To appear in *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence* (AAAI-12), Toronto, Ontario, Canada, July 2012.

Design and Optimization of an Omnidirectional Humanoid Walk: A Winning Approach at the RoboCup 2011 3D Simulation Competition

Patrick MacAlpine and **Samuel Barrett** and **Daniel Urieli** and **Victor Vu** and **Peter Stone**
Department of Computer Science
The University of Texas at Austin
Austin, TX 78701, USA
{patmac, sbarrett, urieli, diragjie, pstone}@cs.utexas.edu

Abstract

This paper presents the design and learning architecture for an omnidirectional walk used by a humanoid robot soccer agent acting in the RoboCup 3D simulation environment. The walk, which was originally designed for and tested on an actual Nao robot before being employed in the 2011 RoboCup 3D simulation competition, was the crucial component in the UT Austin Villa team winning the competition in 2011. To the best of our knowledge, this is the first time that robot behavior has been conceived and constructed on a real robot for the end purpose of being used in simulation. The walk is based on a double linear inverted pendulum model, and multiple sets of its parameters are optimized via a novel framework. The framework optimizes parameters for different tasks *in conjunction* with one another, a little-understood problem with substantial practical significance. Detailed experiments show that the UT Austin Villa agent significantly outperforms all the other agents in the competition with the optimized walk being the key to its success.

1 Introduction

While robots have been trained to capably perform specific tasks in static or relatively simple environments, such as navigating stairs (Shih 1999) or tracking and grabbing an object (Allen et al. 1993), it is still a hard problem to design them to generalize well to dynamic and complex environments. Designing intelligent behaviors for such environments by hand can be difficult, and thus it becomes desirable to automatically learn these behaviors. When the learning process is non-trivial, it can be beneficial to use some kind of decomposition.

In this paper, we investigate learning a multi-purpose humanoid walk in the robot soccer domain. We carefully decompose the agent's behavior into a representative set of subtasks, and then learn multiple walk engine parameter sets *in conjunction* with each other. This results in a learning architecture for a humanoid robot soccer agent, which is fully deployed and tested within the RoboCup[1] 3D simulation environment, as a part of the champion UT Austin Villa team.

The research reported in this paper is performed within a complex simulation domain, with realistic physics, state noise, multi-dimensional actions, and real-time control. In this test domain, teams of nine autonomous humanoid robots play soccer in a physically realistic environment. Though no simulation perfectly reflects the real world, the physical realism of the RoboCup domain enables pertinent research on realistic robotic tasks such as humanoid locomotion. An important advantage of working in simulation is that extensive experimentation and learning is possible without risk of mechanical wear and tear on any physical device.

As each robot is controlled through low-level commands to its joint motors, getting the robot to walk without falling over is a non-trivial challenge. In this paper, we describe a parameterized omnidirectional walk engine, whose parameters directly affect the speed and stability of the robot's walk, such as step height, length, and timing. Optimal parameters would allow the robot to stably walk as fast as possible in all situations. However, the set of possible walking directions is continuous, so it is infeasible to learn specific parameters for each direction. Therefore, the robot learns parameters for common subtasks needed to play soccer such as walking to a target and dribbling a soccer ball. In turn, these subtasks are combined into higher-level behaviors such as dribbling a ball to the goal. The primary contribution of this paper is a methodology for splitting a high-level task, in our case robust omnidirectional humanoid locomotion, into simpler subtasks; and optimizing parameters for these subtasks while respecting the coupling induced by the high-level task.

Additionally, we show evidence that conjunctive parameter set optimization can yield a very competitive soccer agent. The starting point for this work was a competent, but not championship-caliber agent from the 2010 RoboCup competition, UT Austin Villa 2010. That agent finished just outside the top 8 in the competition. Though a key research component of the agent was a machine-learning-based walk, the walk was not able to keep up with the top agents. As fully described in (Urieli et al. 2011), UT Austin Villa 2010's walk was not omnidirectional and could only move in set directions such as forward, sideways, and backwards. A main contribution of this work is an extension and adaptation of the approach from that research to a fully omnidirectional walk engine. The UT Austin Villa 2011 agent described here won the 2011 RoboCup 3D simulation competition by winning all 24 matches it played, scoring 136 goals while conceding none.

