Context Matching for Ambient Intelligence Applications[1]

Andrei Olaru
Computer Science Department
University Politehnica of Bucharest
313 Splaiul Independentei
060042 Bucharest, Romania
Email: cs@andreiolaru.ro

***Abstract*—Reliable and scalable Ambient Intelligence means a distributed system of agents that are capable of working together or autonomously, depending on the requirements of the situation. In previous research we have argued in favor of the use of a representation for context information that can be distributed among agents, so that each agent knows only the information that is relevant to its activity. Recognizing interesting information or relevant situations is done by using context patterns – graph patterns with potentially unknown nodes and edges labeled with regular expressions.**

In this context, a major challenge is for agents to use a graph matching algorithm that is adequate to the possibilities of the devices on which the agents are running. Moreover, it is necessary that the algorithm is able to provide partial matches. This paper presents an algorithm specifically designed for this problem, that uses growing partial matches to obtain the maximum subgraph of the context graph that matches (part of) the context pattern. Experiments were performed with the algorithm and its performance has been compared with that of other algorithms adapted to our problem.

I. Introduction

One of the most important paradigms that have been suggested for the implementation of Ambient Intelligence [1], [2] environments is the agent-oriented approach. Agents can serve well the needs of AmI in terms of distribution, autonomy, fault tolerance and proactive / anticipative behavior [3].

There are two aspects in the real-scale deployment of an Ambient Intelligence system that are central to our work. One of them is scale, and how will the system remain available and useful throughout high loads and/or faults. The other is information transfer. Mark Weiser, considered the father of Ubiquitous Computing, sees such an environment as a "world of information conveyers" [4].

In an ideal future deployment, Ambient Intelligence will be a unified system that interconnects devices that are present in every object, in every material, in the whole world. The system will assist everyone continuously, in any situation, provide them with the appropriate information and actions with no delay, and even with a certain degree of anticipation. This behavior would be the result of processing large quantities of information at the global level. These requirements also call for features like robustness, resilience and availability.

This research is framed by a larger initiative (AmIciTy – *Ambient Intelligence for the Collaborative Integration of Tasks*) to build an agent-based environment for AmI applications in which context is a first-class notion. The elements of this initiative are described in some previous work: at the core of the platform is the context-aware transfer of information between the agents, as well as a generic class of context-aware actions [5]. This context-aware behavior of agents relies on the use of context graphs and context patterns – simple, flexible, yet powerful representations for the user's context and for known situations. By matching context patterns against the user's context graph, an agent can recognize the user's situation and act accordingly, by detecting incomplete matches of context patterns in the context graph and suggesting missing edges [6]. The communication between agents is based on the principle that every agent sends the information that it deems interesting to the agents that share context with it and that may be interested in that piece of information [7]. "Interesting" here means that it matches a pattern from the agent's set of patterns. In fact, most of the generic, non application-specific activity of the context-aware agent is based on context matching (matching context patterns against the agent's context graph). It is used to detect if the information from other agents is relevant to the agent; it is used to recognize the user's situation and propose possible actions; and it is used to detect information that may be interesting to neighbor (context-wise) agents.

While the argument about using context graphs and patterns for AmI applications has already been made in our previous work, we cannot forget that a usable AmI system needs to offer good performance in order to respond promptly to the user's needs and act with anticipation should the situation require it. The question is therefore how the matching of context patterns against graphs (what we call *context matching*) can be done by agents in adequate time, considering the whole range of agents in the system (large and small). The answer that is proposed in this paper is an algorithm for matching context patterns against context graphs that has been created and optimized

[1]The final publication is published by IEEE CPS
Olaru, A., *Context Matching for Ambient Intelligence Applications* in Bjrner, N.; Negru, V.; Ida, T.; Jebelean, T.; Petcu, D.; Watt, S. and 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, pages 265-272

specifically for the problem of context matching.

The proposed algorithm starts from individual matches – only one edge from the pattern that matches one edge from the graph – and continues with merging pairs of matches until a maximum match is created. We have tested the algorithm with several examples from AmI scenarios and we have obtained very satisfactory results compared with classic graph-matching algorithms.

The following section presents some related work in the fields of context-awareness in AmI and graph matching. Based on the formal model presented in Section III, we propose in Section IV a new context matching algorithm, that is further exemplified and tested in Section V. The last section draws the conclusions.

