Learning Pattern Graphs for Multivariate Temporal Pattern Retrieval

Sebastian Peter[1], Frank Höppner[2] and Michael R. Berthold[1]

[1] Nycomed-Chair for Bioinformatics and Information Mining
Dept. of Computer Science, University of Konstanz
Box 712, D-78457 Konstanz, Germany
[2] Ostfalia University of Applied Sciences
Dept. of Computer Science, D-38302 Wolfenbüttel, Germany

Abstract. We propose a two-phased approach to learn pattern graphs, a powerful pattern language for complex, multivariate temporal data, which is capable of reflecting more aspects of temporal patterns than earlier proposals. The first phase aims at increasing the understandability of the graph by finding common substructures, thereby helping the second phase to specialize the graph learned so far to discriminate against undesired situations. The usefulness is shown on data from the automobile industry and the libras data set by taking the accuracy and the knowledge gain of the learned graphs into account.

1 Introduction

As the number of (mobile and/or wireless) sensor networks increases (e.g. health care, climate, earthquakes, traffic), more and more data is gathered periodically and the interest in analyzing temporal data rises. The recorded data usually includes various dimensions and the user is often interested in *typical* or *characteristic* situations. To grasp or encompass these situations, various notions of *multivariate temporal patterns* are employed in the literature. Example applications for multivariate temporal patterns include the discovery of dependencies in wireless sensor networks [1], the exploration of typical (business) workflows [3] or classification of electronic health records [2].

We present a new approach to derive classification rules automatically from multivariate, temporal data. Rather than enumerating all patterns (and forcing an expert to have a look at (too) many of them), we invite the expert to actively work on the patterns – by constructing them from scratch, by extending the patterns proposed by the algorithms presented in this paper or by alternating between both steps. To enable such a successful interaction, it is crucial that the pattern language itself is expressive enough to include the kind of constraints the expert wants to express. We use *pattern graphs* [8] as they provide the necessary flexibility to include not only the order of events, but also different kinds of parallelism, absence of events, as well as constraints on their duration.

The outline of the paper is as follows: We discuss notions of multivariate temporal patterns, including the pattern graph, in the next section. A two-phased

approach to construct rich pattern graphs for classification tasks, consisting of the identification of common substructures and its specialization towards class predictability, is presented in Sect. 3. Results from a real world application are shown in Sect. 4. Finally Sect. 5 concludes the paper.

2 Notions of Multivariate Temporal Patterns

Preliminaries. Due to the fact that we consider a classification task, a case consists of a multivariate data sequence S plus a class label. Rather than building the patterns upon raw data (e.g. the time series itself), we formulate them by means of various conditions (temporal abstractions), such as A≡'speed is above 50 km/h' or B≡'clutch pedaled' etc. We use these abstractions to pose *constraints* over certain periods of time. Besides these constraints on the *values* of the sequences, we also consider *temporal* constraints on their duration. For instance, we may require that there is some period of time in which 'A is present and B is absent for 5-10 time units' (which will be abbreviated by '$A, \neg B$ $[5, 10]$').

Related Work. Many approaches, such as [2], describe multivariate temporal patterns by specifying the relationships between all observed intervals like '*A before B*', '*A overlaps C*' and '*C overlaps B*' in the spirit of [5]. This notation is quite strict and somewhat ambiguous [7], because the qualitative relationship does not carry quantitative information about the degree of overlap or size of a gap. Other approaches contain such information [3], but only consider events without *duration* and thus offer no means to express concurrency as in '*A and B must co-occur for 5-10 time units*'. In practice, we frequently want to forbid some events (e.g. '*no connection to host while running batch*'). Sometimes negative constraints are used to *filter* a large set of enumerated patterns [1], but the pattern language itself seldomly offers the means to express the *absence* of events. For this paper, we settled on pattern graphs, because they offer all these features.

Notion of a Pattern Graph. A *pattern graph* $(V, E, C_{\text{val}}, C_{\text{temp}})$ is an acyclic, directed graph (V, E) with one source $(\top \in V)$ and one sink $(\bot \in V)$. Associated with each node $v \in V$, we have *value constraints* $C_{\text{val}}(v)$ on the necessary observations while in node v and *temporal constraints* $C_{\text{temp}}(v)$ on the duration of node v. A sequence matches (fits) the pattern graph if it is possible to assign subsequences from S to all nodes of the graph, such that the subsequences satisfy all constraints of the assigned nodes simultaneously. The assignment has to be complete in the sense, that (1) the empty prefix of S is assigned to the source, the empty suffix of S to the sink, (2) a non-empty subsequence is assigned to all other nodes, and (3) the subsequences to connected nodes are contiguous.

