Artificial Intelligence Project - SHRDLite
Group 1

John Johansson, Sebastian Ånerud, Roland Hellström Keyte and **Joel Ödlund**

Abstract

In this project an agent was implemented for the SHRDLite-problem. The agent has its own grammar which it uses to parse utterances and then interprets them. The result of the interpretation is a PDDL goal which is then solved by a planner implementing the A^*-algorithm. A list of interesting features that are also implemented includes planning with multiple arms, disambiguation of complex nested ambiguities in utterances, large state spaces and complicated goals.

1 Implemented features

The following is a summary of the features which were implemented.

- All basic requirements
- Quantifiers
- Non-standard interpretation of commands
  - Planner may move things to meet any condition
  - ..unless specifically told not to.
  - More complex planning as a result
- Disambiguation
  - Specific, non-redundant questions
  - Natural language answers with dedicated grammar
  - Stateful dialogue with multiple layers of context
- Large worlds and complicated goals
- Additions to grammar and interpreter
  - Present and future tense
  - Stack command
  - Sort command
- Multiple arms
  - Multiple actions each step
  - Physical constraints on arms
  - Arms displayed in web client

2 Initial assumptions

Some assumptions are made that are significant for many parts of this project.

2.1 Assumptions about the meaning of commands

Consider the utterance ”put the white ball in a box on the floor” in the world given in Figure 1. In the original grammar, this sentence would give two parse trees, namely ”put (the white ball *that is* in a box) on (the floor)” and ”put (the white ball) in (a box *that is* on the floor)”. The corresponding PDDL would be (ONTOP e floor) and (INSIDE e (ONTOP l floor))[1]. There are however more interpretations of the sentence which were not included in the original grammar. The most important of these is the interpretation ”put (the white ball) in (a box *that should be* on the floor)”. In this interpretation, if there were no boxes on the floor to begin with, the planner could modify the state of the world so that there is one, and then put the ball in that box. That is, in Simple world, (INSIDE e (ONTOP k floor)) would also be a valid interpretation. The grammar was extended to account for this interpretation since the goals produced by it are in general more challenging and interesting than the goals produced by the original two interpretations. Hence, the third PDDL goal produced in Simpleworld would be (OR (INSIDE e (ONTOP l floor)) (INSIDE e (ONTOP k floor))).
All three goals mentioned would lead to different solutions in the planner. For (ONTOP e floor), the

[1] Here (INSIDE e (ONTOP l floor)) is a more compact way of describing (AND (INSIDE e l) (ONTOP l floor))

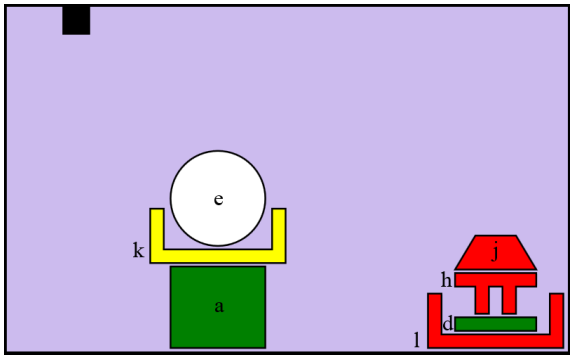

Figure 1: Simple world

planner would simply put the ball on the floor. For (INSIDE e (ONTOP l floor)), the planner would first remove all objects above the red box, and then put the ball in the red box. Finally, for (OR (INSIDE e (ONTOP l floor)) (INSIDE e (ONTOP k floor))), the planner would first put the ball on the floor, then put the yellow box on the floor, and finally put the ball in the yellow box. In the last case, the mentioned solution would be preferable to putting the ball in the red box since it requires less moves to do so. The resulting states from each goal is illustrated in figures 2, 3 and 4.

