Artificial Intelligence Project

Andreas Arvidsson
andarv@student.chalmers.se
Sebastian Lagerman
seblag@student.chalmers.se
Johan Swetzén
swetzen@student.chalmers.se
Robin Touche
robint@student.chalmers.se

Introduction

We have been working on an artificial intelligence project which has resulted in an interpreter, ambiguity resolver and planner for an organizing robot simulator. The robot can react to user commands and move objects in the world accordingly. In this report, we will describe some implementation details of the various parts of the application itself, as well as our experiences with the project and problems we encountered during development.

Workflow

The group decided to work with an agile workflow, with regular meetings and development sessions. With this kind of workflow, we had the ability to focus on one utterance at a time. To avoid as much overhead of working multiple persons on a single code base as possible, we decided to use git for version control. This helped the group to work in different part of the code without conflicts.

Most of the information regarding artificial intelligence was taken from the slides, which we do not have referenced in the report.

Notation

In order to simplify the work for both us and the reader, we will use a notation for describing function types which is inspired by the notation used in Haskell. In case the reader is not familiar with this notation - here's an example:

```
funName :: ArgType → ArgType2 →
    ReturnType
```

The notation starts with the function name ("funName"), which is separated from the rest of the type by a double colon ("::"). Then, the input argument types are separated by arrows ("→") - with the exception of the last variable, which is always the return type. So, to write a function which takes a list of integers and returns a bool, we could write:

```
foo :: [Int] → Bool
```

This is a very concise way to write function types, which is why we've chose to use it. However, this notation will be used sparingly throughout the report to show the most important functions in the various parts of the application.

Relations

There are a few object relations which the user can provide via the utterance (or command). These are "beside", "left of", "right of", "above", "on top", "under" and "inside". To avoid confusion, we will interpret these in the following way:

Beside The assumption here was that the two objects should be placed in two columns that are located next to each other.

Leftof We interpreted this to be that the first object would have to be placed in one of the columns that are left of the second objects' column and vice versa.

Rightof Just like `Leftof`, but the two objects are flipped.

Above This was interpreted to mean that the two objects would be placed in the same column with the first object being above the second one, but there could exist objects between them.

Ontop We took this to mean that the first object needed to be placed strictly on the second object with no objects in between them.

Under This was interpreted just as `Above`, but the two objects were flipped.

Inside We assumed this relation could only be applied to boxes, but otherwise it would follow the `Ontop` relation.

Goals

There are two goals which cover all basic utterances that are valid. First, we have a goal we chose to call a TakeGoal. This goal initially just contained the object Id that was to be picked up. However, when doing the Ambiguity Resolver we were forced to add a quantifier taken from the utterance as well. This is necessary to differentiate the goals based on the quantifier the user chose.

The second type of goal is the MoveGoal. This contains two object ids (and their quantifiers), and also a relation between them. Using these, we can both handle user movement commands, but also simple put commands (by creating a MoveGoal with the currently held object as the first id).

Parser

The input to the application is in the shape of a JSON string containing the world state, the user's utterance and other information that the interpreter and planner needs. This string is sent from the web interface that was provided for us. When the application receives the JSON string, it parses it into a bunch of useful data structures which we can later use in various parts of the application. Much of this code was provided along with the web interface - we've only extended the parser with a few convenience functions for object comparisons. We also extended the parsing to specific data structures like the world state and an object map which maps object ids to their corresponding descriptions.

State

The data type representing world state is, in our case, very simple. It contains the world obtained from the JSON parser, which is just a list of lists where each inner list represents a stack of object ids. The State data type also contains the object Id of the currently held object and, of course, a map for receiving object descriptions based on their object id. This is all the information we need for the interpreter regarding the world and, as it turns out, also all we need for the Planner and Ambiguity Resolver.

Command

A command is created from the user's utterance. This can be one of three data types - Take, Put and Move - depending on the utterance. They contain a description of the objects, and this is used later in the interpreter to convert it to goals for the ambiguity resolver and planner.

Interpreter

The interpreter is meant to be analyzing the world to find objects that match the criteria such that the planner can work with the objects id and not the objects description. Keeping this in mind, we soon realized that we needed a way to find object based on their description. This is the main focus on the interpreter. A few ambiguity errors can occur here - mainly based on which quantifier is used. This has to be included in the interpretation output, so that the ambiguity resolver can handle it.