[1]http://www.robocup.org/

2 Domain Description

Robot soccer has served as an excellent platform for testing learning scenarios in which multiple skills, decisions, and controls have to be learned by a single agent, and agents themselves have to cooperate or compete. There is a rich literature based on this domain addressing a wide spectrum of topics from low-level concerns, such as perception and motor control (Behnke et al. 2006; Riedmiller et al. 2009), to high-level decision-making problems (Kalyanakrishnan and Stone 2010; Stone 1998).

The RoboCup 3D simulation environment is based on SimSpark,[2] a generic physical multiagent system simulator. SimSpark uses the Open Dynamics Engine[3] (ODE) library for its realistic simulation of rigid body dynamics with collision detection and friction. ODE also provides support for the modeling of advanced motorized hinge joints used in the humanoid agents.

The robot agents in the simulation are homogeneous and are modeled after the Aldebaran Nao robot,[4] which has a height of about 57 cm and a mass of 4.5 kg. The agents interact with the simulator by sending torque commands and receiving perceptual information. Each robot has 22 degrees of freedom: six in each leg, four in each arm, and two in the neck. In order to monitor and control its hinge joints, an agent is equipped with joint perceptors and effectors. Joint perceptors provide the agent with noise-free angular measurements every simulation cycle (20 ms), while joint effectors allow the agent to specify the torque and direction in which to move a joint. Although there is no intentional noise in actuation, there is slight actuation noise that results from approximations in the physics engine and the need to constrain computations to be performed in real-time. Visual information about the environment is given to an agent every third simulation cycle (60 ms) through noisy measurements of the distance and angle to objects within a restricted vision cone (120°). Agents are also outfitted with noisy accelerometer and gyroscope perceptors, as well as force resistance perceptors on the sole of each foot. Additionally, a single agent can communicate with the other agents every other simulation cycle (40 ms) by sending messages limited to 20 bytes. Figure 1 shows a visualization of the Nao robot and the soccer field during a game.

3 Walk Engine

The UT Austin Villa 2011 team used an omnidirectional walk engine based on one that was originally designed for the real Nao robot (Graf et al. 2009). The omnidirectional walk is crucial for allowing the robot to request continuous velocities in the forward, side, and turn directions, permitting it to approach continually changing destinations (often the ball) more smoothly and quickly than the team's previous year's set of unidirectional walks (Urieli et al. 2011).

We began by re-implementing the walk for use on physical Nao robots before transferring it into simulation to compete in the RoboCup 3D simulation league. Many people in the past have used simulation environments for the purpose of prototyping real robot behaviors; but to the best of our knowledge, ours is the first work to use a real robot to prototype a behavior that was ultimately deployed in a simulator. Working first on the real robots lead to some important discoveries. For example, we found that decreasing step sizes when the robot is unstable increases its chances of catching its balance. Similarly, on the robots we discovered that the delay between commands and sensed changes is significant, and this realization helped us develop a more stable walk in simulation.

The walk engine, though based closely on that of Graf et al. (2009), differs in some of the details. Specifically, unlike Graf et al., we use a sigmoid function for the forward component and use proportional control to adjust the desired step sizes. Our work also differs from Graf et al. in that we optimize parameters for a walk in simulation while they do not. For the sake of completeness and to fully specify the semantics of the learned parameters, we present the full technical details of the walk in this section. Readers most interested in the optimization procedure can safely skip to Section 4. The walk engine uses a simple set of sinusoidal functions to create the motions of the limbs with limited feedback control. The walk engine processes desired walk velocities chosen by the behavior, chooses destinations for the feet and torso, and then uses inverse kinematics to determine the joint positions required. Finally, PID controllers for each joint convert these positions into torque commands that are sent to the simulator. The workflow for generating joint commands from the walk engine is shown in Figure 2.

The walk first selects a trajectory for the torso to follow, and then determines where the feet should be with respect to the torso location. We use x as the forwards dimension, y as the sideways dimension, z as the vertical dimension, and θ as rotating about the z axis. The trajectory is chosen using a double linear inverted pendulum, where the center of mass is swinging over the stance foot. In addition, as in Graf et al.'s work (2009), we use the simplifying assumption that there is no double support phase, so that the velocities and positions of the center of mass must match when switching between the inverted pendulums formed by the respective stance feet.

