High-level task planning in Robotics with symbolic model checking

Frank Schröder

Aug 11, 2018

Abstract

A robot control system contains a lowlevel motion planner and a high level task planner. The motions are generated with keyframe to keyframe planning while the the tasks are described with primitive action-names. A good starting point to formalize task planning is a mindmap which is created manually for a motion capture recording. It contains the basic actions in natural language and is the blueprint for a formal ontology. The mocap annotations are extended by features into a dataset, which is used for training a neural network. The resulting modal is a qualitative physics engine, which predicts future states of the system.

Keywords: Artificial Intelligence, Computer Graphics, PDDL

Contents

1 Planning

1.1 Symbolic model checking for PDDL planning

Suppose, we have created a pddl model for a pick&place robot. The pddf file contains action primitives like grasp, pickup, place and move and has defined the effects of the actions. Now we are planning something and the planner is producing the plan:

opengripper, move right, down, close gripper

If we want to run the plan on a real robot we will notice a failure. But why? Was the idea of use a pddl domain model wrong, was AI planning in general a bad idea? No, what we have recognized is simple a gap between our domain model and the real world. Such a gap is natural. The problem is, that an abstract symbolic model is not able to predict for all cases the future. Compared to the real world, or a numeric simulation in Box2D such an abstract model is inferior. That means, the plans which are produced by the system can't be executed on the robot, otherwise they will result into failure.

The general question is: can we trust our domain model? This has to answered with no, in case of doubt, the domain model isn't working and only sometimes the produced plans are valuable. But what prevents us, to make a dedicated model checking step first and try out, how good our model is? And if the model checking fails, we simply ignore the generated plan.

The surprising fact is, that it is possible to run a robot with a semi-accurate domain model. If the PDDL file is only correct in 50% of all cases, that means, that in thses 50% cases, it will produces valid plans. Otherwise the planner generates an error message. That the human operator can improve the action model, or he can simply ignore the situation.

Let us make an example, in which we are using a pddl file which contains from the beginning an error. The pddl file knows to actions: “opengripper” and “down”. According to the opengripper command, the effect will be that the robot gripper opens up, what the PDDL file isn’t aware is, that this feature never worked, because the servo motor is broken. In contrast, the other command “down” works great. It brings down the robotarm to the object. The good news is, that we can profit from the partly working domain model. It is possible to control the robot a bit and ignore the non-working parts. That means, the htn-planner wlll produce the correct plan, if the goal is simply to move the robotarm. Only if the goal has to do with open the gripper, the plan isn’t working.

It is important to see a broken domain model not as an exception but as a normal behavior. Because it is impossible to guarantee that the pddl file was implemented perfect and that the abstract action model is able to predict the reality. What we only can say is, that the PDDL sometimes is right.

Perhaps it is time to explain the difference between lowlevel actions and a high-level task model. On the lowlevel side, a robot control system is remarkable simple. A robotarm has not more then 3 dof, and each of them is realized with a motor. What the system can do is activate each of the motors:

motor1=-1
motor2=+0.5
motor3=0.2

As a result, the mechanical construction is moving. More switches and sliders are not available that is the overall robot system. The only problem is, to transfer these lowlevel actions into a longer plan. In the timeperiod of 30 seconds, the motor-movement can have different parameters, and this result into a well known state-space which is very huge in it’s dimension.

The idea behind HTN-planning is to reduce the state space with special heuristics. And for doing so, a so called action model is used. An action model is a constraint which combination of servo commands are useful and which not.

Structuring the state space The major task of a htn-planner is to simulate plan execution. It gets a plan like “opengripper”, “moveright”, “down” and calculates what the future game-state will be. This evaluation is nessary, because we need to test random plans until one of the plans is fulfilling the goals. There are two problems in plan-evaluation: at first it is unknown how the action model looks like, and first it is unclear if a given action model produces correct predictions.

Let us go into the details. The lowest layer of the robot control system is determined by the hardware. If the robotarm consists of 3 servomotors, it is a 3DOF system and each of the motors can be driven forward and backward. Everything above this lowlevel layer can be specified freely. We can define an action model which 6 motion primitives, or with 60. The aim of the action model is to reduce the state space, that means to give a smaller amount of possibilities in which the servo motors can be controlled. For example, if the action model contains only two motion primitives “opengripper” and “Closegripper” the state-space is very small. We can’t do any useful task with the robot.

At a baseline we can suppose a physics engine like Box2D which is similar to to real hardware and is able to predict the reality. The main problem with Box2D is the same as with the real world: the state space is huge. LIke in a physical model, we can build a 3DOF servomotor system in it and let all the motors run forward but it is unclear what the control parameter are to reach a certain goal in the simulation. The Box2D engine is forming a game, and the aim is to find a path through the game.

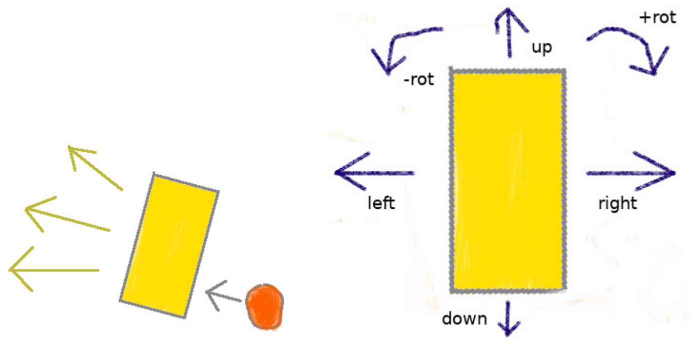

Figure 1: push task

Describing symbol model checking is complicated so perhaps an example will help. At first, we need an action plan which is tested against the HTN-planner:

```
1. opengripper
   servo1=.5 for 2 seconds
   servo1=0 after it
2. moveright
   initmove
     servo2=−1
   moveright
     servo3=−−3
3. down
     servo1=0
     servo2=0.8
     servo3=0.2
```

What is given in the listing is a detailed plan which contains high-level task descriptions and the detailed lowlevel commands for the servo motor. There are many possibilities what to do with that plan. The first option is, to run the plan against the HTN-planner. The second option is to run it against the Box2D engine and the third option is to test the plan on the real robot.

According the introductory remarks, the real robot and the box2D simulation can be stated as equal. So we need to test the plan only against two remaining instances: the HTN-planner and the box2d engine.

1.2 The advantages of a symbolic planner

Suppose a robot arm should push a box in a 2d map with a topview, see the figure 1. The movement of the box is given by the Box2D physics engine, that means, the box will behave realistic. The question is: what are the push-actions to bring the box to a certain point in space?

The first approach might be to program some kind of behavior tree. It is possible to write down in the sourcecode that the robot should push in a certain way to the box. This kind of technique has the disadvantage that it is fixed, the code for running the robot can’t be changed later, it is a heuristics but a bad one.

The better idea is to program a declarative model which can be searched by a planner for an answer. The first model is given by the

Box2D engine itself. Bux2D is able to predict future game states by simulating them:

$$futurestate = box2d(input)$$

Box2D can only predict the next timeframe. We can give as input a value and box2D will print out the box position as a reaction. If we are putting this into a for loop it is indeed possible to determine the plan to push the box. This problem solving technique is similar to computerchess which is also based on simulation.

The sad news is, that the computational effort is to high. If we want to solve larger problems with a long time horizon it will become impractical.

The topic of this chapter isn't given by random. The term "symbolic planner" is the answer to the problem and helps to reduce the CPU consumption. The basic idea is to invent an arbitrary game which follows different rules then the original one. Our original game was controlled by the robot. The robot (a circle) can move on the map and as a reaction the box is pushed. In the new game, the box can be controlled directly, it gets natural language commands like "left", "right" and "rotate clockwise". After the box has received such a command his position is updated on the screen. To bring the box to a certain position is simple, because we need only execute a series of commands like "left, left, up, up" and the box is on topleft of the map.

Now we have to different games, the original one in which the robot can be controlled and the more easier game in which the box can be controlled. The problem of mapping both games together is called in the literature "grounding". Grounding means, that there are at least two physics engine, and a failed grounding is equal that the mapping is not possible. And now comes the magic trick. We can use a box2d solver to realize the grounding. A box2d solver can be used, to determine which movements the robot has to do for execute a high-level task.

THe new game (direct box movements) is the high-level task-layer, while the robot pushing actions are the lowlevel motion layer. A box2d solver produces the lowlevel actions, while the task-planner answer the question which commands has to be given to the box. Eh voila, the problem is solved. What i have described here is called in the literature a hierarchical task and motion planning system. It needs very little cpu consumption and is able to generate the commands for pushing a box.

If a combined task- and motion planner is so powerful why it is not used widely? The problem is, that programming such a planner in software is complicated. The early attempts were done in the 1960s under the term "production system", since the 1990's the literature has understand the subject better and call it correctly "task and motion planning problem" because there is a symbolic planner and a lowlevel numerical planner at the same time active. And it is possible to extend the basic idea with much more features, for example to generate the action model autonomously which makes everything more complicated. The rough basic idea is to use only 2 layers (lowlevel and high-level planner) and to program the action model by hand.

If somebody reads the last paragraph he will perhaps noticed, that even in the year 2018 a task- and motion planner is not used by the mainstream. The reason is, that the most complicated programming exercise for game programming is a physics-engine itself but not more complicated tasks like a solver for a physics engine. And to extend such a solver with a symbolic task model is very advanced. That means, the number of tutorials and working code is limited. That means, from a theoretical standpoint it is the right way to go, but from the programming perspective this is unexplored.

More recent publications since the 2010 are separating between natural language instructions for describing high-level tasks and numeric values for describing lowlevel actions. That means, a symbolic planner is always guided by a lexicon or a grammar and understands action words like "move", "push", "left", while the lowlevel motion planner is based on numerical simulation and number-crunching which leads into physics engines like Box2d.

The reason why the high-level part can be expressed in natural language better has to do, that natural language is per default a high-abstraction language. A concept like "move right" is a broad description of a task which is far away from mathematical terms. Or to explain it from the other side. If the aim is to epxress a task on a high level layer, it is the perfect choice in describing the task with natural language. A description like open the box, take the object, move to the kitchen is very high-level and leaves out all the details.

1.3 Emulating IPL-V

The first AI symbolic reasoning systems were developed in IPL-V which is a list processing language for mainframes. Or to be more specific, it is an assembly like programming language without any syntax which is very powerful but hard to learn. Today, most programmers are not familiar with IPL-V or the later developed LISP system. Their main advantage over today's C++ was, that list processing allows to change the sourcecode at runtime. The early symbolic reasoning systems used this feature heavily.

But there is a bypass to that. In C++ it is possible to include the "python.h" library which gives us access to a runtime virtual machine. The command for executing a python string is:

```
PyRun_SimpleString("print 1+1");
```

That means, we can change the string at runtime and change a program at runtime. This emulates the IPL-V programming language but uses modern C++ and Python programming style. What can we do with an embedded python interpreter? We can at foremost add and delete sourcecode while the c++ program is running. In normal C++ programming technique, all classes and methods are fixed. They are compiled into a binary executable. It is not possible to add a class after the user has send such a request. With the new embedded python interpreter, the C++ program contains a somewhere in a class a string, and the string contains of the python sourcecode. This string is at the first start empty. And the user can add lines to the string at runtime, for example he can add new python classes, new variables and new methods. And this can be executed by the interpreter.

Compared to the powerful IPL-V and LISP language this is not a new invention. It was realized 50 years ago. But it is a bit hard to recommend to learn IPL-V only because we need self-modifying sourcecode. The more modern approach in programming is without any doubt C++. And the idea with an embedded interpreter is a good mixture between both.

1.4 Symbolic planner for regrasping task

Suppose a robot hand is holding a box. In the manipulation task the fingers of the robot hand has to regrasp the object. With manual

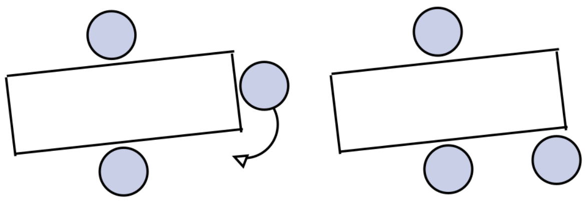

Figure 2: regrasp planner

control this is an easy challenge, because the human operator simply changes the position of the fingers. Realizing such behavior with an automatic planner is much more complicated. At first we can describe, what the robot can do on a physical level. He can move each finger, because a servo motor is in control of the system. The open question is: how does look the servo control parameters in the regrasp task?

The good news is, that in theory the problem can be solved by a symbolic planner which works hierarchical. The top layer of the planner knows high level actions which can regrasp the finger to a goal position, and the lower level of the planner are calculating the exact control movement to reach the subgoals. To understand the advantage we must go a step backward into the domain of reinforcement learning. The main problem here is, that a planner is able to try out alternatives, but he is not aware if the action was an improvement or not. Some authors have mentioned, that this problem can be overcome with guided policy search and reward shaping, which is equal to invent a feedback to the planner.

Let us go into the detail. The planner can execute a random movement. He sends a randomized parameter to the servo control. After executing the action, the system is in a new state. The question is: does this action has improved the situation or not? The idea of a symbolic planner is, to answer this question. A symbolic planner is some kind of subgoal generator. The main goal was: regrasp the object, and the symbolic planner is converting this goal into the subgoal: move the right finger below the object.

The transformation from higher goals into lower goals is driven by simulation. That means, the symbolic planner is able to execute also random actions and can determine the follow up state of the system. If the system is in the goal state, then the subaction was right. From a game-theory perspective, a symbolic planner is inbetween the high-level goal of “win the game”, to the low level possible actions in the game.

Predicting the future A symbolic planner works as a qualitative physics simulation. It is possible to send a command to the engine, and the simulator predicts the follow state. In contrast to a numerical Box2d like simulation, the prediction step is longer. The idea is get a higher abstraction level.

The derivative game consists of 10 possible places and 3 pieces who can moved between the places. The game engine is able to parse an action like “move A to place 1”. And the current state space is described by “A=2, B=7, C=5”. It is some kind of semantic enriched game, because it is not storing the physical location of the robot fingers, but the game engine thinks, that the game has to do with a table like arrangement in which the pieces can be moved on positions. Suppose the piece A is on place 2. Now we want to move also the piece B to the same position 2. If the game engine is programmed

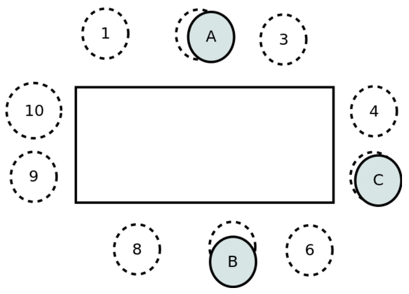

Figure 3: derivativegame

right, it will prevent such a move and call it illegal. It doesn’t fit to the rules. Or suppose we generate another condition, in which the bottom line (contains the positions, 6,7,8) is empty. This will produce an output “object lost”.Because it is also given by the rules, that on every line at least one pieces must be available.

The advantage of a derivative physics engine is, that it is possible to calculate possible moves very fast. We doesn’t need to ask the real Box2D engine what happens in a certain condition, we simply ask the symbolic game engine. In theory, this will save a lot of CPU power. The only bottleneck is, to program the qualitative physics engine.

Suppose a qualitative physics engine is available. The benefit is, that we no longer need the Box2D physics engine. Instead we can play the game against the symbolic game engine. We can enter commands which are executed and the new game state is shown on the screen. It is from a technical perspective a normal simulator. We can start the game, execute some moves, for example “move A to 1, move C to 6” and then we can see how the new state look like. The most important feature is, that it is possible to send even random actions to the game engine and see what will happen. And we can generate a game tree out of these movement to track longer action sequences. This allows us to bring the game engine into a goal state. That means, we only say that we want to reach state “A=3, B=8 and C=10” and the planner will find the correct action sequence autonomously. It is the same method, that be used for playing computer chess.

Physics Engine How can we call a qualitative physics engine, a derivative game or a high level prediction engine? According to the definition it is simply a physics engine, because it can predict the future position of objects in space.There is no need for a new name, the old one is well enough. The only difference to a out-of-the-box physics engine like Box2d is, that our new engine is developed from scratch and isn’t very realistic. For example if we execute the command “move A to place 3”, then without any delay, the piece A is on position 3. That is completely unrealistic, because in reality such a finger movement takes time and will produce steps in-between. The system can be called a bad programmed physics engine, which is usually given by HTML5 cooking games in the internet. They are simulate a situation, but because of any reason, the programmers didn’t implemented a realistic physics engine but only utilized a fall back system from the 1980’s in which cpu time was expensive.

