Artificial Intelligence in the Computer Game Astro Kid

Kasper Reindahl Rasmussen

Kongens Lyngby 2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

The goal of the thesis is to ...

Summary (Danish)

Målet for denne afhandling er at at gøre det muligt at løse problemer ved hjælp af en gennerelle panlæggere i spillet Astro Kid. For at opnå dette analyseres Astro kid domænet og hvordan det kan repræsenteres.Der er også set på læring i domænet. Læringen er ikke udelukket begrænset til Astro Kid men tager udgangs punkt i det. Til dette formål udvider afhandlingen arbejdet i [WL08] og [JJ15], ved Astro Kid....

Fremgangsmåden for læringen er baseret på en logisk sammenhæng mellem de forskellige stadier af universet.

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring an M.Sc. in Engineering.

The thesis deals with interacting and solving problems in the Astro Kid Game using a general purpose planner. It mainly extends the work of [WL08] and [JJ15].

The thesis consists of ...

provide a proposal for general leaning algorithm

Lyngby, 03-January-2016

Kasper Rasmussen

Kasper Reindahl Rasmussen

Acknowledgements

I would like to thank my supervisor on the project Thomas Bolander, and I would like to thank my friends that have assisted me in proof reading.

Contents

Chapter 1

Introduction

Planning is an integrated part of most peoples daily life and appears in everything around us from scheduling to route finding. Basically to complete nearly any task a planning is need if one wishes to do so efficiently.

Planning is in it simplest form, having a problem and finding a series of actions that solves the problem (a plan). To easily find plans and ensure the plans actually are efficient planners can be used to automate process.

Problems comes in many types and shapes, so to simplify the process of automation, planners only works with a subset of the problem's types.

The particular subset of problems for Astro Kid can mostly be categorized as classical planning. This type of planning is discrete and works with an atomic state space (see [RN14] for more on classical planning).

There are two general ways of automating these processes, which is either create a new planner for the particular problem or to use a general purpose planner. The general purpose planner has the great advantage that a lot of time and effort can be saved by using already an existing planner, however this often comes at the cost of poorer performance. To be able to use a planner at all, the problem needs to be described in a way the planner can understand. This is trivially done for a purpose build planner since the description can be done to excactly to fit the needs of the planner. For general planners however the problem needs to be described in general way that can describe a wider span of different types of problems (a planning language). Such a language consists of a way to describe the initial state, goal state and the possible actions available to manipulate the current state. The most common way of describing this is using the planning language PDDL.

In the ideal world it would be possible to throw a generic planner at any type of problem and having it solve it without human intervention. Obviously we are not there yet, and one of the many obstacles on the path, is that the planner needs to know how the domain works before it can solve the problem. In order to mitigate this, the concept of learning is introduced. By allowing the planner to learn from interacting with the domain, it becomes more self-sufficient.

Learning in itself can be different things: It can be learning something new or learning to do some thing better. For instance, the focus of the learning tracks at IPC, isn't to learn something new, but instead optimizing what the planners already can do. The focus here is instead on learning something new.

The project concerns Artificial intelligence and how it can be used interact with flash game Astro Kid[1]. This game is a puzzle game, where an avatar has to move to a designated goal field. In this world the avatar can walk, climb ladders, push objects, and use a remote control for stating robots.

The Astro Kid world varies greatly in complexity depending on the size and number of objects in it. In general it has a slowly growing complexity as it progress trough the levels (due to more and new types of objects being added). One of the challenges with the Astro Kid world is that several actions, have consequences that have a continuous effect over several time steps, and how to represent these effects and interact with them. At the moment no AI exist that can interact with this world.

The main goal of this project is to create an agent that can interact and solve problems in Astro Kid world effectively, without prior or only little initial knowledge. The agent should be able to work with a wider array of problems than solely Astro Kid. To achieve this there will first be looked at how far one can get with a generic planner. This means that there also have to be looked at how and how much of the domain can be described in a planning language. Secondly there will be looked at learning how to interact with the environment and the consequences of these interactions. Keeping the knowledge, as it progresses through the levels, is essential to avoid relearning the same things. Hence, it should be able to abstract away from the particular situation and learn in more

[1] http://www.agame.com/game/astro-kid

general terms. The goal of the learning is not to learn the complete domain, but just learn enough to solve the given problem.

1.1 Related Works

The project revolves around planning and learning. The two main areas in planning is effect over time, and concurrency.

Classic planning normally operates with actions having no duration. This works for many types of problems, such as blocks world, and sokoban. However, in reality things happens over time. The concept of actions having a duration in planning isn't therefore anything new. The concept of temporal planning was introduced in PDDL 2.1 with durative actions. One of the first planners handling durative actions in PDDL was [GFL02]. This concept of durative actions has the notion of time, but it is directed towards scheduling and problems in general, where the end state of an action can be found from the start.

Concurrency in planning isn't something new, either. Various adaptations has been suggest to integrate in a planning system, one of such adaptation is MAPL which builds on multi agent planning[Bre03]. The main attempts of using concurrency in PDDL is based around multi agent systems. The current problem differs from this since it is a single agent system even though concurrency exists.

The concept of learning actions schemas has be looked at multiple times beforeTODO..... example of different approaches.

The concept of learning can involve a vast state space, and therefore it can be a slow process.

A way of speeding up the learning is described in [WL08] with the concept of learning from a teacher. The idea is to only do things that is known to work and thereby not considering a vast number of possibilities. This approach, however, introduces some limitations such as only allowing positive goals and needing a "teacher" (not self-sufficient). This concept was further extended in [JJ15], where the space requirement was reduced and allowing negative preconditions.

A different approach to learning can be seen in [ZYHL10] which is based around Markov Logic Network and general probability. This approach gives the possibility of learning more complex actions using features such as quantifiers. Unfortunately it does have some limitations, since it does not deal in certainties.

1.2 PDDL Background

The Planning Domain Definition (PDDL) Language is a planning language, created in 1998 by Drew McDermott and the AIPS-98 Planning Competition Committee. It was created to make the international planning competition possible and to breach gab between research and application. The development of PDDL is driven around each IPC, and the development of planners is often also effected by this. PDDL is an action based language, based on the language STRIPS with some advanced extensions from ADL (Action description language) such as the use of quantifiers. It provides a deterministic, single agent, discrete, and fully observable planning environment. PDDL also works under the closed world assumption, which is that anything not described is false. The current version of PDDL is 3.1.

PDDL is separated into two parts: The problem, and the domain description. The problem description, is the description of a problem instance, which consist of the state of the world and goal condition. The domain description is the rule set of the world, consisting of the actions[2] which is allowed to be performed on the world, and what type of objects is allowed to exist in the world. An example of this can be seen in Appendix A.1 using the classic blocks domain. This separation means that a single domain description can be paired to many problems, but not necessarily the other way around. For more details on PDDL see **??**.

[2] An action consists of Parameters, preconditions, and effects

Chapter 2

The Astro Kid World

The Astro Kid game is a discrete world where everything moves in a 2D grid, and the world is fully observable. It's a single agent system based around the actions, which can be started by controlling the avatar. These player actions are movement (up, down, left, right), push, and activate control. These are the actions which the player controls, and everything else are consequences of theses actions. This means that these actions are the only ones where the planner have a choice that can effect the final result.

Actions that do fall into this category of consequences, can be such as a robot walking forward when started, objects falling when push over an edge etc. The difference between the two groups is that the planner should not be able to stop an object from falling mid air, but it should be able to chose if the player moves left or right.

When looking at the Astro Kid domain, it is worth noting the main areas where it differs from classical planner domains like the Gripper. These areas are concurrent and continuous actions. Continuous actions are defined as actions that happens over several time steps. The continuous actions all relates to the movement of objects (falling, sliding and walking). An example of this is that falling from the top of the map to the bottom doesn't happen instantaneous, but in steps and it has to pass through the areas in between. As a result of this multiple objects can move at the same time, which leads to concurrent actions.

