Information and knowledge managment final project

Mateusz Grotek, Jože Kraner

Abstract

The report describes the "eXpertIze" program, which is a knowledge application software, for helping users to realize what they have in mind, by asking them some questions. It could be used both as a game, which allows to guess the name of an object a player has in mind by asking him/her some questions, and as a serious software module for discovering what a user has in mind when e.g. learning a language, and not remembering a word. The algorithm used is general. We focused on the game functionality.

Contents

1 Introduction

Let us consider the general knowledge discovery problem. The process of knowledge discovery was defined in Memon (2011) as "the development of new tacit or explicit knowledge from data and information or from the synthesis of prior knowledge". Depending if an individual wants to gain explicit or tacit knowledge, possible strategies are combination of knowledge and socialization.

Let us imagine the following situation: a user does not remember a word in a foreign language. He or she knows exactly what he or she wants to say. A knowledge discovery system could help him or her to find a proper word describing an object he or she has in mind by utilizing what he or she already knows about the thing. The same strategy could be used for the knowledge capturing process, which was defined in Memon (2011) as "the process of retrieving either explicit or tacit knowledge that resides within people, artifacts or organizational entities". The difference between the two situations is the difference between the place the knowledge goes, it could be either a person or a knowledge base. In the knowledge discovery situation a person wants to gain some new knowledge. In the knowledge base situation an organization wants to transfer some knowledge from people's minds to organization's knowledge base. The source of the possibility of using the same algorithm lies in the following observation: in the presented situation a person is the source of the knowledge. Nevertheless the end user of the knowledge is different.

To be able to solve the problem we need to specify it further and to simplify it. The reasons for that are both time constraints and perceived difficulty of fully solving the problem. For this reason, we imagined the following scenario. A user has a material object in mind (but not **necessarily** a name of the object). By the least number of interactions with the user and by using a proper knowledge base we should be able to guess a name of the object.

2 Literature review

There are two general ways of solving the problem. The first question we should ask is: "who should drive the interaction?". The possibilities are the following: interaction could be driven ether by a user, or by a system. For user driven interaction there are many possible algorithms. The problem is very similar to the web searching problem. Let us imagine the following situation. A user has an object in mind, but he or she does not remember the name of it. By using some related queries to a web search engine he or she is possible to find a name of the object. For a way to implement this solution please consult Brin & Page (1998).

We decided to use the other way, which is driving interaction by the system. In this case it is the system, what asks a user some questions. By using this method our system is more general and we could pose the problem as a game between a user and the system. It means, that our algorithm could be used in entertainment. Because of that our problems transforms to the very similar problem of 20 questions game. 20 questions was a popular american radio program airing from 1946 to 1954 (for more information consult Van Deventer et al. (1946)), but the game is based on the popular game "Animal, vegetable or mineral?". The problem of 20 questions game is as follows. A user has an object in mind. By asking him or her 20 questions to which he or she can answer only "yes", "no" or "cannot tell" we have to guess a name of the object. The only difference is the limitation to yes/no/cannot tell answers. Because each question can be transformed to a set of questions of the yes/no type the solution is not less general, but the number of questions can be much larger. We decided to pose our problem as a 20 questions game.

As far as 20 questions problem existing algorithmic solutions are concerned the most comercially successful one is the solution patented by Burgener (2005) and used in a popular electronic device called 20Q. The algorithm used there is an artificial neural network. It maps the set of possible question/answer pairs to the set of possible objects. The main advantage of this system is a possibility of learning: the system asks a user if the answer is correct and changes the weights in neural network accordingly. The main disadvantage of the system is impossibility of influencing choices made by a system by changing knowledge base of the system. The knowledge is hidden in neuron weights and is not easily accessible for selective modification.

