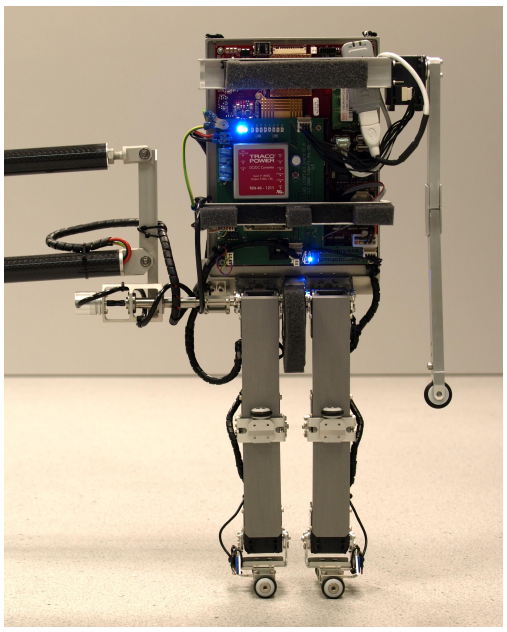
Accelerating flat reinforcement learning on a robot by using subgoals in a hierarchical framework

B. van Vliet

Master thesis – 1177



TU Delft Delft University of Technology

BioMechanical Engineering

Delft University of Technology
Faculty of Mechanical, Maritime and Materials Engineering
Department of BioMechanical Engineering

Master thesis

Accelerating flat reinforcement learning on a robot by using subgoals in a hierarchical framework

Author:
B. van Vliet
Wb1175734

Supervisor:
E. Schuitema

Professor:
Prof.dr.ir. P.P. Jonker

October 14, 2010

Preface

Seeing a robot learn what it is supposed to learn is satisfying.
Seeing it learn something it is not supposed to is fun.

This document contains the report of my research in the form of a scientific paper and an appendix. The appendix contains more detailed information about the software implementation and work that has been done, but which did not fit into the paper.
I would like to thank everyone who supported me during this research. Special thanks goes to my supervisor Erik Schuitema, for answering all the C++ questions, for helping me to find software bugs and for his feedback.

Bart van Vliet

Accelerating flat reinforcement learning on a robot by using subgoals in a hierarchical framework

van Vliet, B.

***Abstract* — Learning a motor skill task with Reinforcement Learning still takes a long time. A way to speed up the learning process without using much prior knowledge is to use subgoals. In this study, the use of subgoals decreased the learning time by a factor nine and we show that tests on a real robot give similar results. The price to be paid, in case the subgoals do not lie on the optimal path, is a worse end performance. Hierarchical greedy execution can (partially) cancel out this problem. For future work, we suggest the use of a method which is able to obtain optimal performance.**

1 Introduction

As robots become more versatile, it becomes harder to design their controllers. Instead, we could use the reinforcement learning (RL) framework. In this framework, the robot learns to perform tasks by maximizing the *rewards* it receives by interacting with the environment. In order to optimize, the robot explores by performing random actions. In this way, RL can find the optimal solution without the need of knowledge about the system. However, learning a task with RL can take a long time. This is why researchers are looking for ways to speed up the learning process. To accomplish this without using a lot of prior knowledge, while still aiming for an optimal solution, we look into the use of subgoals. These subgoals can be regarded as guiding waypoints, placed on the path to the goal of the task, and are either given or discovered by an algorithm, like L-Cut [1].

1.1 Related work

A way to implement subgoals in the RL framework is to use reward shaping, by giving extra reward for reaching a subgoal. In this way, the learner knows when it is on the right way to the goal. In [2] a robot had to grasp a puck and drop it at home. By giving a reward for grasping the puck, the learning speed was improved. However, the robot can quickly discover it can gather many rewards by continuously grasping and dropping the puck, each time collecting the subgoal reward. Such behavior was shown in [3] and to avoid this, a penalty has to be given for such behavior. In the task of [2] this was done by giving a penalty for dropping the puck away from home, but it is not always this straightforward to find a way to penalize the repetitive behavior. In [4] and [5], more general methods to avoid the cyclic movements have been presented, however, these methods require much prior knowledge.

Another way to solve the problem is by adding an extra state variable, which tells which subgoal has been reached. In [6] this is done by making a subtask for each subgoal. The root task learned to execute these subtasks in the correct order. Such a hierarchical approach is believed to be a good way to make RL suitable for complex tasks and more general frameworks have been developed later, of which [7] and [8] are the most popular. These frameworks also add regions in state space for each subtask, where the subtask is allowed to be executed. This limits the choice for the root task, which improves the learning speed. Another advantage of these frameworks is that they can be extended with hierarchical greedy execution (HGE). When this is used, the root task is allowed to go to the next subgoal, while the previous subgoal

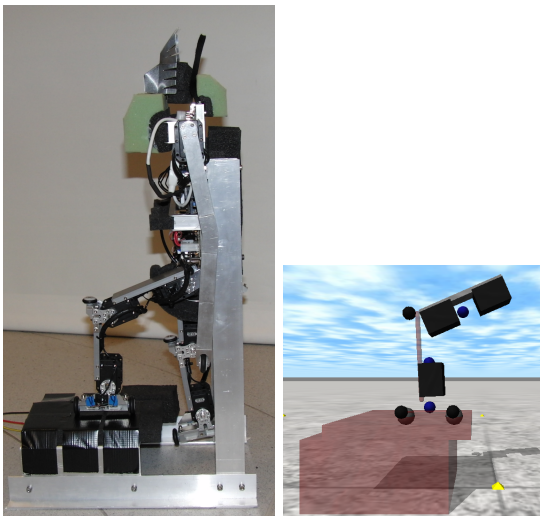


Figure 1: Leo, a 2D bipedal autonomous robot and the simulation of its left leg. Its task is to learn how to make a step onto the platform.

has not been reached yet. This is especially beneficial when the subgoals are not placed on the optimal path. In this case, HGE can bring the solution closer (or make it equal) to the optimal solution.

1.2 Goal

In this paper we test the use of subgoals on a task performed by a single leg of our robot LEO; see Figure 1. In section 4.1 we show that subgoals speed up the learning process. Also, we will show that the robot learns even faster when adding a region for each subgoal where it is allowed to be executed. In section 4.2 we compare these results with the same setting, but in a hierarchical structure. Furthermore, we will test whether HGE indeed helps to bring the solution closer to the optimal solution. Then we show the influence of bad subgoal placement in section 4.3. Lastly, we compare simulation results with results of the real robot in section 4.4, which is, to our knowledge, the first time the hierarchical framework MAXQ [8] is applied to a real robot.

2 Preliminaries

In the reinforcement learning (RL) framework, an agent interacts with the environment. The agent performs actions and perceives the state and rewards. The goal of the agent is to maximize the sum of rewards it perceives. The interaction is usually modeled as a Markov decision process (MDP). It is a discrete time process where, each time step, the agent perceives a state s and chooses an action a. The next time step, the agent ends up in a new state s' with probability $T(s, a, s')$ and receives a reward $r = R(s, a, s')$. This reward function R has to be designed by an engineer or user, while the state transition probability function T can be unknown. The agent chooses actions based on the state it is in, according to: $a = \pi(s)$, where π is the *policy* of the agent. A policy should exploit its current knowledge, but should also explore to find better solutions. To do so, we chose to use an ϵ-greedy policy, which simply chooses a random action with chance ϵ and a greedy action otherwise. A greedy action is the one that exploits the current knowledge; it is the action where the policy expects the highest sum of rewards. In order to find the greedy action, the policy uses the action-value function $Q(s, a)$. It contains the expected sum of discounted rewards for each state-action pair when taking action a in state s and following policy π afterward, as shown in (1).

$$Q(s,a) = E\left\{\sum_{i=1}^{i_{end}} \left[\gamma^{i-1} r_{t+i}\right] | s_t = s\right\} \qquad (1)$$

Where $\gamma \in [0, 1]$ is the time discount factor. The actual learning is done by *updating* the Q values such that they make a good prediction of the rewards that are going to be received.