The number of simultaneous concurrent actions are limited, since effects of continuous actions can't increase the number of moving objects. This means that a maximum of one new action with continuous effect, can be started each step. An example of this can be seen on level 25 (B.1). This level also illustrates that even though one action can be started each turn, the total number is limited by the finite size of the map. In this case a maximum of eleven stones and the avatars can be moving at the same time.

Chapter 3

General Planning

The idea of using a general planner is that it is possible to solve the problem efficiently, without building a planner from scratch, for each problem. The general planer will however nearly often be less efficient than an planner for the specific problem. This is due to it not always being able recognize and utilize features that are specific to a given domain. The main question is therefore if the general planner can solve the problem efficiently enough to be useful.

To begin solving the problem with a general planner, the domain needs to be described in a planning language. In this case PDDL is suitable choice since it is a expressive language and used for the IPC, and there therefore exists a series of planners that support this language.

The general planner there will be used is Fast Downward[1]. Fast Downward was chosen due to it being a widely know open source planner and it has done well in several IPCs, eg. won the Sequential Satisficing and Optimization parts of IPC 2011[2]. It must therefore be considered one of the top planers out there, even though it only support a subset (PDDL 2.2 level 1 + action cost) of the complete PDDL.

[1] http://www.fast-downward.org/

[2] http://www.plg.inf.uc3m.es/ipc2011-deterministic/Results.html

3.1 PDDL

The chosen planner is Fast Downward, which is limited to PDDL 2.2 level 1 and the action cost feature, therefore this subset of PDDL will be used.

Due to how the domain differentiate from the type of problems usually found in classical planning. There are used two different approaches to describe the domain in PDDL. The domain can either be relaxed into a more classical planning domain or use the structure as is, and work around the problems that occur due to it.

Relaxation is based on simplifying the problem by removing or loosen some constraints. This makes it easier to find solutions, but it will not necessarily be valid ones. Invalid solutions can sometimes can be transformed to valid ones using domain specific knowledge.

3.1.1 Problem Representation

There are several approaches to the problem, but the basic structure of the problem remains the same. The differences between the approaches lie in the actions available and the ordering of these.

This is possible due to PDDLs separation of Problem and Domain description. The main part of the problem description is the location of the different objects in the domain. Theses locations are by representing each point in the world's 2D grid as an object and defining its location relative to its neighbours. All other objects locations in the domain are then defined relative to these locations.

```
(relativ-dir pos-01-01 pos-02-01 right)
(at stone01 pos-01-01)
```

Even though the domain description changes, how the metric is used remains the same for both domains. The planner can either be optimised towards plan length or cost. This is useful since the found plan by the planner consist of both player action and consequences actions. The plan length will therefore differ from the number of player actions. Since the quality of solution is solely based on player action, the consequence action needs to be filtered out when the planner finds a solution. this can be done by adding a cost to the player actions while every other action has cost zero, and let the planner optimize by cost instead of the total length of the plan.

3.1.2 Classic Approach

What will further on be described as the "Classic Approach" is built around getting the domain to be more similar to that of a classical planning problem. This approach is centered around avoiding an input update cycle, and by keeping the separation of course and effect as short as possible. The main advantage of doing this, is that general planners are optimized for this kind of problem, since that is what they usually handle at the IPC.

The main problem area with the classical approach is that PDDL doesn't directly allow concurrent actions, unless they are changed/merged into a single action. This could be done by having an action for each possible combination of objects moving, and thereby describing every possible interaction between the objects. This approach would however leads to an explosion of possible actions, since the number of actions would then be exponential depend on the number of objects that can have a continuous effect. Another problem with having the number of actions dependent on the number of objects, is that domain would then either have to be limited to a certain number objects or change depending on the number of objects in the problem description. This would be against how PDDL is intended to be used.

To avoid this problem, the domain can be relaxed by making some assumptions, so it would fit better into a classic planning paradigm. The assumption is that moving objects don't interact with each other. This means that concurrency can be avoided, by letting one of the concurrent actions completely terminate, before the next is considered. To avoid the avatar from standing still unnecessary when executing the plan, only agent actions is considered as part of the solution (when executing the plan the agent doesn't wait for concurrent actions to stop).

This assumption makes the states in-between the continuous actions' start and end states irrelevant for the solution. In the ideal world it would therefore be possible to apply a single of the actions with continuous effect and treat it as a normal action due to the only interesting state being the last of the continuous action. However even with this assumption it isn't feasible to calculate which state a given continuous actions ends in, due to it depending on all the states in between (worst case everything in the domain). It could be done by using PRC or recursion but neither is supported by PDDL.

The assumption holds in most cases due to objects generally moves away from the avatar (actions such as fall, slide, walk, and push). Which also means that any subsequent action won't interfere with others, due to objects moving at same speed. Teleporters, remote controls, and gates can in a few cases interfere with this. When this assumption holds, it would be possible to find the optimal solution.

Applying the assumption however means that minor discrepancies between the real world and the PDDL world can occur. The differences can in some cases be ignored since, what is important, is that the avatars route is clear.

If the discrepancies are too great to be ignored e.g. if problems occurs between multiple moving objects (eg. level 31 (B.2) where the multiple objects needs to move concurrent with the correct timing).

The assumption can in most cases be ensured by inserting noOps (no operation, do nothing) into the plan after a solution is found and thereby ensuring only one thing moves at the time as the assumption requires.

This however results in a not necessarily the optimal solution, and there will still be a few instances where two moving objects needs to interact (e.g. hit each other) to solve the problem, if this is the case the problem cant be solved, due to the limitations of the assumption.

Using this approach means that each action needs a rather large set of preconditions, this is to ensure that not only is the preconditions for the given action fulfilled, but also that no other action with continuous effect is unfinished.

The main precondition for most of the action are therefore a condition that ensures that nothing else is moving (continuous effect). In general several different things is handled in each action, and they are in most cases the same for several other actions.

There are a few cases where this isn't the case, since they are not restricted by other actions. In the case of gates opening and closing, they are handled by a derived predicate (axioms) instead. Further more to simplify the precondition and effects of the actions, part of the functionality are extracted into their own optional actions. This the case for picking up items and teleportation. They are in fact not an optional action, but due to their locations works well as optionals. And in the case of teleportation it is possible to treat it as optional action, using post processing since it is reversible.

3.1.3 Update Approach

Due to the problem with a few unsolvable levels in the classic approach, and the wish of obtaining good solutions without post processing. A different approach is needed. The main idea with this approach, is the idea of using the domain as it is, this is to avoid making assumption on the domain, which then can effect the quality of the found solutions. Basically it is to ensure that the PDDL version of the world is and react at all times the same as the real world. This approach therefore guarantees an optimal and valid solution is possible to find from the domain.

When looking at the domain there are two types of actions: The ones where the planner has a choice (player action), and actions which are consequences of earlier actions. The main idea is therefore to separate the domain into choice and update, and then enforcing an ordering of the actions so a choice is always followed by an update. The ordering of actions can be enforced by using flags. This ordering makes it possible for the action to be limited in purpose and therefore to have fewer and simpler preconditions, and thereby making it more readable.

The update step is in itself split up into multiple parts with a strict ordering, to enforce that the concurrent actions interact correctly with each others. The update step is separated into: removing objects, changing direction of objects, moving objects, collecting items, opening/closing gates, teleportation. To ensure all the continuous effects are applied on the relevant objects universal quantifiers are used.

A problem with this approach is how the strict ordering of actions (choice update cycle), differentiates from the approach usually used in general planning. This means the direct link between action and effect is not as clear as in classic PDDL. This is due to the update sequence delaying when effects of an action later applied.

3.2 Tests

There are various ways of tweaking the performance of the planner. The two most obvious ways of doing this, is changing the parameters of the planner and changing the PDDL domain to fit the planners strengths better. The latter part is where the update and classic approach comes in.

