta_t = \gamma^*_{t-1} + S^{1*}$ and $\gamma(y_t) = \gamma(\pi(y_t)) + S^1_{y_t,\pi(y_t)} + S^0_{y_t}$, which yields:

$$\beta_t + S^0_{y_t} \geq \gamma(y_t)$$

and by assumption, this implies:

$$\mathbb{B}_t + S^0_{y_t} \geq \gamma(y_t).$$

■

Theorem 1 *Let us consider a sequence of calls to Algorithm 3 on layers $2,3,\ldots,t$ $(t \leq T)$. When Algorithm 3 terminates on layer t, the returned vertex y^*_t is the endpoint of the optimal path to layer t. Formally,*

$$\forall y_t : \gamma(y^*_t) \geq \gamma(y_t).$$

Proof We prove the stronger fact

$$\mathbb{B}_t = \beta_t \wedge \forall y_t : \gamma(y_t^*) \geq \gamma(y_t).$$

The proof is by induction on t. The base case for the induction is guaranteed by the initialization step in Algorithm 2 where $\mathbb{B}_2$ and $\gamma(y_1^*)$ are set. We start by showing that

$$\forall y_1 : \gamma(y_1^*) \geq \gamma(y_1) \tag{9}$$

as follows:

$$\begin{array}{rcl}
\gamma(y_1^*) = & \gamma(\arg\max_{y_1} \mathbb{G}(y_1)) & \rightarrow \text{by line 3 (Algorithm 2)} \\
= & \gamma\left(\arg\max_{y_1} S^0_{y_1}\right) & \rightarrow \text{by line 2 (Algorithm 2)} \\
= & \gamma(\arg\max_{y_1} \gamma(y_1)) & \rightarrow \text{by Equation 2} \\
= & \max_{y_1} \gamma(y_1) & \\
& \Downarrow & \\
& \forall y_1 : \gamma(y_1^*) \geq \gamma(y_1). &
\end{array}$$

In order to prove $\mathbb{B}_2 = \beta_2$, we note that Algorithm 2 sets $\mathbb{B}_2$ to $\mathbb{G}(y_1^*) + S^{1*}$:

$$\begin{array}{rcl}
\mathbb{B}_2 = & \mathbb{G}(y_1^*) + S^{1*} & \\
= & S^0_{y_1^*} + S^{1*} & \rightarrow \text{by line 2 (Algorithm 2)} \\
= & \gamma(y_1^*) + S^{1*} & \rightarrow \text{by Equation 2} \\
= & \gamma_1^* + S^{1*} & \rightarrow \text{by definition of } \gamma_1^* \text{ (Table 3) and Equation 9} \\
= & \beta_2 & \rightarrow \text{by definition of } \beta_2 \text{ (Table 3).}
\end{array}$$

Let us now assume that for all $\hat{t}$, $1 \leq \hat{t} < t$:[8]

$$\begin{array}{c}
\mathbb{B}_{\hat{t}} = \beta_{\hat{t}} \\
\forall y_{\hat{t}} : \gamma(y_{\hat{t}}^*) \geq \gamma(y_{\hat{t}})
\end{array}$$

then we prove $\mathbb{B}_t = \beta_t$ as follows:

$$\begin{array}{rcl}
\mathbb{B}_t = & \mathbb{G}(y_{t-1}^*) + S^{1*} & \rightarrow \text{by instruction 6 (Algorithm 3)} \\
= & \gamma(y_{t-1}^*) + S^{1*} & \rightarrow \text{by Theorem 2} \\
= & \gamma_{t-1}^* + S^{1*} & \rightarrow \text{by Equation 10 and definition of } \gamma_{t-1}^* \\
= & \beta_t & \rightarrow \text{by definition of } \beta_t \text{ (Table 3).}
\end{array}$$