We will use a general speed up method: eligibility traces. In short, the trace contains a history of state-action pairs. Instead of only updating the Q values of the last state-action pair, the whole history can be updated. One can argue that state-action pairs further back in history are less responsible for the last rewards. Therefore, the update of a state-action pair of k steps ago, is discounted by λ^k, with trace discount factor $\lambda \in [0, 1]$. However, some state-action

pairs can occur multiple times in the trace. To avoid such pairs getting an update that is too large, we limit the net trace discount factor to 1, as in [9].

2.1 Hierarchical RL

At the basis of hierarchical reinforcement learning (HRL) lies the Semi-MDP (SMDP) framework [7]. An SMDP is equal to an MDP, except that it allows an action to take multiple time steps. The rewards collected after taking an action that took N time steps, have to be discounted by γ^N.

The SMDP framework allows us to break up a single task into smaller tasks: a hierarchy of subtasks. We used the MAXQ framework [8], which was extended in [10] to include eligibility traces: MAXQ-Q(λ). In this framework, each subtask has its own state space, action space and goal. A subtask can only be executed when the current state is inside the subtask's region. The subtask at the top of the hierarchy, the root task, tries to solve the original task, while the other subtasks only try to reach their own goal. The root task usually is an SMDP; its actions consist of executing a subtask on a lower level. Once the root task has chosen a subtask to execute, this subtask is in control until it reaches its goal or gets outside of its region. Then the root task can select another subtask to execute. The subtasks on the lowest level of the hierarchy perform actions from the original task, which are called primitive actions. A MAXQ hierarchy with subgoals can be visualized in a graph as shown in Figure 2.

To make subtasks reach their own goal, each subtask i has its own pseudo reward function $\tilde{R}_i(s, a, s')$. These pseudo rewards are not visible to other subtasks, they merely influence the policy of the subtask.

Once the subtasks have learned sufficiently, they will always reach their goal. However, these goals may not lie on the optimal path of the overall task, the problem that the root task has to solve. To get closer to the optimal path, we can use hierarchical greedy execution (HGE) as described in [8]. The idea is to interrupt a currently active subtask and give the root task the opportunity to choose another subtask to execute. In this way, a new subtask can be given control before the previous executed subtask could reach its goal.

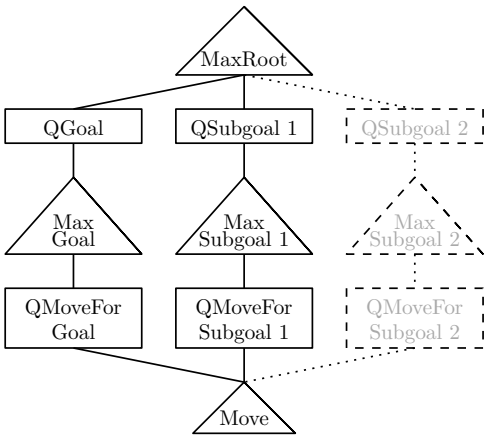


Figure 2: The MAXQ graph for one subgoal, each extra subgoal adds three nodes, like the dotted nodes.

However, interrupting a subtask influences the learning process. The Q value of a task can only be updated when the subtask it executed has finished. Also, when interrupting a subtask, its eligibility trace has to be cleared, since the state-action pairs in the trace can no longer be held responsible for future state-actions. Therefore, we should not use HGE from the beginning but enable it after the subtasks have sufficiently learned to reach their goals.

In [8], HGE is gradually enabled by interrupting the currently executed subtask after a decreasing amount of time steps. This results in interrupting at the same time at each run. Since there could be a correlation between the time of interrupting and learning rate when averaging over multiple runs, we chose to slowly increase the *chance* of an interruption, according to:

$$p_{interruption} = \left\{ \begin{array}{ll} \frac{\kappa}{(t_{hge}-t)+\kappa} & \text{for } t < t_{hge} \\ 1 & \text{for } t \geq t_{hge} \end{array} \right\}, \quad (2)$$

where κ is a parameter to modify the curvature and t_{hge} the time when HGE is fully active; when there is an interruption each time step. For a constant chance p on an interruption, the chance of having k time steps in between two interruptions is given by:

$$f(k, p) = (1-p)^{k-1} p \quad (3)$$

In case the root task has no other option but choosing the same subtask again, interrupting would not lead to a different choice, it would only clear the eligibility trace. Therefore, we do not interrupt in such a case.

More information about the MAXQ-Q(λ) algorithm can be found in appendix A.

3 Experiment

To test the use of subgoals on a robot, we performed experiments on a simulated and a real robot leg. The task we gave it is a motor skill task without a straightforward solution.

3.1 Setup

The robot leg is part of a humanoid robot of approximately half a meter tall [11]. For this research we built a stand (see Figure 1) with which we fixed the torso of the robot and we only used the left leg. Each joint is actuated by a Dynamixel RX-28 motor. Although this is a servo motor, we bypass the internal controller and operate it in voltage mode. To detect foot contact, there are two force sensors at the bottom of the foot, one at the toe and one at the heel.

The simulation of the leg is made with ODE [12]. We used dimensions, masses and inertias equal to those of the real robot. The motor torque M is calculated as a function of voltage U by:

$$M(U) = kG\frac{U - kG\omega}{R}, \tag{4}$$

where k is the torque constant, G the gearbox ratio, R the winding resistance and ω the angular velocity. The inertia of the armature, which has a significant influence for large gearbox ratios, is also included in the simulation.

The primitive actions consist of choosing between 5 voltages (-14, -7, 0, 7, 14) for the hip and knee motors, resulting in 25 possible actions. The ankle motor was controlled to keep the foot perpendicular to the lower leg. The sampling frequency was 75 Hz, the same that was used for early walking experiments. The state of the robot consists of 7 variables: the foot contact and the angle plus angular velocity of each joint. The state was discretized by tile coding [13] with 16 tilings, the resolutions are listed in table 1. The contact variable is binary; at a certain force at the toe and heel, the variable is true.

Joint	Angle step size	Velocity step size
Hip	0.0476 rad	6.67 rad/s
Knee	0.0769 rad	9.09 rad/s
Ankle	0.100 rad	11.1 rad/s

Table 1: State space resolution per tiling

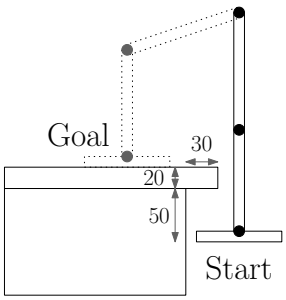


Figure 3: The task to perform: make a step, measures are in mm.

Information about the software can be found in appendix C.

3.2 Task

The task of the robot is to make a step up, as shown in Figure 3. An episode starts with the leg in a straight down position and ends when the goal is reached, or after 60 (simulated) seconds, equal to 4500 time steps. The goal is reached when the foot makes contact at both the toe and heel, which is only possible in a position near the position shown in Figure 3.

We used a simple reward scheme, where a reward of 100 was given when the task was completed. Each time step a reward of -1 was given, to make the robot minimize the time to reach the goal. A subtask receives a pseudo reward of 100 when reaching its subgoal and a pseudo reward of -100 when leaving its region. A subgoal was defined as a small position area, irrespective of the joint angular speeds. We de-

fined subgoals in this way, because a position is easier to imagine for a person.

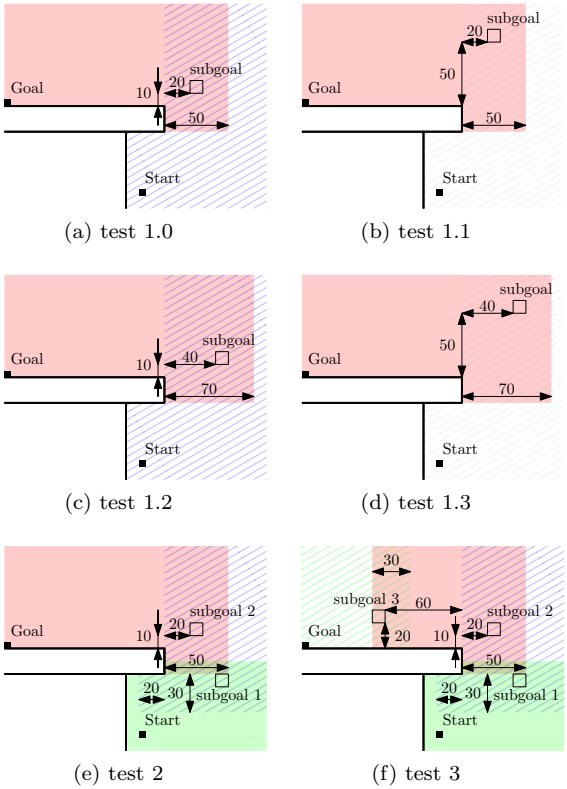
The Q values were initialized with random values between -1 and 1. In this way, a path that has timed out, left the region, or took too much time to reach the subgoal obtains lower values than the initial values, since these paths gathered many negative rewards. This results in more exploration at the early learning stage, as the agent will avoid these negative paths.