1.5 Qualitative physics engine explained

Before introducing qualitative physics simulation, first a short look into traditional physics. In a blogpost is explained what the idea is:

“When you throw a baseball, it moves forward at a constant velocity. [...] Every method of numerical integration for Newton's laws involves some manner of maintaining a clock and doing calculations for tiny little snapshots in time." Burak Kanber, 2012, [1]

From a programming perspective, numerical integration is realized with a vector class and mathematical formulas. In the blogpost sourcecode is given, and it is very helpful if somebody want's to program a physics engine from scratch. The sad news is, that this is not the way what Artificial Intelligence need. Because at the end, we will only have a discrete timestep based physics engine which is comparable to Box2d, ODE and all the other well working engines available. From the perspective of game programming there is no problem with it, because a timestep based simulation is exactly this what Angry Birds and topdown carracing software needs. The game loop is executed 60 times per second, and the physics engine delivers the coordinates of the sprites, which can be displayed by the graphics card.

Qualitative physics engine is about high abstraction. It means to ignore the mathematics and control the simulation with language commands. In the example with baseball throwing the question is: what is the position if we throw it hard, soft or middle? Sometimes such a software is called "Intuitive Physics Engine" [10], but the aim is the same: to predict an outcome of an action on a high-level. What fuzzy logic and qualitative physics is about is to raise the abstraction level. The sideeffect is, that such a physics engine is faster, that means it can predict in one step a large timehorizont into the future. This makes its perfect for running a solver on top of it to determine the correct actions. Such a solver is called a task planner.

ACT-R can combined with a modul called "Predictive ACT-R" which is a qualitative physics engine. [11]. In the paper the term "predictive thinking" is introduced which is heuristics for reducing the state space. Instead of playing according to a policy, the idea is to simulate the game into the future and decide which is the better outcome. Or to make the point clear. Without thinking some steps forward, it is not possible to make a decision.

Programming a naive physics engine There are two possibilities:

1. Data driven approach with neural networks [12]
2. manual programming

The data-driven approach seems at the first impression the better way of implementing a physics engine, because no programming is needed. But the concept didn't work for normal physics engine, so why should it work for naive physics? All the mainstream physics engine like Box2D and others are not generated with deeplearning but programmed by hand. The usual workflow is, that the programmer at first read a theoretical paper about Newton law and linear algebra, programs then some C++ classes and releases version 0.1 of his own physics engine. Another github user is testing out the sourcecode.

According to the literature there were also some trials undergone to build physics engine not with programming but with neural networks. But these attempts failed. So it is rational to accept, that also for a naive physics engine which works more abstract only manual programming will work. That means, somebody has to write down the theoretical idea, for example a grasping model, and then the programmer can implement the model in sourcecode. The bottleneck todate is, that only few theoretical descriptions about naive physics model are available. In most cases such a model is equal to a board game. There are some rules, action primitives and an outcome. It is not so complicated to describe it in a paper, and it makes no sense to discard the idea and switch over to neural networks.

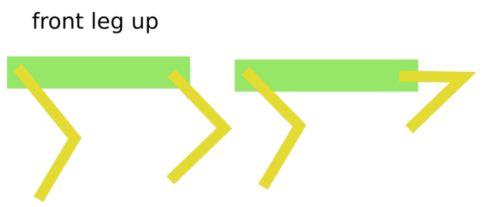

Figure 4: walking model

Example A good example for a naive physics engine is the "Soccero Board Game", it is a soccer simulation for the tabletop and works with dices and movable player position. The main advantage over a normal physics simulater like Box2D is, that Soccero has some rules. That means, the player can only move in a certain way. This reduces the state space. The soccer game is tranformed into an easier to manage game, and the gametree can be searched completely like the chess game can be searched. Transforming a domain into a board game is equal to a naive physics engine and this is equal to be able to program an Artificial Intelligence. Another example is the Formula D boardgame, which comes from the domain of racing and also helps to reduce the state space.

1.6 Intuitive Physics engine for a walking robot

An intuitive physics engine has nothing to do with a mathematical model or differential equations, it is instead a physics engine which receives commands from the environment and is able to predict the outcome. In the figure "walking model" a game is presented which contains of two legs. Instead of using a sophisticated Box2D engine, the physics engine was self programmed, but the model can parse text commands like "front leg up". After receiving such a command, the physics engine shows a reaction, in this case the fore leg position is changed. It is simply an animation system which allows the designer to try out some movement and see how the result will look like.

The most interesting aspect is, that such model isn't an artificial intelligence, it is more an animation simulator. That means, it is created from scratch, contains a class and in the class the angle of the two legs are stored. Additionally the output is graphically to get a better impression. And for the environment, the physics engine provides some basic commands. What the user can do is sending a command to the engine, and then he sees what the result is.

It is obvious, that the engine is not very accurate, because according to the physics laws, the resulting second picture isn't possible in reality. The biped robot would fall down, but in hour model such a condition works. That means, there is a gap between the intuitive physics engine and a realistic box2d physics engine.

The reason why such a intuitive physics engine is valuable has to do with a simplified state space. If the engine contains variables and a fixed amount of commands, then the number of possible states is limited. There are no longer millions of possible ways the legs can look like, but only a limited amount of them. In our case the physics engine can interpret only one command, but it is possible to extend

[1] `http://buildnewgames.com/gamephysics/`

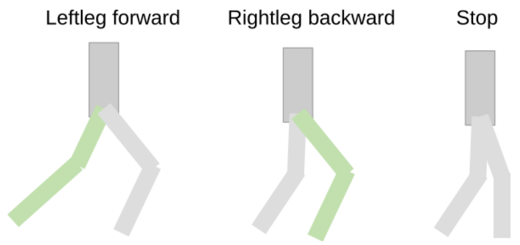

Figure 5: walking physics engine

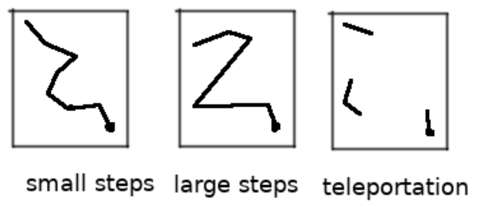

Figure 6: hierarchical pathplanner and graph grammar

the model so that 10 commands are understood. And now comes the magic. If the engine works, it is possible to script a walking gait. It is simply a number of commands which are send in sequence to the physics engine, and this will make the model walk.

Such a script can be interpreted like a trajectory through the state space of the newly created game. The game is invented from scratch, it is called “physics engine” and contains the variables and commands the programmer need. Or to explain it from a different angle: it is a high-level task planner for testing out possible plans. The pipline for creating such a model is remarkable easy. The original vector graphic was designed with the inkscape software, it is simply a concept drawing. After drawing some examples, the next step is to program the engine in a high-level programming language like C/C++. And that is our walking model.

1.7 The benefit of a qualitative physics engine

The best practice method for a robot control system is a physics engine from scratch which is tailored for the domain. The principle is called in the literature a “naive physics engine” and is equal to a predictive simulator. But why exactly should somebody invest his time into programming such an engine, if this new engine is different from the physics which is happening in the normal simulator?

Let us investigate a given physics model in detail. The figure “walking physics engine” shows a model which contains of a torso, two legs and some commands the engine is able to parse. The user can enter an action, and the physics engine will move the legs. What the engine is doing is mainly to convert a textual input like “leftleg forward” into a mathematical notation which changes the position of the legs. On the first look, the model is useless because we want no animate a ragdoll body which is hanging in the air, but a robot who walks in a Box2D physics engine. If we are executing the action “leftleg forward” in a physics simulator like Box2d the robot will not walk, he will loose his balance. So what is the deal, why is the naive physics engine right?

It is important to distinguish between taskplanning and motion planning. It is correct, that the naive physics engine is not a motion planner, the system was never developed to control the servo motors on a lowlevel layer and balance the robot. The better name is a symbolic physics engine, because it gives abstract subgoals. Let us investigate the first frame after the command “leftleg forward”. What the engine is telling us is only how the general idea is. That means, what the robot has to do in the next 2 seconds. The engine is telling a story about which leg has to moved, and how the system will look after the action was executed. A separate motion planner is needed to convert this high-level action into a lowlevel servo command. That means, we are developing the taskmodel for the reason to develop the motion model next.

It is right, a high level task model is useless without the lowlevel motion model. But without a high-level physics engine, we are not able to figuring out what the subgoals for the planner are. A well description for a native physics engine is to call the device a subgoal generator, because the model provides a rough constraint.

1.8 Hierarchical pathplanning

A standard pathplanner is able to parse a sequence of walking commands. The robot can execute the following plan:

up, up, left, left

And the planner determines the position of the robot the last “left” command. The difference between pathplanners like A*, RRT and naive brute force is how they store the sequence of possible plans in a graph. For example, a naive brute force solver doesn’t have any kind of graph, he simply tries out a random walk until the robot is in the goal.

The problem with all pathplanners (including the advanced RRT algorithm) is, that they fail on bigger maps. If the maze in a game is 1000x1000 pixels huge, or bigger then the pathplanner will run into trouble. The generated graph is to huge to calculate it in realtime. The standard technique to speed up the pathplanning is a concept called navmesh. The maze is separated into zones and in each zone the path is planned separately.

To understand the concept of a hierarchical pathplanner better we must look on the possible plans which can be parsed. A normal non-hierarchical pathplanner accepts only a path like “up, up, left, left”. The action model is, that the robot is in a maze can move in four direction for only one field. A navmesh hierarchical pathplanner will accept a multi-modal plan like this one:

up, zone2, down, down

That means, the robot can move one step in four direction but he can also teleport between the zones. The hierarchical pathplanner is able to understand one more command, the zone action. This additional command helps to reduce the state space drastically.

Instead of talking about pathplanner it is time for figuring out action models. A simple action model for a robot in a maze is to move only in four direction by one step: left, right, up, down. A more advanced action model is, that the robot can move larger distances. Here the action model understands commands like “10left, 10right, 10up, 10down”. And the most advanced form of an action model is, if the robot can addtional teleport himself between the zones and bypass the normal laws of physics.

Implementing a reduced action model which contains of 4 possible moves is easy. This is done by every pathplanner. The disadvantage is, that such an action model will fail in larger maps. The more advanced form is an action model which contains more possible actions on

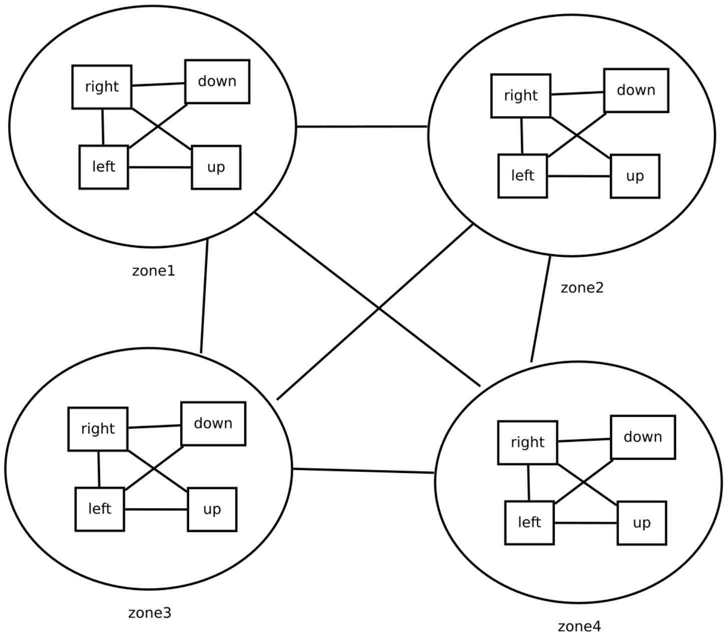

Figure 7: graph grammar

different layers. This is equal to implement a heuristics. The rule of thumb is given by the action model. Such a system can be solved by a STRIPS like planner in a hierarchical fashion which saves a lot of cpu ressources.

1.9 Knowledge based simulation for decision making

A short look into the debate around Artificial Intelligence in the year 1990 comes to the conclusion, that early AI researcher have understood the importance of knowledge. The idea in that area was to use Prolog and LISP to describe knowledge for a machine with the aim to program intelligent software. Unclear was, how exactly Prolog can be used for modeling heuristics in a machine. The answer is simple and is called "knowledge based simulation". That means, the expert system has to be designed as a physics engine for predicting future states. Knowledge means, that the simulator is able to calculate what comes next. A simulator for a jumping robot is able to predict the landing zone, that means in which x/y coordinates the robot will jump onto. And a simulator for an autonomous car will predict what will happen, if the car is driving on a road.

Formalizing knowledge without simulating the process isn't possible. It is only the pre-step for such a system. For example, a textual requirement can be a note, about rigid body physics. But that note isn't the executable software, the requirements has to be translated into sourcecode. Knowledge engineering means to write a simulator for a domain. That can be anything: a pick&place task, a car, a walking robot or a UAV. There is no difference between programming a physics engine and programming a knowledge based simulation. In most cases, LISP and Prolog are not the perfect choice in doing so. In theory, it is possible to program a physics engine in LISP but object-oriented langauges like C++ are better suited for that task. They were developed for simulation purposes in mind and they can be executed faster on standard hardware.

Usually, a physics engine is not recognized as an AI system. But it is the most powerful example available. A physics engine contains all the heuristics about rigid body dynamics. The only problem with today's physics engines is, that they were not programmed with AI in mind. So there is a need for implementing better engines from scratch. The major difference is, that AI capable engines are able to predict larger horizons and that they are containing layers which makes the search in the state-space easier. The overall principle is the same. Like a physics engine in a game, it is able to send a plan to the engine and simulate what will happen. The impressive advantage over behavior trees and simply AI scripting techniques. is, that it is possible to send a malfunction plan to a physics engine. A possible plan for a car simulating engine could be, that that car should drive even the signal on the road is red and the engine will predict what will happen then.

Such feature allows the planner to generate a game tree out of the physics engine. He can send random plans to the engine and create a graph with the outcomes. Every node gets a score and this will give back the best possible plan for every situation. The technology is used in the domain of computer chess since years, but can be adapted to robots as well.

From a mathematical standpoint, knowledge based simulation is a very elegant method for realizing Artificial Intelligence. The basic idea is to simulate a domain with a hierarchy of physics-engines and a solver can try out different possibilities to find the best plan. Realizing such an architecture is not a theoretical problem but depends on software engineering. It is only a question of coding and then the system will work.

Computer simulation is well known since many years, but it's capabilities are underestimated. Usually such systems are not recognized as Artificial Intelligence, but they are the key element. The basic idea behind Computer simulation is, that the system contains some variables, for example a position of the robot, and methods will change these values by executing commands. A simple action might be "walkright", which changes the x-position of the system. Realizing such simulating systems in software is easy: the C/C++ programming language with the object-oriented paradigm is the best choice, but any other modern language like Python, Java and C# are capable in doing so too. The most remarkable aspect is, that a computer simulator has no intelligence in the classical sense. Because the physics engine doesn't tell the robot how to walk, it can only simulate what will happen, if the leg is moved forward. The missing feature of transforming a simulator into an AI is called "AI planning". Here is the idea to utilize a given physics simulator and figuring out random plans. This can be done with well known search algorithm like RRT and A*. The reason why this is successful has to do with the fact, that a physics engine provides a reduced state-space as default. If the engine only accepts two commands "walkleft" and "walkright" the gametree isn't very huge and a brute force solver is able to search for a plan.

I'm not the first author who promotes knowledge based simulation for realizing artificial intelligence. Since the 1980's many attempts were undergone for realizing such systems. In all cases without success. All the major automation projects for improving manufacturing or simplifying computeranimation for making movies ended in failed projects. The reasons why are diverse:

- wrong programming languages (for example LISP)
- slow computer hardware. A typical workstation in the early 1990s was not able to do anything useful.

- lack of communication in the software engineering. Most projects in that time didn't know distributed version control systems for producing software in teams.
- misunderstanding about computer simulation in general
- inflated project goals like automating a complete factory together with limited resources

At the end, it is normal that all knowledge based simulation projects of the past failed, and they will be wrecked in future. But the theoretical concept is right and it is only a question of the details how to realize it.

1.10 Everything about a Physics Engine for bouncing balls

The gaming community is remarkable well informed about physics engine. They are not only able to develop working code for bouncing ball systems but the engines are often much more advanced and can simulate rigid body physics. The problem with this enthusiasm is, that the energy is invested in the wrong position because the question is not how to program a more realistic physics engine, the question is how to implement a different kind of engine. But let us start with a simple example, so previously mentioned bouncing ball example.

The key facts are explained fast: a ball can move in a grid and after colliding with the walls or an obstacle the movement direction changes. Realizing such an easy physics engine can be done by beginners. All what we need is a class ball, which contains a position, a speed and a direction. The class needs further a method, called "update()" which determines the next position of the ball. In the gameloop we are calling the update method every frametick and visualize the position graphically on the monitor. Thats all, our bouncing ball simulator works.

