Dynamic Dictionary = cooperative information sources

José João Almeida
jj@di.uminho.pt

Pedro Rangel Henriques
prh@di.uminho.pt

Universidade do Minho
Braga – Portugal

Abstract

In this article a multi-source system is presented as a case-study in order to show how agents can be used to build a dynamic dictionary.

In this context a dictionary has a much wider sense than usual: anything that can give information about a term (whatever term may be) is a dictionary.

The dictionary is composed by several information sources with different domains (knowledge area and cardinality), relevance/richness and cost (cost may be a function of time, resources, communication traffic, money, etc).

An information source is an agent that, given a term, can return some knowledge about it, in an universal representation schema (feature structure).

To answer the user's query about a term, the system aims at finding an information of maximum relevance at a minimum cost; this requires a working strategy (a compromise solution).

Each information source agent provides meta information about its behavior so that a special agent —strategy builder— is able to define a strategy.

Working in this case-study we learned that sometimes, agents can be as simple as operating system commands (UNIX like) and aplications, properly encapsulated with an interface layer.

1 Introduction

Our main concern was the development of a *dictionary* —a program able to answer queries about the meaning of words (or more generally, terms).

As it is very hard to say what is the *meaning of a term*, we will concentrate in returning any useful *information* related to that term. To represent that information, the system described in this paper adopts an *universal type* based on natural language processing tradition: TFS (typed feature structures) [8]. In order to make this text easier to read the camila specification language [1] will be used to describe all the universal type values. The description of the agents behavior also follows the camila notation.

In traditional approaches, a dictionary has a huge data base(DB) and the heart of the application is a search algorithm that sends queries to the DB. Our experience says that it happens many times to be unable to retrieve the information about a term from that specific data base, while other data bases or even textual documents (also existing in the same computer) could provide the required answer.

Our goal is to avoid such a situation, allowing the simultaneous search of a term in multiple sources of knowledge. In addition to this first aim, we also formulate two more requirements: our dictionary should couple with dynamic evolution of the sources (the system should determine in each moment the resources available), and it should be able to define the best strategy to extract the most relevant knowledge at the less cost.

The proposal that we will discuss in this paper is based on the idea of *agents* [13]; a knowledge source (a document, a data base, ...) will be encapsulated with the necessary procedures (to search, and to pack the answer in TFS format), constituting an *agent*. In this way, our Multi-Source Dynamic Dictionary (MSDD for short) will be set up as a community of cooperative autonomous agents, integrated via a common working environment and controlled by a strategy builder (also an agent). This idea, previously used by [7, 9], seems to be a promising framework in the development of information retrieval systems, as suggested in [10].

Features Structures (FS) are also used to support all the information needed by the system. The environment (that the agents see and change) is a FS. The self-description built by the information source agents, and used by the strategy definer agent, is also a FS.

In section 2 we introduce the architecture of the MSDD; one of its main components, the environment, is detailed in 3; the other elements, the agents, are discussed in 4. As usual, the last section (sec. 5) is used to summarize the paper, to enhance the main lessons that we extract from that work, and to present some ideas for future work.

2 Multi-Source Dictionary: the Architecture

In this section we will introduce the architecture of the MSDD. We assume that there is a basic traditional dictionary (we call it, for instance, dic.ASA) and we intend to be able to search an unknown word (not found in dic.ASA) in other information sources.

The system has two main components:

the agents —a set of independent entities, most of them in charge of the multiple information sources, and the others responsible for the interface issues, and for the definition of an working strategy;

the environment —a global structure that holds information about the society to be shared by the cooperative agents; this information is also provided and maintained by the agents.

The following FS is a formal definition of the MSDD architecture; agents are grouped by their functionality.

```
MSDD = Env        : Environment
       Interface  : Agent
       Strat_Build: Agent
       Conciliator: Rewrite_Sys_Agent
       Inf_Sources: Agent-set
```

A diagrammatic picture of the multi-source dynamic dictionary system can be seen in figure 2.

After this overview of the system's structures, we will begin by describing the environment —the static component of MSDD where all the information is resident; then we are able to characterize the agents —the dynamic part of MSDD— and discuss their behavior.

