A Platform for Matching Context in Real Time

Andrei Olaru and Adina Magda Florea

University Politehnica of Bucharest,
313 Splaiul Independentei,
060042 Bucharest, Romania

Abstract. Context-awareness is a key feature of Ambient Intelligence and future intelligent systems. In order to achieve context-aware behavior, applications must be able to detect context information, recognize situations and correctly decide on context-aware action. The representation of context information and the manner in which context is detected are central issues. Based on our previous work in which we used graphs to represent context and graph matching to detect situations, in this paper we present a platform that completely handles context matching, and does so in real time, in the background, by deferring matching to a component that acts incrementally, relying on previous matching results. The platform has been implemented and tested on an AAL-inspired scenario.

Keywords: graph matching, context matching, context patterns, software agents

1 Introduction[1]

Ambient Intelligence – or AmI – is one of the current priorities in the world of intelligent distributed systems. In order to appear as truly intelligent, and in order to provide the user with the appropriate information at the right time, or with the appropriate, non-intrusive assisting action [1], AmI relies on several essential features, such as system distribution, fusion of information from a large number of sensors, detection of context and context-aware action. Context-awareness [2] is the ability of a system or application to correctly identify the situation of the user based on a large quantity of information, and to take appropriate action in that situation.

This work has as framework the AmIciTy[2] initiative to build a software infrastructure for Ambient Intelligence. The initiative relies on two key features. One is the use of agent technology as an enabler of individual autonomy and of distributed, reliable behavior for the system. The other is a representation of

[1] The final publication is available at http://link.springer.com/chapter/10.1007/978-3-319-19644-2_9.

[1] This work has been funded by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/134398.

[2] See more details at http://aimas.cs.pub.ro/amicity .

context information that is adequate for a distributed system, allowing both an efficient management of context information at the system level, and context-awareness at the individual level, without the mandatory presence of context servers and other centralized components.

In the architecture of the AmIciTy system [3], context information is handled in a distributed manner by persistently storing in each agent the context information that is relevant to its activity. Agents also exchange between them information that is potentially relevant to other agents. Context information is stored in agents as *context graphs* (or CGs), that are very much like semantic networks – graphs having concepts as nodes and relations as edges. Each agent also stores a set of *context patterns* (or, in short, Patterns), that describe situations as graphs with generic (wildcard) nodes. Agents detect the current situation by matching patterns against the CG and take action as indicated by the matching pattern(s). This is what we call *context matching*.

Although graphs are a great way to visually and comprehensively represent information, the drawback of our method is that the general problem of graph matching is NP-complete [4]. However, we have shown in the past that since most of the nodes and edges in context graphs are labeled, the computational effort of matching the graphs is greatly reduced, by using a purpose-built algorithm that starts from single-edge matches and grows them to reach the maximum common subgraph (MCS).

This paper deals with how to perform context matching in an efficient, timely fashion, even when context changes very quickly. In this work we describe the architecture and implementation of a context matching platform that increases the efficiency of matching by relying on two facts: first, changes in the context graph are incremental, even if they are rapid; second, the context patterns remain quasi constant throughout the operation of the system – they are added or removed relatively rarely. This means that, at the expense of keeping part of the (partial) matches in memory, new matches can be obtained quickly when the context graph changes, based on the partial matches stored in memory.

The proposed platform is able to handle rapid changes in the context graph while performing matching in the background. It uses queues of transactions to follow changes in the sequence in which they happened. Matches that are detected in the background are notified to the host process through a mechanism of notifications. This paper not only introduces these new features of platform, but also makes a comprehensive, focused presentation of the whole architecture, in order to allow, together with the open source implementation[3], the use and replication of these results by other researchers.

After discussing some related work, we present the formalism of context graphs in Section 3, followed by the introduction of the Continuous Context Matching Platform and related concepts in Section 4. Section 5 shows some experimental results and the last section draws the conclusion.

[3] The implementation is freely available under a GPLv3 license at https://github.com/andreiolaru-ro/net.xqhs.Graphs .