We decided to use a different method of solving the problem. We used Prolog programming language for implementing a knowledge base and a system operating over it. Prolog is a general purpose logic programming language used in many domains, especially in artificial intelligence and computational linguistics. It was created by Colmerauer and Roussel (see Colmerauer & Roussel (1992) and Kowalski (1988) for the history) and implements a subset of first order logic called horn clauses, with a method of reasoning called SLD-resolution invented by Kowalski (1974). For more information about prolog please consult Clocksin & Mellish (2000), Bratko (2001). As a logic programming language prolog have many advantages when creating artificial intelligence and computational linguistics programs. One of them is that it is very easy to implement reasoning in prolog programs. Another one is the knowledge base operations can be coded inside the language, and transparent knowledge base can be part of a program. These characteristics made Prolog very popular tool for creating logical reasoning programs, expert systems and other types of rule based systems. The addition of definite clause grammars to the syntax of Prolog allowed it to be used as an efficient tool in natural language processing systems. Despite the reputation of logic and rule based systems as beeing unable to implement fuzziness it is actually quite simple to do that in prolog. Because of these and other features of Prolog we decided to use it to create our system. Although Burgener (2005) states that: ”another major problem that is readily apparent in such systems is their incapability to handle inaccurate or misleading information if a player answers one question inaccurately this will cause the system to pursue the wrong branch of the decision tree leading it to the wrong answer or guess”, it was very easy to overcome this difficulty by implementing specific ”soft” rule of reasoning. What we gained is a clear and easy to modify structure of the knowledge base, which was absent in the system based on neural networks.

3 System architecture

3.1 General description

To implement the system we used SWI-Prolog. We used modular architecture, our system consists of four modules: loader, interface, engine and knowledgebase. There is also a separate application called Knowledge Base Manager. The application is not written in Prolog and is described in the knowledge base part of this report.

To run the main application a user should run either `run.sh` or `run.bat` depending on the operating system used. For running the application with debugging output a developer should run `debug.sh` file. The application should run in all operating systems supported by SWI-Prolog. It was developed in the Linux operating system, and checked in the Linux and Windows operating systems.

Each module has a separate file responsible for loading other files residing inside the module. Each module resides in the separate directory. The loader module is used to load other components. It is also the place, where the configuration of application is stored. There are two important parameters stored:

possibility influence and depth limit, which will be described in the engine subsection.

The next part is the interface part, which is responsible for the interaction with a user. It asks the questions and transforms the answers to the form acceptable by the rest of the system, for example it transforms all answers such as: ”yes”, ”y”, ”true”, ”t” to the constant true. The following subsections describe two other modules.

3.2 Engine

The main structure of the application is

Listing 1: main_loop.pl

```
main_loop :- got_new_information, main_loop.
main_loop :-
  setof(O,active(O),L),
  show_list(L).
```

If there is any new information the system keeps collecting it. The loop stops when either the system is able to make a guess, or it is impossible to guess with the current information inside the knowledge base.

The main reasoning part resides in the `getting_information.pl` file. Let use focus on the most important parts of the file.

Listing 2: got_new_information

```
got_new_information :-
  best_attribute(X),
  answer_about(Y,X),
  is_remembered_for(Y,X).
```

This part finds the best attribute to ask about, gets the answer for the question of the attribute, and remembers the answers for the question. We will focus on the `best_attribute` part in the next paragraphs, now let us focus on the two other relationships. `answer_about` finds the question for the attribute, and tells the interface module to interact with a user. `is_remembered_for` removes the question from active questions, so it won’t be asked again, and modifies possibility of objects according to the relation’s possibility reduction value.

Let us describe the algorithm for calculating priorities.

Listing 3: list_of_relevant_attributes_with_priorities_based_on

```
list_of_relevant_attributes_with_priorities_based_on(A,P
    ,[X|R1],[Y|R2]) :-
  possibility_reduction(M,X),
  number_of_active_objects(N),
  possibility_influence(S),
  Y =\= 0,
  Y =\= N,
  Prio is abs(rdiv(N,2)-Y)-S*N*M/2,
  A = [X|Rest1],
  P = [Prio|Rest2],
  list_of_relevant_attributes_with_priorities_based_on(
      Rest1,Rest2,R1,R2).
```

Please focus your attention on the following line:

```
Prio is abs(rdiv(N,2)-Y)-S*N*M/2,
```

Mathematically we can describe the formula as follows:

$$Prio = \left|\frac{1}{2}N - Y\right| - \frac{1}{2}SNM$$

The symbols require an explanation. *Prio* is a priority, that an attribute would be asked as the next question. The question is chosen randomly among the questions with the lowest priority. N is a number of active objects. Each object has a possibility value attached. For each object the default possibility is 1. It can be overwritten by setting the possibility in the knowledge base, in case some objects are very unlikely to be chosen. For each question answered differently to the answer an object has attached through the existence or nonexistence of the attribute, the value is reduced by a number assigned to the attribute the question was asked. If the possibility is less than or equal to 0, the object gets inactivated, and N is reduced. M is the possibility reduction attached to the attribute. S is the influence the possibility calculations have for determining the priority. Y is a number of objects having the attribute.