The learning parameters were not tweaked, since the initial choice was satisfying. We chose $\alpha = 0.25$, $\gamma = 1$, $\epsilon = 0.05$ and $\lambda = 0.9$. For the HGE parameters, we used $t_{hge} = 100$ minutes and $\kappa = 0.0833$.

3.3 Simulation test setup

To see the effect of using subgoals, we compared learning without subgoals (test 0) to learning with one, two and three subgoals (test 1, 2 and 3). The locations of the subgoals and their regions are shown in Figure 4. Each subgoal is defined as an area of 10mm by 10mm, which the toe should reach.

Since the placement of these subgoals often has to be guessed, we will also take a look at the influence of the placement of the subgoal in test 1, by placing it at different positions, further away from the optimal path (test 1.1, 1.2 and 1.3).



(a) test 1.0 (b) test 1.1

(c) test 1.2 (d) test 1.3

(e) test 2 (f) test 3

Figure 4: Placement of the subgoals (squares) and their regions where they are allowed to be executed (colored areas), measures are in mm.

3.4 Robot test setup

Apart from the simulation tests, we also performed tests on the real robot Leo, to see if the simulation results match the results of the robot. Due to the long time real tests take, we only performed test 0 and 2. Real hardware brings several problems that a simulation does not have. One major problem is the time between perceiving a state and performing the action, the control delay [14]. We found that the control delay of our system was on average a quarter of a time step. Such a delay could decrease the learning performance. To avoid this, we tried larger time steps, so that the control delay would be negligibly small compared to the time step. The time step should still be small enough, so that it would still take about 20 time steps to reach the goal, to ensure that a difference in performance can be quantified by the number of time steps it takes to reach the goal. A sampling frequency of 20 Hz, thus 3.5 times larger time steps, turned out to be working well. We also found out we could decrease the resolution of each state variable, without getting instabilities. Since the path to the goal is now 3.5 times shorter, we also scaled λ in order to have a 3.5 times shorter eligibility trace. In order to fully compare this with a simulation, we also simulated with those settings.

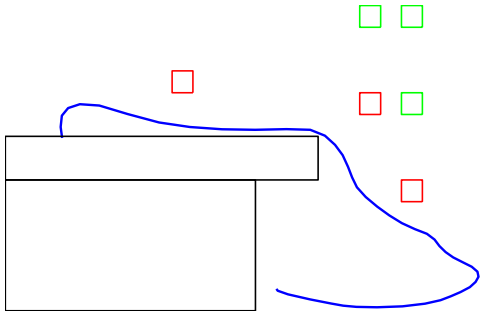


Figure 5: Solution found by flat learning, the line represents the position of the toe over time. Subgoals used by other tests are drawn as squares.

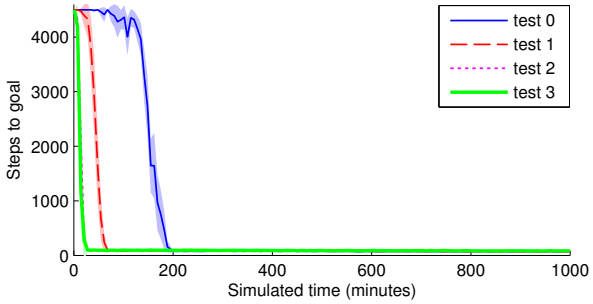


Figure 6: Learning curves, with standard error bars, for 0 to 3 subgoals.

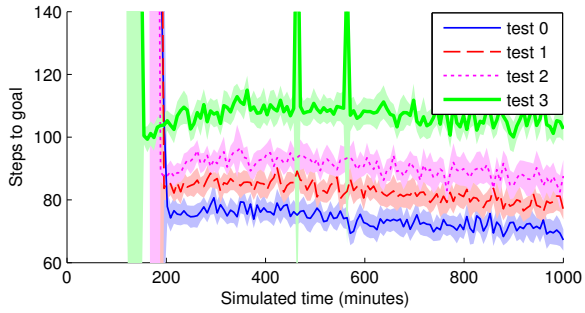


Figure 7: Learning curves for 0 to 3 subgoals, using a flat method with extra state variable, without regions for each subgoal.

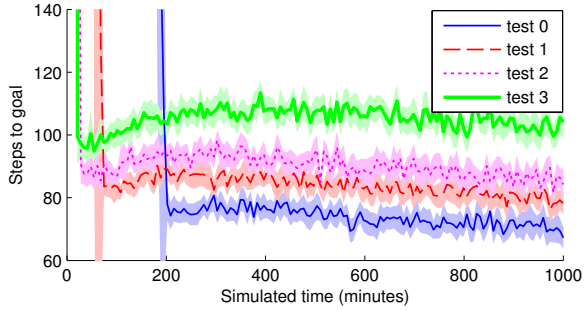


Figure 8: Learning curves for 0 to 3 subgoals, using a flat method with extra state variable, with regions for each subgoal.

4 Results

An example solution, found by flat learning without subgoals, shows that the chosen subgoals most likely lie off the optimal path, see Fig. 5. The performance is defined as the inverse of the average number of steps it takes to reach the goal. Typically the performance increases drastically in a short time period, after which it slowly increases, as shown in Fig. 6. Therefore, we stopped simulating after 1000 minutes and do not know the optimal performance. Consequently, we defined the end performance as the smoothed performance after 1000 minutes. The fall time is defined as the time it takes until the average number of steps reached 10% of the number of steps at the start of the learning. Each test is averaged over 25 runs.

4.1 Flat learning with subgoals

When using subgoals without regions in which they are allowed to be executed, the fall time decreases when using more subgoals (see Fig. 7). However, the end performance also decreases with more subgoals.

If we add regions, the fall times become considerably smaller for tests 1, 2 and 3, as shown in Fig. 8. The fall time of test 1 is approximately three times smaller than test 0, while the fall time of test 2 and 3 are roughly nine times smaller compared to test 0. The end performances remain nearly equal. The small difference of fall time between test 2 and 3 can be explained by the ease of learning the last part of the task; once the robot managed to reach subgoal 2, the path to the end goal was quickly discovered.

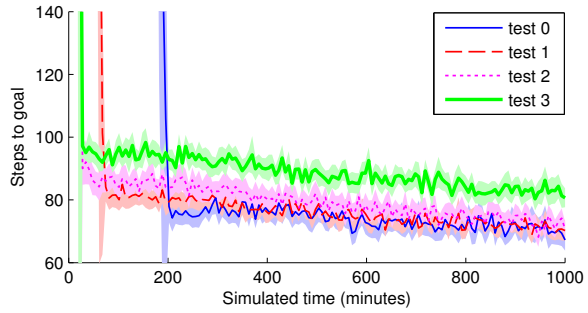


Figure 9: Learning curves for 0 to 3 subgoals, when using MAXQ-Q(λ) without HGE.

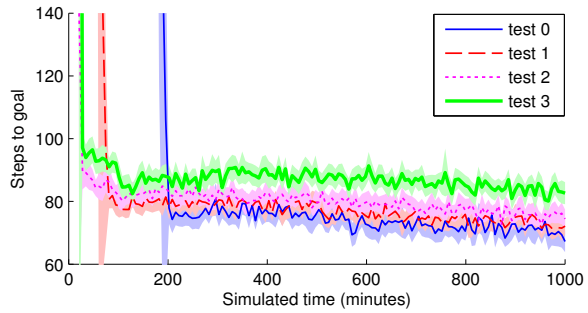


Figure 10: Learning curves for 0 to 3 subgoals with HGE.