II. Related Work

Infrastructures for processing context information and building context-aware applications are usually centralized and use several layers – going from sensors, through pre-processing, storage and management, to the application that uses the context information [8]. In our approach [5], we attempt to build an agent-based infrastructure that is decentralized, in which each agent has knowledge about the context of its user, and the main aspect of context-awareness is based on associations between different pieces of context information. This makes the system be context-aware not only by consuming context information, but also by creating and disseminating it through the system.

Modeling of context information uses representations that range from tuples to logical, case-based and ontological representations [9]. Using graphs and patterns [6] leads to more flexibility and a simple basic mechanism, that is more adequate to a constantly changing dynamic context.

Graph matching algorithms have started to gain momentum at the end of the 70s and were used for pattern-recognition (PR) problems (especially image recognition and segmentation). Recently the interest has risen once again, as increased processing power makes graph matching approachable. It is a process that can benefit applications in image (static and video) analysis, chemistry, document processing, biometric analysis, and biomedical applications. It is notable that in every application domain 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 [10] and algorithms for the matching of a graph against a library of graphs [11]. Some algorithms, especially those for inexact matching [12], are based on powerful mathematical instruments – like expectation maximization [13], graduated assignment [14], and learning of assignment coefficients [15].

We have previously adapted several popular algorithms for graph matching in order to observe their behavior on context matching problems [16]. 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 [17]); algorithms using the equivalence between finding a maximal clique and finding the maximum common subgraph (algorithms by Bron-Kerbosch [18], Durand-Pasari [19], Akkoyunlu [20] and Balas-Yu [21]); and algorithms using the equivalence with the maximal clique, but considering an extended modular product of the edges, not of the nodes (Koch's [22]). While we have found that some of these algorithms have certain advantages with respect to our problem, there was room for improvement. In Section V we offer some comparison of the proposed algorithm with existing ones.

III. Formal Model

A. Context Graphs and Patterns

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. Moreover, it should have a representation of other agents' interests, in order to know whom to inform of potentially interesting information, out of all the agents that share some context with it.

The aim of the representation defined below (introduced in previous work [6]) is to serve as a simple and flexible way of representing the knowledge about the user's current situation. It must be easy to change, by removing or adding associations, and it must be able to adapt to the storage space: an agent must be able to store some relevant information for its function even if it has very limited storage capabilities. The representation must also allow the agent to easily aggregate interesting received information with existing one.

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 $V \subseteq Concepts$

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

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, with the following properties[1]:

[1]We will note with the superscript " ^{P} " the support for generic elements, like ?-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. An extended model endows edges in a pattern with the capability to match entire paths in the graph (described by regular expressions) but we will not focus on that in the contents of this paper.

B. Context Matching

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 or problematic situations and is able to decide on appropriate action to take [5].

The pattern G_s^P *matches* the subgraph $G'_A = (V', E')$, *iff* there exists an injective function $f_v : V_s^P \rightarrow V'$, 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).

We allow partial matches. A pattern G_s^P *k-matches* (matches except for k edges) 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.

One pattern may match various subgraphs of the context graph. A match i between a context pattern G_s^P and the context graph CG_A of an agent A is defined as:

$M_{A\text{-}si}(G'_A, G_m^P, G_x^P, f_v, k)$.

G'_A, G_m^P, G_x^P are graphs: $G'_A \subseteq CG_A$ is the subgraph (partially) matched by the pattern, $G_m^P = (V_m^P, E_m^P) \subseteq G_s^P$ (or solved part) is the part of the pattern that matches G'_A, and $G_x^P = (V_x^P, E_x^P)$ is the rest of the pattern, which is unmatched. There is no intersection (common nodes or edges) between G_m^P and G_x^P (it is therefore possible for G_x^P to contain edges without containing both of their adjacent vertices).

C. A Simple Example

Let us give a simple example of usage for graphs and patterns: Suppose that an Ambient Intelligence system, implemented by means of a multi-agent system, is helping researcher Alex in organizing information on the computer. Among that information, an e-mail has been received: a call for papers (CFP) for the Artificial Intelligence Conference, or AI-Conf. The CFP contains the time when the conference will take place, the deadline for articles, and the date the CFP was issued. Alex read the e-mail, but did not take the time to organize the information inside it. What the AmI system was capable of extracting so far (possibly by means of pattern-matching) are the fact that the CFP is a document, that it contains 2 dates and an interval of time, and that it is about something called AI-Conf (a name that appears throughout the document); it also checked automatically the page linked in the document and the page too contained one of the dates (the deadline) and the interval in which the conference will take place (see also Figure 1 (a)).