2 Related Work

Modeling of context information uses representations that range from tuples to logical, case-based and ontological representations [5]. The most popular approaches are ontologies for representing situations and rules for reasoning, coupled with propositional or predicate logic to represent current context information. We have previously advocated context graphs as an appropriate method to represent context, that couples a simple theoretical formalism with a visual representation and powerful algorithms for graph matching [6]. Ontologies are used in many projects to describe potential situations or situation elements and to establish the relations among the elements of context. Several ontologies have been created specifically for use in context-aware computing (e.g. SOUPA [7]). The main criticism regarding ontologies is a lack of support for temporal relations, the lack of dynamicity, and the large space and temporal complexity required for ontological reasoning [8]. The mechanisms we propose are directed towards system distribution, local storage of context information and local reasoning. The work of Turner et al shows how context-mediated behavior (CMB) [9] can be used to adapt behavior to cases that have not been encountered before, but share similarities with existing cases, much like we use patterns for context recognition. Our research is somewhat similar in behavior but we use structures that are easy to represent graphically and to visualize.

In terms of using graphs, in their works in 2004 and 2014 respectively, Conte and Foggia [4,10] analyze the use of graphs in pattern matching in the last 40 years. Graph matching has gained traction since the beginning of the millennium, as computational power increased and NP-complete problems became more approachable. It is notable that in every application domain where graphs are used there are specific challenges related to the process of pattern matching, but algorithms are customized starting from classical, generic graph-matching algorithms, like the ones enumerated in the rest of this section. None of those fields has, however, the same particular constraints as our problem, therefore in this and previous work, algorithms had to be adapted to solve it.

We may classify graph matching algorithms in two major categories: exact matching, when the reference structure must be found entirely in the examined structure; and inexact matching, when a match might be valid even if the two entities are different to a certain extent. Among the most important algorithms for matching of unlabeled graphs are tree-search algorithms [11] and algorithms for the matching of a graph against a library of graphs [12]. Some algorithms, especially those for inexact matching [13], are based on powerful mathematical instruments – like expectation maximization [14], graduated assignment [15], and learning of assignment coefficients [16].

We have previously adapted several popular algorithms for graph matching in order to observe their behavior on context matching problems [17]. The algorithms that we have focused on were algorithms that can be adapted to the problem of context matching: they rely on label comparison and can be adapted to deal with generic edges and nodes. Among them, algorithms using incremental matching by exploring the entire state space (McGregor's algorithm [18]); algorithms using the equivalence between finding a maximal clique and finding the maximum common subgraph (algorithms by Bron-Kerbosch [19], Durand-Pasari [20], Akkoyunlu [21] and Balas-Yu [22]); and algorithms using the equivalence with the maximal clique, but considering an extended modular product of the edges, not of the nodes (Koch's [23]). While we have found that some of these algorithms have certain advantages with respect to our problem, there was room for improvement. While testing algorithms for matching a pattern to a graph, the algorithm we have developed in previous work was by far the most efficient for all test cases.

While connected with, and sometimes inspired by related research, this work is innovative not only due to the approach we have to using graph matching for context awareness, but especially due to the purpose-built algorithms and methods that we have developed in order to make context matching viable in a distributed setup of resource-constrained devices.

3 Formal Model

In an Ambient Intelligence system, each agent should have a representation of the information that is interesting to it, and also the means of detecting what information is interesting to it from the stream of information that it receives [6]).

Each agent A has a *Context Graph* $CG_A = (V, E)$ that contains the information that is currently relevant to its function. Considering a global set of *Concepts* (strings or URIs) and a global set of *Relations* (strings, URIs or the empty string, for unnamed relations), we have:

$CG_A = (V, E)$, where $V \subseteq Concepts$

$E = \{edge(from, to, value, persistence) \mid from, to \in V, value \in Relations\}$

In order to implement *forgetting* information, or limited validity of information, edges feature an element of persistence. They can be permanent, or they may have an 'expiration time', after which they are removed. The persistence of edges is set when they are added to the graph, according to settings in the pattern that generated the edge (see below).