We now describe the mathematical formulas that calculate the positions of the feet with respect to the torso. More than 40 parameters were used but only the most important ones are described in Table 1. Note that many, but not all of these

Figure 1: A screenshot of the Nao humanoid robot (left), and a view of the soccer field during a 9 versus 9 game (right).

[2]http://simspark.sourceforge.net/

[3]http://www.ode.org/

[4]http://www.aldebaran-robotics.com/eng/

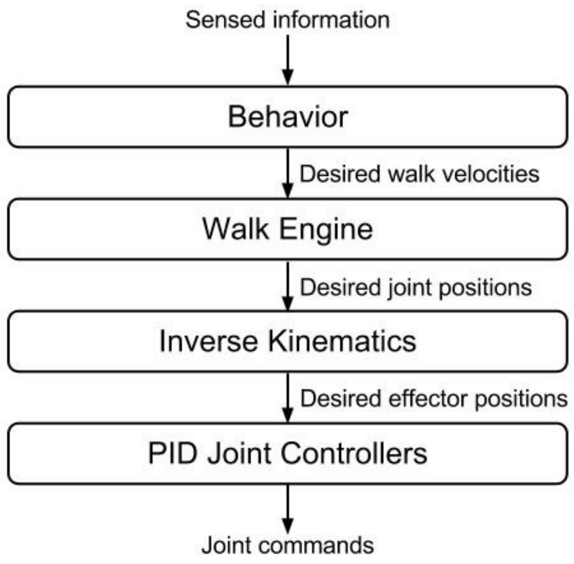

Figure 2: Workflow for generating joint commands from the walk engine.

parameters' values were optimized as described in Section 5.

To smooth changes in the velocities, we use a simple proportional controller to filter the requested velocities coming from the behavior module. Specifically, we calculate $\text{step}_{i,t+1} = \text{step}_{i,t} + \delta(\text{desired}_{i,t+1} - \text{step}_{i,t}) \forall i \in \{x, y, \theta\}$. In addition, the value is cropped within the maximum step sizes so that $-\text{maxStep}_i \leq \text{step}_{i,t+1} \leq \text{maxStep}_i$.

The phase is given by $\phi_{start} \leq \phi \leq \phi_{end}$, and $t = \dfrac{\phi - \phi_{start}}{\phi_{end} - \phi_{start}}$ is the current fraction through the phase. At each time step, ϕ is incremented by $\Delta\text{seconds}/\phi_{length}$, until $\phi \geq \phi_{end}$. At this point, the stance and swing feet change and ϕ is reset to ϕ_{start}. Initially, $\phi_{start} = -0.5$ and $\phi_{end} = 0.5$. However, the start and end times will change to match the previous pendulum, as given by the equations

$$\begin{aligned} k &= \sqrt{9806.65/z_{torso}} \\ \alpha &= 6 - \cosh(k - 0.5\phi) \\ \phi_{start} &= \begin{cases} \dfrac{\cosh^{-1}(\alpha)}{0.5k} & \text{if } \alpha \geq 1.0 \\ -0.5 & \text{otherwise} \end{cases} \\ \phi_{end} &= 0.5(\phi_{end} - \phi_{start}) \end{aligned}$$

The stance foot remains fixed on the ground, and the swing foot is smoothly lifted and placed down, based on a cosine function. The current distance of the feet from the torso is given by

$$\begin{aligned} z_{frac} &= \begin{cases} 0.5(1 - \cos(2\pi \dfrac{t - f_g}{f_a})) & \text{if } f_g \leq t \leq f_a \\ 0 & \text{otherwise} \end{cases} \\ z_{stance} &= z_{torso} \\ z_{swing} &= z_{torso} - z_{step} * z_{frac} \end{aligned}$$

It is desirable for the robot's center of mass to steadily shift side to side, allowing it to stably lift its feet. The side to side component when no side velocity is requested is given by