3.2.1 Plan Quality

Comparing the two main approach and looking at results in table 3.1, it can be seen that on problems where the assumption holds the found solutions are equal.

Table 3.1: Plan Quality

	level 4	level 7	level 25small	problem 24	problem 24v2
classic	35	45	x	11	11
update	35	45	32	7	11

Length of the found plans

Some difference can be seen where the post processing is necessary (problem 24 and 24v2 where the robot is activated to late). The post processing can in most cases be optimised to perform close or equal to the update approach. However to achieve the best possible result from the post processing domain specific knowledge would be need. A greater difference is seen on levels such as level 25, where the assumptions used in the classic approach fails completely (moving objects needs to hit each other). The number of levels where this is the case is fairly limited. As one would expect it can be seen that domain only effect the quality of the solution where the assumptions made does not hold.

3.2.2 Speed

When looking at the data in table 3.11 its worth taking note of that most of the time is not always spend in search, as one would expect for a planner, but instead in preprocessing and translating.

This is interesting since it tells something about what the planner have problems with. that the problem lies in the representation of the problem and not the complexity of it. this is supported by the problem being a simple one which also is reflected in the search time.

A significant speed advantage to the classic approach can be seen on all problems it can solve. The difference in speed was expected, due to the domain type being relaxed and more similar to the ones used in the IPC. One would therefore expect the planner to be more optimized for this kind of problem.

What wasn't expected was how large the difference. On a problem such as level 4 is the difference roughly a factor of 20, it is however worth taking note of that the speed difference isnt equal for all the sub parts of the search. It is in particular the translator which is the bottle neck.

The great speed difference makes the update approach less useful. For the it to be able to solve more complex levels this speed difference needs to be reduced. Test have shown that the use of universal quantifiers have a large impact on the translating/instantiation time. Therefore there is created a variation of the domain where a single universal quantifier is replaced by an existential quantifier and an extra action(s).(code can be found at Appendix A.2)

Table 3.2: problem 4

	forall	simple
total Time	26.2	136.2
translator Time	21.9	6.6
relevant atoms	8872	12968
auxiliary atoms	16959	18807
final queue length	25831	31775
total queue pushes	58904	68537
axioms	1	132652
peak memory	100568 KB	208356 KB
task size	42751	573269
preprocessing Time	4.2	127.4
necessary operators	7292	7337
search Time	0.1	2.2

When running the different versions of the domain on various problems, one thing becomes clear, the total times varies greatly depending on the domain and level combination (table 3.7). However when looking at the results (table 3.11), what shows is that the use of universal quantifiers greatly increases the instantiation/translation time.

Replacing the universal quantifiers ensures a quicker instantiation/translation, but it has the cost of the number of axioms exploding, and in general the number of atoms growing by 5-20%, which in the end can greatly hampers the preprocessing that generates the Causal graph.

The tendency seams to be that the more movable objects and complex the problem the better the version with universal quantifiers does, this is especially clear when moving from toy problems to actual levels (tabel 3.4 and 3.5). Generally the existential variation of the domain only really shines when the problem is "small" and isnt therefore useful on the different more complex levels in Astro Kid.

An anomaly is shown in prob4v2 table 3.3 (where a single stone is add to problem04) and not much difference would be expected, but here the planner figures out that the objects is in fact not movable, and the explosion of axioms dosnt happen.

Another way of tweaking the code is to optimize the Problem Definition, this

Table 3.3: problem 4v2

	Universal	Existential
total Time	9.8	5.2
translator Time	7.1	2.2
relevant atoms	6742	8199
auxiliary atoms	13900	14710
final queue length	20642	22909
total queue pushes	40653	44321
axioms	1	534
peak memory	71356 KB	66040 KB
task size	29496	31067
preprocessing Time	2.6	2.8
necessary operators	5469	5485
search Time	0.1	0.2

can be done by removing unreachable states and objects, more precise remove the representation of position that isn't useful. The effect of this can clearly be seen when adding unreachable location to prob02B.4 and the effect can be most clearly seen in table 3.6 for the update approach. The results shows for each variation the time needed roughly doubles. Interestingly enough the version of the domain with existential quantifiers isnt effected nearly as much by it (table 3.7). One version scales with the size of the problem (grounding), the other with the number of objects (axiom explosion). This shows that long running times does not necessary correlate to a more complex problem. It also shows the weakness of a general planner, which is that it cant use specific knowledge of the domain, and therefore cant easily discard non relevant areas of the problem. Even though the update approach is used to illustrate the point its not restricted to this approach. The classic approach isnt effected ass much but goes from 2.2s (prob02) to 5.5s (prob02v4) which is still an increase of a 100% without the complexity of the problem changing.

The simplest way to optimise using this is to trim the edges of the problems. Doing this dosnt require much knowledge, and can be done on nearly any kind of domain. An even better optimization can be achieved by actually analysing the possible paths in the problems, eg. most of the fields under the platforms are unreachable in B.4. The problem here is that to actually realise this for a given level is often more or less equivalent to actually solving it, this is especially the case for more complex and compact levels.

Table 3.4: level 4

	Universal	existential
total Time	175.4	x
translator Time	164.4	x
relevant atoms	27782	x
auxiliary atoms	40327	x
final queue length	68109	x
total queue pushes	201441	x
axioms	13	x
peak memory	231780 KB	x KB
task size	138081	x
preprocessing Time	10.0	x
necessary operators	24525	x
search Time	1.0	x

3.2.3 Heuritics

The main parameter in Fast Downward that can be tweaked is the heuristic. This choice of heuristic is fairly limited due most of the them supported by fast downward dosnt support the use of axioms or conditional effects, which are used in the domain description, and those heuristics that does, do so barely[3]. The only supported heuristic that is admissible is Blind, and as the name suggest isnt the most advanced of the heuristics.

The importance of the choice heuristic varies greatly depending on the level. and if speed, quality or a combination is wished for. For the update approach choice of heuristic is fairly unimportant since the bottleneck is at the translator and preprocessing. When looking at the classic approach this changes. The effect of the heuristic on the different levels varies greatly, eg. on level 4 the time consumed by the search(heuristic) is minor. when looking at the heuristics performance over a wider array of problems, some things appears.

The heuristics blind and context enhanced additive both does well on all the level except a single one each.

The additive heuristics consistently gives fast results but solutions of a poor quality.

The blind heuristic gives a high quality due to it being admissible and it is fast

[3] (in the sense that the planner won't complain – handling of axioms might be very stupid and even render the heuristic unsafe) `http://www.fast-downward.org/Doc/Heuristic`

Table 3.5: level 9

	classic	Universal	existential
total Time	11.5	301.7	x
translator Time	5.6	280.3	x
relevant atoms	8797	61431	x
auxiliary atoms	169254	64980	x
final queue length	178051	126411	x
total queue pushes	746595	359476	x
axioms	265	23	x
peak memory	195568 KB	386872 KB	x
task size	43197	265076	x
preprocessing Time	5.4	17.2	x
necessary operators	4283	49496	x
search Time	0.5	4.2	x

except on a single level where it is 60 time slower than the second to last.

The simplest possible explanation for why some heuristics does terrible on some levels, is that they cant handle axioms well and will therefore handle certain situations "stupid"[4]. This cant explain why Blind fail, due to i fully supporting all features of the domain. The reason for the failure instead lies in that its an uninformed search, which means that the no knowledge of the domain is used and is therefore prone to explore the state space less efficient. Therefore it is not unexpected that blind fails on a level, but more that it dosnt fails on other levels. That this does not happen, must be credited to the preprocessing of the problem.

This leaves the heuristics FF and Max which are not the fast but does not have any notable problems. with FF being the on average bit Faster than Max, and will therfore be used further on.

3.2.4 Remarks on general planning and Astro Kid

The Astro Kid world can be represented in PDDL, but the domains use of continuous actions, forces the representation to either be relaxed, or made in an awkward way, where the planner have trouble handling it. For PDDL to be able handle the Astro Kid domain properly, there needs to be introduced a proper way of handling concurrency.