Alex's agent is supposed to be able to notify Alex of the approaching deadline of the conference and of the fact that Alex has not yet submitted a paper. The agent contains a pattern describing a conference call for papers, like the one displayed in Figure 1 (b). The agent's task is difficult because while the pattern is defined by relationships, the context information only contains names for its nodes – but the nodes are not named in the pattern.

By matching the pattern against the graph, the agent is able to obtain a partial match ($k = 3$), inferring the fact that the CFP is indeed for a conference, the deadline of the conference, and the fact that there is no article yet submitted to it. The solved part and remaining unsolved part are shown in Figure 1 (c).

IV. The Proposed Algorithm

The purpose of this paper is to present a novel algorithm for context matching (matching a context pattern to a context graph). This algorithm takes inspiration from existing approaches, but has been created from scratch in order to best fit the context matching problem and the agents doing the matching.

The problem is characterized by the fact that all nodes in the graph, as well as most of the nodes in the pattern, are labeled. Most of the edges in the graph and the pattern are also labeled. The size of the patterns is expected to be small. The size of the context graph, however, depends on the size of the device: a powerful device can hold more information and has the power to match patterns against a larger graph.

A. Algorithm description

Tha algorithm is focused on matching edges. It works with valid partial matches (as defined in Section III-B) of various sizes (between 1 and m_s edges, i.e. with $k \in \{0..m_s - 1\}$) and merges them in order to form larger (better) matches. The algorithm has two phases. First, it generates a set of all possible single-edge matches. Then, it selects pairs of compatible matches that it merges in order to create new matches. The search for new matches is close to a depth-first search, in order to get better matches faster. The gist of the algorithm is that it does not test the compatibility (from the point of view of merging) of the matches before each merger, but instead uses for each match a set of data structures (a frontier, a set of adjacent merger candidates and a set of non-adjacent merger candidates) that allow the algorithm to know precisely if two matches can be merged or not.

The pseudo-code of the algorithm is given in Figure 2. In this section, we will consider a match as being a tuple

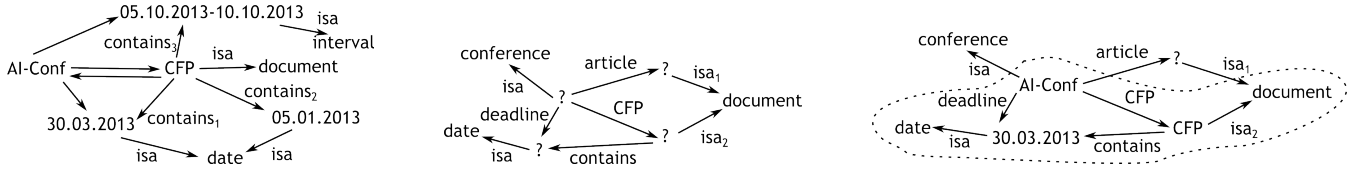

Figure 1. An example of context graph (a), context pattern (b), and the match between them (c) (the solved part is emphasized by the dashed enclosure). Some relations have been annotated with subscripts to be able to differentiate them in the text.

$M(G', G^P_m, f_v, f_e, fr, MC, MO, k)$

that is, a match is characterized by the matched subgraph, the matched (solved) part of the pattern, the correspondence functions for vertices and edges ($f_v : V^P_m \to V'$, $f_e : E^P_m \to E'$, both are bijections), the frontier ($fr = \{v^P \in V^P_m \mid \exists e^P \in E^P_x, e^P \; adjacent \; to \; v^P\}$), the adjacent (or *immediate*) merger candidates, the non-adjacent (or *outer*) merger candidates, and k.

Let us give some details on how the algorithm works in the following paragraphs.

In the **first phase** (illustrated by the pseudo-code function $AddInitialMatches()$, the algorithm creates all possible single-edge matches. That is, it creates a set of all matches of an edge from the graph pattern and an edge from the graph.

An edge $e^P_{kp} \in E^P$ matches and edge $e_{kg} \in E$ if the vertices adjacent to the pattern edge match (are generic or have the same value as) the vertices adjacent to the graph edge, and if either one of the edges is unnamed or the labels of the edges match. The frontier of the newly created match will be formed of the two vertices adjacent to the pattern edge (considering both have other incoming or outgoing edges).

The match can be immediately merged with any other match whose frontier intersects the frontier of the current match, provided that all vertices in the intersection have the same corresponding graph vertex in both matches. That is, for two matches M_1, M_2 (with frontiers fr', fr'' and merger candidates MC', MC'') the condition for $M_2 \in MC'$ (or, equivalently, $M_1 \in MC''$) is that

$fr' \cap fr'' \neq \emptyset$ and $\forall v^P \in fr' \cap fr'', f'_v(v^P) = f''_v(v^P)$

The match could be potentially merged, later, with any other match that is not adjacent to it. All these potential merger candidates are gathered in the set MO.