If someone is interested in the concrete sourcecode, I'm sure that stackoverflow and github can help. They have at least 100 repositories with bouncing ball simulation and every month some new beginner program their own variant. And now comes the difficult part. To improve the simulation we must program a slightly different physics engine. The new requirement is, that apart from the update() method a more advanced method has to be implemented. To understand the needed feature in detail we must go a step back and describe what our physics engine from the introduction is able to provide.

The normal physics engine works in the single step mode. We call the update() method, and the ball position is calculated for the next frame. The aim is to use the new ball position for drawing the ball onto the screen. IF the gameloop produces 25 frames per second, the update() method in the physics engine is called 25 times per second. Such a physics engine is tailored for the need of a graphics card.

In contrast, a qualitative physics engine has a different goal. Here is the idea to provide what if scenarios with different time scales and different init condition. A possible request to a qualitative physics engine might be:

- what happens, if the ball is in the center and moves to the left?
- what if not one but 100 timesteps are executed?
- suppose the ball is on the left corner, has a small speed and a direction with 45 degree. Will it hit the wall in the next 2 seconds?

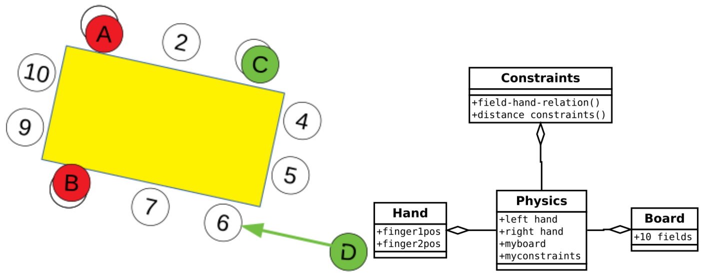

Figure 8: Model for handover task

That are some example questions, a qualitative physics engine has to answer. Like the normal physics engine, it should also answer the request "what happens, if we are going only to the next frame and let all parameters unchanged". It is clear, that the advanced form of a physics engine works more like a interpreter which is able to execute a domain specific language. Instead of a single command "update()" it provides much more methods. Implementing such a physics engine is possible from a technical perspective, but it is seldom realized because most developers don't see the advantage. The reason number one is, that a qualitative physics engine can work hand in hand with a solver, and the solver is executed by a BDI agent framework. The agent can send a request like this one: "what sequence of actions is needed to bring the ball in the left corner?" Answering this question can only be realized with a solver pipeline. That means, different simulations have to be run after each other and a scoring method is used to determine the best plan.

1.11 Modeling a handover task with two robot hands

That models are important in robotics is widely known, but how exactly can such a model be created? Do we need the QSIM software, neural networks or the R programming language? No, in it's easiest form a so called model is a vector graphic created with Inkscape. The file is exported into the PNG format to insert it into a document. What we see on the picture is an object with annotations. It is like a board game because the robot-hands can take a certain place relative to the object. The places are numbered by 1 until 10. The

left hand is symbolized with 2 red fingers, while the right hand is colored in green.

The functionality of the picture is limited. It's not a robot controller and it's not possible to insert the picture into sourcecode. The idea is only to modeling a prototype. That means, to get an idea how the system would works, if it is implemented. What the model provides is a notation. We can say for example, that the fingers of the left hand are on position 1 and 8. And now the goal is to bring the right hand to the position 3 and 6. Without the picture, such a description don't make any sense, but in the context of our newly created board game, it is clear for everybody what the meaning is.

In my opinion, such drawn models are the key concept to develop advanced robotics systems. It isn't a new modeling language, it isn't a mathematical concept, it is only a vector drawing which explains a board game to a wider audience.

The next step after creating the picture is twofold. At first, the picture has to be converted into executable C++ code. This can't be done automatically, it is a manual process. The programmer takes the image and creates a class for storing the positions and creates another class to bring the values onto the screen. The second open question is the animation of the handover task. We must define, in which sequence a task should be done, and which potential failure can happen. This is only possible, if we are extending our model with constraints, for example to prevent that two fingers can be on the same position. It is possible to convert the drawing into a pddl description, which is a declarative description of the interaction with the environment. A pddl file can handle textual commands, which have to be send to the model.

Why a model? Perhaps we should go a step backward and discuss, if an abstract model is useful in general. Without a model, the task of handover an object is defined by the robot itself. The robot has overall 4 fingers, A, B, C, D and each of them can have a position. That is the complete model. The task is to move the fingers to the right position over the timescale and this results into an handover. From a hardware perspective, this description is correct, because the robot has indeed 4 fingers which have coordinates. But it is naive to assume, that we are able to handle the state space without an explicit "handover model". That means, it might be possible to control the fingers, but we will never get a useful result. The number of potential movements is endless and a model-free reinforcement learning algorithm is not able to realize the handover task.

Implementing a model in C/C++ Until now, the question was left open how to convert a graphical model into executable sourcecode. A C/C++ program contains of classes, variables and methods, but can't be programmed graphically. In contrast, the output of a drawing program doesn't know a programming language but only the PNG format. So what is the deal? How can we overcome the gap?

A graphical model is a game description. Such a game can be realized as a boardgame, as a card game or as a computergame. The game contains usually of a description in natural language and some drawings. The transformation of formal game description into executable computercode is not a problem. Nearly all kind of boardgames have been realized as a computersoftware too, for example backgammon, monopoly, chess or reversi. And if a new boardgame was invented it is only a question of time until the computer version is available. So i would guess, that not the transformation into computercode is the problem, but the invention of the game itself.

The problems with formal models from robotics domain is, that they are usually very complex. The above description with the handover task contains 4 fingers, 10 possible positions and an object, but it is only a prototype. Such a model has to be described in detail to make any useful decision with it. The question is: how many pages do we need to describe such games, 10 pages us-letter, or 100 pages with lots of figures? That is unclear. A well working robot model is similar to a complex board games, with hundred of rules and lots of subgoals. Inventing such a game from scratch isn't easy. A working example, which is very complex is called Knowrob and was realized in the ROS ecosystem. It can be compared with a highly developed board game with endless rules and constraints.

To understand the difficulty it is a good idea to imagine a new domain, which has to be modelled as a board game, for example robot-soccer. Drawing the board, the players and describe the rules can be called a large project. Even if no lines of code has to be programmed, the board game itself has to be written down.

Model syntax A formal model is not connected to an ontology, to the Simulink Simscape multibody software or to a deeplearning architecture. The easiest form of a model contains of a concept drawing, a UML chart and some textual description. The general idea behind a model is not the model itself, but a development phase in the software engineering pipeline. Modeling is the step in which a requirement is transfered into prototypes. The constraint under which modeling takes place is given by a time frame. Modeling can take 1 week effort, or one year of effort. It can be done by a single person or 100 persons. In all cases, the result of the modeling phase is a PDF document. It contains figures, natural descriptions, UML charts and pseudocode. The model itself is the pdf file but not executable sourcecode. This will created in a different step after the modeling phase.

At the same time, a model can also be described as a mathematical model, aka a game. A physics engine is a model of the reality, it is used to simulate something. Bringing both concepts together is easy. Modeling means, to create a prototype for a physics engine. That a preliminary steps as preparation for the programming itself. We can call this modeling a model, because the idea is to draw the UML chart for a qualitative physics engine.

1.12 Central pattern generators for biped walking

A central pattern generator (CPG) is a small neural network which is trained for biped walking. Usually the overall network has not more then 30 neurons. It can be any type of neurons, but in most cases recurrent neural networks are used, because they can express more complicated input-output relations. But how exactly works a CPG?

According to [9] a neural network produces a state-space trajectory. To express it more colloquial, the neural network is able to draw a spline curve. Sometimes the term oscillator is used because it looks similar to what a sinus-oscillator can do: it draws curves. A biped motion can be seen as a walk through the statespace. A sequence of joint-angles controls the servo motor and this produces a motion. The problem in biped walking and any other robotics task is, that the state-space is very huge. There are without exaggeration billions of possible trajectories in the state space. And trying them all is not possible because it would takes years.

How it is possible that some pattern generators are able to produce a natural walking gait? What learning algorithm were used? The answer is called model based optimization. A model is a heuristics to reducing the state space. Perhaps an example would help to understand the case better.

At first, a motion capture device is required to record two sequences of angles:

sequence1: 10,20,10,30,20
sequence2: 20,10,20,30,21

Now we define, that this is our model. Our model contains two possible motion sequences. The only choice the controller has from now on is to switch between the sequences. He can chose to run sequence 1 or sequence 2. That means, our state space is reduced to a selection of a given trajectory. Now we can train a neural network to follow the state space. It is an easy task because the number of potential decisions is small. That is the basic principle behind any central pattern generator. At first, a model is constructed (often with the help of motion capture devices) and secondly this model is controlled by a neural network. The overall workflow can be called an adaptive model.

Let us go a step backward to understand the principle in detail. The major problem in any neural networks is to find the parameters. Without the parameters, the neural network will produce only a chaotic pattern. The parameter problem is difficult to solve, even for smaller neural networks which contains not more then 30 neurons. And to be honest: in the domain of central pattern generator no one has solved the parameter problem really. If we are connecting 30 neurons together and want to train this system for doing a practical task, it is not possible, because there a billions of possibilities how the weights of the neurons can be selected. Even on modern nvidia deeplearning hardware and with modern back propagation learning algorithm it would takes million of years.

To overcome the problem, the state space has to reduced drastically. Only a subpart of all possibilities makes sense, and the question is which part exactly. Reducing the state space is equal to create a model. In the domain of biped walking the model has to do with a walking character. That means, the legs can't move in any direction, they can only move in a certain way. And now comes the misconception: the neural network or the CPG isn't able to create the model. The model has to be given before the network is trained. The model is taken from a medical book or from animation course. This model gives detailed information about how legs can move and which movement is not allowed. A biped walking model is equal to a board game which has rules.

This given model will be extended by a neural network. That means, some open parameters in the model are adjusted, That means, the neural network is controlling the model and decides minor parameters. The learning process is easy, because the number of possibilities is smaller. So the model can be trained to walk slow or to run fast.

At the end we investigate the worst case. We are taking the biped model out of the loop and only use the 30 neurons network. Can we solve the problem? No, from a technical perspective the CPG is able to produce the needed pattern, but we are not able to find the correct parameters. We can only use trial and error until the network will find a walking gait. Like any beginner of the OpenAI gym framework knows, this won't work, because after some iterations the network will not increase the increase the performance. The state space is too huge, it will never find a path through it. What i want to say is, that a central pattern generator without a model of a walking robot is useless. The other way around makes more sense. We can use only the walking model and ignore the central pattern generator, and it is also possible to program a walking controller. Instead of a neural network it is possible to use a normal sinus-function or a PDDL Solver to find the right parameters in the models. Because the keyfactor in reducing the state space is the model but not the CPG.

CPG Model What a cpg is, is very simple. It is a neural network which can be implemented in tensorflow or any other neural network. But, this cpg neural network will never work, and it won't produce any walking gait. The most papers are not about the cpg itself, but about something which is called a "CPG Walking model". The difference is, that a cpg walking model contains much more then only 30 neurons which are connected together. It is more a detailed description of the walking process. For example, there are legs and arms, and the walking process is divided into steps. Such a animation model has nothing to do with a central pattern generator, it is the result of a motion capture capture anlysis together with an understanding of biped motion. The funny aspect is, that CPG and a walking model is often combined together. The reader gets the impression that the neural networks control the servo motors, but in reality it is the walking model.

It is important to separate between them. A cpg aka neural network is a spline drawing system. It can parse huge amount of data and can adapt the paramters to emulate the spline. But, using CPGs or neural network without an additional model will fail. Because there is no learning algorithm known which convergates the neural network into the desired state. In practial application this results into networks, which can't reduce their error rate. This is especially true for CPG like networks.

For getting better results, it is important to give the neural network a prestep. Instead of learning the raw data, it learns the paramters of a model. For example, if the model contains action primitives like "leftleg up", "rightleg down", "hip up" it is possible to use a neural network to generate these commands. And it is also possible to give these commands a simple parameter which can also be learned by the network. The trick is, that the model itself has only a reduced state space of less then 1000 possible sequences and that it is possible to test them all which is equal that the neural networks learns its parameters. Again, the idea is not, that the parameters of the neural network are adapted to the raw data, but only to the parameters of an existing walking model.

The capabilities are not determined by the neural network itself but by the biped model. If the model is able to climp on stairs, then the CPG can produce also these signal. And if the model is very simply and only supports walking ahead, then the CPG is limited to that feature.

sinusoidal pattern generator A more simple to understand pattern generator is called sinusoidal. Instead of a neural network a sinus-function is required. Some papers claiming, that with this method it is possible to control a biped walking robot. So the sinus-function must be a powerful Artificial Intelligence powerhouse, right? In reality, a sinus-function isn't able to drive a robot, because sinus means only a mathematical term like this one: $sin(x+2)+sin(x)$. Such a term has no artificial intelligence inside and it is not possible to control a complicated system with it. The reason, why it is possible to use a sinusoidal pattern generator has to do with the walking model. This has nothing to do with the sinus-function but it is a semantic description what walking means. For example, walking consists of two legs, which are connected with the torso, and there is a certain sequence of movement, called gait-pattern. Running this model will result into a movement.

What in the papers about sinusoidal pattern generators was described is a combination between a sinus-pattern generator and a linguistic walking model. Both combined together is indeed capable of controlling a robot. The model is even robust under certain conditions and can adapt the walking style to new requirements with a learning procedure.

What I want to express is, that it makes no sense to focus on a sinus-generator, on neural networks or on central pattern generators. All of these tools are very boring and in case of doubt it is possible to not use it. The more interesting part is the semantic walking model. That is everything apart from the pattern generator. The walking model reduces the state space into a smaller one and this is the core feature for robotics control.

time	servo1	servo2	servo3
1			
2			
3			
4			

Figure 9: motion capture for walking gait

1.13 Reducing the state space for a walking robot

Programming a walking robot is advanced but possible. That means, somebody has done so before. The main problem is the state space. If the robot has two legs and each of them has three servos, then we need 6 servos in total what is from the hardware perspective no real problem, but from the software side will the question arise who to control all these servos in realtime. A single servo motor can have a position from 0 to 360 degree, and six of them will multiply the number of possibilities. The number of states in a single step can be handled it is only 360^6but what is, if we want to plan a sequence of 10 steps and each of them can have all possibilities. And this is the main problem in robotics, that this state space is way to huge.

Classical algorithms like neural networks backpropagation, genetic algorithm and RRT planner are not able to search in this state-space for a certain plan. The good news is, that the answer is there. It is called heuristics and the idea is to reduce the state space with a domain model. The walking process isn't chaotic process it can be described. Such description are used in computeranimation and they are textual and visual. A possible description is a sequence of possible movement steps, the biped robot can do. Let us investigate the most simple walking model. We are recording the servo angles with a motion capture device and creating a table. And now we are saying that the recording angle are exact the correct angles in our model, it is a fixed trajectory to simplify the situation.

From the perspective of state-space everything is better. Instead of searching in an endless space of numbers, the number of possibilities is small. To be honest, the model knows only 1 possibility (the recorded motion trajectory). And the playback of the model is to use the value of the table and set the servos to that parameter. That means, the planner has nothing to do, no cpu time is wasted.

Sure, the model will make a lot of trouble because a simple playback isn't able to deal with distortions. So we can think about how to make the model a bit more flexible. One example is to use random variation, that means we let the parameter fluctuate a bit, another option is the use paremeters from servo a for servo b even this wasn't given by the recording. In all of these cases, the state space remains small. Even with fluctuation the number of potential walking gait will be smaller then 100.

What modern robotics is about, is figuring out how to develop better models. Somebody come to the conclusion, that the gait model has to be programmed from scratch, other are trying out to record given trajectories and so forth. The shared goal is to reduce the state space with some kind of model. The holy grail is a model which contains not more then 1000 possibilities and after playing around with the free parameters the robots walks straight ahead.

The interesting fact is, that it is not important which optimizer or solver is in the loop. RRT, neural networks, genetic algorithm and so on are working all great, if the state space is small. Even a brute force solver who is testing out all parameters will result into a walking gait, if the underlying model was well designed.

Every model works with the same principle. It maps input values to output values.In the given example with recorded mocap trajectories, the input value are the raw data of a trial, and the output data is the newly created table. A good model is grounded semantically, that means major terminology from the domain is used as action primitives to control the output of the model. In the case of a biped robot, that would be a command like “left leg up” which produces a certain gait pattern. A general rule is, that complex models are difficult to program, while simpler (fixed) model are easy to program but they are not robust against different scenarios.