[4] Fast downwards description of the situation..

Table 3.6: white space. for each version the width of the problem have been increased by 5,

domain update,add data for classic

	prob02	prob02v2	prob02v3	prob02v4
total Time	11.2 s	24.0 s	49.3 s	91.0 s
translator Time	7.6 s	17.8 s	38.8 s	75.5 s
relevant atoms	12394	22257	34922	50387
auxiliary atoms	14389	20021	25656	31291
final queue length	26783	42278	60578	81678
total queue pushes	58980	101633	155153	219523
axioms	1	1	1	1
peak memory	100800 KB	157492 KB	229452 KB	317760 KB
task size	56625	104505	166385	242265
preprocessing Time	2.4 s	5.8 s	9.9 s	24.5 s
necessary operators	10774	20079	32184	47089
search Time	0.2 s	0.4 s	0.6 s	1.0 s

Table 3.7: Running times

	prob00	prob01	prob02	prob02v2	prob02v3	prob02v4	prob02v5	prob03
universal	0.892	0.887	11.326	24.798	52.617	95.08	x	0.794
existential	0.646	0.788	6.777	12.148	18.863	27.405	36.86	0.704
	prob04	prob04v2	prob07	prob09	prob10	prob11	prob12	level 4
universal	23.905	9.197	36.326	5.842	13.45	2.4 2	21.484	429.9
existential	124.794	4.434	14.228	2.444	5.789	1.713 s	7.346 s	x

Relaxing the domain ensures that a solution is found quickly, but this solutions is not always valid. This can in some cases be solved by adding noOps at appropriate places. Adding noOps to the the plan when necessary takes a little away form the idea of using the general planner, by requiring some specialised knowledge about the domain, that cant easily be reused on other domains.

The Update approach guaranties to give a valid optimal solution, due to simulating the complete domain. This approach however have the cost of time and memory used. This is due to the domain not fitting well into the classical domain. The variations of this approach shows this clearly by also getting into problems. In general the update approach is only useful on domain where the assumption for the classic approach does not hold, due to large speed difference an nearly equal quality of the found plans. Even then it only works on simple domains, since it often fails to find a solution due to finite memory and time

Table 3.8: level 4

translate+preprocess 7.425 s

name	time (s)	lenght
Additive	0.329	35
blind	0.196	35
ContextEnhancedAdditive	0.193	35
causalGraph	0.182	35
ff	0.19	35
max	0.188	35

Table 3.9: level 7

translate+preprocess 8.086 s

name	time (s)	lenght
add	1.5	77
blind	14.8	76
Context-enhanced additive	13.1	76
causal graph	17.47	76
ff	22.0	76
max	26.9	76

constraints.

The heuristics causal graph, FF and Max have shown to be the most reliable of the available heuristics with a slight advantage to FF.

Table 3.10: level 10

translate+preprocess 21.248 s

name	time (s)	lenght
Additive	0.878	34
blind	2.174	34
ContextEnhancedAdditive	0.717	34
causalGraph	1.024	34
ff	0.755	34
max	1.326	34

Table 3.11: level 14

translate+preprocess 12.389 s

name	time (s)	lenght
Additive	12.719	273
blind	11.367	158
ContextEnhancedAdditive	104.799	181
causalGraph	20.722	158
ff	33.153	158
max	28.949	158

Table 3.12: level 32

translate+preprocess 24.389 s

name	time (s)	lenght
Additive	0.934	20
blind	131.198	17
ContextEnhancedAdditive	0.712	17
causalGraph	2.213	17
ff	0.748	17
max	0.961	17

Chapter 4

Learning

For an agent to be able solve a problem without prior or very little knowledge, it needs to be able to learn how to handle things it has not seen before. To do this the agent needs experience to learn from, there are various ways of obtaining this experience and how to use it. In the two extreme ends of the spectrum are the two approaches "optimisic" and "pessimistic". which is best depends on the goal of learning and the environment.

Optimistic is build around everything being possible, unless proven otherwise. It focus on exploring the environment by trial and error. This approach works best if all actions in the environment is reversible or it can be reset. This is due to that the agent while exploring the domain inevitable will do something irreversible, (possible resulting in a non solvable level). It is also helpful if the environment can be modified to help exploring hypothesis about the environment.

The approach can be compared to a "breath first search", and it will learn a wide array of possibilities in the domain, this wide knowledge helps finding solutions of a high quality. however as with BFS it is also slow, especially since time can be wasted on irrelevant knowledge but effective once learned.

An advantage with this approach is that everything is assumed possible unless proven otherwise. This feature is especially advantageous on large domains where only a small subset of objects effects any given action. It also means that it will always find a plan but it can find faulty plans, which means it can consist of impossible actions or that the result of the plan would deviate from the predicted result. That results of a plan can be guaranteed means it will invertible end in undesired or unsolvable states.

Pessimistic this approach is the polar opposite, where nothing is excepted to work unless proven otherwise. It is build around only doing actions that are guaranteed to work (already knows how to do), the agent will therefore never do something new. The only way of gaining new knowledge are from receiving and analysing traces, since the approach dosnt do something new, the traces it self can produce are not that helpful. the main way to obtaining these would be to ask for help from a "teacher" each time it is stuck as suggested in [WL08]. The advantage of this approach is that everything the agent does is guaranteed to turn out as expected, and it will therefore never produce a faulty plan. It can however not guarantee to find a plan without the teacher. The approach can be compared to a "depth first search" due to it exploring in a single direction (know action), which means it will not necessarily find the best solution. This approach learns quickly what it already have seen but only that, and the fact that everything always turns out as expected means it is useful if the environment is irreversible.

4.1 Learning in Astro Kid

It has been shown earlier that to solve problems in the Astro Kid domain using Fast Downward and PDDL, it is not feasible to model the complete world, and that it has to be relaxed. Therefore the relaxed version of the domain will be used for the purpose of learning. The PDDL states in them self can be considered as an interpretation of visual environment since the domain is fully observable. It is furthermore assumed that the possible actions names and parameters are given. This information are more or less equivalent to a human player knowing what the controls of the game are.

To simplify the learning problem, there will initially be started with learning a restricted version of the domain and extending from that. This initial restricted version will consist of the basic actions: left, right, up down and push (no continuous non player movement), since they in there simplest form can be described with solely conjunctions. This means that only parts of the domain are learned and the remaining is left as it is (destroy, gates etc.). The Action cost feature of PDDL is not considered and will be hard coded. However in general the action cost can be considered as an effect and learned in mostly the same way.

When looking a the Astro Kid domain it is worth noting that it has a slow learning curve. Where most of the actions are introduced by a trace in a tutorial. Using these traces can speed up the learning, by quickly narrow down the search space, and give hints to what useful. The traces are therefore a good place to start for a pessimistic approach, they are however not complete enough to solely base a pessimistic approach on (e.g falling is not introduced).

4.2 Basic learning

The learning algorithm used is based around observe hypothesise experiment. (algorithm 1). It works by analysing a series of action in the domain, and from the knowledge obtained creating a hypothesise of how the world is believed to work, finally experimenting to see what happens when it is applied on the domain. This process is repeated until the enough is learnt to solve a given problem.

Algorithm 1 Learning algorithm

```
while !goal do
    Analyse given trace
    make an hypothesis about how the domain works
    run planner with hypothesis
    get trace from simulation
end while
```

The main idea of how to analyse these traces and learn from them, is that if an action is applied on the domain and something happens, what happens must be the effect of that action, and that the action is applicable means that the preconditions of that actions is fulfilled. It can thereby be concluded that only the literal present before the actions is applied are part of the precondition. An action applied on the domain is considered a applicable as long as the simulations does not reject it. This means an action does not have to have a visible effect to be applicable. This is due to masked knowledge, this is the case where the effect of an action already being present in the domain, and therefore no changes occur when the same effect is applied again. This also means that it is not necessarily possible top find all effects of an action, due to the masked knowledge. In the same way as effects can be deduced so can what is not effects also be found.