In the **second phase** of the algorithm, pairs of matches are picked to be merged. In order to do this, all matches (initial or merged) are added to a queue ($MatchQueue$), that is sorted so that larger matches come first.

For each match, new matches are created by merging the match with candidates from MC. It is guaranteed that all matches in MC are compatible. Moreover, as matches are merged, newly created matches also have MC and MO sets that are guaranteed to be correct.

A match M resulted from the merging of M_1 and M_2 is produced by the pseudo-code function $Merge(M1, M2)$. As the matches are disjoint (except for the common frontier vertices), it suffices to add in G', G^P_m, f_v, f_e all the elements in the corresponding components of M_1 and M_2. The resulting frontier is formed of the reunion of the two frontiers, except for the vertices which are now included completely in the matched part of the pattern.

The immediate merger candidates of M will be formed of three disjoint sets: first, $MC' \cap MC''$ – the matches that are merger candidates for both M_1 and M_2 – these are acceptable merger candidates for any of the component matches; second, $MC'' \cap MO'$ – the matches that were acceptable, but not immediate, candidates for M_1 (in MO'), and were also immediate merger candidates for M_2 – as M contains M_2, these candidates are immediate merger candidates, and M_1 did not reject them; third, $MC' \cap MO''$ – the set symmetric with the previous one. The matches left out by the merger are matches that were compatible with one of the component matches, but not with the other.

The outer merger candidates are candidates that were acceptable by both M_1 and M_2 but were not immediate merger candidates for any of the two.

B. Complexity Analysis

The classic problem of unlabeled graph matching is NP-complete, and so is the problem of context matching if all nodes in the pattern are generic and all edges in the pattern are unnamed. However, it is interesting to look into the complexity of the proposed algorithm and see how the various operations can impact the execution time in real-life scenarios. It is notable that we are especially interested in performance on small devices, which will however host agents that are dedicated to relatively simple functionality, and will therefore hold small context graphs and small patterns. Such small devices would be mobile phones and tablets. Very simple graph matching would be suitable for microcontrollers existing on wireless sensors. In the case of sensors, context matching does not bring a performance improvement, but has the advantage that it is a uniform method of detecting appropriate action, used throughout the entire system, making an agent on a small device structurally identical to one on a larger computer.

Throughout this analysis we will use the example in Section III-C, which is a medium-to-difficult case for a small agent. The example is further analyzed, from a functional point of view, in Section V. In this section, we will consider $m = ||E||$ and $m^P = ||E^P_s||$ – the number of edges in the graph and in the pattern. The number of nodes has little impact on complexity. For our example, the two numbers are 11 and 8, respectively.

function $Match(G, G^P)$ with $G = (V, E), G^P = (V^P, E^P)$

1. $MatchQueue \leftarrow AddInitialMatches()$
2. **while** $MatchQueue \neq \emptyset$ // matches should be kept sorted by k
3. $M \leftarrow pop(MatchQueue)$
4. **for each** $Mc \in MC$
5. $MO' \leftarrow MO' \setminus \{M\}$
6. $M_{result} = Merge(M, Mc)$
7. $MatchQueue \leftarrow MatchQueue \cup M_{result}$
8. **return** best match ever generated (lowest k)

function $AddInitialMatches()$

1. $MatchQueue \leftarrow \emptyset$
2. **for each** $e^P_{kp} = (v^P_{ip}, v^P_{jp}, val_p) \in E^P$ // edges in pattern
3. **for each** $e_{kg} = (v_{ig}, v_{jg}, val_g) \in E$ // edges in graph
4. **if** v^P_{ip} *matches* v_{ig} **and** v^P_{jp} *matches* v_{jg} **and** val_p *matches* val_g // it is a match
5. $MC \leftarrow \emptyset$; $MO \leftarrow \emptyset$ // immediate & outer match candidates
6. **for each** $M_a \in MatchQueue$ matching e^P_{ap} with e_{ag}
7. **if** $e^P_{ap} \neq e^P_{kp}$ **and** $e_{ag} \neq e_{kg}$ // matches don't overlap
8. **if** $\{v^P_{ip}, v^P_{jp}\} \cap \{v^P_{ap}, v^P_{bp}\} \neq \emptyset$ // matches are adjacent
9. **if** common nodes have the same correspondent in G' // matches are compatible
10. $MC \leftarrow MC \cup M_a$ // the other match is a merge candidate
11. **else** $MO \leftarrow MO \cup M_a$ // the other match is an outer candidate
12. $MatchQueue \leftarrow MatchQueue \cup$ // add the single-edge match to queue
$M(G' = (\{v_{ig}, v_{jg}\}, \{e_{kg}\}), G^P_m = (\{v^P_{ip}, v^P_{jp}\}, \{e^P_{kp}\}),$ // matched subgraph, *matched part*
$f_v = \{v^P_{ip} \rightarrow v_{ig}, v^P_{jp} \rightarrow v_{jg}\}, f_e = \{e^P_{kp} \rightarrow e_{kg}\},$ // vertex function, edge function
$fr = \{v^P_{ip}, v^P_{jp}\}, MC, MO, k = ||E^P|| - 1)$ // frontier, match candidates, k
13. **return** $MatchQueue$