In order to prove $\forall y_t : \gamma(y_t^*) \geq \gamma(y_t)$ we start by noting that at the end of the main loop of Algorithm 3 it holds $\mathbb{G}(y_t^*) \geq \mathbb{B}_t + S^0_{y_t}$. Also, for any $y_t \sqsubseteq_t y_t'$ we have (by Definition 2–Equation 5): $\mathbb{B}_t + S^0_{y_t} \leq \mathbb{B}_t + S^0_{y_t'}$. It follows that $y_t \sqsubseteq_t y_t' \Rightarrow \mathbb{G}(y_t^*) \geq \mathbb{B}_t + S^0_{y_t}$. Using Lemma 1 we have

$$y_t \sqsubseteq_t y_t' \Rightarrow \mathbb{G}(y_t^*) \geq \gamma(y_t). \tag{10}$$

8. We note that our definitions give no meaning to $\mathbb{B}_1$ and β_1. We define them to be equal regardless their value: this simplifies the discussion allowing an easier formulation of the properties being stated and proved. They are not used in the algorithm nor in the argument anyway; the definition is thus safe.

Moreover, since the algorithm scans the vertices in the order given by $\sqsupseteq_t$, all vertices y_t, $y_t \sqsupseteq_t y'_t$ have been considered by the main loop. Then by line 4 (Algorithm 3), by our inductive hypothesis ($\forall \hat{t} < t : \mathbb{B}_{\hat{t}} = \beta_{\hat{t}}$) and Theorem 2, we have that for each such vertex $\mathbb{G}(y_t) = \gamma(y_t)$. Moreover, due to line 5 (Algorithm 3), $\mathbb{G}(y^*_t) \geq \mathbb{G}(y_t)$. Putting together the two statements, we conclude that

$$y_t \sqsupseteq_t y'_t \Rightarrow \mathbb{G}(y^*_t) \geq \gamma(y_t). \tag{11}$$

Equation 10, Equation 11, and the fact that $\sqsupseteq_t$ is a total order, yield

$$\forall y_t : \mathbb{G}(y^*_t) \geq \gamma(y_t).$$

By noting that y^*_t is open (and exploiting again Theorem 2), we have:

$$\forall y_t : \gamma(y^*_t) \geq \gamma(y_t).$$

■

Theorem 2 *Let us assume* $\forall \hat{t} < t : \mathbb{B}_{\hat{t}} = \beta_{\hat{t}}$, *then after opening vertex* y_t, $\mathbb{G}(y_t) = \gamma(y_t)$.

Proof By line 7 (Algorithm 4), $\mathbb{G}(y_t) = \mathbb{G}(y^*_{t-1}) + S^1_{y_t, y^*_{t-1}} + S^0_{y_t}$. Then, our main goal is to prove

$$\mathbb{G}(y^*_{t-1}) + S^1_{y_t, y^*_{t-1}} + S^0_{y_t} = \gamma(y_t).$$

Replacing $\gamma(y_t)$ with its definition (Equation 2) yields:

$$\mathbb{G}(y^*_{t-1}) + S^1_{y_t, y^*_{t-1}} + S^0_{y_t} = \max_{y_{t-1}} \left(\gamma(y_{t-1}) + S^1_{y_t, y_{t-1}} + S^0_{y_t} \right).$$

The above equality is satisfied if the following two properties hold:

$$y^*_{t-1} = \arg\max_{y_{t-1}} \left(\gamma(y_{t-1}) + S^1_{y_t, y_{t-1}} \right) \tag{12}$$

$$\mathbb{G}(y^*_{t-1}) = \gamma(y^*_{t-1}). \tag{13}$$

The proof, by induction on t, proves that the Equations 12 and 13 are satisfied at the moment (and after) $\mathbb{G}(y_t)$ is set. `CarpeDiem` starts by opening the most promising vertex in layer 2, this is the first time Algorithm 4 is called and hence the base case of the induction. Let us consider what happens when a node y_2 is opened. Since all vertices in layer 1 have been opened by the initialization step, the first loop in Algorithm 4 iterates on all of them and the second loop is never entered. Then, just before line 7, it holds

$$y^*_1 = \arg\max_{y_1} \left(\mathbb{G}(y_1) + S^1_{y_2, y_1} \right).$$

Since the initialization step guarantees $\forall y_1 : \mathbb{G}(y_1) = \gamma(y_1)$, then properties 12 and 13 are satisfied.