In order to detect relevant information, or to find potential problems, an agent has a set of patterns that it matches against graph CG_A. These patterns describe situations that are relevant to its activity. A pattern s is defined by a graph G_s^P. We will use the " ^{P} " superscript to mark structures that support generic elements, such as generic nodes:

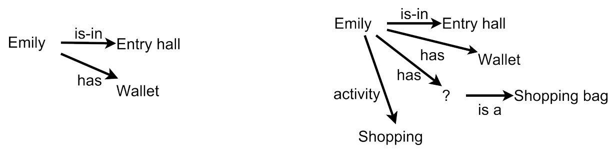
$G_s^P = (V_s^P, E_s^P)$

$V_s^P \subseteq Concepts \cup \{?\}$

$E_s^P = \{edge(from, to, value) \mid from, to \in V_s^P, value \in Relations \cup \{\lambda\}\}$

We have used λ as a notation for the empty string. Examples of a pattern and a graph are shown in Figure 1.

By using graph matching algorithms – matching a pattern from the agent's set of patterns against the agent's context graph – an agent is able to detect interesting information and is able to decide on appropriate action to take.



(a) (b)

Fig. 1. Example of context graph and pattern. The pattern 3-matches the context graph.

The pattern G_s^P *matches* the subgraph $G'_A = (V', E')$, *iff* there exists an injective function $f_v : V_s^P \to V'$, so that the following conditions are met simultaneously:

(1) $\forall v^P \in V_s^P, v^P =?$ or $v^P = f(v^P)$ (same value)

(2a) $\forall edge(v_i^P, v_j^P, rel) \in E_s^P$, $edge(f(v_i^P),\ f(v_j^P),\ value) \in E'$, $value \in \{rel, \lambda\}$

(2b) $\forall edge(v_i^P, v_j^P, \lambda) \in E_s^P$, $\exists value \in Relations$, $edge(f(v_i^P),\ f(v_j^P),\ value) \in E'$

That is, every non-? vertex in the pattern matches (has the same label) a different vertex from G'_A (f_v is injective), and every edge in the pattern matches (same label for the edge and vertices) an edge from G'_A. Subgraph G' should be minimal (no edges that are not matched by edges in the pattern). One pattern may match various subgraphs of the context graph.

We allow partial matches. A pattern G_s^P *k-matches* a subgraph G' of G, if conditions (2) above are fulfilled for $m_s - k$ edges in E_s^P, $k \in \{1..m_s - 1\}$, $m_s = ||E_s^P||$ and G' remains connected and minimal. The k number of a match is the number of edges in the pattern that have not been found in the graph. For a complete match, k is null.

Partial matches are useful because, depending on a set threshold for k, they indicate actionable cases. A small k indicates that the user is indeed in the situation described by the pattern. A strictly positive k indicates however that there is something missing. Depending on the settings of the pattern, this either indicates that the agent should issue a notification about the missing edge(s), or, if the missing edges are *actionable*, the agent may add them to the graph, with a certain persistence.

We have also developed a formalism, called *timelines*, for higher-level patterns which match a certain temporal sequence of pattern matches. While the context matching platform currently supports the detection of timelines, the focus of this paper is on efficiently matching individual patterns.

4 Platform Architecture

The architecture of the platform that is presented in this paper has been designed specifically for the problem of context matching. More precisely, having a context graph, presumably quite large, and a number of context patterns (this number

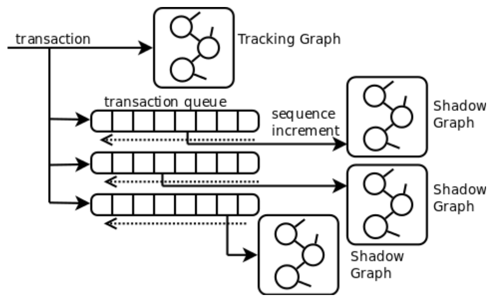


Fig. 2. Shadow graphs take transactions from transaction queues to which the tracking graph appends operations.

depends on the complexity of the functionality of the device / agent), that have a relatively small size compared to the graph, we desire to obtain notifications when a match is found between the current CG and a Pattern (the k is chosen by the user), and also when a previously reported match disappears.

Performance-wise, we desire that the notification comes in a timely fashion (while allowing some small delay), and that no changes are overlooked. For instance, if, as a result of a perception, an edge appears in the CG, and then immediately disappears, and that edge completes a match, we wish that match to not be missed, even if before these events the system worked on obtaining other computationally intensive matches.

The challenges in developing the platform were, on the one hand, to not miss any changes, and on the other hand, to obtain reasonable performance at the expense of some memory space.