function $Merge(M1, M2)$
with $M_1(G'' = (E', V'), G^{P\prime}_m = (V^{P\prime}_m, E^{P\prime}_m), f'_v, f'_e, fr', MC', MO', k')$
and $M_2(G''' = (E'', V''), G^{P\prime\prime}_m = (V^{P\prime\prime}_m, E^{P\prime\prime}_m), f''_v, f''_e, fr'', MC'', MO'', k'')$

1. $V = V' \cup V''; E = E' \cup E''$ // edges are guaranteed to be disjoint
2. $V^P_m = V^{P\prime}_m \cup V^{P\prime\prime}_m; E^P_m = E^{P\prime}_m \cup E^{P\prime\prime}_m$
3. $f_v = f'_v \cup f''_v; f_e = f'_e \cup f''_e; fr = \emptyset$ // functions are guaranteed compatible
4. **for each** $v^P \in fr' \cup fr'', \exists e^P$ adjacent to $v^P, e^P \notin E^P$
5. $fr \leftarrow fr \cup v^P$
6. $MC' \leftarrow MC' \setminus \{M2\}; MC'' \leftarrow MC'' \setminus \{M1\}$
7. $MC = (MC' \cap MC'') \cup (MC' \cap MO'') \cup (MC'' \cap MO')$ // matches that are merge candidates (immediate or outer) for both M_1 and M_2, but are immediate candidates for at least one of them
8. $MO = MO' \cap MO''$ // matches that are outer to both M_1 and M_2
9. **return** $M(G' = (V, E), G^P_m = (V^P_m, E^P_m), f_v, f_e, fr, MC, MO, k)$

Figure 2. The matching algorithm for a graph and a pattern – $Match$ – is based on two phases: adding the initial, single-edge matches to the queue – $AddInitialMatches$ – and merging pairs of candidate matches from the queue, using the function $Merge$.

The first phase of the algorithm – adding initial matches – tries to create $m \times m^P$ initial matches. However, in a realistic scenario, most edges will be labeled, therefore the actual number is much smaller. Even in our example, which is very unfavorable, there are 19 initial matches, even if $m \times m^P = 88$. However, for each new initial match, all previously created matches are explored to find potential merger candidates. This means a total complexity of the first phase of around $MatchQueue.size^2$, considering the size of the queue at the end of the first phase. For our example, 370 edges are compared by the end of the first phase. There are also a considerable number of node label comparisons, which depends on how many generic nodes exist in the pattern (the more the better), but these are the only node comparisons that are done in the whole algorithm. In our example there are only about 200.

It is worth noting that it is only the first phase that deals with label comparison. The second phase deals only with

reference management and comparison, which is much faster and can benefit from compiler and execution environment optimizations.

The second phase takes matches from the $MatchQueue$ and merges them with all their potential candidates. The merger process is characterized by the set operations that create the MC and MO components of the result. These operations are considerably optimized by using efficient set implementations (hash sets). While iterating over all merger candidates takes long if there are many candidates, the depth-first approach leads to large matches being created first and maximal matches are reached considerably fast compared with the full matching process. In our example, the maximal match is found after 1200 edge comparisons, but the total matching process (completely exhausting the match queue) takes over 2400 edge comparisons. If the maximal match is detected correctly, the process can stop at half the time.

The conclusion is that, although any graph matching is time consuming, for a real situation the process is quite fast.

V. Experiments and Results

The proposed algorithm has been implemented (in Java) and tested on several hand-crafted graphs and patterns (resulted from Ambient Intelligence scenarios) and on some randomly created tests, yielding good results.

A. Experimental Tools

During the testing of the algorithm, some visualization tools have been created.