Let us consider what happens if the possibility influence equals 0. It means the formula reduces to: $Prio = \left|\frac{1}{2}N - Y\right|$. This value is the lowest one if the number of objects having the attribute is near to the half of active objects. It means, that the questions which allow to reduce the number of active objects by half would be the most prefered ones. The reason for that is, that it is a faster way to discover the object a user has in mind (a logarithmic number of questions in comparison with a linear number of questions in a trivial algorithm).

Now let us consider what happens if the possibility influence equals 1. The formula reduces to: $Prio = \left|\frac{1}{2}N - Y\right| - \frac{1}{2}NM$. The minimal value of the minuend is 0 and the maximal value is $\frac{1}{2}N$. M is a number between 0 and 1. It means, that an attribute with the possibility reduction 1, but for which no or all objects have the attribute would have the same priority as an attribute with possibility reduction 0, but for which half of the objects have this attribute (priorities of both attributes would be 0).

It means the parameter should have a value between 0 and 1. Our experience showed the number 0.3 works well, but it is possible to modify it by an end user.

Let us move on to the last part of the engine. In `loader.pl` there is a depth limit parameter set. The meaning of it is to prevent loops when reasoning inside the knowledge base. It describes how many reasoning steps are allowed. If the number of steps is greater than the limit the system will stop trying to prove the relation about an object and return false. It is very helpful if the knowledge base is big, because it is relatively easy to overlook something when adding new knowledge. This feature will allow to reason even is there are some problematic entries in the knowledge base.

3.3 Knowledge base

In this subsection the knowledge base will be described in more details.

Why do we need a knowledge base? The system is asking questions, and this questions must be stored somewhere. But, if the knowledge base would store the questions only, it should be called a database. So, the knowledge base stores more than just the questions. It also stores objects, relations and logical relationships between attributes. It allows complicated reasoning to be applied over the knowledge residing inside the knowledge base. It was also created in such a way, that is is very readable and easy to modify for both humans and computers. It is easy to add and remove knowledge from it.

Each of the relations stored in the knowledge base represents a question about the object in mind. We can say that each question about the object represents one attribute of an object. Here is an example of a relation named `can_put_in_backpack.1.pl`.

Listing 4: Relation can_put_in_backpack.1.pl

```
question_about('Can all variations of it be put in a
    backpack?',can_put_in_backpack).
possibility_reduction(0.5,can_put_in_backpack).
can_put_in_backpack(X) :- can_put_in_pocket(X).
```

In the relation there is the following information stored: the question about the attribute **`'Can all variations of it be put in a backpack?'`**, the name of the attribute (`can_put_in_backpack`), and the possibility reduction value (`0.5`). The possibility reduction value is always defined as a value between 0 an 1 — it represents how good, or how bad is the question. The question "Can **it** be put in a backpack?" is not very good, because if, for example, a user has a projector in mind, he or she might answer with "yes". But if the question was "Can **all variations of it** be put in a backpack?", the answer should be "no", because there are big cinema projectors, which cannot be put in a backpack. The problem exists because not all users would think of the big projectors, so they will answer in a wrong way. Because we want to avoid the influence of users making mistakes in answering the questions, we have set the possibility reduction value. A simplified description goes like that: 0.5 means, that if a user answers with "no", the engine will subtract 0.5 from all possibilities of the objects, that have this attribute. When the value reaches 0 (or less), the engine stops asking questions about the object (it deactivates it). So, it is good if bad questions have a lower possibility reduction value and good questions, such as ("Is it an animal?") should have a higher possibility reduction value. With the help of this value, we allow a user to make mistakes when answering the questions. In relation `can_put_in_backpack.1.pl`. there is also, line '`can_put_in_backpack(X) :- can_put_in_pocket(X).`'. This means that if something can be put into a pocket, than it also can be put into a backpack. This is a logical relationship between relations and it enables advanced reasoning inside the engine.

