|  |
| --- |
| Chalmers university of technology |
| Artificial intelligence project report |
| Group 20 |
|  |
| **Pierre BOUTRY, Dan DOLONIUS, Julien MICHELET, Emeric ROVERC'H** |
| **21/05/2014** |

|  |
| --- |
| [Tapez le résumé du document ici. Il s’agit généralement d’une courte synthèse du document. Tapez le résumé du document ici. Il s’agit généralement d’une courte synthèse du document.] |

Table of content

[1. Project Presentation 2](#_Toc388044889)

[2. The World 3](#_Toc388044890)

[3. Grammar 4](#_Toc388044891)

[4. Interpreter 5](#_Toc388044892)

[5. Planner 7](#_Toc388044893)

[6. Ambiguities handling 8](#_Toc388044894)

[7. Output 9](#_Toc388044895)

# Project Presentation

The goal of this project was to implement a dialogue system for controlling a robot living in a virtual block world. The robot we implemented can move around objects of different forms, colours and sizes, and it can answer questions about the world and ask for clarification whenever it finds request ambiguous.

This has been performed using grammaticalframework (gf) for the grammar and java/prolog for the implementation. The world is described using a json file.

Structure global du projet

Parser : query => arbre

Grammar/interp: arbre => Goal/objectif

En fonction du type de goal, il y a plusieurs cas possible

If singleton

Prolog : prend query appel dgc parser pour produire un arbre

Below is a macro explanation of how work the global application.

On client side :

The user type a query as an input. This query is then

couche client:

on tape une query dans un input qui est traité via un appel ajax en jquery

cet appel ajax en particulier va exécuter notre main en prolog ( shrdlite.pl )

couche server:

l’appel ajax qu’on a fait contient plusieurs champs :

world': currentWorld.world,

'objects': currentWorld.objects,

'holding': currentWorld.holding,

'state': currentWorld.state,

'utterance': userinput.split

donc world objects holding state utterance

( en fait tout ca est stocké dans un JSON qui contient le monde avec les objets et notre requête )

ce JSON est lu par prolog et traité

première étape : Trouver le GOAL ( l’objectif )

par exemple si notre query est : move the white ball in te red box

le goal sera moveinside([white ball],[red box])

et en fait ca c’est fait via l’interpreter

( que dan a codé )

qui va interpréter notre requête pour la rendre compréhensible au programme ( fichier interpreter.pl )

Une fois qu’on a notre goal il faut notre plan de résolution

pour ca on a implémenter dans Planner.pl l’ensemble des fonctions permettant d’exécuter tous les mouvements possibles

et ces fonctions sont appelés via le fichier solver.pl

qui contient la fonction solve qui nous permet de “résoudre” la demande de l'utilisateur

une fois que tout est résolu on renvoie coté client le résultat

et avec quelques fonctions JS on gère l'animation

et l’affichage des étapes

ben en gros la grammaire est chargée lors de l’éxécution du programme prolog pour reconnaitre les mots

elle est utilisée dans l'interpreter

# The World

The world is represented by a floor on which several objects (of different forms, colors and sizes) can lay. The objects can stand in/on each other (if it is permitted by the world’s laws).

The goal of this project is to move around the objects, according to the request of the user, through a robot arm which can pick up and put down objects.

The floor is divided in n spaces, meaning that there is room on the floor for no more than n objects at the same time. Each of this space is represented has a column, so that the world can be describe as a list of n columns of objects stake on each other.

The world we implemented can contain all the objects of different forms, colors and sizes listed on the course homepage, plus the possibility to define an object of medium size (in order to create a more complex/realistic world).

Bellow the list of forms, colors and sizes available in “our” world:

Forms: bricks, planks, balls, pyramids, boxes, tables.

Colors: red, black, blue, green, yellow, white.

Sizes: large, medium, small.



This world is described using a json file describing the world as a list of columns of objects.

# Grammar

We started with the initial grammar (given at the beginning of the project) that we have completed and improved.

The major improvements we implemented are the possibility for the user to ask 3 new types of questions: where, what and count.