First, we have developed a linear textual representation of directed graphs, for the purpose of displaying a human-readable form of graphs in the output console. It uses vertex and edge names, arrows, stars and parentheses to completely display a graph. Each edge is shown only once, and nodes are repeated once per graph cycle. For instance, a graph that is formed of three nodes (A, B, C) linked by two edges a and b is represented as $A \xrightarrow{a} B \xrightarrow{b} C$ (or, in ASCII output, as `A -a-> B -b-> C`). A graph which is the cycle ABC is represented as $A \rightarrow B \rightarrow C \rightarrow *A$ – the star marks the fact that A has appeared before and its outgoing edges have been already defined. A tree with the root A and the children B and C is represented as $A(\rightarrow B) \rightarrow C$. Finally, a graph formed of the 3-node cycles (ABC and ABD) is represented as $A \rightarrow B(\rightarrow C \rightarrow *A) \rightarrow D \rightarrow *A$. This linear text representation makes it easy to follow the workings of the algorithm, as one can observe in the snippet in Figure 3.

Based on the linear textual representation, we have also built a graphical representation for graphs and for matches, as the one in Figure 4. The figure shows how the maximal match is obtained of our example, by merging a 4-match with a disjoint single-edge match (7-match) to form a 3-match.

B. Experimental Results

We have tested the proposed algorithm on a large number of examples. The variables of the examples were the size of the graph G, the size of the pattern G^P, the number of unlabeled edges, and the number of generic nodes. In comparing the proposed algorithm with others, we have focused on counting the number of edge comparisons, as the number of node comparisons is very low. To this number we have also added the number of edge hash comparisons resulted from set operations, as the second phase of the algorithm relies exclusively on the set operations required by match merging.

Table I presents the number of matched edges for various test examples and various graph-matching algorithms. The displayed examples are the following: a small example with a 7-edge graph and a 2-edge pattern; the initial example presented in Section III-C of this paper; the same initial example, but with no edge labels in the graph or in the pattern – making this a difficult case because of a large number of initial matches; the initial example, in which all nodes in the pattern are generic, making this a difficult case both for node and for edge matching; and finally, a large example with a 20-node graph and a 12-node pattern.

All test show that the proposed algorithm is at least as good as the algorithms compared against. We have compared it against other algorithms as well, notably McGregor and Larossa. However, the McGregor algorithm is more focused on node expansion, therefore it is difficult to compare against it, but we know that it is very inefficient (it relies on backtracking). The Larossa algorithm [23] is constantly much more efficient than the proposed algorithm, however its nature makes it unable to obtain partial matches, and is therefore unsuitable for the problem at hand [16].

The good results of the proposed algorithm are due to the fact that it uses a greedy approach, building larger matches every time. Each match is a good match, and at any step, for each match, we have the correct set of compatible merger candidates. While the elements of the match (the frontier and the match candidates sets) do make the algorithm more memory-consuming, this consumption is worth the tradeoff for increased time-performance.

VI. Conclusion

Using a flexible but powerful knowledge representation in AmI-purposed multi-agent systems is key to a reliable, intelligent and anticipative Ambient Intelligence environment. Context graphs and patterns offer such a representation, but all context-aware processes involving them use graph matching to detect the current situation.

While the problem of matching graphs is computationally difficult, it is possible to create purpose-built algorithms that rely on the specific properties of the context matching problem to increase efficiency in real-world scenarios.

This paper presents such an algorithm, which relies on a search in the match space, starting from single-edge matches and merging pairs of matches, while always keeping tabs on the merger candidates for a match. We have implemented and tested the algorithm with very satisfactory results.

Future work includes not only improving the proposed algorithm, but also testing it in more harsh conditions, and

```
...
merging match [-] (k=5): AIConf (->CFP) (->300311) ->conftime
                          : ?#3 (-article->?#2) (-CFP->?#5) -deadline->?#2
                                              fv: {?#5=CFP, ?#3=AIConf, ?#2=300311, ?#2=conftime}
and  match [2:5] (k=7):  CFP-contains->conftime
                          : ?#5-contains->?#2
                                              fv: {?#5=CFP, ?#2=conftime}
new  match [-] (k=4):    AIConf (->CFP-contains->conftime) (->*conftime) ->300311
                          : ?#7 (-CFP->?#8-contains->?#6) (-deadline->*?#6) -article->?#4
                                              fv: {?#8=CFP, ?#7=AIConf, ?#4=300311, ?#6=conftime}
...
```

Figure 3. An example of output from the matching algorithm, showing the merging of two matches.