Let us now assume by induction that after opening a vertex y_{t-1} in layer $t-1$ ($t > 2$) it holds $\mathbb{G}(y_{t-1}) = \gamma(y_{t-1})$. We focus on the execution of Algorithm 4 on a vertex y_t in layer t. Let us denote with O_{t-1} the set of vertices presently open in layer $t-1$, and with C_{t-1} the set of closed ones. When the first loop ends, it holds:

$$y^*_{t-1} = \arg\max_{y_{t-1} \in \mathsf{O}_{t-1}} \left(\mathbb{G}(y_{t-1}) + S^1_{y_t, y_{t-1}} \right).$$

Also, since all vertices for which we have taken the arg max are in layer $t-1$ and open, we apply the inductive hypothesis and conclude that:

$$y^*_{t-1} = \arg\max_{y_{t-1} \in \mathsf{O}_{t-1}} \left(\gamma(y_{t-1}) + S^1_{y_t, y_{t-1}}\right). \tag{14}$$

The second loop moves some vertices from C_{t-1} to O_{t-1}. At the same time, however, it updates y^*_{t-1} so that the above equality is preserved. Then, on exit we can conclude (14) and (for the particular y'_{t-1} that caused the loop to exit):

$$\mathbb{G}(y^*_{t-1}) + S^1_{y_t, y^*_{t-1}} \geq \mathbb{B}_{t-1} + S^0_{y'_{t-1}} + S^{1*}. \tag{15}$$

Also, by definition of $\sqsupseteq_{t-1}$ (Definition 2–Equation 5), $\forall y_{t-1} : y'_{t-1} \sqsupseteq_{t-1} y_{t-1}$ implies:

$$\mathbb{B}_{t-1} + S^0_{y'_{t-1}} + S^{1*} \geq \mathbb{B}_{t-1} + S^0_{y_{t-1}} + S^{1*}. \tag{16}$$

Since vertices are considered in $\sqsupseteq_{t-1}$ order and since y'_{t-1} is the first vertex that has not been opened, it follows that all closed vertices follow y'_{t-1} in the $\sqsupseteq_{t-1}$ order. Using this fact along with (15) and (16), it follows:

$$\forall y_{t-1} \in \mathsf{C}_{t-1} : \mathbb{G}(y^*_{t-1}) + S^1_{y_t, y^*_{t-1}} \geq \mathbb{B}_{t-1} + S^0_{y_{t-1}} + S^{1*}.$$

By induction, $\mathbb{G}(y^*_{t-1}) = \gamma(y^*_{t-1})$. Moreover, Lemma 1 implies $\mathbb{B}_{t-1} + S^0_{y_{t-1}} \geq \gamma(y_{t-1})$. Using these facts, along with $\forall y_t, y_{t-1} : S^{1*} \geq S^1_{y_t, y_{t-1}}$ (Definition 2–Equation 3) we obtain:

$$\forall y_{t-1} \in \mathsf{C}_{t-1} : \gamma(y^*_{t-1}) + S^1_{y_t, y^*_{t-1}} \geq \gamma(y_{t-1}) + S^1_{y_t, y_{t-1}}$$

Which yields:

$$\gamma(y^*_{t-1}) + S^1_{y_t, y^*_{t-1}} \geq \max_{y_{t-1} \in \mathsf{C}_{t-1}} \left(\gamma(y_{t-1}) + S^1_{y_t, y_{t-1}}\right).$$

This and (14) yield:

$$y^*_{t-1} = \arg\max_{y_{t-1}} \left(\gamma(y_{t-1}) + S^1_{y_t, y_{t-1}}\right).$$

Also, the fact that y^*_{t-1} is open and the inductive hypothesis, yield $\mathbb{G}(y^*_{t-1}) = \gamma(y^*_{t-1})$.

■

Appendix B. Complexity

Theorem 3 `CarpeDiem` *has $O(TK^2)$ worst case time complexity and $O(TK\log K)$ best case time complexity.*

Proof Let us consider the final step of an execution of `CarpeDiem`, and assume that for each layer t, exactly k_t vertices have been opened. In our proof we separately consider the time spent to process each layer of the graph. We define the quantity $\mathcal{T}(t)$ to represent the overall time spent by Algorithms 3 and 4 to process layer t. Let us define:

$a(y_t)$**:** the number of steps needed by Algorithm 3 to process vertex y_t;

$b(y_t)$: the number of steps needed by Algorithm 4 to find the best parent for node y_t.