The platforms relies on an algorithm developed in previous work [6], that is limited to matching one pattern to one graph. In short, the algorithm grows and merges matches. It starts with creating one potential match for each pair of matching edges in the graph and in the pattern (same edge label and matching labels for adjacent vertices). For each match a set of candidates for merger is computed, based on common frontier vertices in the two matches and on the fact that matches should not overlap. Matches that share a common frontier are 'immediate' merger candidates. Matches that are compatible, but not adjacent, are 'outer' merger candidates. In the second phase of the algorithm, using these two candidate sets for each match, matches are merged incrementally with immediate candidates. The gist is that compatibility between matches does not need to be checked in the second phase, because it is insured by computing new candidate sets as set operations between the candidate sets of the merged matches. For instance, the resulting outer candidate set is the intersection between the outer candidate sets of the merged matches – that is, candidates that were compatible (not overlapping) with, but not immediate merge candidates for any of the two merged matches.

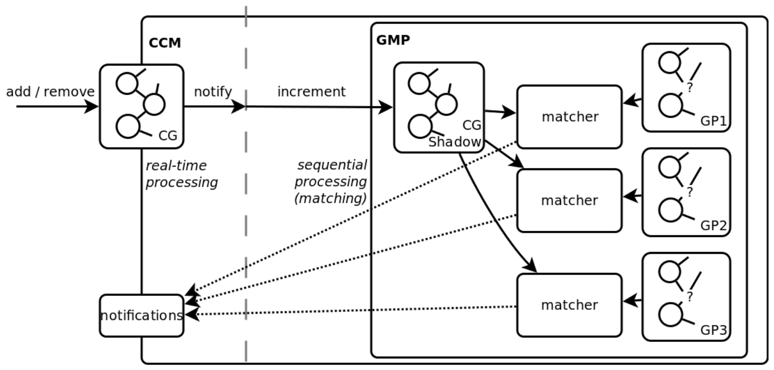


Fig. 3. Architecture of the Continuous Context Matching Platform. The dotted line separates the part that works in real time from the components that do matching sequentially, in the order of transactions applied to the Context Graph.

4.1 Tracking Changes

In order to correctly track changes in the context graph, we have created a special structure called a *tracking graph*. As the normal operations in a graph are addition and removal of nodes and edges, the tracking graph is modified by means of *transactions*. A transaction contains any number of operations, each of the operations being the addition or removal of a node or edge. The only condition is that the same transaction does not simultaneously contain the addition and the removal of the same graph component. Whenever a transaction is *applied* to a tracking graph, the operations in the transaction are applied immediately. A transaction is considered atomic – there is no state in which only part of the operations in a transaction are applied.

However, a tracking graph may have any number of *shadow graphs* (see Figure 2). When it is created, a shadow graph is a snapshot of the tracking graph (they are identical). Whenever a transaction is applied to the tracking graph, it is added to the transaction queue of each shadow graph. Shadow graphs only update their state when an *increment* operation is invoked. One increment invocation causes the shadow graph to take one transaction from the queue and apply it to the graph.

This way, the matching process can be safely performed on the shadow graph, regardless of what changes happen in the tracking graph in the mean time. When the matching is finished, and if changes have occurred, the shadow graph is incremented and the matching process is started again.

4.2 Incremental matching

When using the algorithm presented at the beginning of the section, whenever the context graph changes, the matching process must be done again completely, wasting resources even if, for instance, an edge has been added that cannot be found in any pattern. Resources could be saved, because, if the CG changes only slightly, for every pattern most of the partial matches, and potentially the matching process entirely, remain unchanged with respect to the previous matching.

It is therefore possible to remember partial matches that have been created previously. When a new edge is added, and it matches an edge in a pattern, a single-edge match is created, candidates are computed, and potentially the match is merged with a pre-existing, hopefully maximal, match. Computing the candidate sets is not even too difficult: the match is compatible with all matches that do not contain the pattern edge, and it can be immediately compatible only with candidates containing neighbor edges (in the pattern).

Of course, it may be that with modifications in the graph, some partial matches that are stored become invalid. Therefore an index is stored of what edges in the graph are part of which matches. Whenever an edge is removed, all matches that it was part of are removed as well.