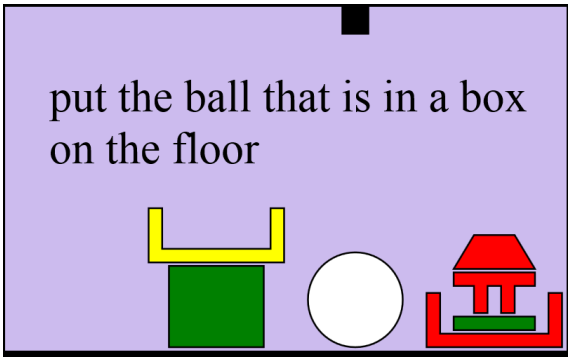

Figure 2: (ONTOP e floor)

In the grammar, this interpretation was added by enabling the user to specify the desired tense. When using ”that should be”, future tense is used, and when using ”that is”, present tense is used.

2.2 Assumptions about what an action is

Possible actions in the world are ”take” something from a stack, or ”put” something on a stack. The movement of the arm is not modelled, and is not considered an action. Neither the distance between stacks or the vertical travel distance of the arm is modelled. This implies that

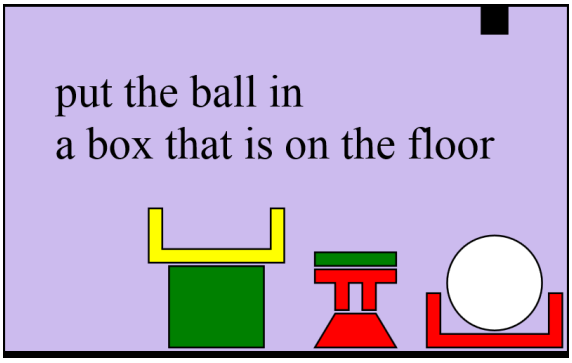

Figure 3: (INSIDE e (ONTOP l floor))

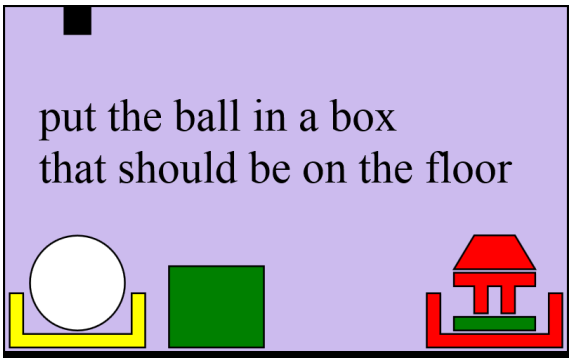

Figure 4: (OR (INSIDE e (ONTOP l floor)) (INSIDE e (ONTOP k floor)))

moving some object from some stack to another is always considered exactly two discrete actions.

3 Interpreter

The grammar parser outputs one or more parse trees to the Interpreter. These are recursively traversed, and checked against the world with a filtering process. If a parse tree uses present tense in some part of a sentence, the description in the parse tree is matched with objects currently present in the world. If future tense is encountered however, the description is matched with all objects which could be arranged to match the description in the world in the future. A logical expression of constraints (a PDDL goal) is then generated that is formulated in terms of id’s of objects in the world.
There is a contract between the interpreter and the planner which states that the PDDL goals need to be in a certain format. The planner solves all goals as long as they have valid PDDL syntax, but if the format is as expected, the planner will work at its maximum efficiency. This contract states that all goals must:

- Constitute valid logic with all the physical constraints of the world respected
- Be in either CNF or DNF formats

There is an example in the attached code which illustrates what happens when the contract is violated (the test is named ”testMoveObjectImpossibleGoal”).

3.1 Logic

The logic in the goals are generated depending on the quantifiers used in the users utterance. In most cases, the ”ALL” quantifier is converted to an AND expression and the ”AN” quantifier is converted to an OR expression, and so on. Complicated utterances can lead to logically complex goals. The interpreter does however always simplify these goals to a conjunctive normal form (CNF) or a disjunctive normal form (DNF). Also, redundant logic is removed. Since the planner is more efficient in dealing with DNF’s, the interpreter will produce DNF’s if a goal in question can be practically represented as a DNF. However, conversion between CNF and DNF is an NP-hard problem since (informally) a DNF equivalent to a CNF is basically an enumeration of all combinations of propositions from the clauses in the CNF which satisfies the CNF expression. Furthermore, we know that the SAT-N problem is NP-complete. From this, we know that all expressions cannot be practically representable as DNF’s.