We note that $a(y_t)$ does not include the time spent by Algorithm 4 since such time is accounted for by $b(y_t)$. Similarly, $b(y_t)$ does not include neither the time spent by Algorithm 3, nor the time spent by recursive calls to Algorithm 4. In fact, the time spent in recursive calls is taken into account by b values of vertices in previous layers. Then we can compute $\mathcal{T}(t)$ as:

$$\mathcal{T}(t) = \sum_{y_t} a(y_t) + b(y_t)$$

The total complexity of `CarpeDiem` is then:

$$\mathcal{T}(\texttt{CarpeDiem}) = \text{time for initialization} + \sum_{t=2}^{T} (O(1) + \mathcal{T}(t))$$

where the "time for initialization" includes the $O(K)$ time spent in the first loop of Algorithm 2 plus the $O(TK\log K)$ time needed to sort each layer according to $\sqsupseteq_t$. It follows:

$$\mathcal{T}(\texttt{CarpeDiem}) = O(TK\log K) + \sum_{t=2}^{T} (O(1) + \mathcal{T}(t)). \tag{17}$$

Let us now note that $a(y_t)$ is at worst $O(k_t)$. In fact, since only k_t vertices have been opened at the end of the algorithm, it follows that the steps needed to analyse a vertex y_t by (the loop in) Algorithm 3 is at most k_t. We notice that we are overestimating the cost to analyze each node since k_t is the *overall* number of iterations performed by the mentioned loop. However this overestimation simplifies the following argument without hindering the result.

$b(y_t)$ is, at worst, $O(k_{t-1})$. In fact, since only k_{t-1} vertices have been opened at the end of the algorithm, it follows that the two loops in Algorithm 4 iterate altogether at most k_{t-1} times. Moreover, since the steps performed by recursive calls are not to be included in $b(y_t)$, it follows that all operations are $O(1)$, and the complexity accounted for by $b(y_t)$ is $O(k_{t-1})$. In both cases no computational effort is spent to process closed nodes.

From the above discussion it follows that:

$$\begin{aligned}
\mathcal{T}(t) &= \sum_{y_t} a(y_t) + b(y_t) \\
&= \sum_{y_t \text{ in open vertices}} O(k_t) + O(k_{t-1}) \\
&= k_t \cdot (O(k_t) + O(k_{t-1})) \\
&= O(k_t^2 + k_t k_{t-1}).
\end{aligned}$$

Putting together the above equation and Equation 17 we have:

$$\begin{aligned}
\mathcal{T}(\texttt{CarpeDiem}) &= O(TK\log K) + \sum_{t=2}^{T} \left(O(1) + O(k_t^2 + k_t k_{t-1})\right) \\
&= O(TK\log K) + \sum_{t=2}^{T} O(k_t^2 + k_t k_{t-1}).
\end{aligned}$$

The worst case occurs when `CarpeDiem` opens every node in every layer. In such case: $k_t = K$ for each t and the above formula reduces to $O(TK^2)$. In the best case `CarpeDiem` opens only one node per layer, $k_t = 1$ for each t and the complexity is $O(TK\log K)$. ■

References

Steve Austin, Pat Peterson, Paul Placeway, Richard Schwartz, and Jeff Vandergrift. Toward a real-time spoken language system using commercial hardware. In *HLT '90: Proceedings of the Workshop on Speech and Natural Language*, pages 72–77, Hidden Valley, Pennsylvania, 1990.

John S. Bridle, Michael D. Brown, and Richard M. Chamberlain. An algorithm for connected word recognition. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP 1982*, pages 899–902, San Francisco, CA, USA, 1982.

Michael Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 1–8, Philadelphia, PA, 2002. Association for Computational Linguistics.