Interpretation. In Fig. 1 we see an example pattern graph with two parallel paths which is read as follows:

1. The temporal constraint of a node is represented above the node. A star represents an unlimited duration.
2. The value constraint(s) of a node is (are) shown inside the node.

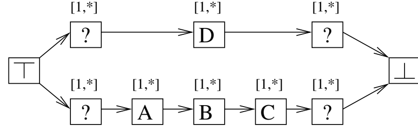

Fig. 1. Example pattern graph with two parallel paths from $\top$ to $\bot$.

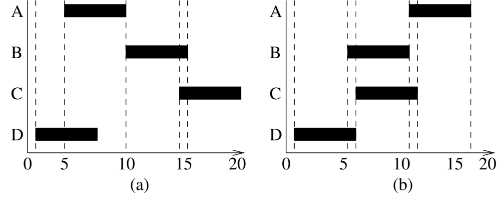

Fig. 2. Two example sequences with four binary properties A-D.

3. A node without any value constraints is labeled '?' (*don't care*).

To match a given sequence to a pattern graph, it has to be subdivided into several parts (on the time axis), such that each part can be assigned to a node of the graph. The nodes therefore represent a part of a sequence and pose constraints on the values and the duration of the subsequence: The temporal constraints restrict the length and the value constraints restrict the behavior of the respective subsequence. The edges between the nodes enforce the order of the parts. If a node has an outgoing edge it means that there has to be another part directly after the associated subsequence that fulfils the constraints of the successor node. If a node has two or more outgoing edges, all parts belonging to the following nodes have to begin at the same time. On the other hand, if a node has two or more incoming edges all parts belonging to the preceding nodes have to end at the same time and the part of the node has to begin immediately afterwards. Thus, to express that two conditions A and B occur simultaneously, we include both, A and B in *one* node (as value constraints). In contrast, to express that A and B are concurrent but independent from each other, we introduce parallel paths for A and B. Any condition A may continue to hold before or after the node unless a condition $\neg A$ forbids this explicitly.

Mapping sequences. Fig. 2 shows two sequences where the vertical axis shows a number of value constraints A-D that hold over certain periods of time (black bars, time on horizontal axis). We now discuss whether these sequences can be mapped validly to the pattern graph in Fig. 1. The graph shown in Fig. 1 can be decomposed into two different paths: For the lower path the sequence has to be divided in five contiguous parts, so that the first part satisfies the '*don't care*' constraint, during the second part the property A has to hold, the property B in the third, etc. The last part is again a '*don't care*'-part. All of these five parts require a duration of at least one time unit (but have no upper bound on the duration). Parallel to the lower path, the upper path requires '*don't care*', 'D' and '*don't care*' again with durations ≥ 1 time unit.

The sequence shown in 2(a) can be mapped to the graph, because we can clearly see that A is before B and B before C. And D is present during this subsequence as well. Due to the fact that B and C are overlapping, it is possible to assign different subsequences to the B and C node (the overlapping part may be assigned to any of the respective nodes or may even be split up into two

parts). This means that the pattern graph has more than one valid mapping. On the other hand we cannot find a valid mapping for the sequence shown in Fig. 2(b), because there, A does not occur before B. If A were true within $[6, 9]$ (rather than $[10, 15]$), we would have another valid mapping.

For a more formal definition of a pattern graph and a valid mapping, as well as an efficient algorithm that decides if a sequence matches a given pattern graph or not, we refer the interested reader to [8]. This algorithm may either simply state whether a match is possible, or it provides us with detailed information about possible matches in the following sense: For each edge $e = (u, v) \in E$ in the graph, we obtain a set of valid positions $p(e)$ in case of a valid mapping, that is, a set of positions t that satisfy all value constraints of node u for $t' < t$ and all value constraints of node v for $t' \geq t$. For the graph in Fig. 1 and the sequence in Fig. 2(a) for the edge e from node A to B we have only one valid location $p(e) = \{10\}$, but for the edge e' from B to C we have $p(e') = \{[14, 15]\}$.