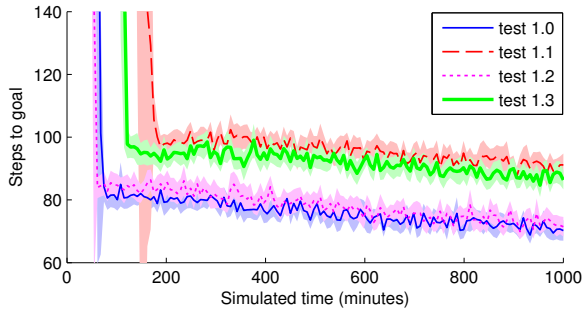


Figure 11: Learning curves for different places of the subgoal, without HGE.

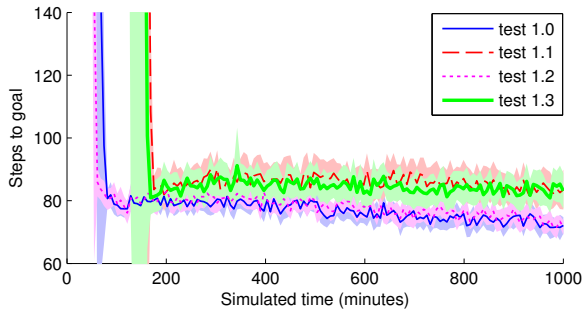


Figure 12: Learning curves for different places of the subgoal, with HGE.

4.2 Subgoals in a hierarchy

When we compare flat learning (Fig. 8) to MAXQ-Q(λ) (Fig. 9) with the same subgoals and regions, we see that the hierarchical method has the same fall times, but better end performances. Since the only difference between the two methods is that MAXQ-Q(λ) has an eligibility trace for each subtask, instead of one trace for the whole task, it seems that emptying the trace when a subgoal has been reached has a positive effect on the exploration.

When using HGE, the fall times and end performances remain practically the same, as can be seen in Figure 10. However, between 100 and 300 minutes, just after HGE was fully activated, the performance of test 2 and 3 is higher than without HGE.

4.3 Subgoal placement

In Figure 11, we see that tests 1.1 and 1.3 have a substantially lower end performance and a longer rise time compared to tests 1.0 and 1.1. Apparently, the subgoals of tests 1.1 and 1.3 lie further away from the optimal path than the subgoals of tests 1.0 and 1.2, which can also be seen in Figure 5.

By using HGE, there is an increase in fall time for tests 1.1 and 1.3 (see Fig. 12), which could be caused by the fact that t_{hge} (as in formula 2) is smaller than the fall time. The end performance on the other hand, increased. Again, between 100 and 200 minutes the performance is higher, compared to the same interval without HGE.

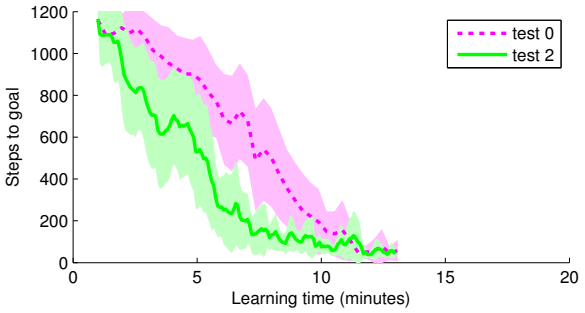


Figure 13: Learning curves of the real robot.

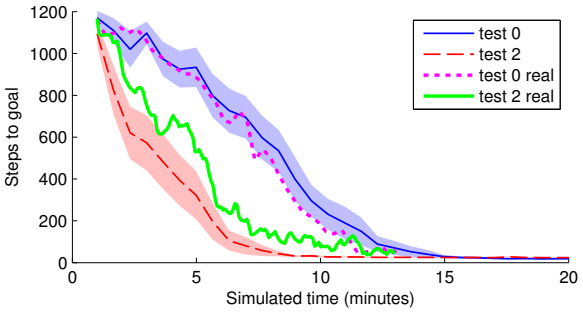


Figure 14: Learning curves of a simulation with the same settings as for the real robot.

4.4 Tests on Leo

On the real robot, the tests were performed multiple times. After removing runs in which hardware problems occurred, we could average test 0 over 12 runs and test 2 over 13 runs. The hardware problems that occurred, were a broken gearbox at the hip actuator and a toe force sensor that got stuck multiple times, which lead to the robot receiving a reward for kicking its behind. Due to a software bug, the runs were aborted after 13 minutes of learning time, instead of 20 minutes. The results can be found in Fig. 13. Due to the smaller state space, the likeliness of finding the goal by random actions is higher. This explains the smaller difference in fall time between test 0 and test 2, which is around 1.5 times. The end performance of test 2 seems to be equal to that of test 0.

The learning curves of the simulation with the same settings as we used on the real robot, together with the average learning curves of the real robot, are plotted in Figure 14. When comparing simulation results with the results of the real robot, we can see that the learning curves of test 0 are roughly similar. Test 2, however, has a lower fall time and a higher end performance than test 2 on the real robot. This can be explained by observations we made when the robot was learning. We noticed how the robot was shaking when nearly reaching a subgoal, indicating it was hard to reach the subgoal. We suspect this was caused by sensor noise and backlash, which makes it hard to precisely reach the small subgoal area.

5 Discussion and conclusions

When learning motor skills with RL, the use of subgoals can lead to a faster learning rate. However, this will usually lead to a worse performance after convergence. A hierarchical framework like MAXQ adds a region for each subgoal in which the subgoal is allowed to be executed. These regions decrease the state space and thus speed up the learning process. Another advantage of MAXQ is the possibility to use HGE. It can result in a policy closer to the optimal policy, especially when the subgoals lie further away from the optimal path. Such subgoals also result in longer learning times. The real robot also benefits from subgoals, but performed worse than the simulation due to the small subgoal areas.

We conclude that the use of subgoals is very useful for RL on a robot, as long as the subgoals lie close to the optimal path and are not too small in comparison with the robot's precision.

6 Future work

It is interesting to test the use of subgoals for a more complicated task, like standing up. This task was originally meant to be used for this paper, but we decided to start with a simpler task. We already found that the stand up task is suitable to be guided by subgoals, see appendix B.

With the methods we described in this paper, there is a trade-off between learning speed and optimality.

We believe there is a way to achieve both fast learning and an optimal policy. This can be accomplished by a method called "all goals updating" [15]. With this method, not only the subtask which is in control gets Q updates, but also other subtasks are updated. In this way a subtask is learning while not in control. We can use this by learning a flat policy, while learning in the hierarchical way. This can be implemented by performing Q updates for the last subtask, the one that brings the robot to the final goal, outside its region. At some point in time, the last subtask is allowed to be executed outside its region, where it already learned with the all goals updating method.

References

[1] Ö. Şimşek, A.P. Wolfe, and A.G. Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd international conference on Machine learning*, pages 816–823. ACM New York, NY, USA, 2005.

[2] M.J. Mataric. Reward functions for accelerated learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, volume 189, 1994.

[3] J. Randløv. *Solving complex problems with reinforcement learning*. PhD thesis, University of Copenhagen, 2001.

[4] A.Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning (ICML'99)*, pages 278–287, 1999.

[5] A.D. Laud. *Theory and application of reward shaping in reinforcement learning*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.

[6] S.P. Singh. Transfer of learning across compositions of sequential tasks. In *Proceedings, Eighth International Conference on Machine Learning', Morgan Kaufmann, Evanston, Illinois*, pages 348–352, 1991.

[7] R.S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999.

[8] T.G. Dietterich. The maxq method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 118–126, 1998.

[9] S.P. Singh and R.S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine learning*, 22(1):123–158, 1996.

[10] Erik Schuitema. Hierarchical reinforcement learning. Master's thesis, Delft University of Technology, Delft, The Netherlands, 2006.

[11] Erik Schuitema, Martijn Wisse, Thijs Ramakers, and Pieter Jonker. The design of LEO: a 2D bipedal walking robot for online autonomous reinforcement learning. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, Taipei, Taiwan, October 2010.

[12] http://ode.org/. last visited in Oct 2010.