Where:

The “where” question allow the robot to answer questions such as “where is the white big ball?” Or “where are the boxes?”. The synonym for the “where” questions are the words: Find, Where is and Where are.

What:

With this “what” question, the robot can answer questions such as “what is under the red box?” Or “what are the object in the world?”. The synonym for the “what” questions are the words: What is and What are.

Count:

This last new possible question aim to permit the user to get answer to requests such as “count the boxes in the world.” or “how many ball are in the world?”. The synonyms for the “count” questions are: Count and How many.

Another major improvement we made from the original grammar is the possibility for the user to request actions and/or ask questions about stack instead of simply the whole world. The user can then perform request like “what are the objects in stack 2?” Or “count the small blue balls in stack 0.” Or “what are the objects on the right of stack 2?”

We also added 2 minor improvements to the original grammar: the alias “world” for “all the stacks” and the possibility for the user to put question marks at the end of his questions without having the robot answering with an error message.

Of course all those modifications regarding the grammar had to be echoed to the other layers of the application (parser and planner).

# Interpreter

//first info from Dan

After the grammar layer we then proceed to the interpreter layer.

For starters we have a world like [[a,b],[c]...], where the letters correspond to some element defined according the world’s rules on forms, colors and size. Those Objects are of the form:

Objects = json([a = json([Type,Size,Color]),...]

In order to query an object of defined form, size and color we created the following function:

getobj([anyform,-,-],PossibleObjects,SelectedObject) :-  
    member(SelectedObject=json([form=\_,size=\_,color=\_]), PossibleObjects).

where PossibleObjects is a list from which we want to query and of the same form as Objects.

Then to get the element, we need to make sure that SelectedObject is satisfying the form, size and colour of the world and is a member of  PossibleObjects.  
  
Further we implemented some functions to check if, for example, an object is besides any other object:  
isbeside(X,Y,World) :-  
    member(ColS,World),member(X,ColS), nth0(IdxS,World,ColS),  
    member(ColR,World),member(Y,ColR), nth0(IdxR,World,ColR),  
    (IdxS is IdxR-1;IdxS is IdxR+1).  
  
When we want a Goal for a given tree we use the following function:  
interpret(move(X,relative(beside, Y)), World, Holding, Objects, movebeside(SelectedObject,RelativeObject)) :-  
    interpret(X, World, Holding, Objects, SelectedObjectTAA),  
    interpret(Y, World, Holding, Objects, RelativeObjectTAA),  
    handleQuantifiers(SelectedObjectTAA, RelativeObjectTAA, SelectedObject, RelativeObject).  
  
The interpret function is recursive, but the output for SelectedObjectTAA is assumend to be of the form [a,b,..] or any([a,b,..]).  
The function handeQuantifiers uses cuts to choose one possible action, when several are possibility are available to the robot (for example, any([a,b])).

Below are a few examples of the cuts performed by the function:

|  |  |
| --- | --- |
| Original query | Interpreted query |
| movebeside([a,b],[c,d]) | movebeside([a,b],[c,d]) |
| movebeside(any([a,b]),any([c,d])) | movebeside([a],[c]) |
| movebeside([a,b],any([c,d])) | movebeside([a,b],[c])) |
| movebeside(any([a,b]),[c,d]) | movebeside([a],[c,d]) |

If we look at this rule:  
interpret(relative(above,X), World, Holding, Objects, SelectedObject) :-  
    interpret(X, World, Holding, Objects, RelativeObjectTAA), (any(RelativeObject) = RelativeObjectTAA ; RelativeObject = RelativeObjectTAA),  
    findall(SelectedObjectAux,  
    (member(RelativeObjectAux, RelativeObject), isabove(SelectedObjectAux,RelativeObjectAux,World)),  
    SelectedObject),SelectedObject \== [].  
  
Here we don't really care about any (it's handled elsewhere) so we simply extract the list of RelativeObject. We then find all objects satisfying the relation.

isabove can return a list so we need to check every member of that to check relations.empty lists satisfy the relation as well, so we disregard those.  
  