3 Learning Pattern Graphs

Next we propose a two-phased approach to learn pattern graphs from labeled data. In the first phase we build a pattern graph that is mappable to *all* instances of the target class. This is done for the following two purposes: Firstly, the resulting pattern graph describes the structure shared by all instances of the class – even if they were not necessary for the purpose of classification – and thus supports the interpretability of the class. Secondly, the graph represents a good initialization for the second phase of the algorithm, which is a pattern graph refinement via beam search. The beam search is tailored towards a quick improvement in terms of discrimination capabilities, but with real-world problems it is impossible to discriminate classes with a pattern graph if it consists of a few nodes only. If we were starting the beam search from scratch (empty graph), it would waste considerable time to build up a pattern graph that is complex enough to eventually include the important aspects for the classification task. In [6] we have experimentally shown that a two-phased approach for learning (a restricted set of) temporal patterns may not only increase the understandability but additionally may increase the accuracy of the patterns as well. Here, we have extended this approach to the case of full-fledged pattern graphs (whereas the earlier work used only 'limited patten graphs', which consisted of a single path from $\top$ to $\bot$ only).

3.1 First phase: identifying the shared structure (within a class)

As already mentioned the primary goal of this phase is to find key aspects of the target class. As we expect to obtain quite complex graphs from this step already, we do not employ frequent sequence mining algorithms (e.g. [9]), as they would waste considerable time on the enumeration of a huge number of frequent subgraphs. Furthermore, such methods are usually not suited for including *absent* items (or constraints). The problem of finding a pattern common to all instances

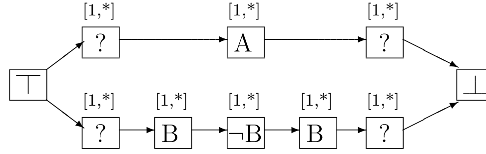

Fig. 3. Resulting pattern graph from step 1 where A occurs once and B two times

(of one class) is closely related to the alignment of multiple sequences, which is known to be NP-complete [10]. As we are aware that those parts of the graph, which are important for the classification task, will be inserted during the refinement phase anyway, there is no need to find some kind of *best graph* in the first phase, therefore we settle for a heuristic approach. The following proposal exploits the possibilities of the pattern graph to incorporate *temporal*-, *present*-, *absent*- and '*don't care*' -constraints into the pattern graph. It consists of three steps: In the first step the relevant nodes of the graph are computed, then we add relations between the nodes and finally remove redundant parts.

Univariate paths. As a preprocessing step, we scan through all instances of the target class once and determine the (contiguous) intervals in which a constraint holds. For every constraint C the minimum number n_C of intervals per instance is determined. From this information we create the first pattern graph: For every constraint C we create a path from $\top$ to $\bot$ always starting and ending with a '*don't care*'-node, consisting of an alternating sequence of n_C *present*- and *absent*- nodes. For example, if constraint A holds at least once and constraint B twice, the resulting graph is shown in Fig. 3. The resulting pattern graph has at least one valid mapping on all sequences of the target class, as we only state the minimal number of occurrences and require no specific relation between different constraints, because they are aligned in different paths.

Linking paths. The second step inserts connections between nodes from different paths. As already mentioned, the matching algorithm [8] returns for an edge $e \in E$ all valid edge locations $p(e)$ in a given sequence. For two nodes u and v, let $A = \bigcap_{(u,w)\in E} p(u,w)$ contain all valid 'end positions of node u' and let $B = \bigcap_{(w,v)\in E} p(w,v)$ contain all valid 'start positions of node v'. If there are $a \in A, b \in B$ such that $0 < b - a \leq t$, the gap between node u and v is at most t time units wide. If it is possible to satisfy this condition for every individual sequence, we may safely introduce a new '*don't care*'-node between nodes u and v with a temporal constraint $[1, t]$ without risking the match of any of the sequences to the extended pattern (because its feasibility has already been checked). The function '`check(u,v,t,S)`' indicates by its return value whether this condition is satisfied for all sequences S. To support the understandability of the graph we do not allow the connection of ' *don't care*' nodes. Furthermore, an edge connecting u to v is not added if a (possibly longer) path already exists from u to v . On some rare occasions the additional node might inject a cycle

into the graph; to ensure the acyclic property of the graph, we disallow such cases. A sketch of this procedure is shown in Alg. 1.