Notation	Description
maxStep^*_i	Maximum step sizes allowed for x, y, and θ
y^*_{shift}	Side to side shift amount with no side velocity
z^*_{torso}	Height of the torso from the ground
z^*_{step}	Maximum height of the foot from the ground
f^*_g	Fraction of a phase that the swing foot spends on the ground before lifting
f_a	Fraction that the swing foot spends in the air
f^*_s	Fraction before the swing foot starts moving
f_m	Fraction that the swing foot spends moving
ϕ^*_{length}	Duration of a single step
δ^*	Factors of how fast the step sizes change
y_{sep}	Separation between the feet
x^*_{offset}	Constant offset between the torso and feet
x^*_{factor}	Factor of the step size applied to the forwards position of the torso
err^*_{norm}	Maximum COM error before the steps are slowed
err^*_{max}	Maximum COM error before all velocity reach 0

Table 1: Parameters of the walk engine with the optimized parameters starred.

$$\begin{aligned} y_{stance} &= 0.5y_{sep} + y_{shift}(-1.5 + 0.5\cosh(0.5k\phi)) \\ y_{swing} &= y_{sep} - y_{stance} \end{aligned}$$

If a side velocity is requested, y_{stance} is augmented by

$$y_{frac} = \begin{cases} 0 & \text{if } t < f_s \\ 0.5(1 + \cos(\pi \frac{t - f_s}{f_m})) & \text{if } f_s \leq t < f_s + f_m \\ 1 & \text{otherwise} \end{cases}$$

$$\Delta y_{stance} = \text{step}_y * y_{frac}$$

These equations allow the y component of the feet to smoothly incorporate the desired sideways velocity while still shifting enough to remain dynamically stable over the stance foot.

Next, the forwards component is given by

$$\begin{aligned} s &= \text{sigmoid}(10(-0.5 + \frac{t - f_s}{f_m})) \\ x_{frac} &= \begin{cases} (-0.5 - t + f_s) & \text{if } t < f_s \\ (-0.5 + s) & \text{if } f_s \leq t < f_s + f_m \\ (0.5 - t + f_s + f_m) & \text{otherwise} \end{cases} \\ x_{stance} &= 0.5 - t + f_s \\ x_{swing} &= \text{step}_x * x_{frac} \end{aligned}$$

These functions are designed to keep the robot's center of mass moving forwards steadily, while the feet quickly, but smoothly approach their destinations. Furthermore, to keep the robot's center of mass centered between the feet, there is an additional offset to the forward component of both the stance and swing feet, given by

$$\Delta x = x_{offset} + -\text{step}_x x_{factor}$$

After these calculations, all of the x and y targets are corrected for the current position of the center of mass. Finally, the requested rotation is handled by opening and closing the groin joints of the robot, rotating the foot targets. The desired angle of the groin joint is calculated by

$$\text{groin} = \begin{cases} 0 & \text{if } t < f_s \\ \frac{1}{2}\text{step}_\theta(1 - \cos(\pi \frac{t - f_s}{f_m})) & \text{if } f_s \leq t < f_s + f_m \\ \text{step}_\theta & \text{otherwise} \end{cases}$$

After these targets are calculated for both the swing and stance feet with respect to the robot's torso, the inverse kinematics module calculates the joint angles necessary to place the feet at these targets. Further description of the inverse kinematic calculations is given in (Graf et al. 2009).

To improve the stability of the walk, we track the desired center of mass as calculated from the expected commands. Then, we compare this value to the sensed center of mass after handling the delay between sending commands and sensing center of mass changes of approximately 80ms. If this error is too large, it is expected that the robot is unstable, and action must be taken to prevent falling. As the robot is more stable when walking in place, we immediately reduce the step sizes by a factor of the error. In the extreme case, the robot will attempt to walk in place until it is stable. The exact calculations are given by

$$\begin{aligned} \text{err} &= \max_i(\text{abs}(\text{com}_{expected,i} - \text{com}_{sensed,i})) \\ \text{stepFactor} &= \max(0, \min(1, \frac{\text{err} - \text{err}_{norm}}{\text{err}_{max} - \text{err}_{norm}})) \\ \text{step}_i &= \text{stepFactor} * \text{step}_i \, \forall i \in \{x, y, \theta\} \end{aligned}$$

This solution is less than ideal, but performed effectively enough to stabilize the robot in many situations.