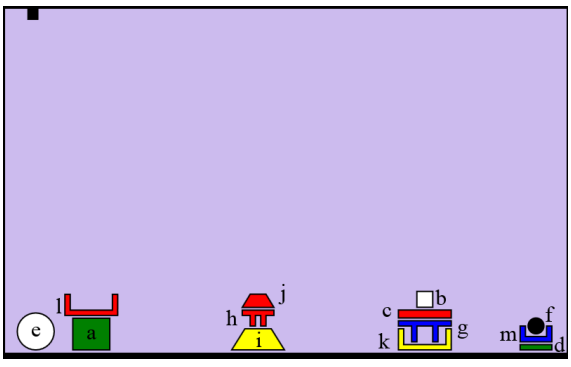

Figure 5: Medium world

Consider, for example, the following user utterance in the standard medium world (see Figure 5): ”put all small objects above an object”. Since, in the medium world, there are six small objects and 13 objects in total, representing this goal as DNF would require roughly $12^6 = 2,985,984$ clauses. This is a result of having to enumerate all possible ways of placing all small objects above an object. Representing this goal as a CNF however, only requires six clauses, each with 12 atoms. Not only would the conversion be time consuming in some cases, but the efficiency benefits of DNF in the planner would quickly be lost because of the extra clauses the planner would have to evaluate. In these cases, the goal is simply left as a CNF for the planner to handle.
Clauses which are impossible to fulfil in the world are also removed in most cases. For example, goals such as (ABOVE a a) will be removed. The intepreter also has some more advanced validity checks such as detecting false logical combinations. One example of such a case is the goal (AND (a ONTOP b) (b ONTOP a)). Another is (AND (INSIDE a b) (INSIDE c b)). These clauses would also be removed. The cleaned-up format is preferable for the planner since the planner assumes the goal state is valid. That is, without this feature, the planner would spend a lot of time trying to solve impossible goals or clauses.

4 Disambiguation

A user command can be ambiguous in two different respects:

1. The utterance results in more than one valid parse-tree (as explained in Section 2).
2. A reference using the quantifier THE matches more than one object in the world.

The ambition of the disambiguator is to make assumptions when it would be intuitive to do so, and otherwise ask clarification questions until an unambiguous command is understood. In addition, the clarification questions should be as specific as possible, listing all possible alternatives in a natural way with no redundant information. The user should be able to answer the question directly, without having to rephrase the entire command. The diambiguator does not work on pure PDDL goals, rather it is integrated in the interpreter. A goal is generated once the disambiguation is complete.

4.1 Multiple grammatical interpretations

If a command generates multiple parse trees, some of them may be ignored if there is no valid

interpretation of them in the world. There may also be a valid interpretation, but one that is ambiguous in the second sense described below. If there is exactly one valid, unambiguous interpretation it is assumed to be the one intended by the user.

If there is more than one valid, unambiguous parse trees a clarification question is sent to the user, where all the possible interpretations are listed, even the invalid or the ambiguous ones. The different possible interpretations are a result of the assumptions that was made regarding the meaning of user commands, that was described in Section 2. Consider once again the example from Section 2 in Figure 1. It would result in the following output:

-Put the ball in a box on the floor.

-Do you want me to move the white ball inside any box that is on the floor, move the white ball inside any box that should be on the floor or move the white ball that is inside any box on the floor?

After this question is asked the user will have to rephrase the command, which is fairly easy since the GUI keeps the phrase in the textbox after submitting it.