An action failing does not give the same amount of information, since it is not known which precondition is the cause of failure. However by combining knowledge from several attempts some information can be deduced. The idea is that at least one of the predicate not present is a precondition. The set of non present predicates is referred to as a candidate precondition. A candidate can be reduced by removing those predicates present at a success full actions. Only when reduced to a single predicate, can it be guaranteed that predicate is a precondition, that leaves the problem of if the action failed due to multiple precondition violations. If this is the case the best the a candidate preconditions can do is giving a hint to what the preconditions could be.

When looking at the predicates present or not a special kind of predicate exists Equality, even though it differentiated from the classic predicates in how it is evaluated it can be deduced in the same way.

When deducing the preconditions typing reduces the problem since it simplifies the grounding of the possible predicates (eg. in a domain with 100 location, 1 stone, the predicate (at ?thing ?location) can be grounded 100 different ways, and without typing $101^2 = 10201$ different ways), however in worst case all objects would be of the same type, which would make the typing meaningless, and would therefore have no effect on the algorithm. In a properly typed domain the complexity would often be reduced considerably. However even with typing, grounding can be a problem since there is no restriction on the number of objects and how they can be relate to each other. It can clearly be seen with the positions and there relation to each other (a 10x10 map means 100 locations and 4 directions and their relative position defined by the predicate (relative-dir ?location ?loaction ?direction) can be grounded $(100^2 = 10000) * 4 = 40000$ different ways).

One way to further reduce the grounding problem is to assume that only predicates present (with that particular grounding) and the negated equivalent in the problem description are relevant, until proven possible by an effect. The idea is to avoid taking various impossible mutations of static objects into consideration. This is especially effective on problems with a underlying fixed structures, such as maps and levels.

This leaves how to handle "new" predicates not seen before created by effects. When a new predicate is discovered the negated version is add to all candidate preconditions. This however means that a candidate preconditions cant be proven unless all predicates is discovered.

Its worth remembering that even though a particular grounding influence if a given action is applicable at that moment, the grounding in it self and the type variables is not what is interesting, it is the relation between the predicates and therefore generalising of the parameters is need, to easier detect relations between predicates it is not enough just to know if a given predicate is present, but also what its relation to the other predicates is. A graph representation is therefore used, what is represented is how the objects is related to each other through known predicates.

4.3 Generating the action schema

One thing is to acquire knowledge from traces, the information also needs to be used, this is where the optimistic and pessimistic approach is required. They are used to generate the action schema. Due to the lack of a teacher an the optimistic approach is more advantageous, since it self-sufficient and it is possible to reset the environment. The preconditions and effects using the optimistic can be generated as follows.

Preconditions The preconditions is generated from the narrowed down candidate preconditions, even though that it is not known which part of the candidate preconditions that is the precondition, it is know that the particular combination fails. This can be used when generating the action schema, by adding all the candidate set as preconditions. when the sets are reduced enough the number of candidates can compensate for the lack of precision. This also ensures that no mistake is repeated

Effects Everything happens unless proven otherwise,(every possible effect are applied unless the effect is disproven), this can be meaningless sometimes since the effects of not x and x would remove each other. To avoid this problem, each time both effects are applicable one is chosen by 50/50 chance.

4.4 How does it work?

Applying the learning approach using an optimistic works quiet badly due to how Fast Downward, handles the large number of unrestricted parameters, in the earlier stages of the learning. It is in particular the preprocessing which takes to long. This is not a problem solely for Fast downward, the YAHSP[1] planner have the same problem except it dies on 5 parameters instead of 6 (results can be seen in table 4.1).

These results means that if learning is going to be applied using Fast Downward, it needs to be a more pessimistic approach, that restricts the parameters and

[1] `http://v.vidal.free.fr/onera/#yahsp` 2. at IPC 2004

Table 4.1: Unristrected search using level 1 and 4.1

Parameters	2	3	4	5	6
YAHSP	0.00	0.01	1.49	x	x
Fast Downward	0.00	0.02	48		x

Figure 4.1: Unrestricted action

```
(:action Walk
        :parameters (
                ?p0 - player
                ?p1 - location
                ?p2 - location
                ?p3 - location
                ?p4 - location
                ?p5 - direction
        )
        :precondition   (and
        )
        :effect         (and
                (at ?p0 ?p2)
        )
)
```

thereby simplifies the preprocessing. Using a completely pessimistic approach leaves one problem, that there is in fact no teacher to ask for help. Some traces can be obtained from the tutorials, but they are not complete enough to solely rely on. If they are processed in order they occurred (falling is missing in the first tutorial).

The pessimistic schema is is generated as follows.
preconditions all possible preconditions, unless they are disproven.
effects all proven effects.

To solve the problem the solution lies somewhere in between pessimistic and optimistic. The idea is therefore to relax the pessimistic approach little by little, when it cant find a solution, and thereby ensuring that some restrictions still exists. This is done by adding effects and removing preconditions. The effects add are selected from what effects could have been masked and each precondition is removed with a likely hood of 1/n (proven preconditions cant be removed). The main disadvantages of this approach is that some of the pessimistic approach still remains, which means that it is possible that it turns up no plan and as a result of that no new information can be learned from the experiment. To minimize the chance of this happening often the chance of changes increase with each no plan found. The difficult part here is in choosing the correct increase. Furthermore to avoid making the same the mistakes multiple times the preconditions from the optimistic approach is add.

The approach for this limited version of the domain works well since given traces can covers most of the domain, and what isn't covered can the relaxation compensate for. This can for example be seen on problem 1B.3 and given trace walk left, only two predicate preconditions needs to be relaxed ((not (= ?p5 right)) and (= left ?p5)) before the problem can be solved. If given the trace right instead it is solved without relaxation. The solution for the grounding have a great impact on the relaxation since it removes a lot of non useful predicates (such as (not (relativ-dir pos-01-01 pos-02-02 left))) which means the chance of the relaxation effecting the right predicate increases.

4.5 Extending the domain

4.5.1 Conditionals

To further extend the support for the domain and allow a wider array of actions such as sliding. Conditionals and Disjunction needs to be add, however a working domain without these features can be created, but doing so would defeat the purpose of the learning by conforming the domain to the learning and not the other way around, by moving away from directly representing the controls available, used for controlling the avatar, and also quickly become unreadable.

When looking at conditionals and disjunction they both have the property that they can be compiled away by splitting the action into multiple actions.

This can done for conditionals by moving the condition from effects to precondition and treating it as a non conditional action. at the same time adding conditions so only one of the new actions is applicable at any time. If a conditional have multiple effects they can also be considered separately. $p \wedge q \implies x \wedge z$ is equivalent to $(p \wedge q \implies x) \wedge (p \wedge q \implies z)$

This means that each effect of an action can be considered as a separate sub-action with a single effect. So as soon as a given effect is present in a trace, it can be used for deducing the precondition for that effect, using the earlier described approach.

The method for deducing preconditions relies on that it is possible to connect a unique effect to its preconditions, therefore it needs the assumption that any given effect can at most appear once as an effect in each action.

The trick is then to merge all the separate sub-actions when generating the action schema, for the pessimistic approach this is done by using the intersection of all candidate preconditions as preconditions and the difference as conditionals.

using this approach for learning conditionals should not effect the effectiveness of the learning. The speed of learning an action with or without conditionals isn't effected (neither in time or plan attempts). This is due to the extra work add are dependent on the number of effects which in praxis must be considered negligible. Furthermore the learning builds on the same principals ... worst case would be that that the action solely consist of conditionals.... only learning for a single effect each time.... sub actions equivalent to action.... therefore no different than learning separate actions..