The implementation of the interpreter began by writing a suitable type for it. We knew that the interpreter needs to have access to some kind of world state, in which it will search for objects. And the goal of the interpreter is to interpret a user command into a list of possible interpretations for the planner. Using these observations, the type could be written as:

```
interpret    :: State → Command →
    Either InterpretationError [Goal]

interpretAll :: State → [Command] →
    Either InterpretationError [[Goal]]
```

The purpose of `interpret` is to interpret a single command, and `interpretAll` just interprets all the provided commands one after another. The second one is used from the main application to interpret all the possible parsings of the user utterance (since often there could be multiple interpretations).

We decided to make an internal monadic interface for the interpreter, which is why the return type looks a bit funky. This was partly because we didn't want to pass the world state around all helper functions, but also to be able to handle possible errors. If the interpreter can't, for example, find a certain entity matching the user command, we can return an error which can be used in other parts of the program to give the user a nice output.

Goal

The end goal of the interpreter is to convert the different user utterance interpretation (from the parser) to a list of different PDDL goals. In order to do this, we need to, as mentioned above, translate the object descriptions to their corresponding id. This is done by running the different search functions. When we receive the result from them,

we just create the goals by checking location constraints etc for all the matching objects. The location checks could have been moved to the Ambiguity Resolver, but we decided against this since without it the interpreter becomes almost trivial to implement.

An important addition that we did late in the project was to include the quantifiers in the goals, so that the ambiguity resolver knows how to handle quantity-based ambiguities. The plan was to handle this in the interpreter first, but we found that it was a bit clearer to move this to the Ambiguity Resolver because it seems to fit better there.

Object search

The implementation of the interpreter started with the realization that we needed a way to find objects in the world based on their appearance. In order to do this, we started fleshing out simple search functions, so that we can construct PDDL goals that references object ids as opposed to the object descriptions available from the user command. The plan was that if we find multiple matching objects, we can either return one of them, or return an ambiguity error - depending on the user utterance.

The function for searching for matching objects takes as input the world state (implicitly via the monadic interface), an object description, a quantifier (from the utterance) and an optional relative location. It returns a list of all the correctly matched items when successful, but of course there is a possibility of error here that we can easily express with the monadic interface. This function was relatively straightforward to implement, although it should be noted that a few bugs were discovered during the test writing.

Error handling

The interpreter does not handle any ambiguity errors. This means that it will return a list of goals (coupled with quantifiers). However, there can still be errors. For example, some functions might expect a certain id to exist. If the matching object cannot be found, an interpetation error will be propagated throughout the interpretation process and ultimately returned to the caller.

Ambiguity resolver

In order to resolve any ambiguities from the output of the interpreter, the ambiguity resolver needs the state of the world and, of course, the list of goals themselves. The type of this function can therefore be written as:

```
resolveAmbiguity :: State → [[Goal]] →
    Either AmbiguityError Goal
```

We have a list of lists of goals, since there will be one list of goal for each possible interpretation of the user utterance. The return type of our ambiguity resolver is either an error (in case of ambiguities due to quantifiers or other reasons) or a single goal which will be sent to the planner.

Physical laws

Once we have received the goals from the input of the main ambiguity resolver function, we need to check all of them if the laws of physics will apply on all of them. The check is fairly simple. We iterate over the list of goals, check which type of goal it is (move or take goals), and process them each to remove any impossible goals. Notable examples that are being removed in this stage are if the user tries to place a brick on top of a ball (which violates the physical laws) or the following more complicated example. If the world contains three objects: a large ball, a small ball and a small box, the utterance “put the ball in the box” could be considered ambiguous. However, since the large ball cannot be placed in the small box, this invalid goal is removed and the ambiguity is resolved so that only one goal is left.

Quantifier check

The ambiguity resolver also deals with ambiguities arising from the use of quantifiers. If we get an utterance specifying “the ball”, we make sure that only one ball (for which the goal is physically valid) is found in the world. Or if the user writes “put the ball in the box”, both “the ball” and “the box” needs to be unambiguous. Variations on this, like “put the ball in a box”, only require one of the objects to be specific whereas the other can match any number of unique objects.

In the case of an ambiguity we tell the user what different objects could match the utterance, starting with the first object description. So if “put the ball in the box” is ambiguous in both the ball and the box, the user will be told about the ball ambiguity first. If the next utterance then has specified this, for example “put the small ball in the box”, the second ambiguity is handled. We have chosen to do it this way since any combination of balls and boxes could lead to quite a large response.

If the quantifiers do not lead to any ambiguities, we know that any of the remaining goals are valid, so the first one is picked. An improvement of this would be to let the planner choose among the goals to find the simplest one, but this is not done at the moment.

Planner

Planning The planner takes the goals that the interpreter and ambiguity resolver have created from an utterance and finds a way to actually fullfill those goals. The planner uses the A* algorithm to explore possible solutions and shortest solution (smallest number of take/drop steps) based on heuristics as explained below in more detail.