4.2 Ambiguous references using ”THE”

Under our interpretation, using the word THE when referring to an object, implies that there is exactly one unique object in the world that matches the description used. This means that an ambiguity occurs if the reference matches more than one object. This scenario is handled using clarification questions, but in a somewhat different way than above. The user is asked to clarify what object was intended with the reference. There is no need to rephrase the entire command, the user simply answers thew question in a natural way. The answer is then parsed in the grammar, and a parse tree is generated which is then injected into the parse tree of the original command. The evaluation continues, and hopefully the reference matches a single object. If not, another question is generated, but a more specific one based on the previous answer. The result is a dialogue with the user where all answers combined are used to identify a unique object. Consider the following examples in the Medium world.

”What can I do for you today?”

”take the thing”

”I can see thirteen objects. Did you mean the small ball, the large ball, the small box, the red box, the yellow box, the large brick, the small brick, the small plank, the large plank, the small pyramid, the large pyramid, the large table or the small table?”

”the large one”

”Yes, but did you mean the ball, the red box, the yellow box, the brick, the plank, the pyramid or the table?”

”the table”

”Ok.” [takes the large table]

——

”put the ball in the box”

”I can see two balls. Did you mean the small ball or the large ball?”

”the large one”

”I can see three boxes. Did you mean the small box, the red box or the yellow box?”

”the red one”

”Ok.”

5 Planner

For the planner the initial strategy was to focus on implementing a simple algorithm *well*, but with a very good (possibly complex) heuristic. This philosophy was very successful, as the planner effectively solves a wide range of complicated goals. The simplicity of the algorithm left more time for rigorous testing, which in turn enabled the authors to make the planner robust for many obscure corner cases. For examples on these, please see the attached test files.

Given a goal from the interpreter, the planner will try to find a minimal length sequence of actions that makes the requirements of the goal true. The planner implements the A^*-algorithm [1]. The algorithm sorts the states according to the cost function $c(s) = v(s) + kh(s)$, where $v(s)$ is the real distance from the start state to the state s and $h(s)$ is a lower bound estimation of the distance from the current state s to the goal state. For $k > 1$ the algorithm works as a k-approximation algorithm where the sacrifice of optimality is often rewarded by finding the solution in less iterations [1]. This is exploited by the planner when the state space is large and an optimal solution cannot be found in reasonable time.

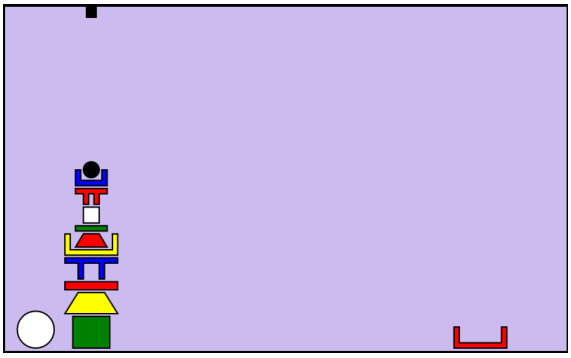

Figure 6: A stack of objects in the Medium world

5.1 Heuristics

The heuristic function used by the algorithm derives a lower bound of the number of actions required to reach a goal state. Let's assume that the goal is (ONTOP *a b*) which means that the planner has to move *a* immediately ontop of *b*. To do so the planner must first clear all objects lying on top of *a* in order to move it. It also has to remove all objects on top of *b* in order to place *a* on top of *b*. If five objects are placed above *a* and three objects are placed above *b*, where *a* and and *b* are placed in different stacks, it means that it must first move at least eight objects and then move *a* on top of *b* resulting in at least 18 actions. If *a* and *b* are in the same stack they might share objects that has to be moved. These objects are only counted for once. In this way the heuristic function estimates the minimum number of actions for more complex and nested goals.