1.14 States and constraints

A domain model can have a state. For example, a car can be on the start position. All planning systems are trying to bring the system into a goal state. A state is something which is happening at a time step. The transition between states can be specified with constraints. For example, a goal state of the car is the parking loft. And the constraint is, that the car shouldn't collide with any obstacle. If the constraint is different a transition from start to goal state can be altered.

A model has a process flow, this is called a simulation. The flow has to do with running the game loop which increases the frame counter. Visualizing this is possible on a time chart. x-axis is the timescale (0 seconds, 1 seconds, 2 seconds) and y-axis is a value from the system, for example the car position.

Model based exploration A domain model is created with a special need. The main purpose is, to use the model as an exploration device for searching future states. A simple domain model which can the car only in four direction will a have different exploration behavior than a complicated one. The term exploration means what the model can do if it is run by a random generator. Suppose the model has four possible actions: left, right, up, down. Exploration means, to try out different plans with that actions. A potential sequence could be: left, left, right, up. If we are testing out many thousands of these sequence this produces a graph, which is equal to the game tree.

What every model has to offer is a way to explore much of his environment. It is not only a problem which can be summarized as RRT, but it is the question of the model itself. RRT assumes, that the model is given. That means, that are only 4 directions possible. The better approach is to extend the original model by new possible actions, for example with “long left, longup, longright, longdown”. Sometimes this is called multimodal planning, because now the planner has in total 8 possible actions to execute. This allows him to explore much more of the map.

Why is exploration important? Because this allows us to search for a goal. If the goal is to bring the car into the left upper corner, then the logical step is to explore this movement first in a simulation. If the planner has found a valid sequence of movements, they can be executed. Let us define a simple example to show in which cases the model isn't powerful enough.

Suppose our car-model has only 4 possible movements: left, right, up, down. Each action moves the car for exact 5 pixels into the direction. Now we define as a goal to bring the car 10000 pixels away from the starting position. The problem is, that the exploration capability of the model isn't ready to fulfill the task. Even with creating a RRT graph, we won't find the path to the goal. Only with unlimited cpu ressource we are able to plan a path to the goal.

But what is, if the car has two modes: at first the well known small movements steps to adjust the positions and then a movement to beam the car in one of the cells of the maps. This converts the planning problem into a hierarchical one. At first, we are searching for a path through the cells, and then we can investigate how to come to each cell. Our model was extended with new features, which allows to explore better the environment.

RRT action model A vanilla RRT algorithm has the aim, to extend the graph in all directions. A new node is add to the graph with a uniform random sampling method. But, RRT makes no assumption about the action model, that means in which direction the robot can explores his environment. Understand the multimodal extension is not easy. Perhaps a small example from the pathplanning domain will help.

Many game designers have recognized that the vanilla RRT and A* algorithms are not very useful in planning on larger maps. They will generate millions of nodes and this takes to long. What in games are used today are nav-meshes. That are heuristics enrichted pathplanner. The high-level layer works on cells, and only the lowlevel layer works with a classical pathplanner. At first, the robot is trying in which cell the goal is, and then he plans a way from the center of the cell to the exact position. This helps to reduce planning costs and makes it feasible to planning paths on huge maps. From a theoretical point of view, a combination between highlevel and lowlevel planner is called a multimodal planner. Because the system has two layers. The robot can execute a command from the high-level layer (move to a cell) or he can execute an action from the lowlevel layer (move left, right, up and down). The terms multimodal and hierarchical layer planning are talking about the same.

In very easy planning task like planning the path on a small map, it makes no sense to implement a multimodal planner. A simpler vanilla RRT algorithm will solve the problem much easier. But in large maps and especially in complex robotics task a multimodal planner is indispensable. Let's describe a robotics task which only can solved with this high end layer planner.

A robot should jump on a wall, move to the box, switch to the other side of the box, and push the box downstairs. This task is more complicated then simple reach a certain point on the map, instead the robot has to do a sequence of actions. From a input perspective the game is surprisingly easy, because the robot has only 4 possible movements: left, right and jump. The problem is, that these simple commands generate a complicated state space, which can't be explored with a vanilla RRT algorithm.

Pushing a box Suppose a robot and a box are both in game available. An RRT algorithm is exploring the potential movements of the robot. Because of the uniform sampling the algorithm will explore every position the robot can have in the maze. But, what happens if the robot pushes the box? This is a second state space, which wasn't explored by the RRT sampler. From a previous game play it is known, that is possible to move the box in any corner of the maze. But the RRT planner will not recognize it, because he only samples the positions of the robot,.

1.15 Modeling is easier then expected

The term modeling is usually used as a synonym for deeplearning models or for differential equations. In the Scilab software it is possible to create with Xcos package so called models, which can be used for simulate multi-body physics and inverted pendulum. But what is the essence of a model? A model is equal to an object-oriented model, because this is the language which can be implemented as computercode. Even complex mathematical formulas and non-linear systems can be expressed with the UML notation language. Let us give a short example: a walking robot. The robot has two legs, each leg is a C++ class. The class contains the angle values for the servo motors. Additional we need some methods to manipulate the leg-class e.g. setting up a value for the gait pattern 0, pattern 1 and pattern 2, which means to store absolute values of the joint angle to the class.

What can we do with this model? Surprisingly very much, at first it is possible to draw the content on the screen. The result is a nice looking visualization of the legs. And then we can execute a gait-pattern, which results into an animation. That principle can be transferred to any other domain outside of a walking robot. The basic idea is to use an UML editor and create a model like developing a software program. This principle is called object-oriented modeling of physical systems. The idea is, to use the UML notation not only for describing a business application or a standard computer program, but for describing the working of non-linear physical systems.

The concept behind a model is to transform high level natural language into low level numerical data. The user sends to the model a command like "leftleg forward" and the model converts this term into a value like {20,10}. A model is equal to an abstraction. It simplifies the situation.

1.16 Model based Central pattern generator for biped walking

The major problem in Artificial Intelligence is called state-space explosion. It is a situation in which the game-tree of a domain contains billions of nodes and solving the game means to find a specific node. If somebody is able to master huge state-spaces the Artificial intelligence will work. Let us take a look into literature how so called Central pattern gait generators have answered the state space problem.

The first impression might be, that the neural network is the part of the system drives the walking robot. In a central pattern generator the network contains of neurons which are connected together and a learning algorithm is used to determine the parametern. Learning means to reduce the state space, so the CPG is an answer to that problem? Nope. Nearly all working biped robots are working slightly different. The first step in implementing such a system is to create a so called model. In the case of a walking robot a model is equal to the angle trajectory. An angle trajectory is a chart which describes the joint angle for each timestep. On the chart the keyframe points are given, which a connected with a smooth spline. If we are modifying the curve, the walking pattern is slightly different. If the robot walks slow, the chart looks different from running.

The trajectory spline together with a parametric adjustment is the major tool for reducing the state space of a walking robot. Only for detail adjustments like determine the right parameters a so called solver is needed. That is a software module which works with genetic algorithms, brute force search or with neural networks. A central pattern generator is equal to such a a solver. That means, a CPG isn't controlling the robot directly but it is only the solver for a previously created model. Leaving out the CPG is possible, but not implementing the model will fail the project.

The main problem with optimization techniques like neural networks, genetic algorithm or stochastic search is, that they only efficient for small state space. Small means a number of possibilities lower then 1000. None of todays robotics problems like walking, grasping and autonomous driving has a small problem space. That means, it is not possible to use any of these optimizer directly on the problem. Neural networks are surprisingly bad in solving problems with a huge state space. What the user will see is a learning rate who stands still, that means, the neural network isn't able to improve his performance. The problem has to do with the number of possibilities in the state space. There is no magical learning algorithm available which is able to search in a huge amount of space.

1.17 Model-based game tree search

Artificial Intelligence can only be realized with game tree search. The graph has to explored for potential solutions and the node with the highest score is the goal. The question is how to search large game trees in short amount of time. One possibility might be the RRT algorithm which is exploring the state space in all directions. But, RRT fails for larger problems and uniform sampling is not enough. A potential alternative is to use a model for searching in the game tree. Such a model is a equal to a heuristic. To explain the principle let us take the example of hierarchical pathplanning. A standard pathplanner can explore only nodes which are in the neighborhood of the robot. If we extend the model with a new option to jump somewhere to the map, larger parts of the game-tree can be searched in a small amount of time. The options the robot has to move in the maze is equal to his behavior model. Improving the behavior model results in a better heuristic and this allows to search the game tree more efficient.

In the literature the concept is called multi-modal planning and hierarchical task networks. In both cases a model is used. Instead of sampling lowlevel actions, synthetic macro-actions are sampled. What most literature didn't answer is how a model looks like for solve a certain domain. Some authors are trying to generate a model from the input data on the fly, but the more promising method is to use classical software engineering techniques like concept drawings, prototypes and object-oriented programming for realizing models.

Perhaps some example for behavior models from different domains. A walking robot needs as a behavior model a prescripted gait animation which contains walk patterns. If that model is parametric it become very easy to search a large state space with it. An autonomous car isn't able to profit from that kind of model, it needs a different one. This time the model is equal to a symbolic traffic simulator which is able to generate what-if-cases, for example: "what if the car is driving and the light is red?", "What if, the car wheel is left?" and so on.

In most cases behavior models are equal to mini-games. They are simulating aspects of reality. The question which has to answered by the programmer is, how to generate mini-games in a short amount of time. The previously called techniques (concept drawings, prototypes and OOP) are one example for it. PDDL and ABL are another example for a prototyping languages to create a mini game. Such a mini-game has the purpose to speed up the game-tree search. It is a formalized heuristic

.

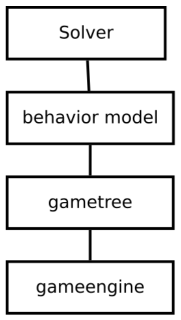

Figure 10: layered architecture

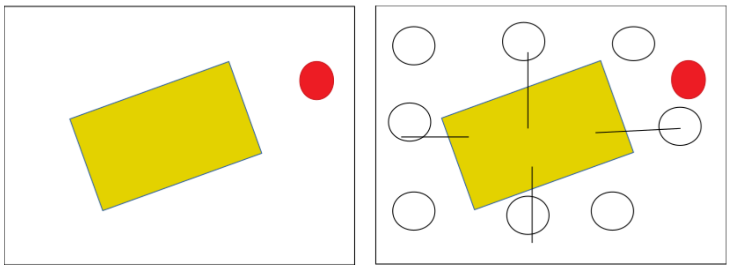

Figure 11: robot in maze together with a box

Example box pushing Suppose a robot is in a maze and has the task to push a box to a certain position. The first naive idea is to describe the problem as a state space. The robot can move and every action results into a node in the state-space. Pushing the box is equal to wander around a path in the game tree.

We can call this approach not very practical because a simple calculation will show, that the state space is too big to search it in realtime. If the robot has to execute 20 moves, the number of potential movements is endless. The answer to the problem is to think about a behavior model for the robot. This is shown in the figure right on the screen. The task is the same: push the box. But this time we define, what the robot can do on a semantic level. At first he is able to gain a absolute position on the map. .This is symbolized with 9 circles. That means, the robot can decide to go to one of these position. And secondly the box can be pushed in 4 possible directions, this is symbolized with lines. The behavior model contains of two possible actions:

- moveto #1 until #9
- boxpush north, south, east, west

A possible plan might be to go to position #3 and then push the box to the right. A different plan might be to go to position #2, then to position #3 and then push the box to south. With the newly invented semantic behavior model it is easier to define possible exploration of the state space. The number of possibilities is reduced, and the actions are defined on a high-level-layer.

Task and motion planning In the literature many names are used to describe model based gametree search. For example, multi-modal planning, Hierarchical task networks or in newer literature often the term "task and motion planning" is used. [3] for example, describes a forklift who navigates in a maze and pick ups boxes. The system contains a geometric lowlevel planner and a high-level symbolic planner.

1.18 From action planning to video surveillance

Most problems in Artificial Intelligence are synthetic problems to reduce the failure rate. Instead of modelling a humanoid robot the idea is to program a pathplanner which finds a way through a maze. Even the synthetic challenges are sometimes to complex to solve them with today's algorithm. On the first impression it makes no sense to increase the difficulty but it is important to be aware of a potential alternative, called video surveillance. Here is the idea not to invent an artificial problem which can be solved by robots, but take a look on real situations which are solved by humans.

The difference is, that usually humans are aware of how to play soccer, drive cars or work in a factory. They are doing tasks all the time. Unclear is only what the machine translation of these tasks is and how to store them. Video surveillance on the naive approach means only to record the video stream as a H.264 file. But what AI is interested in is to convert the data into a semantic description of the task. Let us go into the details and think about possible simplifications. At first, video surveillance needs no real cameras, it is possible to track also gaming interactions. The advantage is, that in a computer game it is known as default what the player has pressed on the keyboard and what his mouse movements are. The new reduced question is: why has the player pressed a certain button?

Video surveillance means in most cases to construct a scoring function around a game. What is happen on the computer monitor are actions which are part of a larger system. Player 1 passes the ball because he want's to win. And the other player do not make random moves, they want to win too. If it is possible to describe the game on a formal level, video surveillance will become successful. So the better description is: video surveillance of games.

Like in classical Artificial Intelligence an easy example is a pathplanner for a maze. This time, not a software-agent has to find the goal but a human. The human is taking actions and the AI has to monitor the progress. The new thing is, that the human will find the solution in any case, because he is an expert for navigating in a maze. The question is only, if the AI is able to track the actions, that means to explain why the human is taking the longer way and not the shorter.

From a technical point of view, video surveillance goes into the direction of a head up display. The human player stays in the loop and is the only one who takes action, but a computer is looking over his shoulder and give advice which action would make sense. What the computer is providing is a situation awareness as a decision support system.

1.19 Formalizing task and motion planning

The idea behind Task and motion planning (TAMP) is divide a problem into a high-level layer which is solved by a symbolic planner and a lowlevel layer which is solved by the motion planner. The task layer consists the heuristics for a domain which will speed up the search process. The problem is, that usually the task model isn't known. Only the motion layer is given in the name of the Box2D game engine. A possible way to create a task model is "learning from demonstration". That means, a human operator is doing a task and the actions are recorded. These recordings are used for a task prediction engine.

Let us make the example very short. At first we have a Box2D physics engine for implementing a game. The goal is to grasp an object with a gripper. A human operator is doing this task. A very simple form of a task model is to save a keyframe every 2 seconds. This task model says, that the task has to be done in a certain order without the possibility of variation. A different kind of task model would be to store also alternative ways of executing the task, which are collected by different demonstrations.

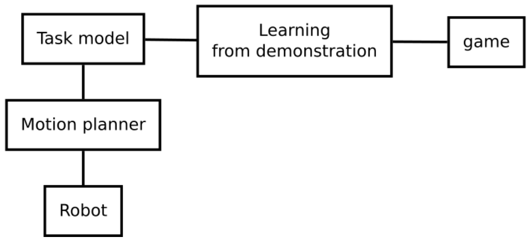

Figure 12: Task model, modified from [8] page 94

1.20 Discretization the state space with macro-actions

Suppose we have a steering problem in which a robot has to drive to a goal in a maze. What the robot can do in each step is to move the steering wheel or move one step forward. Because the simulation is very accurate, in one step the robot will move only 5 pixels forward. The reason is, that the movement will gets very smooth. For a human, this results into a wonderful game, but what will happen if a solver should navigate the robot? Right, he has the problem, that the controller needs to calculate a sequence of 1000 movements to bring the robot into the goal and such a sequence generates lots of possible plans.

The most impressive way to minimize the effort for the CPU is to use macro action. In a simplest form a macro action like "10forward" moves the robot not only 1 step forward, but presses 10 times on the simulated forward button. And this will allow the solver to calculate a plan? It seems like magic, but it is true. With some simple macro-actions the problem can be solved.

The next surprising information is, that the concept of macro-actions can be extended. For example it is possible to create virtual macros. That are action primitives which are not possible in the real game. For example a macro with the name "beam(pos)" which transfers the robotposition to the goal coordinates directly without pressing any button. This tool is useful, if obstacles are in the way. It is possible to calculate a path to the goal only with "beam" actions. And now a second solver has to calculate who the robot can move to the next waypoint.

Another option to improve the idea of macro-action is learning from demonstration. Here is the idea to let the problem solve first by a human and generates of his demonstration a list of macro-actions. These macro-actions can be used to solve a different planning problem faster. If we are combining all these techniques into one system, we will get a powerful solver. It is mainly a brute-force solver which brings a system into a goal state, but thanks to the macro-operators it works very fast.