4.5.1.1 Disjunction

The base (non disjunction part) of an action can be deduced as earlier described (section 4.2), the disjunction part cant be found this way, since the different parts of the disjunction would eliminate each other. Using this approach, it is however possible to detect if a disjunctions appears in the action, this can be seen when contradictions happens (an action failing when fulfilling all the predicted precondition). When such a contradiction to the base happens it is guaranteed that at least one predicate from the disjunction precondition is not presents. When only looking at the simplest version of disjunctions of the form $x \vee y \vee z ...$ with a maximum of a single disjunction per. action. This simple disjunction can be deduced by using that all present predicates from a contradiction traces cant be part of the disjunction, since it would otherwise have succeed. Furthermore since it possible to treat each effect as an separate sub-action it is also possible using the same approach to deduce a single disjunction for each sub-action, which means a maximum of one disjunction precondition and one disjunction for each conditional.

This approach however falls apart as soon as more or more complex (eg nested conjunction) disjunction are allowed. The main problem here is that the used approach relies on that no predicates from the disjunction being present on failure where the base is fulfilled, and this can no longer be guaranteed eg. in $w \vee x \vee (y \wedge z)$ a failure with y present would remove y.

To avoid this problem it is necessary to identify which part of the disjunctions are related to the failure or success. This is however difficult to determine due to uncertainty add by the disjunctions. The uncertainty is rooted in that multiple disjunctions can hide each others "effects". When looking at a failure it is therefore not possible to eliminate any predicates, since potentially any combination of the non present predicates could be part of a disjunction and therefore the cause of the failure. The only thing that can be conclude is that particular complete set of preconditions isn't allowed. On a success the same problem occurs. So without a way to connect course and effect it is basically down to something is missing from some disjunction. In the same note adding more complexed disjunctions to the domain also makes it nearly impossible to determine when an action is completely learned. This is due to that every combination of preconditions needs to be tried, unlike earlier where preconditions could be removed if not present a single time. This means that a different approach is need for action schemas using more complex disjunctions, however the simple approach should be sufficient to interact with the Astro Kid domain.

Deducing disjunction introduces a new aspect, that effects the learning process. Which is that it is no longer possible to discard already learnt episodes, without

Figure 4.2: Example of merging two subactions

```
subaction 1
pre: (and (at ?p1 ?p2) (relativ_dir ?p2 ?p3 right)
        (relativ_dir ?p3 ?p4 down))
effect: (and (at ?p1 ?p3))

subaction 2
pre: (and (at ?p1 ?p2) (relativ_dir ?p2 ?p3 right)
        (relativ_dir ?p3 ?p4 down) (clear ?p4))
effect: (and (falling ?p1))

merged action
pre: (and (at ?p1 ?p2) (relativ_dir ?p2 ?p3 right)
        (relativ_dir ?p3 ?p4 down))
effect: (and
        (at ?p1 ?p3)
        (when (clear ?p4)
                (falling ?p1)
        )
)
```

the risk of loosing knowledge. The problem lies in that knowledge are obtained from contradictions to the base, and some of these cant be discovered until more knowledge is obtained. Furthermore the speed of learning is also effected since the disjunction cant be started to be learn before contradiction can be detected.

4.6 Generating the action schema

The basic structure for generating the actions remains the same, the differences lies in that multiple sub-actions needs to be merged into a single one. This can be done by taking the intersection of the sub-actions preconditions as the preconditions, and having the difference as the conditionals(figure 4.2). Whit the additions of disjunctions the approach moves slightly more away from the completely pessimistic approach. This is due to the disjunction part being deduced using an optimistic approach, which meant the generated disjunction being a disjunction of all possible disjunction predicates.

The presence of universal quantifiers can be detect if effects not associated with parameters are present. That the effect predicate isn't associated with parameters also gives a handle on where to start when deducing the preconditions (a single unique point). The handle means that the quantifier can be deduced nearly as any other effect. The two main differences is that the predicates considered is no longer only the subset containing the parameters but all present predicates. the second difference is that the graph representations is essential.

universal quantifiers used as solely preconditions using imply can be transformed into existential quantifiers by inverting the set of conditions

4.7 How does it work?

The consequences of extending the domain have is that learning an single action takes longer (number of plans)**table ....**, this is as expected since there are also more to learn. Learning conditionals in it self have no effect on how many steps it takes to learn a action, when taking into consideration that, if the conditionals had been compiled away (into multiple action) it would be worse since what is in common could not be learned simultaneously.

The disjunction part have an impact since the base have to be learned before the disjunction can be learned, the consequences of this mostly mitigated by the optimistic approach used for this part.

As can be seen in **table ....** if to many new situations are introduced at the same time, it can be difficult to create traces even with relaxation. The problem lies in that multiple specific preconditions needs to be relaxed simultaneously, and even more relaxation dosnt necessary solve the problem, since it then reintroduces the problem from section 4.4.

The reason to why many new situations are introduced at once is that for a human, many of the new situation dosnt require an explanation eg. when introduced to walking on ground, it obvious that the avatar can walk off an edge. Basically the learning approach is missing some background information to make assumptions about the world on. However even with this background knowledge it isn't even possible to use for the learning since concepts as up, down and gravity dosnt translate to PDDL. This means that even though the tutorials for Astro Kid is complete enough for a human to understand the domain, this is not the case for learning using PDDL and some essential concepts is not explained. To compensate for this either more traces are need or a planner that allow a more optimistic approach.

Chapter 5

Conclusion

The Astro Kid environment has been described in PDDL and integrated with Fast Downward in such a way that it is able to solve most of the levels in the domain with some limitations concerning concurrency. This has been combined with a learning process which enables it to to learn from given traces. The learned knowledge from different traces can easily be combined using the PDDL representation. And also learning system can generates its own new traces when the given isn't sufficient. This is done using a mostly pessimistic approach to learning, with a little use optimism when relaxing. The created learning system works well within its limitations, but if it needs to be extend further, a different approach is need.

Working with Fast Downward have shown that general purpose planners still have some way to go before truly becoming general purpose planners. The problems with Fast Downward illustrates some of the weakness for general purpose planners and their strengths. The main weakness is that the concept of a general purpose planner needs to be able to span a wide array of problems and sometimes problems not considered or prioritised by the creators of the planner. This means the performance of the planner can vary greatly depending on the problem description.

On the other hand this also means that when conforming to the type of problems the planner is expecting everything works well. This means that it can be an artform in it self to optimise the PDDL description of the problem and require detailed knowledge of the planner used.

5.1 Future Work

It would be interesting to see how the complete Astro Kid domain could be supported by PDDL. The main problem to achieve this is the concurrency created by the continuous actions. This could be solved by either extending the language support for temporal planning or switching to a multi agent version of PDDL. The idea of switching to a multi agent version eg. MA-PDDL is to threat all objects that can be involved in continuous actions and treating them as independent agents. This could help handling the concurrency issues and thereby allowing the planner to solve levels where this is an issue such as level 25. Switching to a multi agent system would however introduce noise (which agent caused what) and thereby more uncertainty to the system when considering learning. Since the used learning approach already is limited by uncertainty, such a change would require some modifications to the learning part to deal with the add uncertainty.

To truly be able to drop an agent in a new universe and make it act in it. The agent needs to be able to handle not knowing which parameters are useful. So future work would be concentrated around learning using only quantifiers. Such an approach could be based on Markov logical network used in [ZYHL10].

Chapter 6

Proof of concept/implementation

The complete implementation can be found at Insert Link . . . . . .

6.1 World Simulation

Due to the complete set of rules in the Astro Kid world not being be described. The rules are constructed by watching the tutorials and playing the game. This means that there most likely are some few inconsistency compared to Astro Kid game itself.