[13] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[14] Erik Schuitema, Lucian Busoniu, Thijs Ramakers, Robert Babuska, and Pieter Jonker. Control delay in reinforcement learning for real-time dynamic systems: a memoryless approach. In *Conference on Intelligent Robots and Systems (IROS-10)*, 2010.

[15] T.G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

A The MAXQ-Q(λ) algorithm

Here we will describe the MAXQ-Q(λ) algorithm into more detail.

In MAXQ, the value function of subtask i is defined as

$$Q_i(s,a) = V(s,a) + C_i(s,a) \tag{5}$$

$$V(s,a) = \begin{cases} Q_a(s,\pi_i(s)) & \text{if } a \text{ is a subtask} \\ V_p(s,a) & \text{if } a \text{ is primitive} \end{cases} \tag{6}$$

Since the actual policy of a subtask is based on rewards plus pseudo rewards, each subtask also has a pseudo value function $\tilde{C}_i(s,a)$, which predicts the expected pseudo plus real reward. Thus, the greedy action of the policy for subtask i is:

$$\pi_{i,greedy}(s) = \arg\max_{a'} \left[\tilde{C}_i(s,a') + V(s,a')\right] \tag{7}$$

The eligibility trace is stored in $e_i(s,a)$, which contains the discount factor for each state-action pair.

A subtask terminates when it reached its goal or when it got outside of its region. It also has to terminate when its parent has to terminate. In all those cases $T_i(s)$ is true.

The pseudo code of the MAXQ-Q(λ) algorithm, including HGE, can be found in table 2. In the real time implementation, the algorithm has to be cut open between lines 3 and 4. At line 3 the actuation commands are sent to the simulator or motors. The next time step, the program should continue at line 4, where the new state is perceived. In order to achieve this, the complete calling stack has to be stored, as well as the temporary variables.

The part that usually costs the most calculation time is to find the maximum Q value (line 15), especially for the root, as it has to loop through all C values of all subtasks lower in the tree. At this moment, once a subtask has been chosen, this subtask will calculate its maximum Q value again, while this has just been done. A future improvement would be to cache the maximum Q value.

1: **function** MAXQ-Q(λ)(MaxNode i, State s)
2: **if** i is a primitive MaxNode **then**
3: execute i // Send control commands
4: receive $R(s,i,s')$, and observe result state s' // Receive state and rewards
5: $V(s,i) \leftarrow (1-\alpha_i) \cdot V(s,i) + \alpha_i \cdot R(s,i,s')$
6: **return** 1
7: **else**
8: $count \leftarrow 0$
9: initialize $e_i(s,a) = 0$ for all s, a // Reset trace
10: choose an action a according to the current exploration policy $\pi_i(s)$
11: **while** $T_i(s)$ is false **do**
12: $N \leftarrow$ MAXQ-Q(λ)(a,s) // Recursive call
13: observe result state s', reward $R(s,i,s')$ and pseudo-reward $\tilde{R}_i(s,i,s')$
14: choose an action a' according to the current exploration policy $\pi_i(s')$
15: $a^* \leftarrow \arg\max_b[\tilde{C}_i(s',b) + V(s',b)]$
16: **if** s' is terminal and absorbing **then**
17: $\tilde{C}_i(s',a^*) \leftarrow 0, \quad C_i(s',a^*) \leftarrow 0, \quad V(s',a^*) \leftarrow 0$
18: **end if**
19: $\tilde{\delta} \leftarrow \gamma^N[R_i(s,a,s') + \tilde{R}_i(s,a,s') + \tilde{C}_i(s',a^*) + V(s',a^*)] - \tilde{C}_i(s,a)$
20: $\delta \leftarrow \gamma^N[R_i(s,a,s') + C_i(s',a^*) + V(s',a^*)] - C_i(s,a)$
21: $e_i(s,a) \leftarrow 1$ // Replacing trace
22: **for all** s,a with $e_i(s,a) > e_{min}$ // Limited trace length **do**
23: $\tilde{C}_i(s,a) \leftarrow \tilde{C}_i(s,a) + \alpha_i \tilde{\delta} e_i(s,a)$
24: $C_i(s,a) \leftarrow C_i(s,a) + \alpha_i \delta e_i(s,a)$
25: **if** $a' = a^*$ // Greedy action **then**
26: $e_i(s,a) \leftarrow \gamma^N \lambda(i) e_i(s,a)$ // Let the trace decay
27: **else**
28: $e_i(s,a) \leftarrow 0$ // Cut off trace on exploration
29: **end if**
30: **end for**
31: $count \leftarrow count + N$
32: $s \leftarrow s'; a \leftarrow a'$
33: **if** HGE **and** $i \neq$ rootnode **and** rootnode has no other subtask to choose than i **then**
34: break
35: **end if**
36: **end while**
37: **end if**
38: **return** *count*

Table 2: The MAXQ-Q(λ) learning algorithm in pseudo-code, including HGE

B Stand up with subgoals

When Leo is learning to walk, it often falls, hence it would be nice to have a fast stand up policy. This is a suitable task to be learned with subgoals.

To keep the learning time low, the action space can be reduced. First of all, symmetry can be exploited by coupling left and right leg actuators. This reduces the number of actuated DOF from seven to four. To reduce the action space even further, we designed a simple stand-up sequence in which only three actuators are active at the same time. The subgoals of this policy are shown in Figure 15, while the unused actuators are listed in table 3.

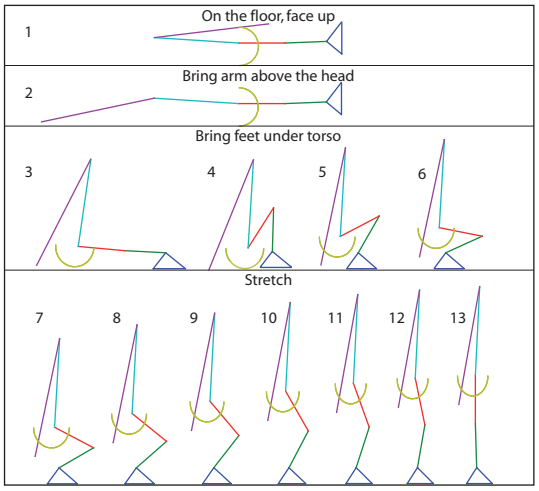


Figure 15: Stand up sequence for face up

We tested this sequence on the real robot, by using a PD controller which brought the angles to the given subgoals. The robot was able to stand up with this sequence, as shown in figure 16, meaning we can use the subgoals for RL.

Transition	Shoulder	Hip	Knee	Ankle
1–2		x	x	x
2–3			x	
3–4	x			
4–5			x	
5–6	x		x	
6–7	x			
7–8	x			
8–9	x			
9–10	x			
10–11	x			
11–12	x			
12–13	x			

Table 3: Unused actuators during transitions from subgoal to subgoal for the stand up sequence with face up