The most easy to understand implementation are manually designed macro-actions. That means, no machine learning or learning from demonstration is used, instead every macro action was programmed by hand in the source code. This helps to get the project clean and makes it easier to understand by others. The job for the knowledge engineer is to identify possible macro-actions which make sense for a certain domain. In the steering car problem, possible actions are:

- move 10 steps forward
- beam to a waypoint
- move in a circle
- move forward and then right in a long line

The idea behind macro actions is to bridge large amount in the state space. In most cases, the programmer is not very motivated to implement macro actions. The idea is to implement only a small amount of them, until the solver finds a solution in realtime. From the perspective of a programmer the best is to implement no macro actions. The car will have only the three standard actions: left, right, forward. But the solver will make trouble, if the should find a plan only with these lowlevel actions. So at least 1 or 2 macro-actions are needed until the solver will be able to handle the state space.

With macro actions, the search process looks familiar. The solver will generate the game tree and searches for a node which fulfills the conditions. He sends random actions to the robot, and if one of the node is in or near the goal, then the solver has found the plan. The new thing is, that the planner no longer sends lowlevel actions, but macro-actions. Let us make an example to detect a potential bottleneck.

Suppose, the solver has generated the following sequence to bring the robot into the goal:

1. beam(10,20)
2. beam(20,30)
3. beam(30,30)

Each macro actions moves the robot a large amount of pixels, The plan is short and is very high-level. Now the planner wants to execute the plan. The first two beam-actions makes no problem, because no obstacle is in-between. Only the last action “beam(30,30)” can’t be executed because on the map there is no direct way. What’s next? At the first impression, the macro action solver has failed and we must search for another problem solving technique. But not so fast, not the macro actions are wrong, only a minor detail problem is open. The solver has only to generate a second plan with beam-actions, for example this one:

1. beam(5,15)
2. beam(25,30)
3. beam(30,30)

And this time, all actions are possible on the map. What I want to express is, that Macro-actions and hierarchical planning are a powerful search technique which can solve even complex domains. The only bottleneck is, to implement such an algorithm in real software and fix the detail problems.

Macro actions = prediction engine On a formal level a macro action predicts the future. If we are sending a command like “100steps-forward” to the physics engine, the physics engine will move forward and answers the request with a new state. In contrast to a normal lowlevel action, a macro action spans over more timesteps, usually more then only one step. Suppose we have a library of macro actions. Each of them predicts the future of the current system. If we run macro1 the result will be different from action2.

A normal physics engine isn’t able to predict more then one step. A physics engine is updated step by step. That means, the graphics are drawn to the monitor and then the game-loop moves the physics forward by only one frame. Macro actions are about longer horizons. A macroaction can take 10 steps or 100 steps. It is possible to run the macro against the original game, or it is possible to store the outcome in a hash table. In any case the response to a macro action is a information about future game states. These information are aggregated by the solver to a plan. If we know, then after executing an action the robot will lost the game, it is easy to prevent such an action.

The basic idea of macro action is similar to a normal pathplanner. That means, from the current state some random actions are executed. The new thing is, that these actions are more powerful.

1.21 Symbolic planning

The term symbolic planning isn’t clearly defined in the literature. The STRIPS system can be called a symbolic planner, but also Hierarchical task networks are part of the idea. A pathplanning algorithm is usually not called a symbolic planner because it operates on the lowlevel euclidean space. The idea behind a symbolic planner is to formalize heuristics. The best example is the PDDL language, which operates as default on a symbolic level. But what is the difference between gametree search and symbolic planning? Let us make a simple example.

The robot want’s to move from A to B. A gametree search would take the possible directions of the robot (north, east, west and south) and create a graph. Then the planner searches through the graph until the goal is reached. It is obvious that this technique ignores possible heuristics, instead the raw gametree is searched which takes a huge amount of cpu-time. A symbolic planner implemented in PDDL would first define some actions:

- walk-10-steps-north
- beam to point A, B, C
- walk in a half circle

Then the planner would try to use these motion primitive to construct a path to the goal. The important difference is, that the symbolic planner can execute moves which are usually not available for example “beam to point A”. Such actions are called symbolic because they operate on a higher abstraction level. They are equal to a description by a human. He would also argue, that the robot should go first to a waypoint, move then in a circle and will then finds the goal. Symbolic planning is sometimes called task planning because it describes the domain from a higher perspective.

The confusing aspect of task planning is, that the plans can’t executed directly on the robot. If we are sending the command “walk in a half circle” to the robot he doesn’t understand the action. The robot will only understand lowlevel actions like “north, south, east and west". Additional to the task planner itself, some kind of translater from task level descriptions to lowlevel descriptions has to be implemented. The same is true for pddl like planners. Usually a motion primitives prints out only the result of an action. A command like "beamt to point A" would result into the state "robot is on place A". But it is unclear how the robot has moved to this place. The output of a task planner is equal to subgoals. These subgoals devide the statespace into smaller subspaces.

AI planning languages The first planning language was STRIPS. It evolves into today's PDDL, but the difference between both language is small. The idea of a planning language is very different from normal programming. In a language like C++ the program flow is determined by the sourcecode. The order of the execution is fixed. A program contains of steps like input, evaluation and output. In contrast, AI planning languages are describing the domain in a declarative fashion. That means, only actions are provided but the order how to execute them. It is up to the planner to find a sequence. Planning languages have at their major problem, that the state space has to be searched. For example, it is possible to formulate a chess problem in PDDL, but if the planner should find the plan to the goal state he will search a huge state space of potential alternatives and this results into a high-cpu usage.

In the gaming community, AI planning is called GOAP planning (goal action planning). Instead of defining a behavior tree, the sequence of actions is determined at runtime and can change on each situation. In the area of machine learning the idea is not only to fluctuate the action-sequence but the actions itself. That means, after starting the robot he has no PDDL description and must search first in the state space of potential actions.

All planning languages are working like a qualitative physics engine. That means, they are simulating a situation. A simulator can be called a metaprogram because the system itself has no preference instead the idea is to execute movements which are given by the user. For example, a grasping simulator waits for commands like "opengripper", "closegripper" and "movearm" and can execute these actions. It is equal to an interpreter which is used for execute a domain specific language.

PDDL can be understand as an extended form of a bison grammar. It defines a language and the task of the user is to write a program in that language. The program is called a plan and specifies the sequence of the actions. Programming in PDDL is equal to what is called in computer science a compiler generator. That is the idea not to program in one of the given languages like C or Python, but to invent and new language from scratch and define the grammar and the compiler to translate this language into machine code.

To formalize AI planning it makes sense to see it as sentence generation.[6] PDDL means usually high-level task planning, and natural language is per default well suited for describing high-level actions. What the planner has to do is to output natural language, which fulfills certain constraints.

Natural language generation It is interesting to observe how natural language is stored in a computerprogram. The answer is, that natural language has no meaning, instead it is stored as strings. A typical example in C++ might be:

```
std::string s = "This is a sentence";
```

Why is this important? Because the string s can't be executed, it is nothing which has an influence to the program flow. It is only a sentence but not an executable command. But, it is possible to use a string for language games. That means, we can sort strings, add new strings to a blackboard and search for strings. And as result it is possible to derive from the strings actions. Here is an example:

```
if (s=="go") robot.go();
```

This time a string is connected to the normal program flow.

1.22 Formalizing task and motion planning

In the literature many keywords are used to describe the same concept: PDDL, symbolic planning, task and motion planning. What is missing right now, is a readable description about the general idea. [7] has given a definition based on the option idea. At first, Markov decision process is defined which contains

"the low-level sensor and actuator space of the agent"

on top of this markov process it is possible to run an option. This bypass the actuator space. The details are given by the option policy which defines a state-transition and is written in the PDDL language with it's precondition/result syntax. A plan is defined as a sequence of options.

I like this definition because it needs only the option definition. An option is equal to a action in the pddl syntax to move the agent to any state. It is like beaming in Startrek which can bypass the normal markov statespace. Perhaps an example would be helpful. According to a Markov decision process the robot in a maze can move left, right, up and down. An option would be, that the agent gets a new command "gotoxy()" which moves the agent to any position in the map without any effort. It is some kind of cheating.

Even the option idea is hard to understand, because in a game usually rules exist which prevents the agent to move anywhere. So what is the meaning, if we are first defining rules only to bypass them with options? Perhaps another explanation makes more sense. In the domain of computeranimation there is a concept called keyframe. A keyframe defines a situation in the animation process. A keyframe follows no rules, it can be anything. An option is equal to a keyframe.

The question which wasn't answered yet is how the options have to look for a certain domain. The answer has to do with inventing games. Defining the options is equal to invent a new board game. That means, there are no rules to follow, instead the idea is to invent the rules. Let us investigate the process of board game inventions further. Usually the author has an empty sheet of paper and some pieces. Then he is drawing lines of the paper, and moves the pieces according to rules. These rules have to be modified to improve the gameflow.

1.23 High-level taskplanning with simulation

A high-level planner is based on domain specific knowledge. In the literature some topics are discussed to store the knowledge in a machine readable format, notable ontologies, semantic web and RDF-triples. The problem is, that until now the next step is unclear, which request can an ontology answer, and where exactly is the knowledge stored? A better approach to describe knowledge is a simulator. The question, how the knowledge is stored isn't important instead the question is: what happens in a certain situation.

From a programming perspective a knowledge based system is a task simulator. That means it is a physics engine which takes a natural language input and prints out the following state. Such an engine is similar to what is called in the gaming community a textadventure. A textadventure can be programmed with ontologies and RDF-triple but it can also be programmed in any other programming language. The question is not, how the knowledge is stored in the engine, the question is if the task-simulator is realistic and comes to the same conclusion like the reality.

Let us make a simple example of a simulated water-in-cup-example. The first thing what the textadventure is telling to the human player is the current situation:

bottle is empty and the room has a water point.

Then the human is entering a textual command: “fill bottle with water”. The textadventures parses the command and prints out the new status:

bottle is filled with water.

Now the human player can enter the next command “drink”, and the textadventure is answering with:

bottle is empty and the room has a water point.

And so on. The open question is only how to program such a textadventure with current programming technique. It may be possible to build such a textadventure with an existing adeventure-game-engine, or it might be possible to program the textinterface from scratch. IT is possible to use a XML description language developed for chatbots to describe the interactions, or it is also possible to use object-oriented programming. In any case the result has to look like a task-simulator. That means, the physics engine can parse commands, and changes its state through interaction with the environment.

The advantage of the concept is, that the engine can be verified. It is possible to say how many words the textinterpreter understands and how realistic he reacts. From a programming perspective, the domain has to be converted into a textadventure, thats all. A textadventure has no graphics, instead it knows only natural language. But the most interesting feature is, that a given textadventure can be solved with a planner. If the system is in a start-state it is possible to find a path to the goal state by randomized actions and storing the outcome in a gametree. Brute forcing a textadventure is not very complicated, because the state-space is small. Usually in each state a small number of actions can be entered and testing all of them is possible with a standard computer. The more demanding task is to construct the textadventure. That means to invent the game, the vocabulary and the outcomes of actions. From working textadventure of the past for example Zork, it is known, that many man hours were invested until the engine works reasonable stable.

In most cases, a textadventure for a robotics domain will be organized like a maze game. That means, the robot is somewhere, can go in some directions and finds on the way objects which can be picked up. Planning the complete game is equal to execute high-level actions, for example at first the robot goes to the kitchen, takes an object there, then the robot goes to another room and puts the objects there and so on. Reaching the goal of the game is similar to do a sequence of high-level actions which are part of the game. The question which has to be answered by the task-simulator is, what will happen if the robot executes a certain plan. Will the result be positive, or negative, or will the game-engine prevents a certain move, because it is not possible to pick up an object if it’s not in the room. From a birds view perspective a textadventure is reducing the potential interaction with the world. The robot can only execute actions which are given by the game-engine parser. And in most cases, the parser rejects non-sense actions very early. Solving the game means, to search in the remaining state-space for a plan and executes it in reality.

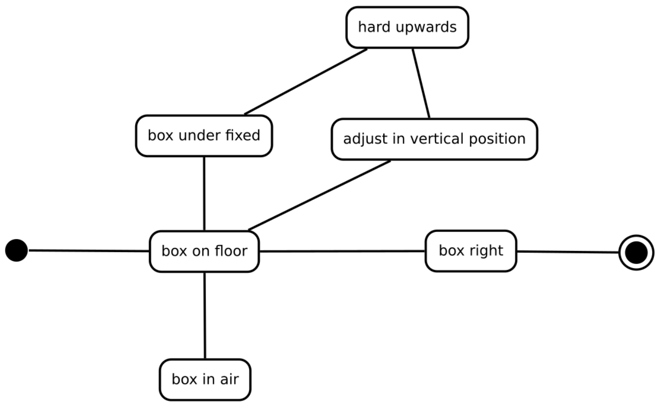

Figure 13: UML state-chart for box moving task

1.24 State transition with petrinets

The pddl language contains a feature, called precondition. With that feature, the sequence of actions is defined. It is not possible to execute actions in any order but only if certain conditions are true. The graphical visualization of preconditions are petrinets, which are part of the UML specification. They are called state-diagram and certain symbols are available to modeling event-driven systems. An example is given in the figure. In contrast to a class diagram, the state chart represents an activity. There are possible ways in the graph. The most important aspect is, that it is possible to convert such a diagram into a pddl syntax. The preconditions are equal to the connections between the states.

Let us go into the details. The chart contains situations, e.g. “box on the floor” or “adjust upwards”. It is not possible to call these transitions in any order, but only in a sequence given by the connection lines. To move the box hard upwards, we must first adjust the box in vertical position. It is comparable to a maze map, in which some roads are blocked by obstacles. The difference is, that a state-charts expresses a model on an abstract level.

UML statecharts are usually used outside of the pddl domain, for modeling software systems. For example, it is possible to define shortcuts in an application. That means, a certain key combination will start an action but only in menu 1 but not in menu 2. A similar diagram in the UML notation is called “activity diagram”. This looks similar to a flowchart used for modelling the programflow. Like in a state chart, only certain transitions are possible.

Behavior trees Statecharts have the problem, that they become messy if the diagram is growing. In the gaming community an alternative approach, called behavior tree, was developed. A behavior tree can be seen as a hierarchical petri net. It simplifies the preconditions because subactions can only form a sequence within the same task. In the xample, the action “hard upwards” can not be executed in the state “in air”.

Let us compare the difference in detail. In a plain petrinet or PDDL file the actions are seen as isolated entities. It is possible to take one of the actions and it has all the definitions to understand the

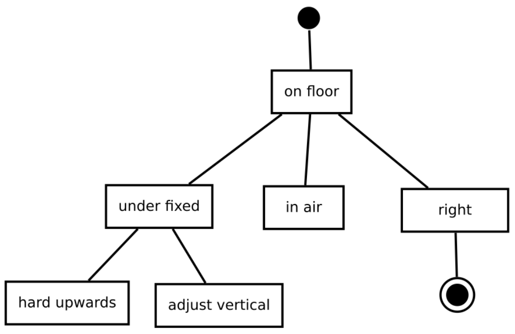

Figure 14: task graph as behavior tree

context. In contrast, the actions in a behavior tree are depended from its surrounding. They are grouped hierarchical and this gives implicit the preconditions.

Motion graph In the domain of computeranimation so called motion graph are describing the transition between keyframes. They are used as a template to generate transitions between states. It is also possible to use "task graphs", which have also a state-action syntax but describing high-level tasks.[14] A more recent concept is a "neural task graph" [13] which stores a task graph into a neural network and learns the graph through interaction with the environment.

According to the literature motion graphs are often used in practical application. Their advantage over petrinets and behavior tree is their simplicity. They only contains node which can have transitions. No actions and no hierarchies are given, instead the motion graph is forming a maze like structure which can be generated automatically with mocap recordings and can be traversed by a planner. A motion graph is walk in the keyframe space.

Let us give a short example in the domain of biped walking. A biped walk character contains of gaits. That are 6 keyframes with different poses. The walking animation is produced by playback the keyframes in a linear order from frame 1 to frame 6. The steps between the keyframes gets interpolated to become a smooth transition. The order of the 6 keyframes is described as a motion graph. It defines, that after frame 2 the frame 3 can follow but no other keyframe. Playback the animation means to walk through the motion graph: frame 1,2,3,4,5,6,1,2,3,4 and so on.

If the character should become more realistic he needs apart from walking pattern also a jumping pattern. The jumping sequence can't start everywhere but only at keyframe 3. That means on frame 3 it is possible to move to the known keyframe 4 which is about walking or to new keyframe 3a which is called jump.

1.25 Task planning with motion graph and tracking control

All major problems in Robotics have to do with tracking control. Tracking control isn't an algorithm but a challenge. The human demonstrator is doing a task, and the robot should repeat the movements. Usually, the robot fails and this bug helps to understand the limits of a robot control system. Let us make an example: an object is on the table, the human operator controls the robotarm with a joystick to pick the object and place it on the goal position. This part isn't very difficult because no artificial Intelligence but the human operator has produced the control signal. But now the robot should repeat this action alone. And this time, he needs some kind of planning to solve the task. In most cases, such a planner isn't available that means, the challenge is very difficult.