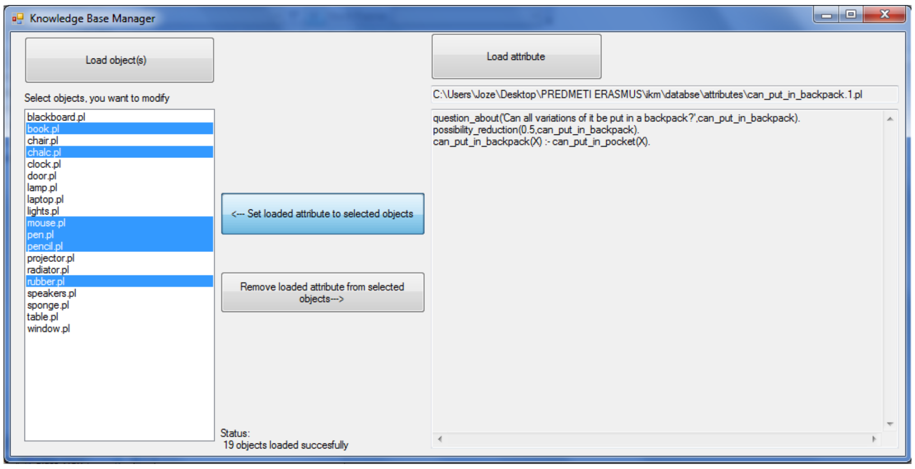
Our knowledge base, consist only of few objects. We have created small but complete set of objects. But the aim was not to create a big knowledge base, but to create a good system, so the knowledge base could be expanded in future. Here is an example of an object in the knowledge base.

Listing 5: Object book.pl

```
can_put_in_backpack(book).
can_be_opened_closed(book).
consist_of_pages(book).
```

Each object that is stored, includes the information about the names of the attributes the object has.

We have created a simple window application called **Knowledge Base Manager**, to work with objects and attributes. The application is very simple and it just presents an idea of the direction the system could be developed.


Knowledge Base Manager
Load object(s)
Select objects, you want to modify
Load attribute
<-- Set loaded attribute to selected objects
Remove loaded attribute from selected objects-->
Status:

```
blackboard.pl
book.pl
chair.pl
chalc.pl
clock.pl
door.pl
lamp.pl
laptop.pl
lights.pl
mouse.pl
pen.pl
pencil.pl
projector.pl
radiator.pl
rubber.pl
speakers.pl
sponge.pl
table.pl
window.pl
```

```
C:\Users\Joze\Desktop\PREDMETI ERASMUS\ikm\databse\attributes\can_put_in_backpack.1.pl
```

```
question_about('Can all variations of it be put in a backpack?',can_put_in_backpack).
possibility_reduction(0.5,can_put_in_backpack).
can_put_in_backpack(X) :- can_put_in_pocket(X).
```

```
19 objects loaded succesfully
```

Figure 1: Knowledge base manager application

As showed in the figure, the application allows manipulation of objects, which is assigning attributes to objects or removing attributes from objects.

How to build a good knowledge base? There is an article about 20 question game Nilssen (1997) that helped us to build the knowledge base. There we found some good advice, for example, which questions should be asked in the beginning. An example of a very bad question is "Is it big?". This is a bad question because it is a question of degree, when it should be a question of comparison. So a better question is "Is it bigger than a backpack?".

4 Limitations

As said before, the algorithm used in our system is general. It may be used for entertainment or in some other, more serious way, for example when learning languages. One of the limitations this system has is the method of filling of the knowledge base. For now, the objects and the relations (the attributes) have to be put in manually. Still, it is possible to create some automatic means of doing that.

The second limitation is that our system is unable to learn by its mistakes. Nevertheless, by changing possibility values of relationships it is possible to create a learning algorithm for our system. For example, if the system asked for some feedback from a user, it would be possible to change possibility values depending on the comparison of the user's answers and the answers stored inside the knowledge base.