4 Walk Movement and Control of Walk Engine

Before describing the procedure for optimizing the walk parameters in Section 5, we provide some brief context for how the agent's walk is typically used. These details are important for motivating the optimization procedure's fitness functions.

During gameplay the agent is usually either moving to a set target position on the field or dribbling the ball toward the opponent's goal and away from the opposing team's players. Given that an omnidirectional walk engine can move in any direction as well as turn at the same time, the agent has multiple ways in which it can move toward a target. We chose the approach of both moving and turning toward a target at the same time as this allows for both quick reactions (the agent is immediately moving in the desired direction) and speed (where the bipedal robot model is faster when walking forward as opposed to strafing sideways). We validated this design decision by playing our agent against a version of itself which does not turn to face the target it is moving toward, and found our agent that turns won by an average of .7 goals across 100 games. Additionally we played our agent against a version of itself that turns in place until its orientation is such that it is able to move toward its target at maximum forward velocity, and found our agent that immediately starts moving toward its target won by an average of .3 goals across 100 games. All agents we compared used walks optimized by the process described in Section 5.

Dribbling the ball is a little different in that the agent needs to align behind the ball, without first running into the ball, so that it can walk straight through the ball, moving it in the desired dribble direction. When the agent circles around the ball, it always turns to face the ball so that if an opponent approaches, it can quickly walk forward to move the ball and keep it out of reach of the opponent.

5 Optimization of Walk Engine Parameters

As described in Section 3, the walk engine is parameterized using more than 40 parameters. We initialize these parameters based on our understanding of the system and by testing them on an actual Nao robot. We refer the agent that uses this walk as the *Initial* agent.

The initial parameter values result in a very slow, but stable walk. Therefore, we optimize the parameters using the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) algorithm (Hansen 2009), which has been successfully applied previously to a similar problem in (Urieli et al. 2011). CMA-ES is a policy search algorithm that successively generates and evaluates sets of candidates sampled from a multivariate Gaussian distribution. Once CMA-ES generates a group of candidates, each candidate is evaluated with respect to a *fitness* measure. When all the candidates in the group are evaluated, the mean of the multivariate Gaussian distribution is recalculated as a weighted average of the candidates with the highest fitnesses. The covariance matrix of the distribution is also updated to bias the generation of the next set of candidates toward directions of previously successful search steps. As CMA-ES is a parallel search algorithm, we were able to leverage the department's large cluster of high-end computers to automate and parallelize the learning. This allowed us to complete optimization runs requiring 210,000 evaluations in less than a day. This is roughly a 150 times speedup over not doing optimization runs in parallel which would have taken over 100 days to complete.

As optimizing 40 real-valued parameters can be impractical, a carefully chosen subset of 14 parameters was selected for optimization while fixing all other parameters. The chosen parameters are those that seemed likely to have the highest potential impact on the speed and stability of the robot. The 14 optimized parameters are starred in Table 1. Note that maxStep$_i$ represents 3 parameters. Also, while f_g and f_s where chosen to be optimized, their complements f_a and f_m were just set to $(1 - f_g)$ and $(1 - f_m)$ respectively.

Similarly to a conclusion from (Urieli et al. 2011), we have found that optimization works better when the agent's fitness measure is its performance on tasks that are *executed during a real game*. This stands in contrast to evaluating it on a general task such as the speed walking straight. Therefore, we break the agent's in-game behavior into a set of smaller tasks and sequentially optimize the parameters for each one of these tasks. Videos of the agent performing optimization tasks can be found online.[5]

[5] `www.cs.utexas.edu/~AustinVilla/sim/`

5.1 Drive Ball to Goal Optimization

We start from a task called driveBallToGoal,[6] which has been used in (Urieli et al. 2011). In this task, a robot and a ball are placed on the field, and the robot must drive the ball as far as it can toward the goal within 30 simulated seconds. The fitness of a given parameter set is the distance the ball travels toward the goal during that time. The agent thus optimized, which we refer to as the *DriveBallToGoal* agent, shows remarkable improvement in the robot's performance as the distance the ball was dribbled increased by a factor of 15 over the *Initial* agent. This improvement also showed itself in actual game performance as when the *DriveBallToGoal* agent played 100 games against the *Initial* agent, it won on average by 5.54 goals with a standard error of .14.