Tracking control makes visible that in the software a task model is missing. To plan motions, some kind of constraints and a model must be given first. If a model is there, the solver will find a sequence of actions to reproduce the motion. The question is: how can we convert a domain into a model?

Let us first explain which part of a solver is easy. Lowlevel planning between two keyframes makes no trouble. If keyframe 1 shows the robotarm, and keyframe 2 shows the same robotarm with a slightly different position it is possible to determine the movement between both keyframes. The algorithm is based on searching: a random generated gametree is used for searching a node which is near the goal keyframe. In the literature this concept is called motion planning, because it is on the lowest level.

And now comes the part which isn't fully understood and which makes trouble in robotics, the task planner. Planning of longer sequences is made with a keyframe generator. That is a module, which generates a list of keyframes which describes how to bring the initial system into a goal state. Usually a task planner works on a higher abstraction level. In the literature the following ideas are discussed:

- Hierarchical task networks (HTN) which are described by a PDDL file
- natural language command language
- qualitative physics
- Motion graph, Task graph
- object oriented modeling with UML notation

The shared goal is to model a domain in machine readable description. So that a planer can search a path through the gametree. The problem is, that every domain is a different kind of game. A robotarm works with inverse kinematics, while an autonomous car works with traffic rules. What a solver has to provide is an outlook into future. He must predict a state if the user enters a command. This works on Hierarchical task networks, but also with motion graphs.

Let us compare HTN-planning with motion graphs first. A HTN-planning domain is described in the PDDL syntax. The system contains of actions like grasp, open, close, ungrasp. The user can enter one of these commands, and the system answer the request with a new system state like in a textadventure. The innerworking of the parser is given by the PDDL preconditions and effects. From the outside view a PDDL textadventure works with the following syntax: $action(name) \rightarrow newsystemstate$

All possible actions in all possible sequences are generating a gametree which can be searched for a node. This node is the goal and the planner can show the path to that node. Now we can compare a HTN planner with a motion graph. A motion graph contains of keyframes which are connected by edges. For example keyframe #1 is connected with keyframe #3. This idea was introduced for describing motion pattern, for example a walking gait. Like in a HTN-planner the user can enter an action, and the parser reacts with the follow up state: $action(edgetransition) \rightarrow newsystemstate$

The difference to a PDDL description is, that a motion graph contains less information. For keyframe #1 the user has the choice to switch to keyframe #3 or #5 next. That means, he can enter as command "action(switchto#5)".

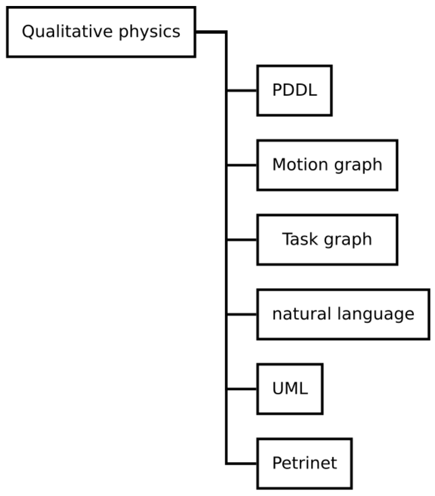

Figure 15: qualitative physics consists of many implementation techniques

Let us summarize the ideas behind motion graphs and Hierarchical task planning. The user interactions with a physics engine which accepts commands. The commands bring the system into a future state.

Qualitative Physics A "qualitative physics engine" is a general term to describe a prediction system, which accepts an input command and reacts with a new system state. In the figure some examples are given how a qualitative physics engine can be implemented: PDDL, motion graph, task graphs and so on. All these techniques can be used to model a dialog based userinteraction for predicting future game states. From its implementation in the Python language, a qualitative physics engine is a class which takes the userinput and then the class is calculating what happens. In the case of a motion graph, the calculation contains of a transition matrix. Here is qualitative physics engine contains a variable called "state" and a command brings the system to new state. This is very similar to a petrinet. From the user perspective the user can enter transitions, that means he enters the number of the next keyframe and if the transition is allowed then the Python class will change the state-variable.

If the aim is to implement a PDDL like system but not a motion graph, then the internal situation of the Python class looks a bit different. A pddl domain model has more variables and the commands are more complicated then only the number of the next keyframe. But what we can say is, that any of the task planning techniques has to be implemented as a qualitative physics engine. That means, there will be a python class which accepts commands and calculates the future state of the system.

Literature An early attempt for formalize a qualitative physics engine was made in a paper of the year 1994, [1]. They are describing a task manipulation problem. It is correct, that the robot needs a manipulation graph to generate the keyframes. The problem with that paper is, that it describes only the problem on a mathematical level. This is done with geometric constraints which has to be fulfilled. But how does the planner will look like, how can the system predicts future states? That is not given by the paper. At page 6 of the paper is a nice looking figure given, which shows the keyframes for the manipulation problem. What we see is, that the robot is doing some actions in the map and brings the system into a goal state. A human operator can produce these keyframes easily, by playing the game. Like in the famous sokoban game, he moves around with the robot and pushes boxes. But it is very hard to program a solver which can do the same task autonomously.

1.26 Task model grounding

Between a task model and the user generated input is a difference. That means, the user can drive the robot freely in the game, while the task model only knows certain game state. If it is a simple motion graph, the task model contains only keyframes which are stored in the database. If the taskmodel is more elaborated, it is only aware if the robot is in his raster. In all these cases, the task model is under defined.

The first important step to overcome this problem is to be aware of it. If the taskmodel contains only of 3 keyframes, there will be a problem, if the real robot is outside of these keyframes. That means, the game and the taskmodel doesn't fit together. How can we solve the mismatch? The first idea would to improve the task model. But this is very complicated, because task models are usually created by hand. The other option is to match the real game to the task model. In the literature this is called grounding because it has to do with a gap between two models.

Let us give a short example: suppose we have modeled in Box2d a sokoban game. Thanks to the physics simulator the game is highly realistic. As a task model we have programmed a simplified sokoban game which isn't very realistic but works with a chessboard. That means, for the task model the objects can't rotate, and there is no gravity. If we a overlay the taskmodel over the box2d game we will notice a gap between both models. That means, our task model isn't accurate.

The answer to the problem is easy. At first we suppose that our task model is static and we can't improve it. So the box2d game must change it's behavior. At first, we are searching for a state in the taskmodel which looks similar to the box2d state, and then we adjust the box2d game to match with the taskmodel. If the grounding algorithm works, the box2d simulation will looks like a discrete taskmodel. Everytime we rotate the boxes a bit, the system will try to adjust the rotation.

Learning from demonstration A simplified approach to determine the keyframes of a task is learning from demonstration. The idea is, that the human demonstrator controls the robotarm, and each second a keyframe snapshot is created. In the reproduction phase, the keyframe sequence (trajectory) is used as a template for the motion planner to calculate the inbetween steps. The flexibility can be increaded while recording many demonstrations and give them different names:

- demonstration1 = walk to left
- demonstration2 = walk to right

The reason why learning from demonstration is attractive is because any task can be demonstrated. The human operator controls the robot, this generates the keyframe trajectory and the trajectory can then be reproduced by the robot autonomously. The disadvantage is that learning from demonstration is something different from a planner.

It is not possible to generate the keyframes from scratch for a new situation.

The recorded keyframes in learning from demonstration are equal to a task model. They are a path through the state-space. The open question is how does look this path for a new situation not seen before? The answer to the problem is unclear. Perhaps it is necessary to store the keyframes in some kind of task model? That means, no only record them but also annotate and create aggregate models from it.

A general description might be to transform a learning from demonstration trajectory into a qualitative physics engine. A qualitative physics engine is per default more advanced the a simple database. It contains a so called prediction model which gives the next state for any situation. In contrast, a recorded demonstration is linear, after keyframe #1 follows keyframe 2.

A simple example could be to use Learning from demonstration together with a motion graph. Learning from demonstration means that a human operator is controlling the robot, while in the background the transition between the keyframes are stored in a matrix. Such a motion graph is not advanced like a pddl model with complicated precondition, but it is more powerful then a simple linear keyframe recording. The advantage of a motion graph is, that it can store many demonstrations at once. And the solver can traverse through the network for finding a goal node.

Motion graph as prediction engine A motion graph contains keyframe nodes which are recorded in a Learning from demonstration session. – This explanation is right from a technical perspective but it doesn't explain why motion graphs are created. Their purpose is task modeling, that means to create a prediction engine which gives future states back to the user. Let us make an example. The robot stays on the start position in a maze. Now he goes up, up, left, up. What is his new position? The task model will answer the question. It searches in the motion graph for previous logfiles, finds a similar path and answers the request. The robot will be on position (42,40) if he executes the movements.

A motion graph is one but not the only technology to construct a qualitative physics engine. All the connected nodes of a motion graph can be seen like a PDDL model, which is also able to predict "What-if-scenarios". The idea is, that the prediction engine can parse a sequence of actions (up, up, left, up) and then prints out the future state. It is some kind of simulator. PDDL files are usually created for symbolic simulation which have to do with natural language, while motion graph are geometrical simulators to predict future states of a gait-pattern in computer animation.

1.27 Task model as a Finite state machine

A Task model is a tool for planning longer horizons on a symbolic level. A task model isn't an AI script which tells the robot what to do next, instead it is a simulator which can run different plans in trial mode. Let us make a simple example: in a Box2D physics engine a grasping task is given. The robot gripper should take an object and bring it to the goal. A task model isn't about the lowlevel actions and the details of Box2D, instead it simulates a high-level plan. An example action in the task model could be "move object up". After executing this action, the object will have a position 50 pixels upwards from it's current position. According to the box2d engine, this task isn't possible, because in Box2d objects can't move freely through space, there are laws of physics which prevents such a behavior. But the task model ignores the detail problems. It delegates the realization of an action to lower levels.

For the task model there is no box2d engine, instead the game contains of objects which can move freely. The task model can change the position freely without any constraints. Only the task model itself restricts the movements. In reality, the task model has more in common with a goal planner. It defines high-level objectives.

Let us go back to the example. The task model has four possible actions: objectup, objectleft, objectright and objectdown. A possible plan could be:

objectright, objectright, objectup

Like I mentioned above, it is not possible to execute this plan in Box2D. Because Box2d hasn't an action called objectright. Box2d only knows forces which can applies to objects. The task model works on a different layer which has nothing to do with a physics engine. The task model creates it's own physics engine.

1.28 Sampling the state space with a animation language

From the perspective of the rapidly random exploring tree algorithm (RRT) the planning works is easy: all what the solver has to do is creating a uniform sampling of the entire state space, because this maximize the chance that one of the nodes is equal to the goal node. The uniform sampling is reached with a concept in which a random point in the map is selected first and the nearest existing RRT neighbor is selected then. The problem is, that the vanilla RRT algorithm contains no action model. Instead the simplified assumption is, that the robot can move up, down, left and right.

Now it is time to introduce a task planner. A task planner is a high-level action generator. Sometimes a task planner is mistaken by a strategy to find a path to the goal, but in reality is task planner allows a uniform sampling of the state space. On the first hand a task model is extending the actions of the robot. It adds to the existing up, down, left and right new possible actions for example, pushobject, gotoxy, releaseobject and so on. With each action the state-space gets sampled better. And this time a huge state-space will help to solve a problem. Let us explain it on an example.

Every planner starts at a current situation. The robot is in the middle of the maze and the RRT planner is generating a graph of possible movements. In the vanilla version the graph grows around the robot, because he can move only a small number of steps. Now we add some high-level tasks to the RRT planner. He can now grow the graph into much more directions. It the RRT literature this is called extended RRT because the graph can grow on two positions at the same time. A task model can define much more possible actions, if it is elaborated the graph extends on many places at the same time with different actions.

The most important feature of RRT is not a huge number of nodes in a graph, but that every node is different. The aim of extending the graph is to explore unknown paths. That means, every node is different from the others. In a vanilla RRT graph, the nodes are created by the same actions: node1 is the result of a sequence like left, left, up. node2 by a sequence like up, right, left, left and so forth. If we have 100 nodes, everyone contains the same 4 possible actions. But what will happen if we extend the number of possible actions? node1 can have the sequence "opengripper, left, left, up". node2 is "up, up, gotoxy" and node3 contains of "gotoxy, longup, left, left".The result is, that the statespace will be sampled different than in the previous example.

On the first look, the huge state space makes no sense, because we don't want to have a graph with thousands of trajectories, we only need one which is the goal trajectory. The problem is, that normal sampling of the state space results into a huge game tree but this game tree doesn't contains the goal node. Instead all the nodes are equal. The idea behind RRT is to improve the random-quality of the sampling process. That means to grow the graph more intelligently.

The goal finding procedure for robots works a bit different from what humans can do. The advantage of an algorithm is, that it can investigate many hundreds of trajectory in a second, and that it can run the solver again and again. In contrast, the robot isn't very intelligent. A well working AI can do both.

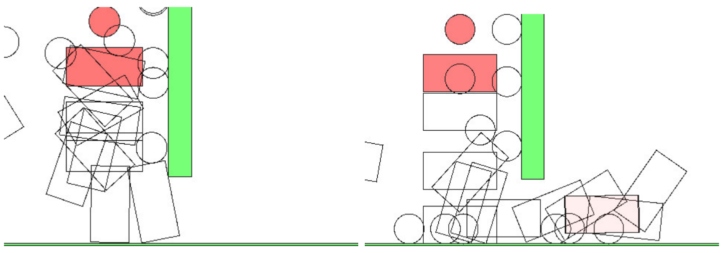

Figure 16: Multimodal push planning
left: without any modes, only lowlevel actions are planned. The box doesn't find a path to the goal
right: with a mode "boxonbottom". The planner is able to push the box to the goal.

Multimodal planning A multimodal planner is based on a animation language. The aim is to combine symbolic and geometric planning for a uniform sampling of the state space. But first we must define what a mode is? A mode is something which isn't reachable from the current state. A robot which is inside a maze can only move up, left, right and down. But he can not beam himself directly to a position and he can't also move an object. Such high-level actions are only the result of up, left, right and down actions. Such high-level behaviors can be called a mode.

What most researchers in the past have done is to ignoring modes. The idea was that only planning with the lowlevel actions make sense. A multi-mode planner is overcoming this ideology and is utilizing symbolic modes too. The main reason in doing so has to do with subgoals. A complicated task contains subgoals and waypoints. The robot can't move straight to the exit of a maze, first he must go to waypoint A, B and C. But what are the right waypoints? That is unclear, and to figure out which subgoals / waypoints are needed a high-level task planner is the right choice.

Let us investigate how a multi-modal planner works in reality. The system is able to try out a subgoal for example "movetowaypointA", and the planner can also try out lowlevel actions like "up, down, left and right". The solver is based on the "What if" principle. That means it creates a graph for everything. One graph is managing potential subgoals, and a second graph is about lowlevel actions after a subgoal was reached.

Because the topic is a bit complicated a pathplanning example may be right. Suppose the robot is on leftbottom of the maze and his goal is to go the goal which is on topright. The multimodal planner is finding a plan which is the sequence "waypointA, waypointD, goal". It is a very short plan because it contains only 3 actions. Every action in the plan is a mode-action, that means the action can not be executed on the robot. The planner has generated a fictional plan. But that isn't a problem, because we can run the solver again, and this time he figures out the details. He finds a plan to go from waypointA to waypointD, the plan "up, up, left, up". These lowlevel actions are realistic and can be executed on the robot.

Multimodal planning is the same like Hierarchical task networks, it contains high-level and lowlevel actions at the same time. Simulating lowlevel actions is easy, we can execute the action against the game engine. That means, we give the "up" action to the engine and the engine will move the robot by 100 pixels up. A bit more complicated is to simulate the execution of modes. For example, what is the result of executing the "movetowaypointA" action? Answering this question is called "Motion language". A motion language is a vocabulary which defines what a action can do. Traditional it is written in the PDDL syntax.

A multimodal planner contains two subsystems: a lowlevel motion simulator and a high-level task simulator. The planner is using the actions on each layer for a uniform sampling of the state space. If the statespace is sampled the planner can pick a node from the graph and knows the plan to reach that node.

Example A push task is given in the figure. The round robot should push the box to the right side of the maze. The game uses the Box2d physics engine to simulate the actions realistic. To solve the task a planner is used. But a normal planner would never find the correct trajectory for the robot. Because there are endless ways to interact with the object. So let us observe how the multi-modal planner works.

It can handle lowlevel and high-level actions at the same time. A low level action, that the robot can change his position in small steps, he can move up, left, right and down. A high-level mode is, that the boxposition is changed directly to the bottom of the maze and then the robot can push from the left side. The result is, that the box will move to the goal.