The planner makes sure that only valid solutions (that upholds the given physical laws) will be considered. However, the algorithm only validates the possible steps taken and avoids impossible states, which means that it assumes the initial world is valid as well. If not it cannot reach a conclusion.

Heuristics

We started out by just implementing the "take" utterance, so our heuristics were simply the number of objects on top of the sought-after object. As we continued to add more utterances, that part was left unchanged until we started getting major speed issues. We long thought that this was an issue with our state being too big, because we saved the whole world in every node of the graph that the A* algorithm was searching through. After rewriting the graph representation multiple times we realised that the problem was not with the graph, but with the heuristics. The number of objects above a given object turned out to be a very bad idea, and twice that number is much closer to reality. This cut our search space down from 7000 nodes (basically bredth-first search) to 40 nodes visited when picking up an object with about five objects above it.

Take Firstly the hueristics checks what the arm is holding at the moment. If the objects id matches the one we wish to take then there are no more step needed. However if it's another object then we need to remove it from the arm which adds one step. After which we find the object we are looking for and remove the objects lying above it, this is calculated to take two steps for each object. Once all that is done then there is only one more step needed to pick up the object we are after. One the other hand when we are not holding anything then we do the same calculations, but without adding the one step for clearing the arm.

Move to floor A special case was created for puting an object on the floor since the floor otherwise can't be interacted with. The hueristics starts of by checking if there are any empty floor spaces. After which both cases checks what it is holding and if it's the object we wish to place on the floor. If this is true then when there exists a floor space then it's only one step to drop it down. However if it doesn't exist any space then we decided that the number of steps it would take to remove a minimum would be the length of the shortest column multiplied by two since it takes two steps to pick up and drop down an object. After which it took three aditional steps, one for puting the object down before clearing away the smallest column and then two steps to pick the object up and drop it down on the floor.

Ontop The utterance started of by checking what it was holding. If it was the object we wished to place on the second one then we only checked how maney objects were place on it. For each object we count two steps to remove them in addition to three more steps to put down the first object we were holding then picking it up again and placing it on the second object. On the other hand if it isn't one of the objects we are suppose to handle then we need to remove it from the arm which takes one step. Then we need to clear away all the object on the two objects which takes two steps for each. Finally we move the first object onto the second object which also takes two steps.

Inside Works exactly like the `Ontop` utterance.

Beside This utterance will firstly check if the goal is satisfied and return zero steps. Otherwise it checks what is the arm is holding, if it's nothing then it will look up how maney objects are placed on the two objects. Then we take two steps of each object on the smallest pile to then be moved beside the other object which takes two additional steps. If the arm is holding something then we check whether

it is one of the object and then assume it will only take one step to put it in the column beside the other object. Otherwise we need to take a step to free the arm from the object and then go about clearing away the smallest pile of objects on the two objects we wish to place next to eachother as mentioned before.

LeftOf Works much like the `Beside` utterance except we don't have to be as strict since we interpret `LeftOf` as any column as long as it's left of the other objects column.

RightOf Works like the `LeftOf` utterance except right instead.

Above This utterance works as `Ontop` except it checks whether the one of the objects above the second one is the first one.

Under Exactly like `Above` except we have flipped the first with the second object.

A*

The planner uses our own implementation of the A* algorithm which is based on the description by Lester (2005). It takes as input a graph, a heuristics function, a goal-checking function and a starting node. To represent the nodes, we have chosen to use the world (a list of lists of id strings) and the object that is currently being held. Our graph is described as a function that takes a node and returns the possible nodes following it, so we do not have to hold an infinite graph in memory. We use a priority search queue for keeping track of the "open" nodes, the ones we have yet to visit, and their estimated cost, i.e. path cost + heuristic. At all times we also save the visited nodes, the cheapest path cost for getting to any node and the parent of each node. Iterating through the open queue, taking the cheapest node every time, we will eventually end up with a solution that is optimal with respect to the number of pick and drop actions required.

Extensions

- Ambiguity resolution by listing possible different objects, eg. "You could mean the Yellow Box or the Red Box", if the user wanted to pick "the large box" although there were two of them.
- Find the shortest solution, measured by the number of pick and drops. This is inherent to the A* algorithm that we are using.
- Support for differentiating between quantifiers. 'a' and 'any' means the same, but 'the' requires the specification to refer to a single object, or we output an ambiguity error.

Final thoughts

The project was immensely helpful for our understanding of artificial intelligence for problem solving. It would've been nice if all the groups didn't have exactly the same problem, since we had some cool ideas and watching the same solution in the presentation slots might be a bit boring.