5.2 Multiple Subtasks Optimization

While optimizing walk engine parameters for the driveBallToGoal task improved the agent substantially, we noticed that the agent was unstable when stopping at a target position on the field or circling around the ball to dribble. We believe the reason for this is that the driveBallToGoal task was not very representative of these situations frequently encountered in gameplay. When dribbling a ball toward the goal, the agent never stops as it often does in regular gameplay. Additionally, good runs of the driveBallToGoal task receiving a high fitness occur when the agent perfectly dribbles the ball toward the goal without losing it and being forced to approach and circle the ball once more.

Go to Target Parameter Set To better account for common situations encountered in gameplay, we replaced the driveBallToGoal task in the optimization procedure with a new goToTarget subtask. This task consists of an obstacle course in which the agent tries to navigate to a variety of target positions on the field. Each target is active, one at a time for a fixed period of time, which varies from one target to the next, and the agent is rewarded based on its distance traveled toward the active target. If the agent reaches an active target, the agent receives an extra reward based on extrapolating the distance it could have traveled given the remaining time on the target. In addition to the target positions, the agent has stop targets, where it is penalized for any distance it travels. To promote stability, the agent is given a penalty if it falls over during the optimization run.

In the following equations specifying the agent's rewards for targets, $Fall$ is 5 if the robot fell and 0 otherwise, d_{target} is the distance traveled toward the target, and d_{moved} is the total distance moved. Let t_{total} be the full duration a target is active and t_{taken} be the time taken to reach the target or t_{total} if the target is not reached.

$$\begin{aligned} reward_{target} &= d_{target}\frac{t_{total}}{t_{taken}} - Fall \\ reward_{stop} &= -d_{moved} - Fall \end{aligned}$$

- Long walks forward/backwards/left/right
- Walk in a curve
- Quick direction changes
- Stop and go forward/backwards/left/right
- Switch between moving left-to-right and right-to-left
- Quick changes of target to simulate a noisy target
- Weave back and forth at 45 degree angles
- Extreme changes of direction to check for stability
- Quick movements combined with stopping
- Quick alternating between walking left and right
- Spiral walk both clockwise and counter-clockwise

Figure 3: GoToTarget Optimization walk trajectories

The goToTarget optimization includes quick changes of target/direction for focusing on the reaction speed of the agent, as well as targets with longer durations to improve the straight line speed of the agent. The stop targets ensure that the agent is able to stop quickly, while remaining stable. The trajectories that the agent follows during the optimization are described in Figure 3. After running this optimization seeded with the initial walk engine parameter values we saw another significant improvement in performance. Using the parameter set optimized for going to a target, the *GoToTarget* agent was able to beat the *DriveBallToGoal* agent by an average of 2.04 goals with a standard error of .11 across 100 games. Although the goToTarget subtask is used in the driveBallToGoal task, varying its inputs directly was more representative of the large set of potential scenarios encountered in gameplay.

Sprint Parameter Set To further improve the forward speed of the agent, we optimized a parameter set for walking straight forwards for ten seconds starting from a complete stop. The robot was able to learn parameters for walking .78 m/s compared to .64 m/s using the *goToTarget* parameter set. Unfortunately, when the robot tried to switch between the forward walk and *goToTarget* parameter sets it was unstable and usually fell over. This instability is due to the parameter sets being learned in isolation, resulting in them being incompatible.

To overcome this incompatibility, we ran the goToTarget subtask optimization again, but this time we fixed the *goToTarget* parameter set and learned a new parameter set. We call these parameters the *sprint* parameter set, and the agent uses them when its orientation is within 15° of its target. The *sprint* parameter set was seeded with the values from the *goToTarget* parameter set. This approach to optimization is an example of layered learning (Stone 1998) as the output of one learned subtask (the *goToTarget* parameter set) is fed in as input to the learning of the next subtask (the learning of the *sprint* parameter set). By learning the *sprint*

3dsimulation/AustinVilla3DSimulationFiles/2011/html/walk.html

[6]Note that we use three types of notation for each of driveBallToGoal, *DriveBallToGoal*, *driveBallToGoal*, to distinguish between an optimization task, an agent created by this optimization task and a parameter set. Similarly for "goToTarget", "sprint" and "initial".