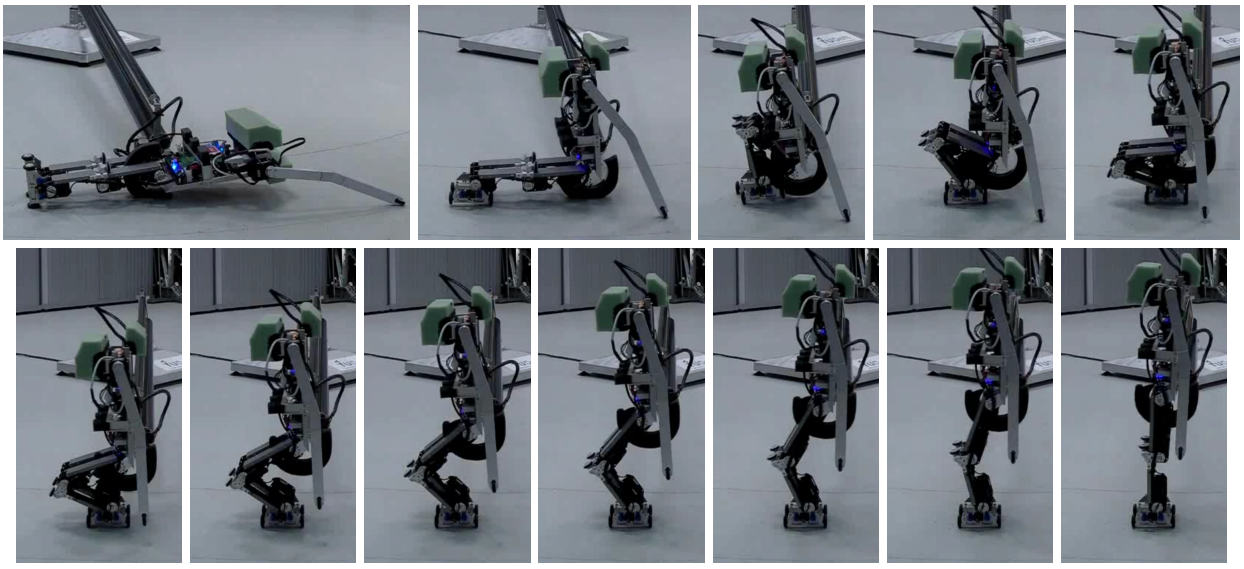


Figure 16: Leo standing up with the calculated sequence

C Software architecture

In this appendix, the software architecture is presented in class diagrams. The software is made in C++ and we made use of an existing framework, developed in our lab. This framework is made to work for both a real time environment and simulation. The State Transition Generator (STG) is the part that provides the new state plus time and can receive actuation commands, see Figure 17. The STG can be either a simulator or a real hardware device, both use the same communication, so a policy can be used for both the simulation and the real robot. The policy player chooses a policy and lets this policy choose the actuation commands to send to the STG.

In Figure 18 the class diagram of the program for the real robot is shown. The program has three policies: one to cool down when the actuators become too warm, one to bring the leg back into its starting position and the policy that learns to make a step up. The latter is shown more detailed in Figure 20.

The program to run the simulation is an extension of an existing program, which was used for walking experiments on the robot. In the extended program, the policy and simulation to be used are settings in an XML file. In this way, the program can run both the step up as well as the walking simulation, depending on the settings in the XML file. Furthermore, when the program has been given an XML file as parameter, the console version of the program will be started, which is why the class diagram splits in two (see Fig. 19). The console version is especially useful when multiple simulations have to be performed, and enables the user to run the program on a remote machine via SSH.

The policy CLLSMaxq is the policy which learns to perform the step up task, the diagram of this policy is shown in Figure 20. The implementation of subgoals in a hierarchy is shown in the path on the left. Each subgoal has its own Max-node, as in Figure 2. Flat learning without subgoals was implemented as a single Q-node (CLLSQMoveForGoal). To implement flat learning with subgoals, we made a separate root node (CLLSMaxRootPlus) with a single Q-node. All the lower Q-nodes are connected to the same primitive node, that sends out the actuation command. The number of subgoals, subgoal areas, subgoal regions and learning parameters of the nodes are defined in the XML file.

The inheritance trees of most classes used in the programs are shown in Figures 21, 22, 23 and 24. The listed functions are public functions and are not listed in derived classes, even if they are overridden.

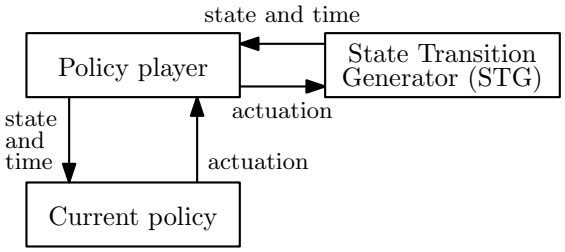


Figure 17: Overview of the framework.

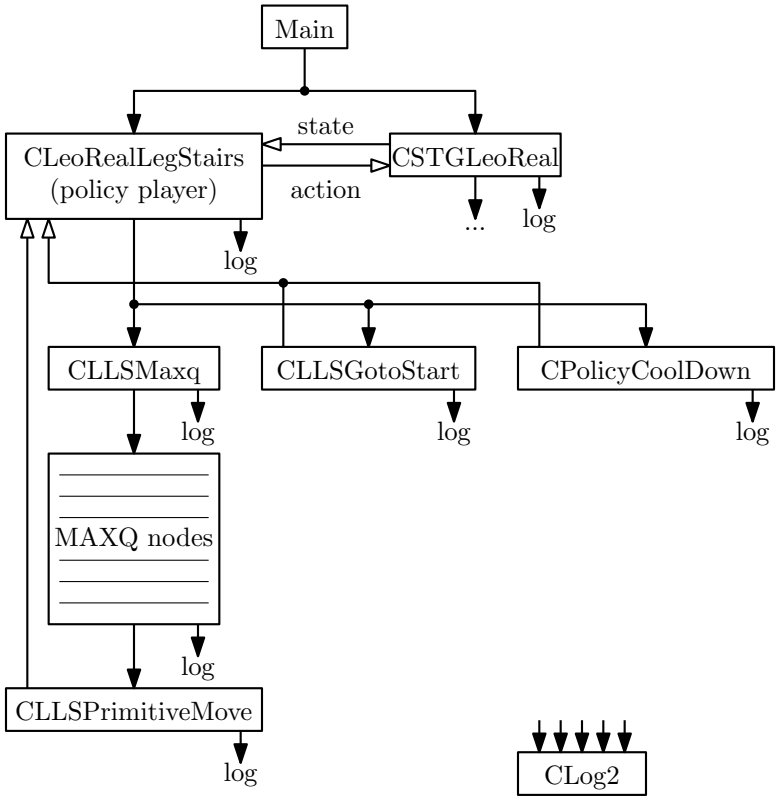


Figure 18: Class diagram of the program for the real robot. The CLLSMaxq policy and its nodes are shown more detailed in figure 20

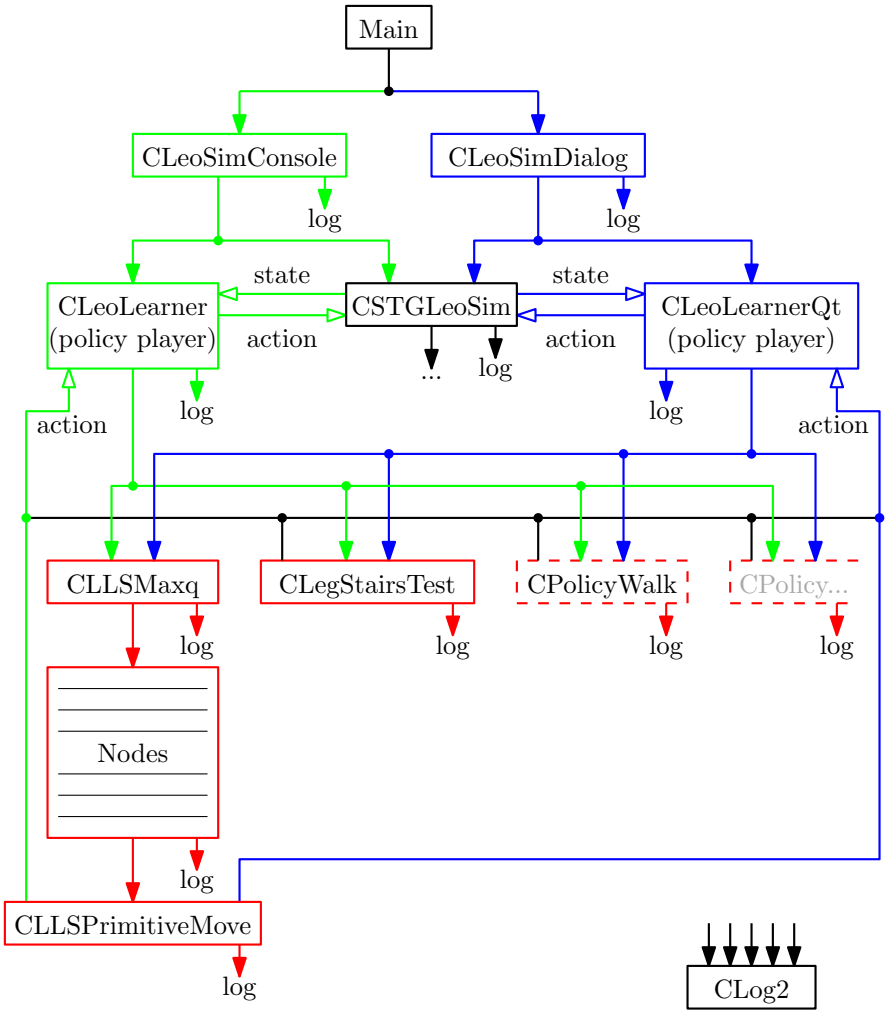


Figure 19: Class diagram of the simulation program. The red part (bottom) are the policies, the blue part (right) is used when the user chooses a dialog, else the green part (left) is used.