3 The environment

In our MSDD system, the environment is a shared structure that contains information about:

- the word[1] (one, or more) that is being analyzed and the accumulated information about it (WI)
- the strategy to be used in the analysis (S)
- the behavior of the Information Sources (ISDs)

[1] More generally, the *term*.

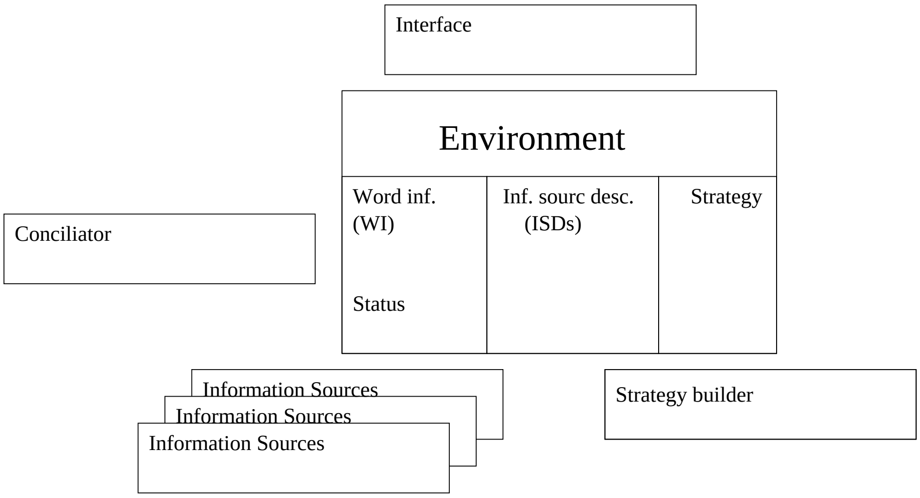

Figure 1: The Architecture of MSDD

- the current status

Formally, we would say that the environment is the following structure:

```
Environment = WI      : WordInfo-seq
              S       : Strategy
              ISDs    : ISD-set
              status  : Int
```

The information kept about the word under analysis (`WI`) is defined by the FS below

```
WordInfo = word        : String
           info        : FS
           relevance   : Int
           submissions : FS
```

The `relevance` feature will describe the quality[2] of the information produce by the sources and available in the `info` feature. In this context, relevance is expressed as a number between 0 and 100.

The `submissions` feature is used to record each individual contribution provided by the information sources; `info` represents the synthesis of the `submissions`.

The working `Strategy`, that prescribes how to use the available information providers, is described by a pair:

[2]Some measure of the completude, confidence and usefulness of the extracted data.

```
Strategy = Autom  : Automaton
           Use    : state -> IS-set
```

The first component is a finite state automaton, defined by:

```
Automaton =  T : [the alphabet, or terminal symbol set]
             Q : [the states, a set of Int]
             Z : [set of states corresponding to a good relev.]
             S : [the initial state]
             D : [the transition function]
```

Terminals are produced by the Conciliator agent as a function of the relevance of the data returned by the Information Source agents; states describe different stages in the quality of the information obtained; final states correspond to the achievement of an `WI` with a good relevance.

The second component is a mapping that associates each state of the automaton with the subset of information sources that should used in that stage.

The `ISD` is a structure containing the features necessary to describe each Information Source agent. In each `ISD` we can find information like: maximum expected relevance of the data produced by that source; expected cost of the retrieving process; domain cardinality (or domain width), a measure of the probability to find in that source an answer for the given term; and any other feature that could be useful to define a strategy.

All of this features will take values representing guesses about the respective characteristics of the information source; precise values for these entities make no sense.

```
ISD = relevance : Int
      cost      : Int
      domCard   : Int
      .......   : ...
```

4 The Agents

4.1 Interface Agent

This agent receives a word from the user —the term to be searched for— and sets the `WI` component of the environment to this value; initially, the relevance factor (the confidence in the answer obtained) is set to zero.

```
env.WI <- [word      -> gets(),
           relevance -> 0
          ]
```

The Interface agent shows to the user the content of `WI` when the relevance factor changes, or when it attains a good enough value.

In an incremental system, this Interface agent can also allow the Conciliator to implement *interactive choices.*

4.2 Strategy Builder Agent

The active information sources are not required to cooperate all at a time, during the search process; instead it shall be explicit in the Environment which subset of sources have to contribute at that moment —this decision is, of course, inherent to the strategy.