Move at least twice. In some cases it is possible to determine whether an object needs to be moved at least two times. For example, with the goal (ONTOP j d) in Simple world (Figure 1), the pyramid (j) would need to be removed from the pile first in order to remove the table (h). Then, it would need to be moved a second time when moved back ontop of the plank (d). The heuristic function determines this in all cases possible, which significantly improves the performance of the A-star algorithm. To illustrate the necessity of this function, consider the situation depicted in Figure 6.
Let's say that we want the red box to be under the whole stack. The heuristic without the move-at-least-twice heuristic would evaluate to 22 in this case (11 move actions), since the planner would know that we need to move all objects above the green brick in order to move the green brick inside the red box. However, after removing some objects from the stack in order to restack the pile, the heuristic would still be stuck at 22. After moving the black ball on the floor for example, the planner would know that the black ball still needs to be moved at least once in order to put it back in the blue box. Since the heuristic value does not change in this case, there is no way for the planner to know it is making progress when moving objects off the stack to the floor. With the move-at-least-twice heuristic however, the initial heuristic would be $10 * 4 + 2 = 42$. In this case, after moving the black ball on the floor, the heuristic would decrease with 2 since the black ball no longer needs to be moved at least twice.

CNF and DNF expressions. Recall from section 3.1 that the goals are either in CNF or DNF. The heuristic of a logical goal given in disjunctive normal form is calculated as the minimum of the heuristics of all separate clauses in the formula. This is often very good but it can become a problem if the clause with minimal heuristic is not reachable in the world. An improvement to this would be to take on a probabilistic approach where the planner selects a clause proportional to how good its heuristic is. In this way the A^*-algorithm would move towards very good states with high probability but not get stuck trying to reach unreachable states.

The heuristic of a logical goal given in conjunctive normal form is more complex to calculate. Calculating an accurate lower bound similar to the one for DNF would require all combinations of atoms in the clauses to be enumerated and tested. As explained in Section 3.1, this is infeasible in some cases. Instead, a max-min argument is used in combination with accounting for objects which invariably needs to be moved in each clause. The idea behind the algorithm used is to make the heuristic as high as possible without violating the lower bound guarantee of the A-star algorithm. Consider for example the DNF expression (AND (OR (ABOVE a b) (ABOVE a c) (ABOVE a d)) (OR (ABOVE b c) (ABOVE b d)) (ABOVE c d)). Let c_1 denote the first clause (OR (ABOVE a b) (ABOVE a c) (ABOVE a d)), let c_2 denote the second clause (OR (ABOVE b c)

(ABOVE b d)), and let c_3 denote the third clause (ABOVE c d). The algorithm first checks which objects need to be moved at least once or twice to satisfy each of the clauses. These can be used for the heuristic since they provide sets which are guaranteed to be moved to reach the goal. For c_3, this is easily done, since there is only one option to check. For c_1 and c_2 however, the objects returned needs to be present in all atoms of the clause in order to not violate the lower bound. That is, for c_2, the algorithm will return the objects which need to be moved to fulfil the atom (ABOVE b c) and the atom (ABOVE b d). Secondly, the algorithm looks through each clause c_i for the minimum number of objects needed to be moved for each of the atoms including the objects derived from step 1. Once determined, the algorithm finally chooses the maximum of these objects returned from each clause. If we denote each atom by $c_{i,j}$ (the j:th atom of clause i), this operation can be expressed more compactly as:

$$heuristic\_objects =$$

$$\max_i(\min_j(minObjsToFulfil(c_{i,j})$$

$$\cup(\cup_i(\cap_j minObjsToFulfil(c_{i,j})))) \quad (1)$$

where the function minObjsToFulfil returns sets of objects in the world which need to be moved at least once or twice in order to fulfil an atom. This heuristic is a lower bound, is quick to calculate, and is gives a heuristic almost as good as for DNF expressions most of the times.