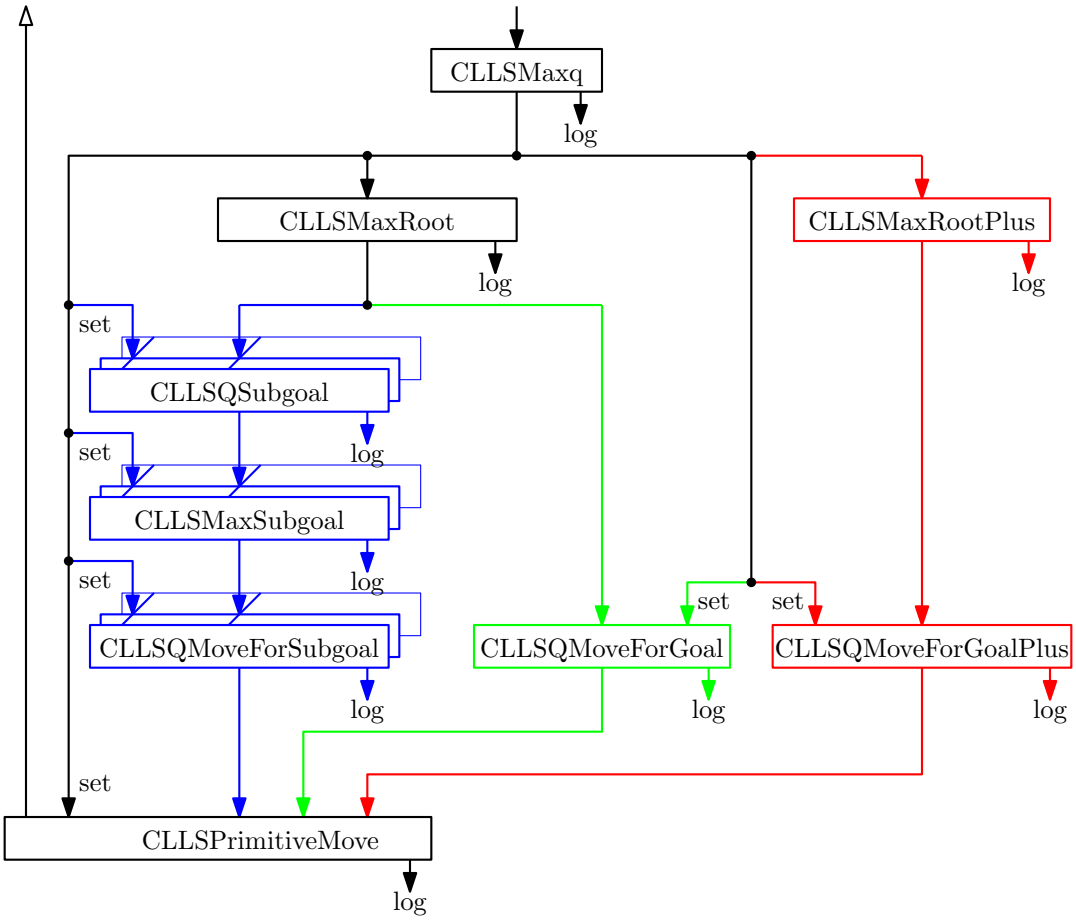


Figure 20: Class diagram of the step up policy. The blue part (left) is used when having one or more subgoals in a hierarchy, the green part (center) when having no subgoals, the red part (right) for flat learning with subgoals

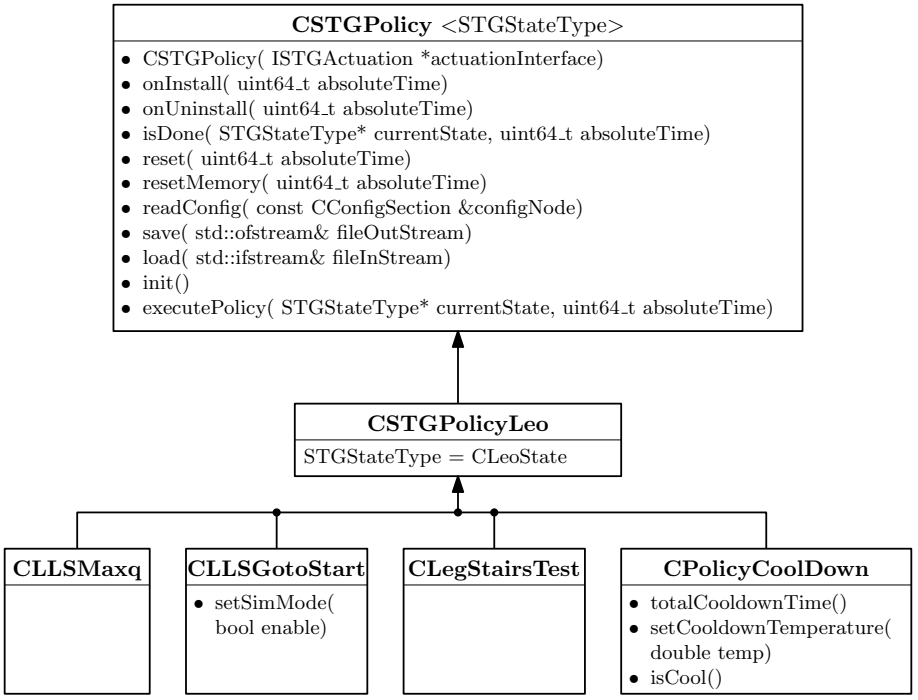


Figure 21: Inheritance diagram of the policies. The base class is on top, derived classes are connected to the base class.

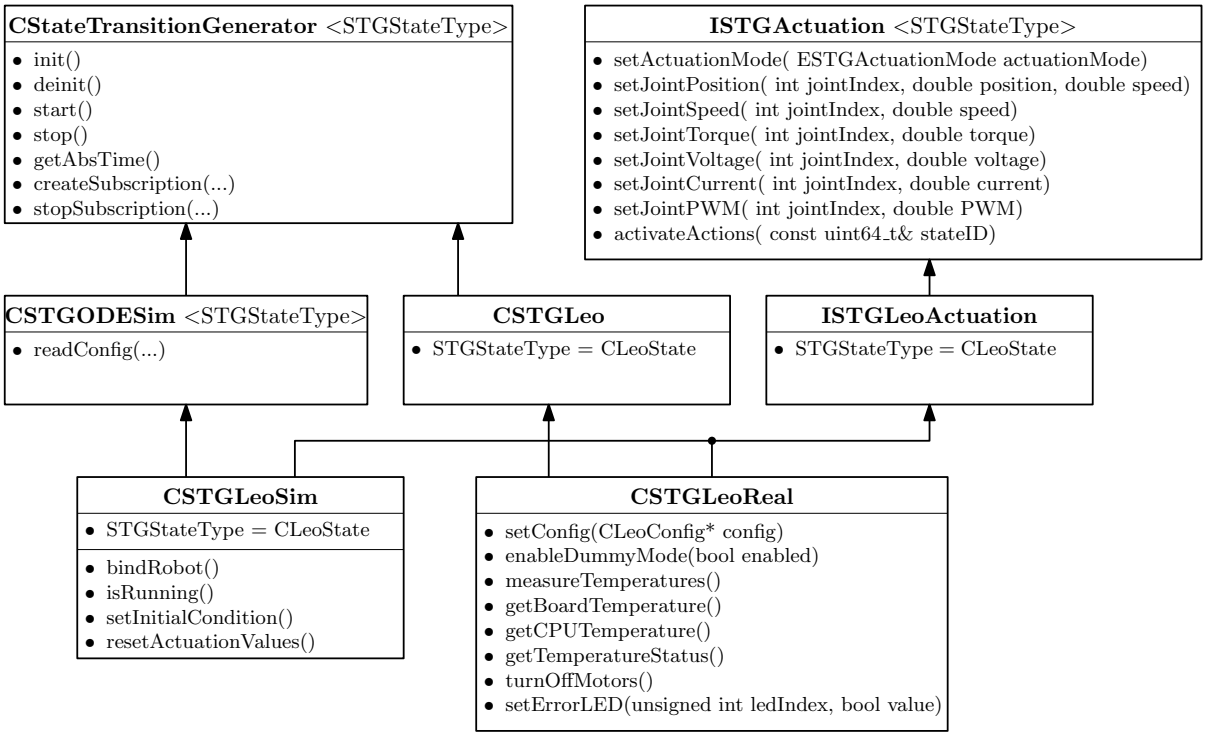


Figure 22: Inheritance diagram of the state transition generators. The base classes are on top, the derived classes point to their base classes.