Algorithm 1 linking paths

Require: start pattern graph $G = (V, E, C_{\text{val}}, C_{\text{temp}})$, set S of target sequences, minlength, maxlength, stepsize

Ensure: return extended pattern graph

1: t ← minlength
2: **repeat**
3:    **for all** $(v_1, v_2) \in V \times V$ **do**
4:       **if** check(v_1, v_2, t, S) and ¬ existsPath(v_1, v_2) and ¬ existsPath(v_2, v_1) **then**
5:          extended G by a '?'-node between v_1 and v_2 with duration $[1, t]$
6:          re-run pattern matcher to update sets $p(e)$, $e \in E$
7:       **end if**
8:    **end for**
9:    $t \leftarrow t$+stepsize
10: **until** t >maxlength
11: **return** G

Removing redundancy. The last step removes those edges and nodes from the graph that are no longer needed because they do not provide additional information (or may be derived from transitivity). In particular, the algorithm checks for each '*don't care*'-node v, with exactly one incoming and one outgoing edge, if a path exists from node v_p directly preceding it to the node v_f directly following it. If this is the case the '*don't care*'-node and the connecting edges are removed. This ensures us, that no synchronization between nodes is removed and all constraints are preserved: synchronization requires at least two incoming or outgoing edges and no value constraints are removed (the parallel path subsumes the '*don't care*' constraint). As an example, consider the case that in step two the pattern graph in Fig. 3 has been extended by an edge connecting A and $\neg B$. This would render the '*don't care*' node following A useless because we can follow the path $A \to \neg B \to B \to ? \to \bot$ to reach $\bot$ as well.

3.2 Second phase: discrimination (between classes)

Next, we propose a method to explore the space of pattern graphs to discriminate differently labeled sequences. The search algorithm implements a general-to-specific search: it begins with the pattern graph from phase one, which matches all instances (of the class), and tries to specialize it further to improve some chosen measure of interestingness (e.g. the J-measure). While a propositional rule can only be specialized by an additional condition (like *outlook=sunny*), there are various ways to specialize a pattern graph: we can examine it in a finer resolution (by splitting a node into two or three nodes), we can change or add a value constraint (for some node), or introduce or change an existing temporal constraint. Furthermore we can add new nodes or connect two existing

nodes with an additional edge to express a temporal dependency. We have settled on five different specialization operators to address each of these aspects. The outline of the beam search algorithm is given in Algorithm 2.

Algorithm 2 Outline of the beam search

Require: start pattern graph G_S, set S of labeled sequences
Ensure: set of k best pattern graphs (acc. to some measure of interestingness)

1: initialize the set T with the single start pattern G_S
2: **repeat**
3: $B \leftarrow T$
4: **for all** $G \in B$ **do**
5: apply all refinement operators to G and insert resulting graphs into T (but keep only the k best graphs in T)
6: **end for**
7: **until** k^{th} best graph in T is not better than the kth best graph in B
8: **return** B

The general idea for all refinement operators is to search for specializations that improve the measure of interestingness, which basically requires that the specialized pattern still matches the positive instances but less negatives. Due to lack of space, we will describe only two operators briefly.

Link Refinement. This refinement operator is almost identical to step two of phase one, except that the criterion for path-inclusion is now an increase in some chosen measure of interestingness rather than matching all instances of the considered class. The connection that discriminates best will be included.

Path Refinement. This refinement operator determines for all nodes n of the graph, whether some condition (temporal abstraction) occurs frequently before or after the subsequence assigned to node n. As an example, consider the node labeled A in the pattern graph of Fig. 4(a) as n. Suppose that in some instances of the same class we observe a presence of condition X after n. If the inclusion of 'X after n' increases the measure of interestingness most (among all other possible combinations), we add a short sequence of nodes: $? \rightarrow X \rightarrow ?$ between the node n and $\bot$ (or $\top$ in case the presence of X is observed *before* n). In our example, we may obtain the pattern graph shown in Fig. 4(b).

Note that this refinement operator does not add the constraint between the nodes labeled A and B, because it did not determine the relationship between X and the node labeled B. We thus insert a path to $\top$ or $\bot$ and leave the determination of a possible temporal relationship to B open for further refinements.