4.3 Continuous Context Matching Platform

The CCM platform (see Figure 3) completely deals with the matching of a set of patterns against a context graph. At any time, the context graph can be modified. The CCM platform uses, internally, a component that is called a Graph Matching Platform (GMP). The GMP is capable of obtaining all matches of a set of patterns against a context graph. An incremental process, called a Matching Process, is attached to each of the Patterns. When there are changes in the CG, Matching Processes are executed by a pool of threads, giving priority to the patterns which are closest to the changed edges (simple label comparison is performed). Matching Processes work on shadows of the CG.

A Matching Process holds two indexes of matches – for each edge in the graph, the list of matches containing that edge, and for each in the pattern, the list of matches containing it. When an edge in the graph is removed, all matches containing that edge are declared invalid and will be removed whenever they are iterated over. When an edge is added to the graph, initial matches are created (against matching edges from the pattern) and merge candidates are added from among the matches of the neighbor pattern edges.

4.4 Complexity Considerations

While the matching of graphs with unlabeled nodes and edges is an NP-complete problem, labels greatly improve the performance of the process. This section extends our previous analysis [6] but focuses on real-time matching.

Since a certain number of matches are kept in memory and the matching is done incrementally, the largest computational effort is when the initial matches are created, that is when new Patterns are added. Later, matching processes are executed whenever an edge is added or removed in the CG. When an edge is added, It is first matched against all edges in the pattern in an attempt to create a single-edge match. For all patterns, this means one label comparison

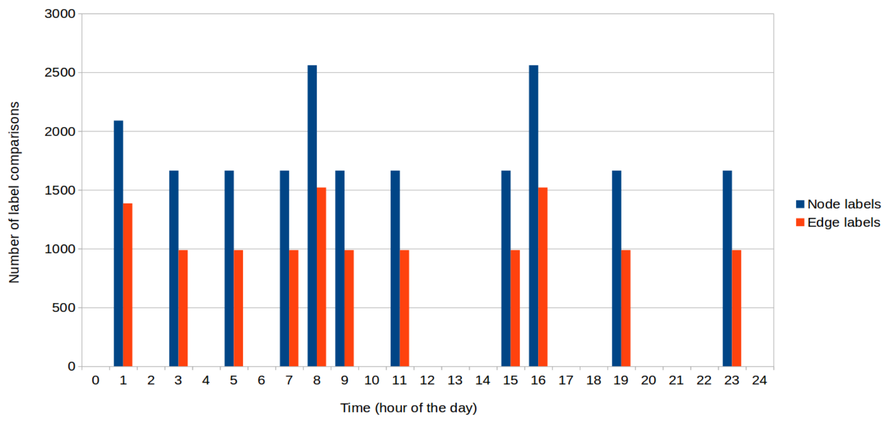


Fig. 4. The number of label comparisons for nodes and edges in every hour-long interval.

for each pattern edge. Next, the candidate sets are computed for each resulting single-edge match, which is done in $O(\overline{m_n})$, where $\overline{m_n}$ is the mean number of neighbor edges for an edge in the pattern. As seen in Section 5.2, only single-edge matches are significant computationally. Practically, the computational effort is proportional to the number and size of patterns, but also to the branching factor of patterns. Thanks to indexing, removing an edge from the CG is done in $O(1)$ for each pattern.

From the point of view of memory consumption, results show that storing partial matches consists mostly in single-edge matches. For a CG and a pattern with no labeled edges and nodes, there is a maximum of $m \times m^P$ matches, but when edges and vertices are labeled, there is one single-edge match per pair of CG edge and pattern edge with the same label and matching vertices. Practically, performance is better when the pattern is less ambiguous.

5 Experiments with Matching Context

The Continuous Context Matching Platform was implemented[2] in Java, so it can be executed on workstation platforms as well as on Android devices. The tested scenario, taken from the AAL domain, takes place in the home of an elderly woman named Emily. Emily is aged 87 and lives alone in a small apartment. Motion sensors in the home track Emily. The system, running on a limited piece of hardware, must be able to promptly detect the current activity of Emily, based solely on location detection.

5.1 Experimental Setup

Each experiment is a simulation using an automatically generated scenario which takes place over 24 hours – one day in Emily's life.