But isn't it cheating if we move the move to a new position without pushing them first? No it is not, it is a normal mode in the planner. It helps to explore the state space. We divided the complex problem into a smaller one.

Let us describe what the programmer has to do for implement such a planner. The lowlevel actions are the same and they are given by the box2D simulation. What the programmer can adjust is the task model. In the example, i have created a new mode, called "boxonbottom". This mode set the absolute box position. But it would be possible to define other modes, for example "boxonright" or something like "robotbelowbox". If the modes are carefully defined it will help a lot to sampling the state space. The planner can try out these high-level actions and prints the result to the screen.

Let us investigate what will happen if the modes are implemented in the bad way. The result is, that none of the random actions will bring the box to the goal. That means, the planner is testing out thousands of possibilities but in all cases, the box remains in the left area and isn't able to go smoothly through the gap.

Modes Because modes are powerful we must describe them in detail. A mode is equal to a pddl statement, but it is also possible to use any other programming language. It overrides the physics engine and changes is value directly. An example mode could be:

```
boxonbottom() {
  pos = {100,200};
  angle = 0;
}
```

In the common PDDL syntax also preconditions are possible but this is only optional. The idea behind a mode is, to start a new game instance in a sandbox and create a scenario. Modes have nothing to do with the normal game-play or the rules from the game. The best way to explain modes is a situation in which no modes were defined. In the figure with the boxpushing task is such a situation given on the left side. The state-space sampling is only done with lowlevel actions which are possible in the given situation. That means, the robot can push against the box, it will react and the outcomes are drawn to the screen. What we see is, that all the boxes stay on the left side. None of them is going at random to the goal. That means only a small subspace was sampled and the goal is outside this subspace. So the question is how to increase the state space. One possibilities might by to increase the number of trials. Instead of testing out 15 random actions we can testing out 150 or more. The problem is, that the resulting nodes will stay also on the left but the CPU consumption is much higher. And here comes the idea of modes into the game. A mode improves the state-space sampling without increasing the number of trials. A mode starts with new situation and explores random actions there.

1.29 The RRT Connect algorithm

RRT connect is discussed in the literature as a pathplanning algorithm. The idea is, that two trees are growing at the same time. The first tree grows around the starting point of the robot, while the second one starts in the near of the goal backwards. After a while it is possible to connect both separate trees into one huge graph and the way from start to goal is found.

For a domain like pathplanning, RRT connect doesn't improve the speed very much. Because pathplanning is in general an easy task and a normal RRT planner will find the way too. But for complicated manipulation tasks, RRT is some kind of goldstandard. As an improvement not only 2 trees are growing in parallel but 20 and more. Is it possible to use RRT connect together with a task model which contains of symbolic actions? Sure, this works remarkable good. The idea is to plan on different positions and on different layers at the same time. The graph not only occupies the surrounding of the robot, but the complete game is explored with nodes. A fully rrt connect graph is equal to a motion graph, except that it is created every time from scratch.

Suppose the graph is huge, what next? At first we must determine which of the nodes is near to the goal, and then we need to search for a path from start to that node. It is possible that the graph isn't fully explored. That means there are gaps between the subgraph, so additional search steps are needed to close the bridge. But then can the robot to the goal.

One question remains open: how to program such an RRT connect algorithm which is able to grow in many layers? That is indeed a problem, a simple 10 line algorithm won't work, what we need is some kind of class only for the purpose to create and monitor the graph. From it's technical perspective the situation is easy: even the fully explored graph has not more then 100 nodes. That means creating the graph needs only a little of cpu time. The more complicated aspect is to monitor the graph, the subgraph and the different actions in the graph.

But let us investige a small example. Suppose we want to pick&place an object. The last node is simply: we open the gripper and release the object. This fulfills the goal. That is the first node we add to the graph. The problem is, that from the current situation to that goal node there is no connection. So we need some nodes between them. What we can try is to create new nodes on different positions with different layers.

1.30 Random-MMP

Kris Hauser has published a paper about the Random-MMP algorithm. He uses modes for planning complex motions. But what is a "mode"?. A mode is a heuristics which guides the search process. It is defined as a motion primitive. "graspobject" is an example for a mode, another example is "opengripper". The idea behind konowledge guided planning is, that robot movements are not only lowlevel servo parameters, but have a semantic meaning on a high-level. What a multi-modal planner like Random-MMP is doing is to sample the set of motion primitives to find a path. Sometimes the problem is called grounding because high-level pddl domain description is connected to lowlevel geometrical planning.

The open question right know is how a mode-switch is defined. A mode switch is used by Random-MMP to extend the existing graph. The interesting aspect is, that a mode-switch isn't defined on a geometrical level. Unlike a lowlevel action, a modeswitch is always a high-level action which can't be executed directly on the graph. We can define a modeswitch as some kind of sandbox game. That means, the normal physics rules are no longer valid, instead the new position of the robot can be anywere.

According to the normal physics, a robot can move left, right, up and down for only 5 pixels. That is given by the game, because the robot contains of wheels and there speed has a maximum. A mode switch is some kind of super-natural action, which moves the robot without delay 100 pixels away from it's original position. Another mode-switch would be, that the robot becomes the object in his gripper even the object was not in the environment. Mode-switches are like cheating, they produce a situation which is not possible.

Why are modes so important for planning? Because it helps to plan on high-abstraction. Let us take an example for normal travel planning done by humans. Suppose somebody wants to travel from New York to Boston. From the physical condition, the person has no superpowers, that means, the person can only walk with his legs. If he prefers to plan on a lowlevel layer, he would think about to leave his house, walk along the street, and moves around in his city. It is well known, that he will never reach Boston, because his planning is broken. The better approach is to assume first, that the person has superpower, that means he owns a plane, has access to a bus and so on. And now we can plan under super-power conditions. The result is a plan, which includes modern public transport system and only the last step is to plan how to walk physical with legs to the next bus-station, for driving to the airport.

That is – in short – the idea behind multimodal planning. That the superpower of the robot is defined by the motion primitives and they were used to plan in an abstract space and only at the end, this graph is matched with the real capabilities of the robot, to move 5 pixels forward.

modes vs. lowlevel actions To understand modes in detail we must compare them with lowlevel actions. A lowlevel action is everything what a robot can do in reality. For example, he can move his servo to 45 degree. The lowlevel actions are defined by the physical constraints. That means, the high-speed of a car is given by the battery. The robot can not drive faster then 5 pixels per second.

In contrast, a mode is everything which is supernatural. A mode is some kind magic spell which is able to teleport the robot to any place he wish, a mode can put a object into the robots gripper or it can clean up the room with a single command. From a technical perspective a mode is realized by direct manipulation of the physics engine. To teleport a robot, we change it's x/y position to a new value. For putting an object into his gripper we change the object-position and so on. That means, at first we create a highly realistic box2d simulator and then we override the simulation with modes.

2 Neural networks

2.1 Converting a domain into a task model

The real bottleneck in robotics is called knowledge transfer. The problem is that the human operator knows, how to manipulate objects and how to control a robot arm, but this knowledge isn't available in machine readable format. So it is not possible to use any kind of planners on this missing knowledge.

At first we must define, what the machine need. The machine needs a task model which is encoded as a physics / game engine. This is a class in written in C++ which takes an input and produces a simulation step. Examples for ready to run task models are the famous Box2D physics engine, a chess simulator or the sokoban game. All of these engine are dialog oriented, that means, the player enters a move, and the game engine is executing the action.

Somebody may ask if the sokoban game is available, why do we need a second game engine on top of the first one? The problem is, that a game can be described on different levels. Available games like Sokoban are only implementing the game rules itself. The result is a huge state space. A task model also creates a space of movements, but the number of possibilities is lower. So we need a game engine with a very small state space.

After describing why the machine needs a task model, we can discuss potential realizations of that need. The question is how to program a high-level task simulator for a certain domain, for example for sokoban, a robot grasp task or for Starcraft. According to the given literature of the last 20 years, this problem is unsolved, that means there is no best practice method in doing so. What we have seen in many projects are concepts like natural language description, learning from demonstration and hierarchical task networks for converting a domain into a task model.

But let us go a step backward. If a human operator plays a game, everything is fine. He controls with the keyboard all the actions and is able to fix the problems in the game. If the robot gripper is a bit out of place, the human operator inserts a corrective move. Remove control systems are well understood and all of them are working great. The problems begin if we want to convert the human action model into a machine readable task model. This is indeed complicated. What humans are doing in games is to follow a walkthrough tutorial. Either, they bring this tutorial in because they know it from the past, or they can read the manual of game and follow a tutorial given by other. In a walk through tutorial is explained, how to play a game. It is a description in which situation which action has to be executed. All tutorials are written in natural language. So it is a common practive, that also a machine readable task model understands words in English. In most cases this is not enough, additional a hierarchical description and a plan library is used inside a task model. Combining these things together is complicated. Some examples are given by the literature, but a general approach is not available.

From a technical point of view, a task model is equal to a what-if system which is based on natural language and a hierarchical plan library. But this description isn't helpful if we want to implement such a system in reality. A potential solution to the problem is, to define a constraints. Instead of programming a task model with C++ we restrict our self to neural networks. The idea is to implement a task model without writing any lines of code. On the first hand this makes no sense, because C++ programming is more powerful compared to neural networks. But neural networks have the advantage that they can be easier understood by beginners. A neural networks is based on standard software, for example on pybrain plus a so called training data which is a excel spreadsheet. Usually a neural network doesn't mean, that we need to write new Python or C++ code, instead we must think about the training data.

I'm convinced that it is possible to implement a task model without using neural networks, for example with writing the qualitative physics engine in C++. But programming new code is difficult. The more beginner friendly approach are neural networks. They are not so powerful like real coding but they can be easily described in papers and on the internet, especially for an audience which isn't familiar with Artificial Intelligence.

Now we can go into the details how to realize a task model with neural networks. Neural networks are some kind of machine learning, that means we have data stored in a CSV-file and these data are converted into a neural network which contains weights. The open question is: how does the CSV-file look like? The good news is, that is impossible to store C++ or Python code in the datafile, instead it contains only pure non-executable data. The bad news is, that data can be anything: numbers, words in natural language, positions in 2D spaces, or force values.

The source for the data are always gamelogs. That means, the human operator is playing the game and this results into our dataset. The problem is, that the raw data have too little information, so we must annotate the data first with natural language descriptions. If we convert this into a neural network, we have build a first task model, without programming any line of code.

Let us make a simple example how annotation works. Suppose the human operator is playing a real time strategy game. The raw data which are produced by the game are mouse movements. The CSV-datafile contains of a timecode, a mouse position and if the button was clicked. To enrich this data we must retrieve also the action the user has activated in the game. An RTS game contains usually on the right screen symbols for building a house, or building a unit. These information are important and should be transfered into the CSV-file.

Surveillance annotations A practical example of how the creation of a task model works can be explained on video surveillance. The idea is not only create a task model but not using any kind of programming language for it. At first, the GUI software is started which is a video playback software. It is possible to enter new categories

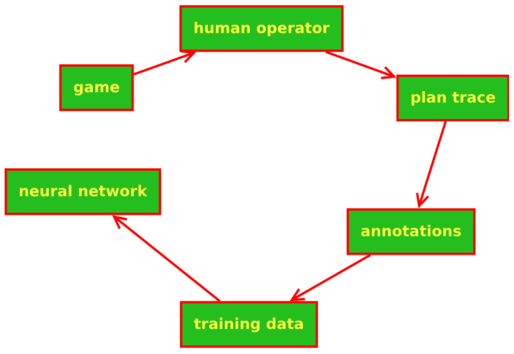

Figure 17: task model acquisition

(activities, persons) and each category contains labels. Now the playback of the video is started. The user can stop the video, and select one of the predefined label to annotate a time-sequence. The result is a csv-file which can be used for training neural networks.

The basic idea is a combination of non-programming and datadriven approach. That means, the annotation consists of a CSV-file and in the file are labels and timecode informations stored. Sure, annotation alone isn't equal to a task model, but the result can be used in a more complex pipeline.

Motion graph annotations A similar approach can be used for annotating computer animations. The user sees on the screen a sportsgame and can annotate the movements with a self-created vocabulary.[2] The result is not executable sourcecode but the result is an XML file with the timecode, actionname syntax. This is used for training a SVM neural network But let us go into the details. A computeranimation consists usually of raw data which are pixel information. It is a sequence of positions in 3d space. For the player of the game, this looks like a walking gait, but for the machine that are only numbers. Annotation with natural language verbs is tool for transforming lowlevel replays into grounded replays. The vocabulary itself is not given at the beginning. It is up to the user to define labels like "jump, walk, run" and match them to the timeline.

The result can't be called a real task model, because it contains no C++ sourcecode. If we want to compile this CSV-file the C++ will print out an error. But that is an advantage, because every step which can be done without programming is a cheap step. It can be done in short amount of time by non-programmers. And the csv-file opens a new interesting question: what can we do with the data? How can we use them to predict unseen actions and how can we use them to recognize goals of human players?

2.2 Complex hierarchical annotations

The standard way of annotate a motion sequence is to use a shallow vocabulary. The user starts the annotation GUI, defines some actions words like "walk, jump, standstill" and then he annotates the timeline. This approach is a good starting point because it connects actions with language, but what we need is an elaborated taskmodel. This contains of hierarchical actions and of situation descriptions.

Let us make an example. On the screen is a robot visible which is controlled by human operator. The robot opens the door, goes to the kitchen and takes the apple from the table. The task can't be described by a simple sequence of action words, instead the workflow is grouped into subtasks. The overall timeline can be called "grasptheappleinthekitchen". This annotation is correct for all the frames. Subtasks like gointokitchen, and takeobject are the next entity, and they can divided further in subsubactions. At the same time, the robot has some properties like an energylevel. This can also be annotated.

On the first look it is hard task to annotate manually all these information. But as a result we get a textfile which contains a blueprint for a task model. It contains of a hierarchical task network and a concrete example. Annotation of a replay is not algorithm, but a best-practice method for creating the task model of a new domain. Their main advantage is, that no programming skills are needed, instead a GUI is used together with a texteditor to create action groups.

Sure, the resulting textfile can't be used direct for controlling the robot autonomously, but it contains a draft taskmodel which can be used in a next step. This next step asks no longer how to convert a new domain into a task model, but it asks how to convert a rich annotated replay which is already there into a robot motion controller.

From plan traces to a prediction engine Suppose the user has generated a annotated replay which contains action names in natural language. The next step is to convert this plain text file into a qualitative physics engine. The first idea might be to program such a engine manually with C++ or Python, but it is also possible to restrict the potential step to a neural network only solution which means not to write any lines of code. So we need a neural network which can predicts future states of a physical system, right?

The input for the network is the current state and a action, and as output the networks gives the future state back. Like i mentioned in the introduction such architecture isn't the only option for realizing a qualitative physics engine, but it is an option which prevents real programming but is datadriven. We have to answer two question: what is the correct dataset for training such a network and which kind of neural network (perceptron, LSTM, CNN) we want to use. As input data we can utilize the annotated korpus of game-replays. After some smoothing they are the perfect input for training a prediction engine. As a result the network is able to predict future states of the physical system and it is very likely that some errors are happening. But this isn't a problem, because now we can think about potential improvements.

[4] gives a survey of previous literature about the subject of physics prediction with neural networks. All these concepts have in common that they are datadriven, that means the physics engine isn't programmed but generated from the dataset which is feed to the network. From a technical perspective this approach isn't very efficient. But from the point of view of beginners and non-programmers it is a here-to-stay method because they can testing out different models without writing a single line of code in C++. All what they have to do is compile some training data as a CSV file, train their neural network and see what the error rate will be.

How exactly the input data and the structure of the neural network have to look like is unclear, but it is very easy to experiment with different kind of possibilities to find it out and write a paper about it.

2.3 Task model building with motion capture annotations

Before a motion planner can determine the lowlevel control movements for a robot a so called task model has to be created. This is a qualitative physics engine to predict future states. A prediction engine can answer the question what happens if the robot opens the gripper while the object is in the gripper. The bottleneck right now is how to build such a task model?

A possible answer to that problem are annotated mocap recording. A motion capture recording is used for translating visual information in absolute positions in 3D space which results into a wireframe animation. It can show the hands, the legs and complete traffic situations in realtime. But a wireframe visualization is different from a high-level task model. For grounding the mocap data natural language is the right choice. For connecting motion with language a manual annotation is the best practice method. The principle is remarkable easy in reality. At first the user defines a vocabulary, for example opengripper, closegripper, pick, place and so on. Then he goes in the mocap recording to a certain timecode and inserts the actionverb in the timeline. The result is a video which has additional verbal descriptions of the meanings.