Complexity. The number of possible pattern graphs grows quickly with the number of nodes and possible constraints in the data set. But we restrict the number of actually explored patterns with the size of the beam. Apparently we cannot guarantee that the best graph will eventually be found, because beam search is a heuristic search technique. For each main iteration, the patterns in the beam are extended by the five mentioned operators. All of the operators directly

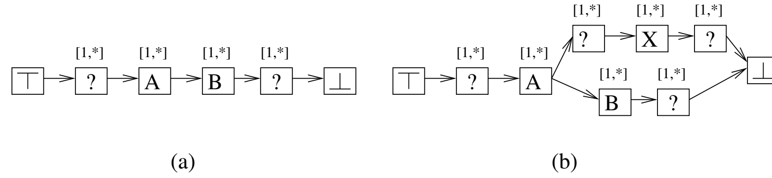

Fig. 4. Example of how the pattern graph (a) is refined by the partial order refinement operator to the pattern graph (b)

work on the result of the matcher, so each instance has to be matched against a pattern graph only once. The complexity of the refinement operators differ: for example, the worst case complexity of finding the best partial order refinement is in $\mathcal{O}(v * n * (m + c * c_n))$ and that of link refinement is in $\mathcal{O}(n * v^2 * m)$, resp. Here v denotes the number of nodes in the graph, n the number of sequences, c the number of different conditions, m the number of matches and c_n the number of occurrences of the conditions in the sequences.

4 Experimental Evaluation

We evaluated the proposed algorithm on real world data from a German car manufacturer. Several cars were equipped with recording devices that captured various measurements, such as current speed, gear, pedal state and angles, etc. The goal is to identify driving cycles with a specific duration in the data, which appears pretty simple at first glance. In a test-bed situation, a driving cycle may be defined as a sequence of acceleration, constant speed and deceleration. However, if we define 'cycle' by such a pattern, it matches far more situations than the experts actually had in mind. Fig. 5 shows one test drive, where the first plot shows the current speed of the car over time and the lower plot the intervals during which various conditions hold. These conditions were derived from the numerical time series data using a priori defined thresholds. We used the following labels:

- gear up/down: indicates whether the last gear shift was up- or down-shift
- g: represents the current gear of the car: no gear (g(..-0.500)), gear one (g(0.500-1.500)), ..., gear 4 or higher (g(4.500-...))
- revolutions: represents the engine revolutions (low, middle, high)
- coupling: clutch is pedaled (coupling(0.500-..)) or not (coupling(..-0.500))

Learning the pattern graph. In [8] we created a pattern graph from scratch by iteratively enhancing the graph with the help of expert knowledge to retrieve the desired driving cycles. In the following we want to show how the above mentioned approach could be used to help the expert specifying the pattern graph or how a pattern graph could be learned if no expert is available.

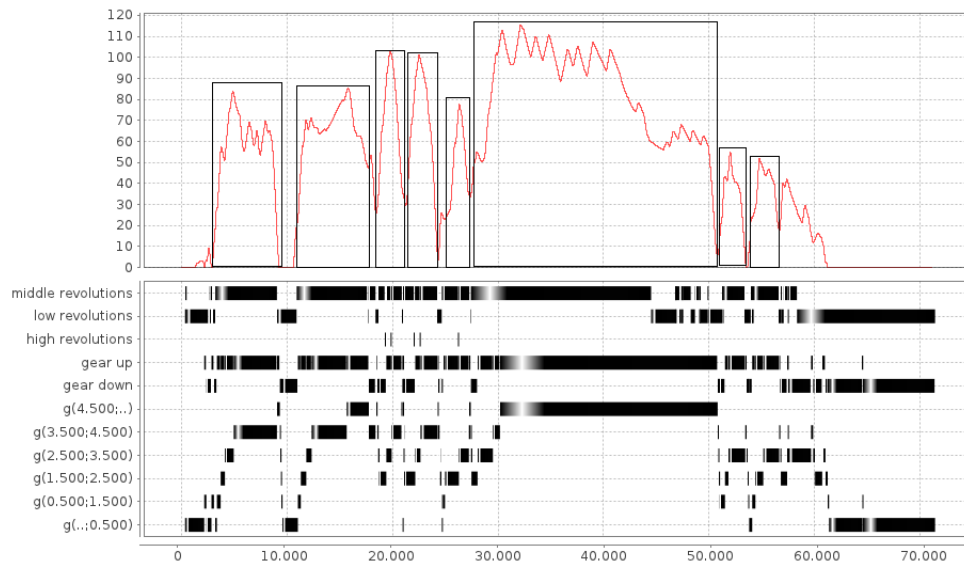

Fig. 5. Possible driving cycles marked in the sequence, which could be used as instances for the target class.