Michael Collins and Brian Roark. Incremental parsing with the perceptron algorithm. In *ACL '04: Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*, pages 111–118, Barcelona, Spain, 2004. Association for Computational Linguistics.

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachussets, 1990.

Thomas G. Dietterich. Machine learning for sequential data: A review. In T. Caelli, editor, *Structural, Syntactic, and Statistical Pattern Recognition*, volume 2396 of *Lecture Notes in Computer Science*, pages 15–30, Windsor, Ontario, Canada, 2002. Springer-Verlag.

Thomas G. Dietterich, Pedro Domingos, Lise Getoor, Stephen Muggleton, and Prasad Tadepalli. Structured machine learning: The next ten years. *Machine Learning*, 73(1):3–23, 2008.

Robert M. Fano. A heuristic discussion of probabilistic decoding. *IEEE Transactions on Information Theory*, 9:64–73, 1963.

Pedro F. Felzenszwalb, Daniel P. Huttenlocher, and Jon M. Kleinberg. Fast algorithms for large-state-space HMMs with applications to web usage analysis. In *Advances in Neural Information Processing Systems*. MIT Press, 2003.

Peter W. Frey and David J. Slate. Letter recognition using Holland-style adaptive classifiers. *Machine Learning*, 6(2):161–182, 1991.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics*, 4:100–107, 1968.

John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 282–289, San Francisco, CA, 2001. Morgan Kaufmann.

Bruce Lowerre and Raj Reddy. *Trends in Speech Recognition*, chapter The Harpy Speech Understanding System, pages 340–360. Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

Andrew McCallum, Dayne Freitag, and Fernando Pereira. Maximum entropy Markov models for information extraction and segmentation. In *Proceedings of the 17th International Conference on Machine Learning*, pages 591–598, Stanford, California, USA, 2000. Morgan Kaufmann.

Shay Mozes, Oren Weimann, and Michal Ziv-Ukelson. Speeding up HMM decoding and training by exploiting sequence repetitions. *Combinatorial Pattern Matching*, 4580:4–15, 2007.

Hermann Ney, Dieter Mergel, Andreas Noll, and Annedore Paeseler. Data driven search organization for continuous speech recognition. *IEEE Transactions on Signal Processing*, 40:272–281, 1987.

Hermann Ney, Reinhold Haeb-Umbach, Bach-Hiep Tran, and Martin Oerder. Improvements in beam search for 10000-word continuous speech recognition. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 9–12, San Francisco, California, USA, 1992.

Bryan Pardo and William P. Birmingham. Algorithms for chordal analysis. *Computer Music Journal*, 26:27–49, 2002.

Lawrence R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, volume 77, pages 267–296, San Francisco, CA, USA, February 1989. Morgan Kaufmann. ISBN 1-55860-124-4.

Daniele P. Radicioni and Roberto Esposito. Tonal harmony analysis: a supervised sequential learning approach. In M. T. Pazienza and R. Basili, editors, *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing, 10th Congress of the Italian Association for Artificial Intelligence*. Springer-Verlag, 2007.

Sajid M. Siddiqi and Andrew W. Moore. Fast inference and learning in large-state-space HMMs. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 800–807, Bonn, Germany, 2005. ACM.

James C. Spohrer, Peter F. Brown, Peter H. Hochschild, and James K. Baker. Partial traceback in continuous speech recognition. In *Proceedings of the IEEE Interntational Conference on Cybernetics and Society*, pages 36–42, Boston, Massachussets, USA, 1980.

Charles Sutton, Andrew McCallum, and Khashayar Rohanimanesh. Dynamic conditional random fields: factorized probabilistic models for labeling and segmenting sequence data. *Journal of Machine Learning Research*, 8:693–723, 2007.

David Temperley. *The Cognition of Basic Musical Structures*. MIT, Cambridge, MASS, 2001.

Andrew J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. In *IEEE Transaction on Information Theory*, volume 13, pages 260–269, 1967.

Yuehua Xu and Alan Fern. On learning linear ranking functions for beam search. In Z. Ghahramani, editor, *Proceedings of the 24th International Conference on Machine Learning*, pages 1047–1054, Corvallis, Oregon, USA, 2007. ACM.