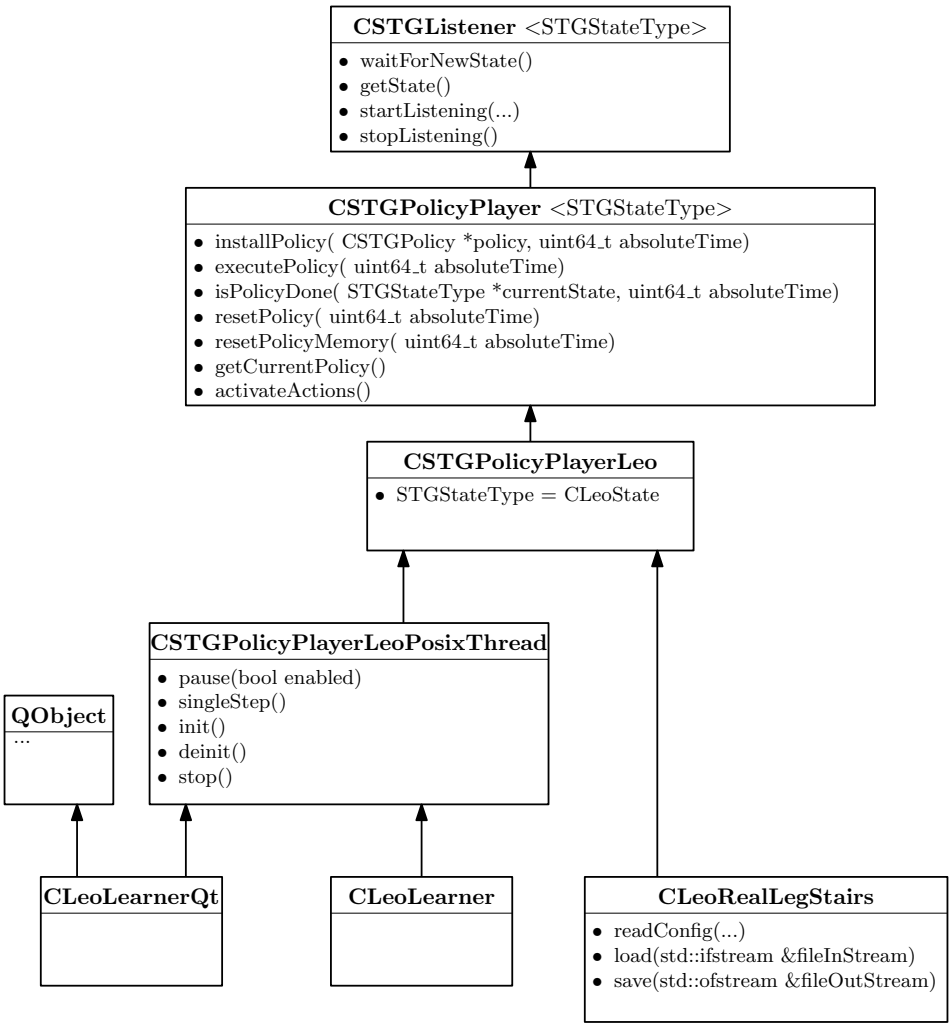


Figure 23: Inheritance diagram of the policy players. The base classes are on top, the derived classes point to their base classes.

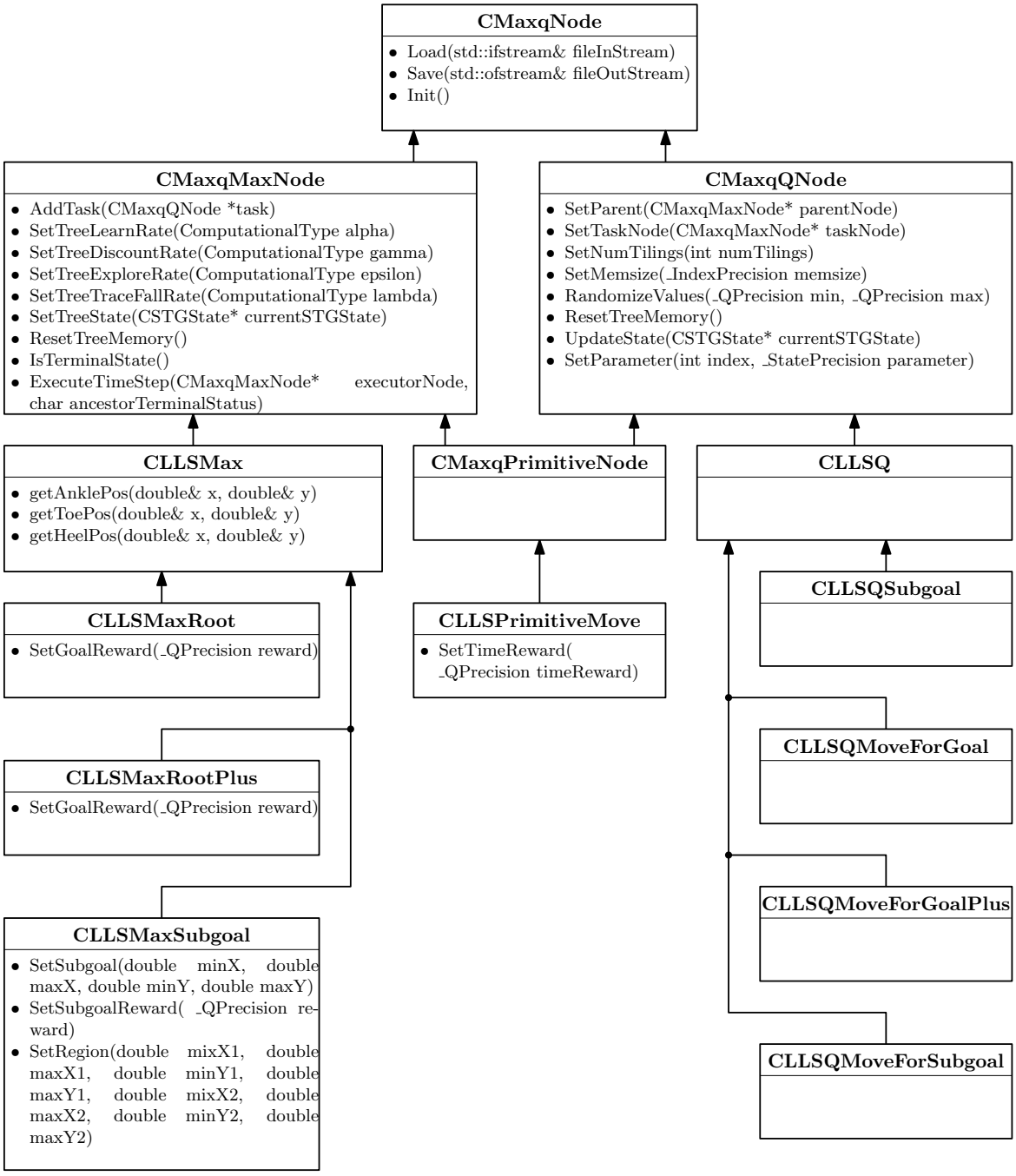


Figure 24: Inheritance diagram of the MAXQ nodes. The base classes are on top, the derived classes point to their base classes.