The next steps are also simple. Af first the user want's to automate the process, because he is not interested in annotating by hands hundreds of mocap videos. So he writes a small parser which searches in the videostream for an action. And the second thing what the user want's to do is to convert the annovated video into a taskmodel which can be searched by a solver. How this step can be done? A recurrent neural network can predict future mocap data:

> quote: "Our goal is to predict the mocap vector in the next frame, given a mocap sequence so far" [5]

2.4 Motion Synthesis with neural networks

To animate a virtual human or to control a biped robot the idea of walking is transfered from humans to robot. Instead of reinventing the wheel a gait-animation is usually copied from examples of the reality. In classical animation a gait-pattern consists of 7 steps which shows individual keyframes of a walking character. In modern style computeranimation these 7 poses are extracted by motion capture device.

The idea behind motion synthesis with neural networks is, not only store 7 keyframes but thousands of them. At first, the character is recorded under any condition: walking slow, walking fast, jumping, avoiding obstacles, grasping objects. These motion capture information are stored in a database which is the learning dataset for a neural network. Instead of neural networks it is also possible to request the database with other machine learning technique to make the transition smoother. The principal is the same like in unit-selection for speech-synthesis. From a given corpus existing examples are extracted and rearranged by a planner.

The idea itself isn't new. In computeranimation with 7 keyframes the animator is doing the same. He has 7 manually recorded keyframes and plays back this poses in a linear sequence. The result is realistic walking animation. The new idea behind neural networks is to increase the numbers of examples and to improve the retrieval process. This technique is called data-driven machine learning because it is based on a corpus of recorded animations.

Is it possible to animate with this technique a virtual human or a biped robot? Yes, it works great and can be called the gold standard. The result isn't only a walking robot, the result is a human-like walking robot.

Grammar based animation The first step is to record a task with a motion capture device. The visual information are converted into positions of a skeleton. The next step is to annotate the movements. The segments gets name like walk, run and jump. Now the mocap data are connected to semantic meanings. What is missing right now is a higher abstraction to generate complex behavior. This can be done with a grammar. A motion grammar is a hierarchical task network which is layered. For example to move to a room, the character must standup, walk, open the door and then he enters the room. That means, the "movetoroom" action contains of subactions and each of them are connected to a motion capture recording.

From the perspective of mocap annotation a grammar is created through forming a semantic network from the annotations. The vocabulary is grouped in a hierarchy. Now a planner can provide the motion pattern for a certain request. That means, the actions in the mocap recording become a meaning, they are part of larger tasks.

The "KIT Motion-Language Dataset" is an example for a language model. It contains many natural languages descriptions which are arranged in a hierarchy.

3 Example

3.1 Creating a task model from scratch

The main problem in hierarchical planning is the question how the task model can be formalized as machine readable sourcecode. At first we need a computergame which is shown in the figure on top left. The player can control a rectangle with the cursor keys and the green box in the middle is fixed. The rectangle obeys the physics law box2d and if it's pushed against the fixed box, it will stop. The normal interaction with the game is manually, that means, the human player presses cursor keys and the game engine will display the result on the screen.

The question is now: how can we formalized the interaction with the game, so that symbolic planning is possible? The usual answer would be that we simply take the PDDL or OWL file and then plan a path to the goal. The only problem is, that we didn't have a pddl file. We must create such a description from scratch. The best to do is a combination between natural language plus keyframes. A first possible action could be to move the rectangle to the top of the world. Then a next action can move the rectangle below the fixed box and so on. The idea is, to describe the movements not for a machine but for human, and the best way in doing so is a sequence of annotated keyframes. It is equal to a comic strip which describes possible actions.

So what's next? At first, the task model is formalized for a human. He can read the figure and make the same movements. The second step is to use this description as a blueprint for a machine readable model. Our model in PDDL or any other planning language contains of the six actions, given by the chart plus the translated keyframes. A description like "move box to the top" has to converted to absolute position informations.

And now comes the magic, we want to try out our newly created model. We bring the system into a random state, and the goal is that the AI planner finds a sequence of actions to bring the system into the goal state. That means, the rectangle should stand in vertical position in the middle of the screen. Is our task model powerful enough to realize this wish, or needs the model some minor improvements?

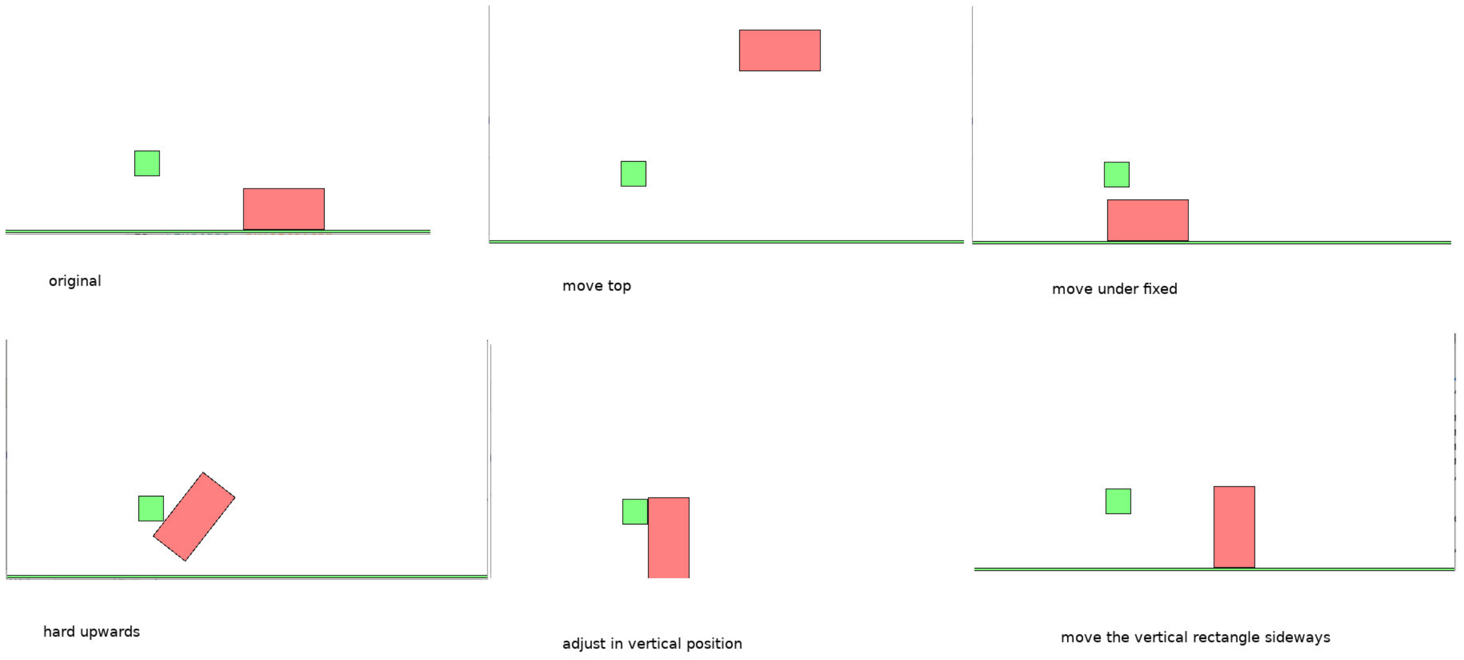
original
move top
move under fixed
hard upwards
adjust in vertical position
move the vertical rectangle sideways

Figure 18: Keyframes with textual annotations

Let us investigate first, which elements an action can have. At first, the keyframe itself. That is the absolute position of the rectangle plus his orientation, then ofcourse the action name, and also a precondition. That means, the action primitives are the same like in the PDDL syntax. But apart from PDDL everything is notated as keyframes in a visual representation. The reason is, that we only want to model spatio-temporal relationships, which is equal to a image flow. We are using the pictures to tell a non-linear story, and what the planner is trying to do, is tell the story again.

Before the AI planner can be active, we need to formalize the story. That means, we need a corpus of keyframes in a certain order which express possible plot-directions. That means, we are reducing the state-space to a small number of useful actions.

Planning can happen on two levels. At frist between the keyframes (motion planning). We have keyframe A and need a transition to keyframe B. The planner has to figure out how exactly this can be done. The second need is a planning capability from a start situation to a goal situation (task planning), that means over many keyframes along. If the difference between two keyframes is too big, or the preconditions are not fulfilled, the transition won't work, and the planner must find a path around this issue.

Motion planning Let us take a look again at figure 18. The subpicture "hard upwards" shows how the human operator is turning the rectangle into a vertical position. He jumps against the fixed green box, and pressing the cursor keys to maintain the balance. After the maneuver the rectangle stands still in the vertical position. The most interesting aspect on this movement is, that it is not possible to script it, because the timing is each time a bit different. It has to do with the box2D simulation who reacts each time a bit different and it is not possible to guess at which millisecond the human-operator has to press the "up" key to maintain the balance. But that isn't a real problem, because the overall maneuver takes only 2 seconds, and this short timeperiod can be planned by a brute force solver. We have the initial keyframe and the goal keyframe and the planner has to find out the keyboard actions to reach this goal. Such a short period of 2 seconds generates only a small gametree, that means the number of possibilities is limited. I would guess it is smaller then 100 and this can be calculated in realtime without stressing the CPU.

To make the point clear, a brute force planner is a powerful tool if the timeperiod between two keyframes is low. If we want to use the same technique for long sequence planning it will fail. Each seconds between the keyframes increases the number of possibilities exponential.

Planning between two keyframes with a small distance is called motion planning. Planning on the larger scale is called task planning. Task planning means to ignore the transition between two keyframes and figuring out larger sequence of actions, for example to decide which sequence is better: keyframe 2,5,7 or keyframes 1,2,7. In a task and motion planner both techniques are used together. The system finds an answer how to move from keyframe A to keyframe B, and also finds a sequence which keyframes are needed to reach a long term goal. This is called a hierarchical planner.

But let us focus on the detail problem of the keyframes at the lowlevel. The motion planner gets an starting keyframe, which contains the position of the rectangle and its orientation (angle). And the planner also gets the target keyframe which contains also a position and an angle. The question can be called a classical question: "how to come from start to the goal?". The most promising hint is, that this time no extra knowledge or taskmodels are needed, instead the planning task can be solved like normal planning in computing, for example, if want to find the shortest route in a graph. At first there are different possibilities, and the planner is calculating the costs, then he is searching for the plan with the lowest cost and this answers the question.

Solver In the figure 19 an example is given how a solver works. What is shown in the image is the well known rectangle with the

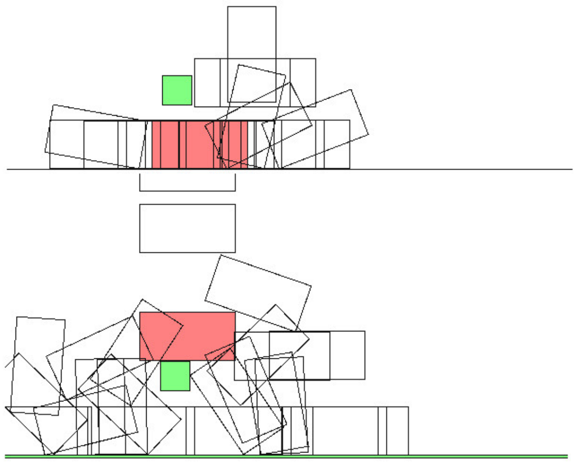

Figure 19: sampling in Box2D

position under the obstacle. This time, not the human operator is pressing a button, but the solver is testing out 50 random plans and paints all of them to the screen. We see, that some of the boxes are moving to left, while other are changing their direction. What the solver has created is called a game tree. For all the simulated boxed he knows the sequence of actions to reach this position.

The sampling technique works well, because the given timeframe is low. We see, that none of the boxes is far away from the starting keyframe. According to the solver setting, only a period of 3 seconds was calculated. That means the game tree has a very short horizon. But for the purpose of figuring out the inbetween keyframe actions this is enough. We don't want to solve the complete game, but only need the action to get to the next keyframe.

In the second example, the rectangle has a different starting position. This time it is ontop of the green fixed box. The solver generates again random plans and displays them to the screen. The result is a different state-space. And like in the first example, we can decide for one of the plans. All what was drawn to the screen is in the reachable area.

3.2 Multimodal solver with unspecified starting point

The assumption of a conventional planner is to start with the current situation and plan from this position a trajectory to the goal. But what happens, if we are ignoring the current situation and planning from any starting position available? The result is visualized in the figure. The real starting position is on top left of the maze, but the planner takes addtional starting points everywhere in the map. The idea is described in the literature as extended RRT because the graph has many starting points. The aim is to sampling not only a path for the robot, but for the map in general.

The advantage is, that the newly created graph can answer "what if" questions. For example "what if the robot is on top right of the map and walks left?". The graph can answer this question. A what if scenario means, that the robot didn't have to be on top right, it is only a hypothetical scenario.

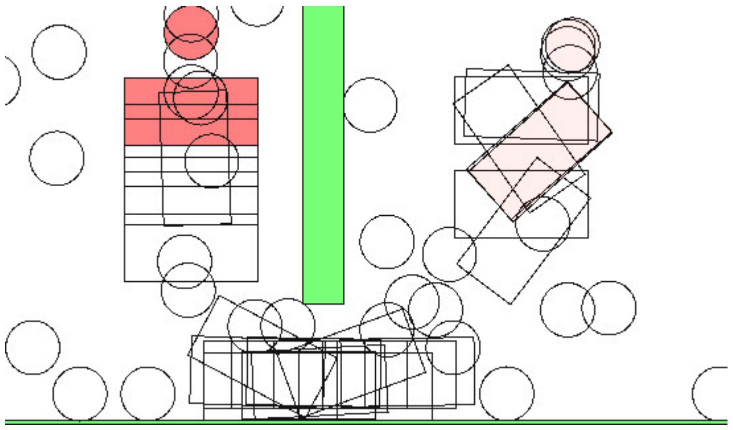

Figure 20: many starting points at the same time

But let us watch the figure carefully, what will we recognize? At first, that the box is nearly everywhere. Now matter if the goal is to push the box to the left, to right or somewhere else, some of the RRT-nodes is near to the goal position. That means, the solver has created a partial order plan and by default he finds a way to the goal. The only problem right now is, that some of the partial order plans are starting with a wrong assumption. But that isn't a real problem, because we can calculate the needed steps from the current robot position to the fictional position. This will close the partial order plan.

References

[1] Rachid Alami, Jean-Paul Laumond, and Thierry Siméon. Two manipulation planning algorithms. In *WAFR Proceedings of the workshop on Algorithmic foundations of robotics*, pages 109–125. AK Peters, Ltd. Natick, MA, USA, 1994.

[2] Okan Arikan, David A Forsyth, and James F O'Brien. Motion synthesis from annotations. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 402–408. ACM, 2003.

[3] Julien Bidot, Lars Karlsson, Fabien Lagriffoul, and Alessandro Saffiotti. Geometric backtracking for combined task and motion planning in robotic systems. *Artificial Intelligence*, 247:229–265, 2017.

[4] Katerina Fragkiadaki, Pulkit Agrawal, Sergey Levine, and Jitendra Malik. Learning visual predictive models of physics for playing billiards. *arXiv preprint arXiv:1511.07404*, 2015.

[5] Katerina Fragkiadaki, Sergey Levine, Panna Felsen, and Jitendra Malik. Recurrent network models for human dynamics. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4346–4354, 2015.

[6] Alexander Koller and Matthew Stone. Sentence generation as a planning problem. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 336–343, 2007.

[7] George Konidaris, Leslie Pack Kaelbling, and Tomas Lozano-Perez. Constructing symbolic representations for high-level planning. In *AAAI*, pages 1932–1938, 2014.

[8] Klas Jonas Alfred Kronander. Control and learning of compliant manipulation skills. Technical report, EPFL, 2015.

[9] Barak A Pearlmutter. Learning state space trajectories in recurrent neural networks. *Neural Computation*, 1(2):263–269, 1989.

[10] D Pentecost, Charlotte Sennersten, R Ollington, C Lindley, and B Kang. Using a physics engine in act-r to aid decision making. *International Journal on Advances in Intelligent Systems*, 9(3-4):298–309, 2016.

[11] D Pentecost, Charlotte Sennersten, R Ollington, Craig A Lindley, and B Kang. Predictive act-r (pact-r): Using a physics engine and simulation for physical prediction in a cognitive architecture. In *Eighth International Conference on Advanced Cognitive Technologies and Applications*, pages 22–32, 2016.

[12] Jonathan Scholz. *Physics-based reinforcement learning for autonomous manipulation*. PhD thesis, Georgia Institute of Technology, 2015.

[13] Sungryull Sohn, Junhyuk Oh, and Honglak Lee. Neural task graph execution. 2018.

[14] Ioan Alexandru Sucan. *Task and motion planning for mobile manipulators*. PhD thesis, Rice University, 2012.