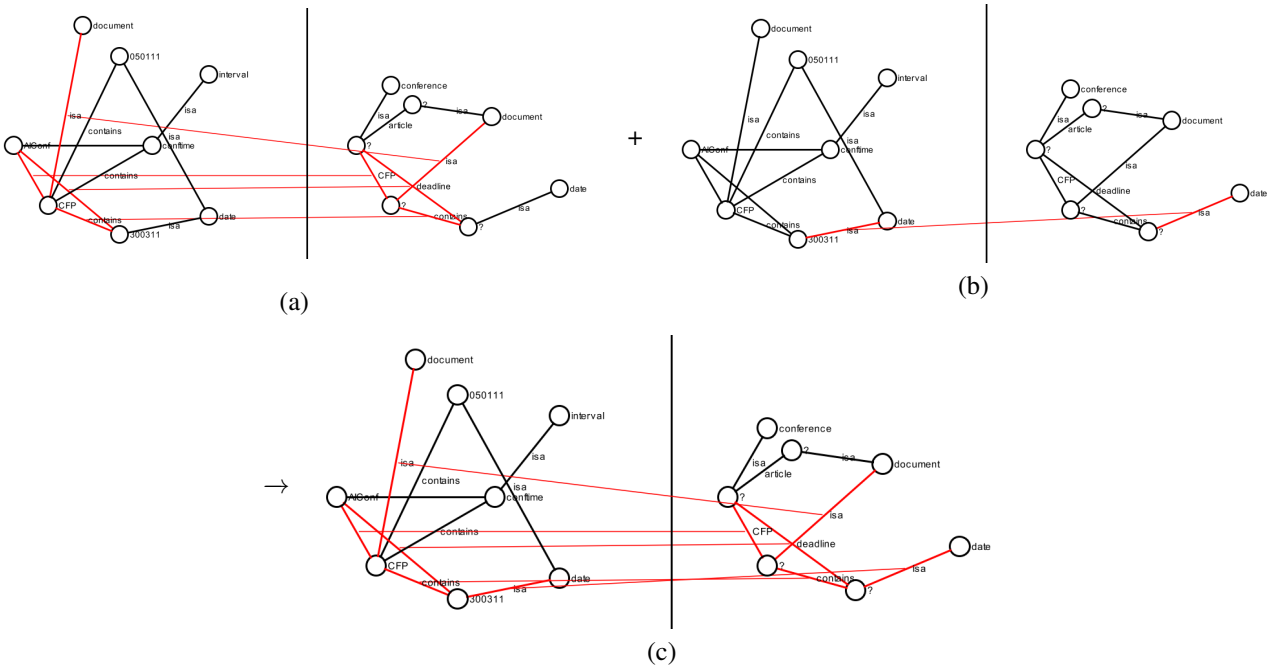

Figure 4. Example of graphical output of the matching algorithm, showing the merging of two matches (a) and (b) into a single match (c). The same example is used as in Figure 1.

Algorithm	Akkoyunlu	Bron-Kerbosch	Balas-Yu	Durand-Pasari	Proposed algorithm
Expanded edges:					
Small example	124	120	135	119	34
Initial example	5431	5440	6423	5219	2459
No labeled edges	7054	9454	15843	9060	7581
No labels	326044	371943	578401	367725	108902
Large example	20470	19989	22170	18322	11834

Table I
COMPARISON OF NUMBER OF EXPENDED EDGES IN VARIOUS GRAPH MATCHING ALGORITHMS.

on mobile devices, as well as integrating it in an agent-based framework for Ambient Intelligence applications.

ACKNOWLEDGMENT

The author would like to thank Adrian Dobrescu for providing some of the references in the Related Work section, as well as the platform used to test various graph algorithms, from which we obtained the comparative results in Table I for other algorithms than the one proposed in this paper.

This research has been supported by project ERRIC: Empowering Romanian Research on Intelligent Information Technologies (FP7-REGPOT-2010-1/264207)[2].

REFERENCES

[1] M. Weiser, "The computer for the 21st century," *Scientific American*, vol. 272, no. 3, pp. 78–89, 1995.

[2] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J. Burgelman, "Scenarios for ambient intelligence in 2010," Office for Official Publications of the European Communities, Tech. Rep., February 2001.

[3] C. Ramos, J. C. Augusto, and D. Shapiro, "Ambient intelligence - the next step for artificial intelligence," *IEEE Intelligent Systems*, vol. 23, no. 2, pp. 15–18, 2008.