Horizontal Planning For horizontal constraints, that is LEFTOF, RIGHTOF and to some extent BESIDE, the heuristic as described in the previous section has limited efficacy. This is because it is difficult to prove a guarantee for which objects need to be moved as a minimum to fulfil an LEFTOF/RIGHTOF/BESIDE atom. Recall our assumption that the planner is allowed to move any object to fulfill a constraint. This implies that if we have a relation (LEFTOF a b), we can move either a or b to meet the goal. If we then have multiple such goals nested together, for example when using the sort command, it is non-trivial to produce a good lower bound on the number of moves to reach the goal. An example of a nested goal is (AND (LEFTOF a b) (LEFTOF c b)). We can see that this is already more difficult to solve since the planner could either move (a left of b) and (c left of b), or simply move b right of both a and c. Or if, say, there was no room for b to the right of a and c, the planner could first move c and then move b right of a and c. For more complicated goals, there are even more corner cases to take into account.

To approach this, a heuristic was designed which is independent on exactly how the problem is solved. The idea is that given a set of horizontal relations, one can infer limits on how close to a wall a given object can be in a goal state. Say for example that we have $a < b < c < d$, where $<$ represents the LEFTOF relation. Then we know a priori that d cannot be in stacks 1, 2 or 3 in a goal state. Similarly we can infer limits on the distance to the right wall for each object. We can then precompute all possible stacks for each object, and use this information as an additional constraint in the heuristic.

To compute the possible stacks, a graph is created of all objects as nodes and their relations as edges. In this graph, a breath-first search finds the longest chain of horizontal relations from a node, and a minimum distance to the wall can be calculated.

The implementation of this heuristic does not take into account that objects may require multiple stacks, even if no relation says so. This occurs frequently when using the sort command. If we sort by color, the heuristic will compute that one stack is needed for all red objects based on the horizontal constraints. In reality, the red objects may require more stacks, and the heuristic will miss this, and the search risks to get stuck in local minima.

In addition to this heuristic, a max-min type heuristic was implemented for horizontal planning in a similar fashion to what was described for CNF expressions. The combination proved to be fairly efficient in solving goals generated by the sort and the stack grammars. For large sort problems (see the attached test file named ”testSortByColorLarge”), there is still room for improvement. Some additional heuristics were considered (there are many options), but were ultimately left as possible future extensions.

6 Stack

A ”stack” command was added to the grammar. The user can ”stack”, ”stack up” or ”build a tower of” arbitrary objects. This is implemented as an addition to the original grammar, as well as an addition to the interpreter. The command generates one possible stack and outputs a single MOVE command to the planner to solve, where each object is ONTOP of another. This implies that the Stack command generally does not produce the optimal stack, with respect to the number of steps required to solve it. Another possible approach would be to generate all possible stacks and send them all to the planner as a disjunctive goal. However, there are $O(n!)$ different possible stacks, so even enumerating all the stacks is prohibitive for non-trivial instances, not to mention solving them. Yet another possibility would be to define the stack in terms of ABOVE and BELOW relations. This would allow a more compact representation, with $O(n^2)$ clause, Where each object is either BELOW or ABOVE all other objects in the stack. However, it would also not guarantee that no other objects appeared in the stack. It was deemed unlikely that the planner would optimally solve such instances, so this approach was not explored further.

6.1 Stack algorithm

A greedy algorithm finds a legal stack of n objects in $O(n^2)$ time, if such a stack order exists. This algorithm exploits the particular physical rules that are in place, and would not work for arbitrary stacking problems. First all objects except balls and boxes are placed in the stack, sorted from the floor first by size, then in the order pyramid, plank brick, table. Then, all boxes are placed one by one inside the stack, as high up as possible in the stack. Finally, if there is a ball, it is placed in the box at the top of the stack. Only one ball can exist in a stack. If either the ball or some box cannot be placed, there is no possible way to stack the objects.