  
The rule:   
interpret(relative\_entity(any,X, Relation), World, Holding, Objects, any(SelectedObject)) :-  
    findall(RelativeObjectAux, ( interpret(Relation, World, Holding, Objects, RelativeObjectListAuxAux),  
                                 member(RelativeObjectAux, RelativeObjectListAuxAux)),  
                                RelativeObjectListAux),  
    sort(RelativeObjectListAux,RelativeObjectList),  
    %Find all objects which supports the relation  
    findall(SelectedObjectAux,( member(SelectedObjectAux, RelativeObjectList),  
                                interpret(X, World, Holding, Objects, SelectedObjectAux)),  
                                SelectedObject).  
  
is simply collecting all objects satisfying a relation. We first find all possible objects satisfying the relation, then sort the list to remove duplicates and finally find all objects satisfying the description as well as being a member of the relation satisfactions.  
These rules are quite similar for "all" and "any", but we modify the return to act as a flag.  
  
  
last example  
interpret(object(Type,Size,Color), World, Holding, Objects, SelectedObject) :-  
json(AllPossibleObjects) = Objects,  
findall(X=json([A,B,C]), (member(Col,World),member(X=json([A,B,C]),AllPossibleObjects),member(X,Col)), PossibleWorldObjects),  
member(Holding = json([A1,A2,A3]),AllPossibleObjects),  
append(PossibleWorldObjects,[Holding = json([A1,A2,A3])],PossibleObjects),  
getobj([Type,Size,Color], PossibleObjects, SelectedObject).  
  
this will return the letter of queried object  
what we do is to extract the objects in a more manageable form, append what the arm is holding, and lastly "get" the objects from the possible objects.

//second info from Dan

//to merge an reformulate

# Planner

The first step in our implementation was to implement terminal case, i.e. the move of an object that does not require another modification of the world. A terminal case could be “Put the black ball beside the yellow box“.

The planner takes a query as an input and build list of triplets, where each triplet represent a feasible action. There are 3 possible terminal moves, each represented by a specific triplet:

|  |  |
| --- | --- |
| Motion | Triplet |
| Pick | [K1,-1,move] |
| Drop | [-1, K2,move] |
| Move | [K1,K2,move] |

Where K1 is the position of the object to pick and K2 the position where the object has to be drop.

Once the terminal moves have been implemented, we can proceed to the implementation on more complex cases. Such as moving an object that is under one or more objects.

The output of our plan function for complex cases is a list of triplets/actions (pick, drop, and move) that lead to the Goal.

In order to create this list of actions, the plan function is call recursively. The list of actions to reach the Goal is then a concatenation of all the actions to get from the original world to the requested world. This can also be view as a concatenation of all the branch of the decision tree used by prolog to find a solution to find a way to the requested state of the world (see below).

Here is an example:

The user query is “Put the blue box in the red box”.

The plan function performs an in-depth research in the prolog decision tree.

|  |  |
| --- | --- |
|  |  |

So the output of the plan looks like:

Plan = [[4,3,move],[4,2,move]]

Heuristic

When the robot as to deal with not straight forward case, such as moving object that are not on top of a stack or moving a ball on top of a table, we need the heuristic in order to optimize handling of those case and to move objects in a better way.

The first heuristic we implemented was really basic. Basically, it was moving objects randomly and simply checking if the resulting state of the world has not been met before. For that, we increment a list of all the states of world we reach and checking if the resulting state is not in the list.

Improvement, in order to move object in a better way.

Answering questions

First,

# Ambiguities handling

The ambiguities handling allow the robot to handle cases where the request from the user is not clear enough. For example, in picture 1, “what is under the box?” would lead to an ambiguity, since the robot would not know if the user want to know what is under the red box or the blue box or the yellow box.

In case of ambiguity, the robot ask the user for a precision. The user then has to precise the object he is referring to. In our example, it could be “the small blue box”.

Once the user has precise the object, the robot will get this information and try to match with all the possible goal he has identified. It then select the unique matching solution (if it exists).

If an ambiguity still occurs the robot then return an error. There is no second question ask to the user since prolog does not handle while loop.

# Output

The particularity of our robot is that the possible outputs are not only an action (such has moving objects around) but also possibility some verbal information such as the number of elements stack on top of each other.