The current search state of the MSDD system is known (it is one of the features of the environment); so the strategy has to express which agents should answer at each state. By other words, the strategy must define which subset of information sources will collaborate in each situation (state of the search).

After receiving answers from the information source agents in the elected subset, the Conciliator merges the contributions (`submissions`) obtained till that moment (see details in the next subsection), and recomputes the state of the system, (as a function of the relevance factor reached). To enable that computation, the strategy must also provide information about the state transition mechanism.

The Strategy Builder (SB) agent uses the self-description provided by the information sources (ISDs) available, and builds the strategy to be used in the cooperative process. This strategy must define how to achieve a good result with a low cost.

According to the capabilities (behavior parameters expressed in the ISD) of each information source agent, the SB agent generates a finite state automaton (`Autom` field below) and builds a mapping defining the set of information sources that should be consulted in each state (`Use` field below).

```
env.S <-  [Autom -> mkAutomaton(env.ISDs),
           Use   -> mkIdTable(Autom.Q,env.ISDs)
          ]
```

Different methods can be used to compute the strategy, depending on the actual `ISD` features available, and on the main strategy goals[3].

Even when we are unable to implement a SB agent that computes an optimal strategy, it is possible to use heuristics to produce a good automaton.

[3]To minimize the cost, we should minimize the agents consulted; to maximize the relevance, we should use all the sources —different compromises between cost and relevance imply different choices.

4.3 Conciliator Agent

The Conciliator, a rewriting system, is the agent responsible for merging (into the `info` field) the `submissions` produced by the information sources agents.

When different agents contribute with different features (for the information about the word under analysis), the merging task is easy —it is only necessary to add these items into the respective FS. However, when different agents produce the same features, the merge was to select one, or to combine both answers.

The merge process is obviously crucial for the performance of the dictionary, but it is a complex operation. The easiest, and also the most dynamical, way we found to write the merge algorithm was by means of a set of rewrite rules; these rules translate the heuristics used to accumulate features and to conciliate when the same feature is provided by different sources.

This agent will also change the current state as a function of the relevance returned. This state change is determined by the transition function of the above referred automaton.

```
env.WI <- [info      -> rewrite(env.WI.info,env.WI.submissions),
           relevance -> eval(env.WI.relevance,env.WI.submissions)
          ]
env.status <- delta(env.status,env.WI.relevance,env.S.Autom),
```

As the SB agent, the Conciliator agent is always present and active in the MSDD.

For the sake of simplicity and due to space limitations, no more details of this agent will be presented in this text.

4.4 Information Source Agents

As previously said, each Information Source (IS) agent announces its properties in the `ISDs` at start time (details about this FS were given in section 3).

Every time an available IS is active (belongs to the subset of IS agents that must collaborate at the current state),

```
Active: Agent in env.S.use[state]
```

it looks at the `WI` for the `word` to be searched, and adds to the `submissions` feature of `WI` (in the environment) the new information that it was able to retrieve about that word.

In order to clarify what kind of process can be thought as an IS agent, and how these agents work and cooperate, we will detail, in the rest of this section, four agents implemented in our prototype MSDD: *ASA dictionary, NLgrep, JSpell, ListOfProverbs.*

Example of Information sources:

The ASA dictionary

This agent has an internal state containing the 5000 entries of that dictionary[11]. Each entry is very rich and is the human synthesis of many other sources. The domain is small.

```
-----------------------------------------------------------
Agent ASA
1  State dic: word -> ANY

2  BEGIN add ISDs [asa -> [relevance -> 100,
                           cost      -> 2,
                           domCard   -> 5000]
                  ]

3  let(w = env.WI[word])
4  in   if( w in dom(dic) -> submit  dic[w] +
                                     [ source    -> asa,
                                       relevance -> 100 ]
                             returns [ relevance -> 100 ]
5           otherwise     -> returns [ relevance -> 0  ])
-----------------------------------------------------------
```

Implementation detail: this agent is a perl [12] program using DB_File database.