6.1.1 Backend of the simulation

All objects in the world is represented as an object with a set of coordinates defining its position in the 2d grid. Multiple objects can have same position, with the exception of that there can only be one moveable object at any given

Figure 6.1: Robot PDDL representation

```
(:objects
        robot0 - robot
)
(:init
        (at robot0 pos-01-05)
        (facing robot0 right)
)
```

coordinate set. Besides the position all objects have some extra based on which kind of object it is eg. a robot object also knows if its moving and which way its facing. This extra information means that each object have complete knowledge of it self and can separately be translated to PDDL (eg. figure 6.1), so generating the problem description consist of merging the description of all the objects in the world and adding formalia to the description. Running the simulations is build around a basic input update cycle, where input is the player commands, which can be left, right, up, down or a set coordinates. When a command is given the validity of it is checked and if valid the involved objects are then marked to be move in the given direction. Nothing is actually moved in this stage and if a new command is given before and update the previous one is ignored.

The update consists of all objects are moved to their new positions, when they are moved, they are checked for if they should keep moving next update or stop. objects in the bottom of the grid are updated first to ensure items fall correctly.

6.1.2 GUI

A basic GUI using Swing is created, it consists mainly of a visual representation of a given Astro Kid world, and some controls for interacting with this world. the world is visualised using a 2D grid of images which is synchronised with the world using a observer observable pattern.

6.1.3 Solving Problems

The first thing that happens when trying to solve a problem is a PDDL description is generated from the world. This generated description is a direct representation of the world, so no optimizations are performed on it.

The generated problem description and a predefined domain description is passed to a new instance Fast Downward as command line parameters. The heuristic is also passed this way, as default the FF heuristic is used as it shown to work best on this domain. The output of Fast Downward is piped to a function that extracts the results of the search. The function extracts all actions with a action cost of 1 (which is the player actions) and translates them to a series of Commands that can be executed on the world. Finally each command is executed and validated on a copy of the world, and error massaged appears if the command isn't valid or no solution is found.

6.2 Learning

The learning part of the implementation revolves around implementing the main algorithm 1, deducing information and generating action schemas.

tutorials traces are stored as series of commands just as a plan from the planner. The only difference between a plan from the planner and one from a trace is that parameters are given from the planner and they are calculated for the trace when executing the command. The plans are executed as normally on the relevant world. The only difference is that a PDDL representation is generated for each step. Each episode (the command and before and after PDDL representation) is then used for the learning process, where there is differentiated between if an command was applied correctly or not to the domain. This distinction is solely used for how knowledge is deduced from the episode as described in section 4. The implementation does not store old plan.

When generating the action schema it is done as describe in section 4. The degree of which the relaxation happens in is chosen experimentally, and might therefore needs readjustment for different types of problems to ensure optimal performance.

Appendix A

PDDL

A.1 Blocks domain

A.1.1 Problem Description

```
(define (problem BLOCKS-4-0)
  (:domain BLOCKS)
  (:objects
    D B A C - box
  )
  (:init
    (clear C) (clear A) (clear B) (clear D)
    (on C t) (on A t) (on B t) (on D t)
    (handempty))
  (:goal
    (and
      (on D C) (on C B) (on B A)
    )
  )
)
```

A.1.2 Domain Description

```
(define (domain BLOCKS)
  (:requirements :strips :typing)
  (:predicates
    (on ?x ?y - object)
    (clear ?x)
    (handempty)
    (holding ?x - box)
  )
  (:types box table - object)
  (:constants t - table)
  (:action pick-up
      :parameters (?x - box ?under - object)
      :precondition (and
      (clear ?x)
      (handempty)
      (on ?x ?under)
    )
      :effect (and
      (not (on ?x ?under))
      (not (clear ?x))
      (clear ?under)
      (not (handempty))
      (holding ?x)
    )
  )
  (:action put-down
      :parameters (?x - box ?under - object)
      :precondition (holding ?x)
      :effect(and
      (not (holding ?x))
      (clear ?x)
      (not (clear ?under))
      (handempty)
      (on ?x ?under)
    )
  )
)
```

A.2 Domain Variation

The original part of the update domain and what its replaced with in the existential approach.

A.2.1 Original

```
(:action updateDestroy
    :parameters ()
    :precondition (and
    (update updateStage1)
  )
    :effect (and
    (forall (?t - thing ?at ?under - location)
      (when
        (or
          (and
            (boarder ?at)
            (at ?t ?at)
          )
          (and
            (relativ-dir ?at ?under down)
            (at ?t ?at)
            (or
              (and (ground blue ?under) (not (wearing blue)))
              (and (ground purple ?under) (not (wearing purple)))
            )
            (not (exists (?r - robot) (= ?t ?r)))
          )

        )
          (and
            (not (at ?t ?at))
            (not (moving ?t left))
            (not (moving ?t right))
            (not (moving ?t down))
            (clear ?at)
          )
      )

    )
    (not (update updateStage1))
    (update updateStage2)
  )
)
```

A.2.2 Changed

```
  (:action updateDestroyBoarder
       :parameters  (?t - thing ?at - location)
       :precondition  (and
              (update updateStage1)
              (boarder ?at)
              (at ?t ?at)
           )
       :effect (and

          (not (at ?t ?at))
          (not (moving ?t left))
          (not (moving ?t right))
          (not (moving ?t down))
          (clear ?at)
        )
  )

  (:action updateDestroyUnder
       :parameters  (?t - thing ?at ?under - location)
       :precondition  (and
              (update updateStage1)
              (relativ-dir ?at ?under down)
              (at ?t ?at)
              (or
                (and (ground blue ?under) (not (wearing blue)))
                (and (ground purple ?under) (not (wearing purple)))
              )
              (not (exists (?r - robot) (= ?t ?r)))
           )
       :effect (and
          (not (at ?t ?at))
          (not (moving ?t left))
          (not (moving ?t right))
          (not (moving ?t down))
          (clear ?at)
        )
  )
```

```
(:action updateDestroy
     :parameters ()
     :precondition (and
            (update updateStage1)
            (not (exists (?t - thing ?at ?under - location)

              (or
                (and
                  (boarder ?at)
                  (at ?t ?at)
                )
                (and
                  (relativ-dir ?at ?under down)
                  (at ?t ?at)
                  (or
                    (and (ground blue ?under) (not (wearing blue)))
                    (and (ground purple ?under) (not (wearing purple)
                  )
                  (not (exists (?r - robot) (= ?t ?r)))
                )

              )

              )

            )
          )
     :effect (and

        (not (update updateStage1))
        (update updateStage2)
      )
)
```

A.3 Levels

insert levels here

Appendix B

Astro Kid

B.1 Astro Kid Rules

The player can do the following things: walk left and right, climb up and down on ladders, and push objects horizontally away from it.