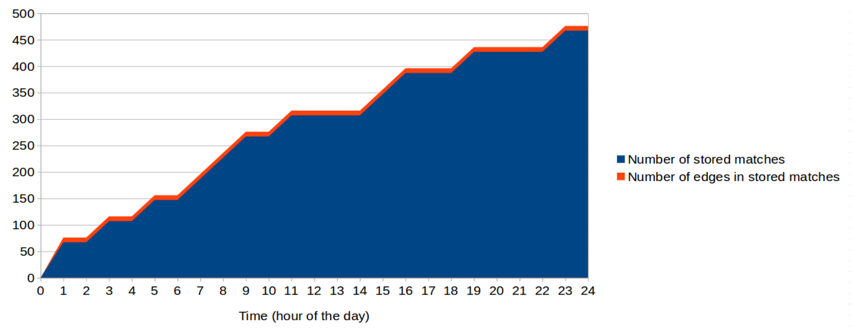


Fig. 5. The number of matches and labels stored in memory to allow incremental matching.

Emily's apartment is consists of a living room, a hall, a kitchen and a bathroom. Each of the first three rooms is equipped with a motion detector, and there is a detector to know whether someone is near the bathroom door. There are no sensors in the bathroom. The context graph of the system contains nodes referring to the current state of Emily, the layout of the rooms, food in the fridge, etc (about 50 nodes).

Emily sleeps between 10PM and 7AM. During the day, she eats two or three times, goes to the bathroom, and may take a shower. At times she wanders around the house and spends time in the kitchen without eating anything, just sitting down and looking out of the window.

In order to generate a 24h long scenario, we use a generator that inserts various activities (one of sleeping, having a meal, going to the bathroom, taking a shower, wandering around the house, and doing nothing) at various moments of time, using a distribution of probability for each type of activity. For example, if Emily just ate, she will not eat again for the following 3 or 4 hours, but after that the probability of her deciding to eat will increase with time. Depending on their type, some activities may have variable durations.

5.2 Results

We have executed the Continuous Context Matching Platform on 24-hour long scenarios in compressed time. We have mapped every second in simulation time to a millisecond in real time, but the platform had no problem completing the matches in that millisecond, even on older, slower, machines.

Incremental matches mean that the number of operations is very low at any given time. In Figure 4 a chart is shown of the number of label comparisons, for nodes and for edges, for every hour of simulated time. We have chosen to show label comparisons because they are the most computationally intensive operation while performing the matching.

Of course, incremental matching brings performance at the expense of memory consumption. In order to evaluate the tradeoff, we have monitored how the number of stored matches evolves over time. We have also inspected the total number of edges in stored matches. These results are shown in Figure 5, in the context in which no optimization of memory space was done (e.g. activating a strategy in which some partial matches are removed in time).

A first observation would be that the number of matches grows steadily with time and does not reach a very high number. The matches can be stored even on devices with low capabilities. Secondly, it is interesting to observe that the difference between the number of matches and the total number of edges stored in matches is very small. Basically, all matches are single-edge matches. The platform could therefore be optimized to keep only larger (much fewer) matches and recreate less used single-edge matches, in case memory is constrained and the device handles many patterns.

6 Conclusion

Although the problem of graph matching is computationally difficult, using graph for representing context is an approach that is flexible, easy to understand, and suitable to detecting context by matching patterns against a graph. This paper presents a platform for graph matching that uses tracking graphs in order not to miss any rapid sequences of operations, and uses incremental matching, at the expense of some memory, to keep the useful results in the matching process for future matchings.

An implementation is currently underway for a set of tools that allows working in a uniform manner with varied datasets of activity data, some very large. The presented platform will be deployed against such datasets in the near future. On the medium term, our goal is to build a large experiment in which context matching will be used by a large number of agents, on the same machine, so as to full understand the impact of the performance and memory tradeoffs that exist in the implementation.

As a long term goal, the deployment of AmIciTy on multiple machines and platforms (such as smartphones) will enable us to apply machine learning to improve patterns and learn new patterns by tracking the user's activity.