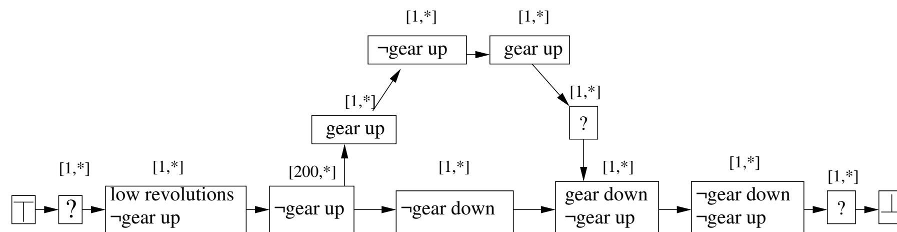

Fig. 6. Pattern graph to query driving cycles.

To derive a pattern for the target class, as with any other classifier we need examples that contain the target class as well as examples that do not (or contain other classes). For illustration purposes, some 'driving cycle' examples are marked by rectangles in Fig. 5. In many applications the sequence labels will be readily available, but in our case we may ask an expert to label some examples manually. However, as we have already specified *a pattern* (rather than marked individual sequences) for 'driving cycles' by means of expert knowledge in [8], which does a pretty good job at identifying cycles, we have decided to use this pattern (shown in Fig. 6) to label example sequences: We have extracted various (random) subsequences and label them 'driving cycle' if the manually constructed pattern matches. The appeal of this procedure is that it allows us to compare the learned graph directly to the manually constructed graph.

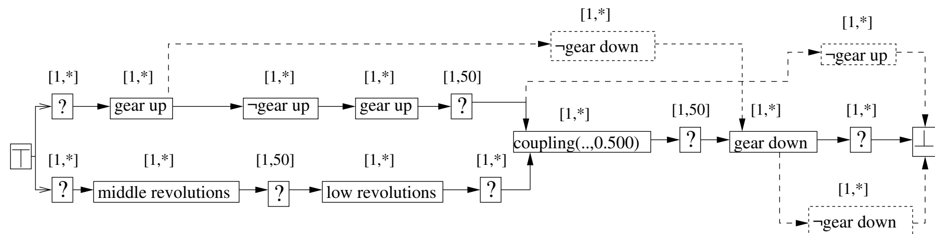

Fig. 7. Pattern graph learned for the driving cycles. The graph without the dashed nodes shows the pattern after phase one and the 3 dashed nodes are added during phase 2.

The pattern graph without the dashed nodes shown in Fig. 7 was found during the first phase of the approach. This graph is already very similar to the graph used for extraction (cf. Fig. 6), the sequence of two up-shifts is prominent in both graphs. The location of the *low revolutions* constraint is different, it appears to occur somewhat later in the extracted graph. However, both parallel paths in the extracted graph synchronize at the *coupling*-node. While the upper part (*gear-up*) is closely connected to this node (time constraint [1,50]), this is not necessarily the case for the lower part. Due to the unlimited temporal constraint of the '*don't care*'-node, the *low revolutions* are allowed to appear before the first *gear up* (as in the original graph). The fact that no temporal dependency is introduced in the extracted path may be explained by the simple fact, that it did not help to discriminate the classes. Furthermore the graph requires *middle revolutions* before *low revolutions*, but of course this is always the case when slowing down before the next cycle starts (except for the first cycle starting from a parking position). An interesting aspect is that between the last *gear up* and the *gear down* node *coupling(..,0.500)* has to occur, which is a stronger constraint than ¬*gear down* because a gear shift requires coupling. If we apply the learned pattern to all instances we retrieve the confusion matrix

$\left(\begin{smallmatrix}300 & 0\\ 174 & 249.\end{smallmatrix}\right)$, where we can see that all ‘cycles’ are matched correctly however we also obtained 174 false positives. This was to be expected as we were not trying to separate the ‘cycles’ from other sequences so far.

This is the task of the subsequent phase, which tries to reduce the number of false positives by adding constraints to discriminate the target class from all other classes. The resulting pattern is also shown in Fig. 7 if we include the dashed nodes. The refinement operators added three constraints to the graph: The first constraint is the ¬*gear down*-node between the first *gear up*- and *gear down*-node, thus stating that no down-shift is allowed during the ‘cycle’. The second refinement is the ¬*gear up* node connected from the *don't care*-node after the last gear up node to ⊥, which ensures that after the last up-shift no further up-shift can occur. The last additional constraint states that after the *gear down*-node no further change in a lower gear is permitted (additional ¬*gear down* node connected between *gear down* and ⊥).