Another limitation is, that a user can answer the question only with "yes" or "no". So answers such as "I do not know" are not possible. The reason for that is the agreement, that only possitive answers should be answered with "yes".

All other answers should be answered with ”no”. The system is designed in a way to make it clear to a user, but if it is not important, the user interface part could be changed easily to incorporate other answers. Nevertheless, the system is based on yes/no answers to each question. It is possible to add other types of questions, for example questions of degree etc., but as discussed in introduction, all questions can be converted to series of yes/no questions.

The final limitation is the limitation of prolog itself. It uses only Horn clauses, and it is not a full implementation of predicate calculus, so it is not possible to state all relationships between relations. For example it is not possible to state: ”(not A) or (not B)”. This is a price we pay for our system being fast (the complexity of full predicate calculus resoning is exponential). Nevertheless Prolog adds a non-logical operator (so called cut operator, symbolized by `!`). With the help of this operator it is possible to describe much more relationships, that with Horn clauses only.

We have to take care not to state any relationships which could make the program to go into a loop (the simplest possible case is `xxx :- xxx`). For this reason we implemented the loop prevention mechanism described in the engine subsection.

5 Conclusions and future work

There is a lot of things in domain of this system that could be improved. Here are some suggestions.

- **Filling the knowledge base** — The knowledge base could be filled automatically, using information found on the internet. For example, it could use Wikipedia to learn about objects and their attributes. This is how the knowledge base could be expanded. The application ”Knowledge Base Manager” could be improved.
- **Learning the system** — It might be possible to implement a learning algorithm for our system. It should allow the system to learn on its own mistakes.
- **More answers** — We could offer a user more answers to questions. For example ”I do not want to answer”. This feature could reduce mistakes users do when they are forced to answer ”yes” or ”no”. Nevertheless, implementation of this feature on the user interface level only could make it more difficult to understand the system by a user. This is the reason we did not implement this simple feature.
- **Different types of questions** - The development of this feature is probably more complex, but it would be very useful. The system could be upgraded in a way, that it would ask questions like ”If you could describe an object with one word, what it would be?”. So the system could connect the answer with the object the user has in mind.

Without doubt, the system that was developed has a lot of potential. The algorithm of choosing questions can be used in a much bigger knowledge base. So the future work should be focused on the expansion of the knowledge base.

References

Bratko, I. (2001). *Prolog: programming for artificial intelligence* (3th ed.). Harlow: Pearson Education Limited.

Brin, S., & Page, L. (1998). *The Anatomy of a Large-Scale Hypertextual Web Search Engine.* In Proceedings of the Seventh International World-Wide Web Conference. Web site: `http://www7.scu.edu.au/00/com00.htm`

Burgener, R. (2005). *Artificial Neural Network Guessing Method and Game* (United States Patent Application Publication, Publication No. 2006/0230008). The United States Patent and Trademark Office. Retrieved from: `http://www.google.com/patents?id=W5iZAAAAEBAJ`

Clocksin, W., & Mellish, S. (2003). *Programming in Prolog* (5th ed.). New York: Springer-Verlag.

Colmerauer, A., & Roussel, P. (1992). The birth of Prolog. *ACM SIGPLAN Notices*, 28(3), 37-52. doi:10.1145/155360.155362

Nilssen, E. (1997). *The game of twenty questions.* Retrieved Dec 26, 2011, from `http://barelybad.com/20_questions.htm`

Kowalski, R. (1974). Predicate Logic as Programming Language (pp. 569-574). In Proceedings of IFIP Congress. Stockholm: North Holland Publishing Company.

Kowalski, R. (1988). The early years of logic programming. *Communications of the ACM*, 31(1), 38-43. doi:10.1145/35043.35046

Memon, N. (2011). *Information and Knowledge Management.* Retrieved from the University of Southern Denmark, lecture notes. Web site: `http://www.mip.sdu.dk/~memon/IKM_2011.htm`

Polesie, H. (Producer), Van Deventer, B., Van Deventer, Fl., Van Deventer, Fr., & Van Deventer, N. (Speakers). (1946). Twenty Questions [Radio broadcast]. Retrieved from: `http://www.otrcat.com/twenty-questions-p-1948.html`