B.2 Astro Kid Levels

Figure B.1: Level 25

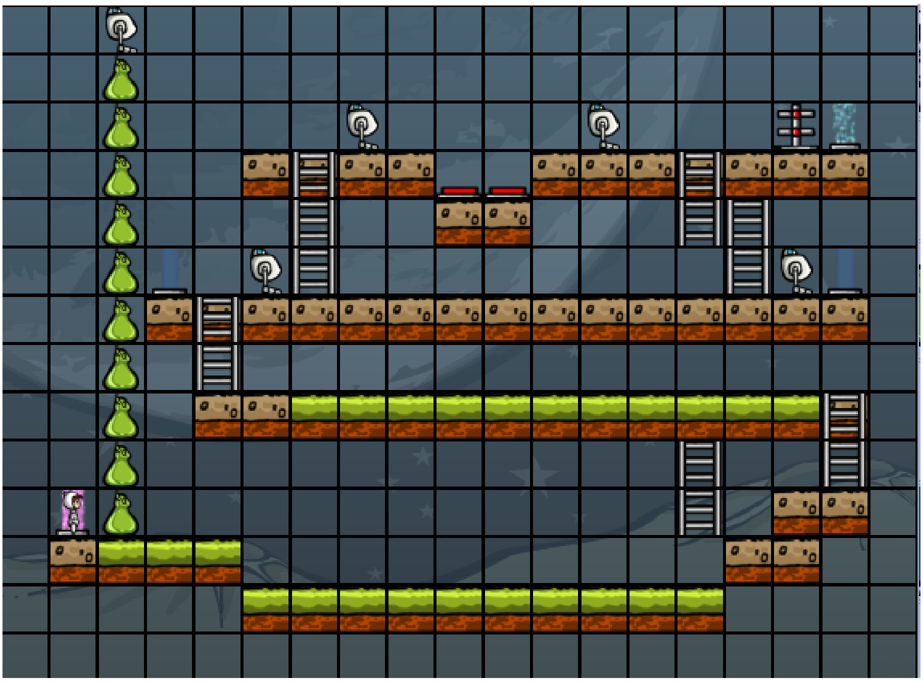


Figure B.2: Level 31

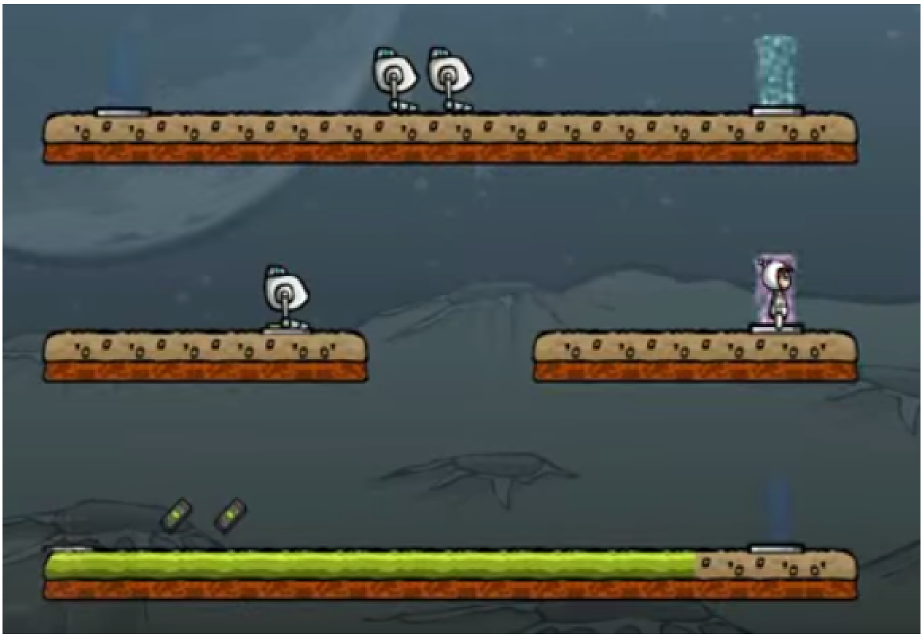


Figure B.3: Toy problem 1

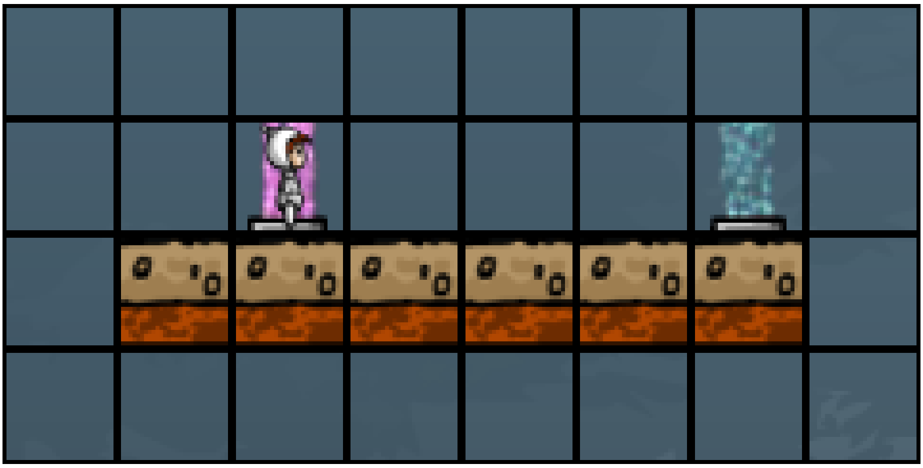
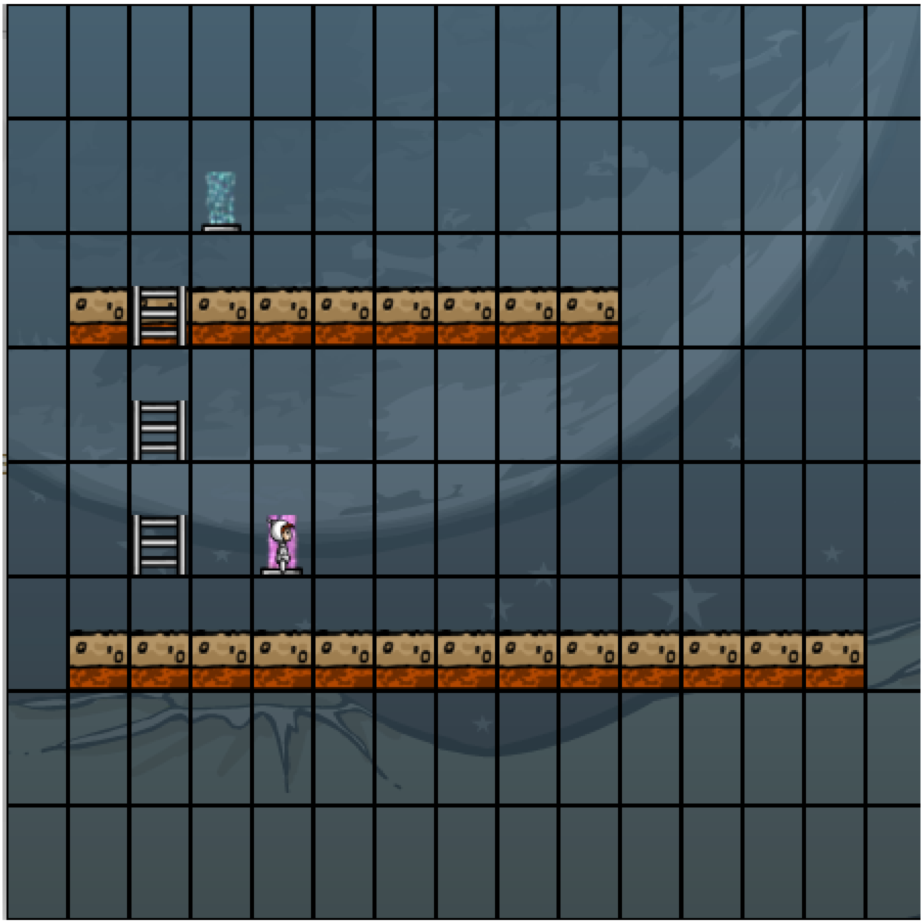


Figure B.4: Toy problem 2



Bibliography

[Bre03] Michael Brenner. A multiagent planning language. In *Proc. of the Workshop on PDDL, ICAPS*, volume 3, pages 33–38, 2003.

[GFL02] Antonio Garrido, Maria Fox, and Derek Long. A temporal planning system for durative actions of pddl2. 1. In *ECAI*, pages 586–590. Citeseer, 2002.

[JJ15] Søren Jacobsen and Jannick Boese Johnsen. Action learning in automated planning, 2015.

[RN14] Stuart Russell and Peter Norvig. *Artificial Intelligence : A modern Approach.* Pearson, 2014.

[WL08] Thomas J. Walsh and Michael L. Littman. Efficient learning of action schemas and web-service descriptions. *AAAI Conference on Artificial Intelligence*, 23, 2008.

[ZYHL10] Hankz Hankui Zhuo, Qiang Yang, Derek Hao Hu, and Lei Li. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence, Artif. Intell, Artif Intel, Artif Intell, Artificial Intelligence an International Journal, Artif. Intell. J*, 174(18):1540–1569, 2010.