Comments:

1. Asa has a dictionary `dic` which maps words in its information Feature Structure (FS).
2. At start time, it tells the system that ASA is good, low cost, and has a not so big domain.
3. It consults the environment `WI` and extracts the word.
4. If `w` is in ASA dictionary, it adds that new information to the `WI` using `submit` function and say that relevance is good.
5. If the word is unknown, it returns that it has no relevant answer.

NLgrep information source

This agent looks for sentences containing a word (or some morphological derived form), using the command NLgrep (grep for Natural Language)

```
-----------------------------------------------------------
Agent NLgrep
```

```
1  BEGIN add ISDs [nlgrep -> [relevance -> 40,
                              cost     -> 10,
                              domCard -> 100 000]
                  ]

2  let(w = env.WI[word]
3       s = txt2seqStr(‘nlgrep w examples.txt | head -40‘))
4  in  if (s == <> )  -> returns [relevance -> 0]
5          otherwise  -> submit  [use       -> s,
                                  source    -> nlgrep,
                                  relevance -> length(s)]
                         returns [relevance -> length(s)]
          )
--------------------------------------------------------
```

Implementation detail: this agent is a perl program that makes an agent with an external command.

Comments:

1 The maximum relevance possible for the information produced by this agent is defined to be 40.

3 `‘command‘` - executes `command` and gets its standard output; `‘nlgrep w examples.txt | head -40‘` - gets the first 40 sentences containing word `w`.
`txt2seqStr(...)` function that translates a text into a sequence of strings.

5 In case of finding less then 40 sentences, the returned relevance is announced to be the number of sentences found.

Morphological analyzer JSPELL

The agent described bellow is a simplification of the real case. In order to make it simple to understand, we will only discuss the situations where a word has just one morphological analysis.

```
--------------------------------------------------------
Agent JSPELL
1  State dic   : word   -> (featureStructure x ruleId-set)
2        rules : ruleId -> rule

   Internal functions
3       morphAna   : word -> FS
4       nearMisses : word -> word-set
```

```
5       evalRelev  : FS -> Int

6  BEGIN add ISDs [jspell -> [relevance -> 40,
                              cost    -> 10,
                              domCard -> 1000000,
                              newGoal -> true]
                  ]

7  let(w = env.WI[word]
8      fs = morphAna(w)
9      relev = evalRelev(fs)
10     hasNewGoal = "radical" in dom(fs)
      )
   in if( hasNewGoal -> submit  fs +
                                [ source -> jspell,
                                  relevance -> relev
                                  newword -> fs[radical] ]
                        newgoal [ word -> fs[radical],
                                  relevance -> 0]
                        returns [ relevance -> relev ]
          otherwise  -> submit  fs +
                                [ source -> jspell,
                                  relevance -> relev ] ]
--------------------------------------------------------
```

Comments:

1-2 the internal state has a dictionary and rule table. The dictionary maps a word to its classification and to the set of rules that can be applied to the word.

3 internal function that performs morphological analysis of a word

4 internal function that computes existing words similar to the original word.

5 internal function that evaluates the relevance of an analysis

6 newGoal property says that this agent can produce new words to be analyzed

8 make the morphological analysis of the word `x`

9 compute the relevance of the previous obtained answer

10 if `fs` contains the feature `radical` defined, then the word `fs[radical]` should also be analyzed

Implementation detail: this agent is a perl script that uses jspell[2] program implemented in C.

List of proverbs and slang

```
-------------------------------------------------------
Agent Proverb

1  BEGIN add ISDs [prov -> [relevance -> 10,
                            cost      -> 10,
                            domCard -> 10 000]
                  ]

2  let(w = env.WI[word]
3      s = txt2FS(‘nlgrep w proverb.txt‘))
4  in  if (s == <> )  -> returns [relevance -> 0]
5          otherwise  -> submit  [idiomatic -> s,
                                  source    -> prov,
                                  relevance -> length(s)]
                         returns [relevance -> length(s)]
          )
-------------------------------------------------------
```

4.5 Other similar cases

The approach used to develop the MSDD system as a community of cooperative information sources agents can be applied, with the same success, to other different problems with similar characteristics.

As an example, we describe below the IS agents used in another system that we have implemented: an *Encyclopedia of Computer Science*.