7 Sort

A ”sort” command was added to the grammar. The user can sort any group of objects with a phrase like ”sort all objects by color”. Any group of objects that can be referred to can be sorted, and any attribute (color,size or shape) can be used for sorting. This command is implemented as a conjunction of LEFTOF and RIGHTOF constraints sent to the planner. There are $O(n^2)$ such constraints generated. Unlike the Stack command, the logical expression generated expresses all possible ways of sorting the objects, and one would expect the planner to find the optimal way of performing the sort. Presently however, the planner does not handle large instances of sort due to limitations in the heuristic function for LEFTOF and RIGHTOF.

8 Multiple arms

A feature was added that allows the use of any number of arms when performing the planning. For each time unit, each arm performs one of the actions *pick/drop/move* where moving to the current position is considering not doing anything at all. The arms may not cross each others path which means that one or several arms might have to move in order for another arm to pick or drop something in a given stack. This gave rise to a more interesting planning problem and implementation challenge. However, the A^*-algorithm and heuristic described above could still be used exactly as it it. The only difference is the expansion of neighbouring states of a given state.

The neighbouring states of a given state are the states where each arm pick, drop or move in/to each of its possible columns. If there are n arms and m columns each arm has $m - n + 1$ columns to work in. Since an arm can only pick or drop, depending on if it already holds something, each arm has $2(m - n - 1)$ possible actions each turn and there are n arms resulting in a total of $(2(m - n + 1))^n$ neighbouring states. In contrast to using only one arm where each state have a maximum of m neighbouring states. This becomes a trade-off since when using more arms the number of neighbouring states that have to be considered is exponential in the number of arms but each state moves faster towards the goal. When using fewer arms a linear number of neighbouring states have to be considered but each state moves a lot slower towards the goal. In fact it is easy to understand that the algorithm explores the minimum number of states when using only one arm. This is due to the fact that using n arms each state moves a factor n towards the goal but explores an exponential number of

neighbouring stats all the time, while using one arm moves a factor 1 towards the goal but only explores a linear number of neighbouring states. However, the problem with multiple arms is interesting in real world problems with for example multiple overhead cranes which cannot cross each other.
In the medium world the utterance ”move the large ball inside a yellow box that should be on the floor” was tested with one, two and three arms. The plans for the different tests can be seen in Table 1. Using only one arm the plan needs eight actions in eight time steps to complete the task. Using two arms the plan consists of ten, whereof two move, actions over five time steps. Finally, using three arms need only four time steps but uses twelve, whereof two move, actions. It is also interesting to look at how many states that were considered by the A^*-algorithm for the different examples. One arm only checks 15 states if it is a goal state and calculates the heuristic for a total of 82 states. The respective numbers for two arms are 16 and 1211 and the respective numbers for three arms are 269 and 52107. The exponential growth in the number of neighbouring states is clearly visible in this simple example.

9 Additional examples

An example which works well and illustrates much of what has been implemented is ”put all blue objects on a red object that should be on a green object that is on the floor” in the Medium world. The resulting PDDL goal is (OR (AND (ONTOP g (ONTOP c a)) (ONTOP m (ONTOP h d))) (AND (ONTOP g (ONTOP c a)) (ONTOP m (ONTOP h a)))) which is a DNF goal. The result of the operation is shown in Figure 7. Note that the green objects must already be on the floor for this operation to work. For a list of examples (some good, some bad), please refer to the test-files in the attached code. The reason as to why the planner is good or bad at handling these should be apparent from the report or from the comments in the code.

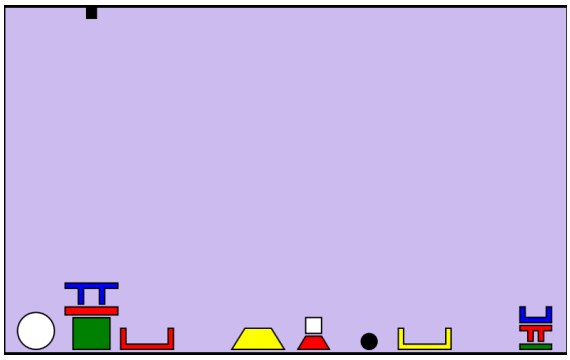

Figure 7: Result of ”put all blue objects on a red object that should be on a green object that is on the floor”

arm0
pick0 7
drop0 1
pick0 7
drop0 2
pick0 7
drop0 3
pick0 0
drop0 7

(a)

arm0	arm1
move0 0	pick1 7
pick0 7	drop1 8
drop0 3	pick1 7
pick0 0	drop1 2
drop0 7	move1 8