By comparing the learned pattern (after phase 2) in Fig. 7 to the pattern used for the extraction in Fig. 6 we can see that the patterns describe a nearly identical ‘driving cycle’. The beam search step only inserted constraints that helped to separate the randomly selected sequences from the ‘cycles’. All these constraints are also found in the manually defined graph that was used for labeling the sequences. As the graphs are nearly the same it is not surprising that the learned pattern performs perfectly as indicated by the confusion matrix: $\left(\begin{smallmatrix}300 & 0\\ 0 & 439.\end{smallmatrix}\right)$.

Libras Movement. We also applied our algorithm to the libras movement data set from the UCI repository [4]. It contains 15 different signs described by their characteristic hand movement over 45 frames, where the current x- and y-positions of the hand were recorded. We extracted features to address the speed of the hand movement in the x- and y-direction only. For each sign we learned a pattern graph on 66% of the data and then matched all pattern graphs to the unseen data. We predict a class only if just one graph matches (and is ‘undecided’ otherwise). We arrive at 98 correct, 2 false and 23 unclassified instances, resulting in 79.675% accuracy and 20.325% error-rate. For the 23 unclassified instances (where no pattern graph matches), we can switch back to manual mode and inspect and change the pattern manually to further improve the classification rate. The dataset contains some cases that deviate greatly from the original hand movement, such that even a human is not able to classify them. By removing these outliers the approach improves to 88.235% accuracy and 11.765% error rate.

5 Conclusion

We consider pattern graphs as useful for capturing multivariate patterns in temporal data, because they are capable of expressing most of the constraints a human expert may want to use when describing a specific situation (e.g. absence of events, durations, different kinds of parallelism). These graphs are thus well-suited for manual construction [8], but in this paper we have demonstrated that they can also be learned automatically from data. We have presented a two-

phased approach to construct pattern graphs for classification tasks. The first phase identifies the common structure in all sequences of the same class and the second phase refines this structure further to discriminate between the different classes. Early results are encouraging, the approach was able to successfully re-discover hand-coded pattern graphs from a set of labeled examples.

Acknowledgements We would like to thank Dr. Werther from Volkswagen AG for kindly providing the data.

References

1. T. M. Basile, N. Di Mauro, S. Ferilli, and F. Esposito. Relational temporal data mining for wireless sensor networks. In *AI*IA 2009: Emergent Perspectives in Artificial Intelligence*, number 5883 in LNAI, pages 416–425, 2009.
2. I. Batal, H. Valizadegan, G. F. Cooper, and M. Hauskrecht. A pattern mining approach for classifying multivariate temporal data. In *Proc. IEEE Int. Conf. Bioinformatics BioMed*, pages 358–365, 2011.
3. M. Berlingerio, F. Pinelli, M. Nanni, and F. Giannotti. Temporal mining for interactive workflow data analysis. In *Proc. 15th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, KDD '09, pages 109–118, 2009.
4. A. Frank and A. Asuncion. UCI machine learning repository. University of California, Irvine, School of Information and Computer Sciences, 2010.
5. F. Höppner. Discovery of temporal patterns – learning rules about the qualitative behaviour of time series. In *Proc. of the 5th Europ. Conf. on Principles of Data Mining and Knowl. Discovery*, pages 192–203. Springer, 2001.
6. F. Höppner, S. Peter, and M. R. Berthold. *Enriching Multivariate Temporal Patterns with Context Information to Support Classification*, volume 445 of *Studies in Computational Intelligence*, pages 195–206. Springer Berlin / Heidelberg, 2012.
7. F. Mörchen. Unsupervised pattern mining from symbolic temporal data. *ACM SIGKDD Explorations Newsletter*, 9(1):41–55, 2007.
8. S. Peter, F. Höppner, and M. R. Berthold. Pattern graphs: A knowledge-based tool for multivariate temporal pattern retrieval. In *Proc. IEEE Conf. Intelligent Systems*. IEEE, 2012.
9. J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *Int. Conf on Data Engineering*, pages 79–90, 2004.
10. L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, 1994.