References

1. Sadri, F.: Ambient intelligence: A survey. ACM Computing Surveys (CSUR) **43**(4) (2011) 36
2. Perera, C., Zaslavsky, A., Christen, P., Georgakopoulos, D.: Context aware computing for the internet of things: A survey. IEEE Communications Surveys and Tutorials **16**(1) (2013) 414–454
3. Olaru, A., Florea, A.M., El Fallah Seghrouchni, A.: A context-aware multi-agent system as a middleware for ambient intelligence. Mobile Networks and Applications **18**(3) (June 2013) 429–443
4. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. International journal of pattern recognition and artificial intelligence **18**(3) (2004) 265–298

5. Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A., Riboni, D.: A survey of context modelling and reasoning techniques. Pervasive and Mobile Computing **6**(2) (2010) 161–180
6. Olaru, A.: Context matching for ambient intelligence applications. In Björner, N., Negru, V., Ida, T., Jebelean, T., Petcu, D., Watt, S., Zaharie, D., eds.: Proceedings of SYNASC 2013, 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, September 23-26, Timisoara, Romania, IEEE CPS (2013) 265–272
7. Chen, H., Finin, T., Joshi, A.: The SOUPA ontology for pervasive computing. Ontologies for Agents: Theory and Experiences (2005) 233–258
8. Bolchini, C., Curino, C., Quintarelli, E., Schreiber, F., Tanca, L.: A data-oriented survey of context models. ACM SIGMOD Record **36**(4) (2007) 19–26
9. Turner, R.M.: Context-mediated behavior. In: Context in Computing. Springer (2014) 523–539
10. Foggia, P., Percannella, G., Vento, M.: Graph matching and learning in pattern recognition in the last 10 years. International Journal of Pattern Recognition and Artificial Intelligence **28**(01) (2014)
11. Cordella, L., Foggia, P., Sansone, C., Vento, M.: A (sub) graph isomorphism algorithm for matching large graphs. Pattern Analysis and Machine Intelligence, IEEE Transactions on **26**(10) (2004) 1367–1372
12. Messmer, B., Bunke, H.: Efficient subgraph isomorphism detection: A decomposition approach. Knowledge and Data Engineering, IEEE Transactions on **12**(2) (2000) 307–323
13. Bengoetxea, E., Larrañaga, P., Bloch, I., Perchant, A., Boeres, C.: Inexact graph matching by means of estimation of distribution algorithms. Pattern Recognition **35**(12) (2002) 2867–2880
14. Luo, B., Hancock, E.: Structural graph matching using the EM algorithm and singular value decomposition. Pattern Analysis and Machine Intelligence, IEEE Transactions on (2001) 1120–1136
15. Gold, S., Rangarajan, A.: A graduated assignment algorithm for graph matching. Pattern Analysis and Machine Intelligence, IEEE Transactions on **18**(4) (1996) 377–388
16. Caetano, T., McAuley, J., Cheng, L., Le, Q., Smola, A.: Learning graph matching. IEEE transactions on pattern analysis and machine intelligence (2009) 1048–1058
17. Dobrescu, A., Olaru, A.: Graph matching for context recognition. In Dumitrache, I., Florea, A.M., Pop, F., eds.: Proceedings of CSCS 19, the 19th International Conference on Control Systems and Computer Science, May 29-13, Bucharest, Romania, IEEE Xplore (2013) 479–486
18. McGregor, J.J.: Backtrack search algorithms and the maximal common subgraph problem. Software: Practice and Experience **12**(1) (1982) 23–34
19. Bron, C., Kerbosch, J.: Algorithm 457: finding all cliques of an undirected graph. Communications of the ACM **16**(9) (1973) 575–577
20. Durand, P.J., Pasari, R., Baker, J.W., Tsai, C.c.: An efficient algorithm for similarity analysis of molecules. Internet Journal of Chemistry **2**(17) (1999) 1–16
21. Akkoyunlu, E.: The enumeration of maximal cliques of large graphs. SIAM Journal on Computing **2**(1) (1973) 1–6
22. Balas, E., Yu, C.S.: Finding a maximum clique in an arbitrary graph. SIAM Journal on Computing **15**(4) (1986) 1054–1068
23. Koch, I.: Enumerating all connected maximal common subgraphs in two graphs. Theoretical Computer Science **250**(1) (2001) 1–30