(b)

arm0	arm1	arm2
pick0 1	pick1 7	move2 8
drop0 5	drop1 6	pick2 7
pick0 0	pick1 7	drop2 8
drop0 7	drop1 8	move2 9

(c)

Table 1: Showing the plan for the utterance ”move the large ball inside a yellow box that should be on the floor” in the medium world for (a) one arm (b) two arms and (c) three arms.

References

[1] A^* search algorithm, Wikipedia. Available: *http://en.wikipedia.org/wiki/A*_search_algorithm* (2014-05-18).

10 Individual contributions

10.1 Roland Hellström Keyte

- Handling of quantifiers
- The logic package, including
  - Conversion of expressions to/between CNF and DNF
  - Simplifying logical expressions
- Grammar for present and future tense
- The interpreter and conversion of parse trees to PDDL goals (filtering idea with Sebastian, visitor pattern by John)
- Implementation of the tests in the class ”InterpreterTest” and fixes of related bugs
- Some initial theory behind the disambiguity resolution, and some initial implementation (The rest done by Joel and John)
- Theory and implementation of the heuristic function, including
  - Handling of CNF and DNF forms
  - moveAtLeastOnce and moveAtLeastTwice heuristics
  - Implementation of the valid-stacks heuristic for leftof/rightof (theory together with Joel)
  - Extension of Sebastians leftof/rightof heuristic
  - Inference of indirect relations
- Implementation of physical laws and spatial relations
- The extension ”Handling of large and complicated goals”
- Profiling and improvement of the code
- The ”world” package

The main contributions I made to the report were on sections 1, 2.1, 3, 5.1, and 9.

10.2 Sebastian Ånerud

The list of theoretical contributions to the project include:

- the idea behind using the A^* algorithm along with the basic idea behind the current heuristic (heuristic mostly implemented by Roland who also extended the idea further on his own).
- the idea of how to represent the state space and how to move between states of the world.
- the idea behind matching objects in parse tree to objects in the world using a filtering technique (implemented by Roland and co-invented with Roland).

The list of practical contributions to the project include:

- The implementation of the A^* algorithm.
- The implementation of the data structures used to represent a state in the algorithm.
- The implementation of the expand method (method for enumerating all neighbouring states) together with John.
- The implementation of multiple arms together with John.

10.3 John Johansson

The list with theoretical contributions to the project include:

- the idea of how to represent the world in an object oriented way
- the idea using N-arms as extension
- the idea behind communication between server and client including saving states.

The list with practical contributions to the project include:

- The implementation of all extension in the web interface, such as more arms, questions and buttons.
- The implementation of the representation of the world (the tree package).
- The implementation of all communication between server and client.
- The parsing from the parsing tree to data nodes.
- The implementation of the expand method (method for enumerating all neighbouring states).
- The implementation of multiple arms together with Sebastian.

10.4 Joel Ödlund

- Planner, general contributor
  - general problem solving
  - idea to use logical normal forms
  - horizontal planning heuristic (idea, not code)
- Disambiguation, main contributor
  - Overall disambiguation scheme
  - Implementation inside the Interpreter
  - Recreate state from a history of questions
  - Parsing of answers in grammar
- Natural language generation, main contributor
  - Description of objects and commands
  - Meaningful questions and error messages
- Stack, sole contributor
  - Grammar and interpreter extension
  - Algorithm to find legal stack
  - Theoretical considerations
- Sort, sole contributor
  - Grammar and interpreter extension
  - Theoretical considerations
- Report
  - All sections regarding my own contributions
  - Various contributions elsewhere