Our project organization could perhaps have been a little better planned with regards to the structure of the code. Late in the project, when we needed to handle errors in a nice way from the user's perspective, we realized that using monads for this would be a nice solution. If we'd begun with this realization (which could have been achieved with more discussions and meetings), we'd have saved some time.

Finally, we did not use a Scrum tool like PivotalTracker. If we would have done this, the project might have finished quicker - but unfortunately we realized this a bit too late. Instead, we mainly used github's issue tracking system for long-lasting bugs (which there weren't many of).

Contributions

Andreas Arvidsson

My main contribution to the project has been to write the interpreter and functions related to the interpretation process. A huge chunk of this work was focused especially on writing search functions for the objects (which took a longer time to get right than expected) and writing unit tests for every helper function in the interpreter. Both were otherwise pretty straight forward to implement. I managed to discover some bugs in the development doing this, though, so it was probably worth it. I could probably have started writing the tests sooner, which is a common good practice anyways.

I also worked along Johan and Sebastian to implement the ambiguity resolver. This implementation happened late in the development process, so it forced the group to rethink the goal representation a bit. Specifically, we needed to include the quantifiers in the goals, in order to separate ”the” and ”any” in the ambiguity resolver. It was actually completed in a day, but it's not covered by the unit tests due to time limitations.

Furthermore, I participed in the group meetings, development sessions and discussion like any good boy would do (except a small period when I was sick due to being too beautiful), so I have a good grasp on the project as a whole.

Finally, I've written the whole interpreter section of this report, the introduction section, roughly half of the ambiguity resolver section, some touch ups on the report as a whole, and last but not lest - this whole contribution snippet. Yey!

Sebastian Lagerman

I have, together with Robin, created the very first draft of the report for the first deadline.

I created some tests for the validation function, which checks that the laws of physics are not violated in the planner (and ambiguity resolver).

In the introduction of this report I wrote how we decided to interpret the relations which the robot can use. I was part of writing the ambiguity resolver in this report. After which I went into details on how the relations heuristics were created and what calculations they perform.

My main contribution to the development of the application is the code I wrote for the the planner - more specifically the check function. This function is used in the A* algorithm to check if a goal is satisfied. I also spent a large chunk of my time together with the help of Johan to create the heuristics function, which also was used by the A* algorithm to calculate how many steps it could be until it reaches it goal. This is what took most of my time since I wanted a solid lower bound. Together with Johan and Andreas did we create the ambiguity resolver. I improved the validate function to be used in the ambiguity resolver. Other than that, I've gone through the code and cleaned it up and moved some stuff around.

Johan Swetzén

I started off early on working on how to use the A* algorithm to solve a graph problem in general. When time came to write the planner, I took up that task and got it working with the "take" utterance and an A* algorithm that we installed from hackage.

I implemented the A* algorithm on my own. As I wrote and rewrote the A* algorithm a few times due to the speed problems mentioned in the Heuristics section of the report, Sebastian continued to add heuristics for the "put" utterances. I came back after finishing the A* algorithm and improved the heuristics a bit as well as fixing the speed problems.

At the end of the development period, collaborated together with Andreas and Sebastian to write the base of the ambiguity resolver. I continued that work alone so that it checks the physical laws and outputs the different objects that could be referred to. The last formatting of the ambiguity messages (eg. "black ball or white ball" instead of "Object Ball Small Black or Object Ball Large White") was not done by me but by Robin.

In the report, I have written the last part of the Physical laws as well as the section on Quantifier check for the Ambiguity resolver. I have also contributed with the introduction to the Heuristics section as well as the section on A* for the Planner.

Robin Touche

Worked with Sebastian on the initial report for the first deadline.

Implemented the world validation checker. It checks how different objects are allowed to interact with each other and makes sure no physical laws as defined in the project are violated. Cleaned up the output of the ambiguity resolver to avoid redundant information. Wrote some tests for the check function. Other minor code snippets and optimizations that I can't really remember specifically.

Created the LaTeX-backend for the report (*i.e.* created and formatted the files, split the report into a format we could work with easily). Made the report follow the specified EACL2014 format. Created and integrated support for references via bibtex (or, more specifically, *a* refernce). While not directly responsible for any specific part of the report I have written paragraphs here and there (can't remmeber exactly where).

Various code cleanup the planner and common files. Documented planner, common and grammar.

References

Lester, P. (2005). A* pathfinding for beginners. http://www.policyalmanac.org/games/aStarTutorial.htm. Last accessed on 2014-05-17.