Although in a different context, we have adopted a similar approach involving the following agents:

```
inf_source0 = FOLDOC  (Free on line dictionary of computer science)
inf_source1 = man
inf_source2 = apropos
inf_source3 = if x in PATH then x -help
inf_source4 = emacs hypertext from emacs
```

5 Conclusions and future work

In this paper we have presented the architecture of a multi-source dynamic dictionary, and discussed its behavior.

The system described is able to find the information concerned with a given term in different information sources (documents, data bases, etc.). These sources need not be always present: they can be added or removed dynamically. The system is capable of determining which sources are available in each moment, and can extract the information choosing a good compromise between relevance and cost.

Following the pioneer work of [7] and [9], to implement the required functionality we built the dictionary as a set of cooperative autonomous agents.

In order to enable dynamic evolution of the system (birth or death of sources) an adaptative strategy should be used. To achieve that, the agents have to make public its properties (domain, expected relevance, cost, ...) so that another agent can plan the system's behavior. A rewriting system is used to conciliate the different answers produced by the multiple information agents.

It is worth notice that the ability to cope with heterogeneous information providers (source agents) resides on the adoption of an universal type —*TFS, typed feature structures*— to represent all the information involved (either the knowledge returned by the information source agents, or the data used to control the society).

One of the interesting outcomes of the work here reported is the fact that operating system commands may become information agents if we add some encapsulating code. This may require that an abstract view of the commands is created (reverse engineering) to allow them to cooperate.

We believe that this approach can cover a wide range of problems, for instance *knowledge extraction from data bases.*

Work is being done in adding more intelligence to the strategy builder and in defining a more powerful representation scheme for the properties of the agents. One interesting feature would be to allow the source agents to modify dynamically the Conciliator rewriting rules.

Acknowledgments

We would like to thank Orlando Belo for the comments and suggestions.

References

[1] J. J. Almeida, L. S. Barbosa, F. L. Neves, and J. N. Oliveira. CAMILA: Formal software engineering supported by functional programming. In A. De Giusti, J. Diaz, and P. Pesado, editors, *Proc. II Conf. Latino Americana de Programacin Funcional (CLaPF97)*, pages 1343–1358, La Plata, Argentina, October 1997. Centenario UNLP.

[2] J.João Almeida and Ulisses Pinto. Manual de utilizador do JSpell. Manual, Universidade do Minho, July 1994.

[3] Orlando Belo and José Neves. An architecture for multi-agent systems communities. In *10th International FLAIRS Conference, Florida, USA*, May 97.

[4] M. Genesereth and S. Ketchpel. Software agents. In *Communications of the ACM*, volume 37, pages 48–53, July 1994.

[5] Denis Howe, editor. *FOLDOC - Free on-line Dictionary of Computing.* http://wombat.doc.ic.ac.uk/foldoc.

[6] J.M.Fonseca, E. de Oliveira, and A. Steiger-Garçao. Multi-agent negotiation algorithms for resources cost estimation: A casa study. In *Progress in Artificial Intelligence, EPIA'97.* Springer-Verlag, October 1997.

[7] Craig A. Knoblock, Yigal Arens, and Chun-Nan Hsu. Cooperating agents for information retrieval. In *Proc. of the 2nd Int. Conference on Cooperative Information Systems.* University of Toronto Press, Ontario, Canada, 1994.

[8] Hans-Ulrich Krieger. Typed feature formalisms as a common basis for linguistic specification. Research Report RR-94-39, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, 1995. To appear in Machine Translation and the Lexicon, Springer, Berlin, 1995.

[9] Alon Y. Levi, Anand Rajaraman, and Joann J. Ordille. Query-answering algorithms for information agents. In *Proc. of the 13. Nac. Conference on Artificial Inteligence (AAAI-96), Portland*, August 1996.

[10] Miguel G. Rio. Indexação distribuida com ponderação de termos. Master's thesis, Departamento de Informática, Universidade do Minho, July 1997.

[11] Mário Vilela. *Dicionário do Português Básico.* Edições Asa, 1991.

[12] Larry Wall and Randal L. Schwartz. *Programming PERL.* O'Reilly and Associates, 1991.

[13] Michael Wooldridge and Nicholas R. Jennings. Agent theories, architectures, and languages: a survey. In *Intelligent Agents, ECAI-94*, pages 1–40. Springer-Verlag, August 1994.