[2]http://www.erric.eu

[4] M. Weiser, "Some computer science issues in ubiquitous computing," *Communications - ACM*, pp. 74–87, 1993.

[5] A. Olaru, A. M. Florea, and A. El Fallah Seghrouchni, "An agent-oriented approach for ambient intelligence," *Scientific Bulletin of the University Politehnica of Bucharest, Series C - Electrical Engineering and Computer Science*, vol. 74, no. 3, pp. 45–58, August 2012. [Online]. Available: http://www.scientificbulletin.upb.ro/SeriaC_-_Inginerie_Electrica_si_Stiinta_Calculatoarelor.php?page=revistaonline&a=2&arh_an=2012&arh_ser=C&arh_nr=3

[6] ——, "Graphs and patterns for context-awareness," in *Ambient Intelligence - Software and Applications, 2nd International Symposium on Ambient Intelligence (ISAmI 2011), University of Salamanca (Spain) 6-8th April, 2011*, ser. Advances in Intelligent and Soft Computing, P. Novais, D. Preuveneers, and J. Corchado, Eds., vol. 92. Springer Berlin / Heidelberg, 2011, pp. 165–172. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19937-0_21

[7] A. Olaru and C. Gratie, "Agent-based, context-aware information sharing for ambient intelligence," *International Journal on Artificial Intelligence Tools*, vol. 20, no. 6, pp. 985–1000, December 2011. [Online]. Available: http://www.worldscinet.com/ijait/20/2006/S0218213011000498.html

[8] M. Baldauf, S. Dustdar, and F. Rosenberg, "A survey on context-aware systems," *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 2, no. 4, pp. 263–277, 2007.

[9] M. Perttunen, J. Riekki, and O. Lassila, "Context representation and reasoning in pervasive computing: a review," *International Journal of Multimedia and Ubiquitous Engineering*, vol. 4, no. 4, pp. 1–28, October 2009.

[10] L. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 26, no. 10, pp. 1367–1372, 2004.

[11] B. Messmer and H. Bunke, "Efficient subgraph isomorphism detection: A decomposition approach," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 12, no. 2, pp. 307–323, 2000.

[12] E. Bengoetxea, P. Larrañaga, I. Bloch, A. Perchant, and C. Boeres, "Inexact graph matching by means of estimation of distribution algorithms," *Pattern Recognition*, vol. 35, no. 12, pp. 2867–2880, 2002.

[13] B. Luo and E. Hancock, "Structural graph matching using the EM algorithm and singular value decomposition," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, pp. 1120–1136, 2001.

[14] S. Gold and A. Rangarajan, "A graduated assignment algorithm for graph matching," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 18, no. 4, pp. 377–388, 1996.

[15] T. Caetano, J. McAuley, L. Cheng, Q. Le, and A. Smola, "Learning graph matching," *IEEE transactions on pattern analysis and machine intelligence*, pp. 1048–1058, 2009.

[16] A. Dobrescu and A. Olaru, "Graph matching for context recognition," in *Proceedings of CSCS 19, the 19th International Conference on Control Systems and Computer Science, May 29-13, Bucharest, Romania*, I. Dumitrache, A. M. Florea, and F. Pop, Eds. IEEE Xplore, 2013, pp. 479–486. [Online]. Available: https://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6569308

[17] J. J. McGregor, "Backtrack search algorithms and the maximal common subgraph problem," *Software: Practice and Experience*, vol. 12, no. 1, pp. 23–34, 1982.

[18] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph," *Communications of the ACM*, vol. 16, no. 9, pp. 575–577, 1973.

[19] P. J. Durand, R. Pasari, J. W. Baker, and C.-c. Tsai, "An efficient algorithm for similarity analysis of molecules," *Internet Journal of Chemistry*, vol. 2, no. 17, pp. 1–16, 1999.

[20] E. Akkoyunlu, "The enumeration of maximal cliques of large graphs," *SIAM Journal on Computing*, vol. 2, no. 1, pp. 1–6, 1973.

[21] E. Balas and C. S. Yu, "Finding a maximum clique in an arbitrary graph," *SIAM Journal on Computing*, vol. 15, no. 4, pp. 1054–1068, 1986.

[22] I. Koch, "Enumerating all connected maximal common subgraphs in two graphs," *Theoretical Computer Science*, vol. 250, no. 1, pp. 1–30, 2001.

[23] J. Larrosa and G. Valiente, "Constraint satisfaction algorithms for graph pattern matching," *Mathematical structures in computer science*, vol. 12, no. 4, pp. 403–422, 2002.