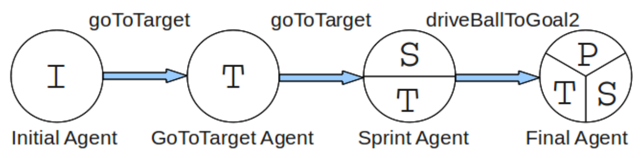

Figure 4: UT Austin Villa walk parameter optimization progression. Circles represent the set(s) of parameters used by each agent during the optimization progression while the arrows and associated labels above them indicate the optimization tasks used in learning. Parameter sets are the following: I = *initial*, T = *goToTarget*, S = *sprint*, P = *positioning*.

parameter set in conjunction with the *goToTarget* parameter set, the new *Sprint* agent was stable switching between the two parameter sets, and its speed was increased to .71 m/s. Adding the *sprint* parameter set also improved the game performance of the agent slightly; over 100 games, the *Sprint* agent was able to beat the *GoToTarget* agent by an average of .09 goals with a standard error of .07.

Positioning Parameter Set Although adding the *goToTarget* and *sprint* walk engine parameter sets improved the stability, speed, and game performance of the agent, the agent was still a little slow when positioning to dribble the ball. This slowness is explained by the fact that the goToTarget subtask optimization emphasizes quick turns and forward walking speed while positioning around the ball involves more side-stepping to circle the ball. To account for this discrepancy, the agent learned a third parameter set which we call the *positioning* parameter set. To learn this set, we created a new driveBallToGoal2[7] optimization in which the agent is evaluated on how far it is able to dribble the ball over 15 seconds when starting from a variety of positions and orientations from the ball. The *positioning* parameter set is used when the agent is .8 meters from the ball and is seeded with the values from the *goToTarget* parameter set. Both the *goToTarget* and *sprint* parameter sets are fixed and the optimization naturally includes transitions between all three parameter sets, which constrained them to be compatible with each other. As learning of the *positioning* parameter set takes the two previously learned parameter sets as input, it is a third layer of layered learning. Adding the *positioning* parameter set further improved the agent's performance such that it, our *Final* agent, was able to beat the *Sprint* agent by an average of .15 goals with a standard error of .07 across 100 games. A summary of the progression in optimizing the three different walk parameter sets can be seen in Figure 4. The results reported throughout this section are summarized in Table 2.

6 Results

The results we present highlight the substantial increase in game performance achieved through optimizing the agent's

[7]The '2' at the end of the name driveBallToGoal2 is used to differentiate it from the driveBallToGoal optimization that was used in Section 5.1.

Table 2: Game results of agents with different walk parameter sets. Entries show the average goal difference (row − column) from 100 ten minute games. Values in parentheses are the standard error.

	Initial	DriveBallToGoal	GoToTarget
Final	8.84(.12)	2.21(.12)	.24(.08)
GoToTarget	8.82(.11)	2.04(.11)	
DriveBallToGoal	5.54(.14)		

omnidirectional walk. As can be inferred by the data presented earlier in Section 5, and seen in Table 2, optimization continually improves the agent's performance from the *Initial* agent on up through our *Final* agent. This result accentuates the utility of using machine learning techniques for optimization (going from the initial to *driveBallToGoal* parameter sets), the importance of finding a subtask that is best representative of scenarios encountered in the domain (improvement in play from *driveBallToGoal* to *goToTarget* parameter sets), and the benefit of combining and switching between parameters learned for multiple subtasks (increase in performance from adding two additional parameter sets to the *goToTarget* parameter set).

The *Final* agent was the one that we entered as UT Austin Villa in the RoboCup 2011 3D simulation competition, which consisted of 22 teams from around the world. UT Austin Villa 2011 won all 24 of its games in the competition, scoring 136 goals and conceding none. Even so, competitions of this sort do not consist of enough games to validate that any team is better than another by a statistically significant margin.

As an added validation of our approach, in Table 3 we show the performance of our *Final* agent when playing 100 games against each of the other 21 teams' released binaries from the competition. UT Austin Villa won by at least an average goal difference of 1.45 against every team. Furthermore, of these 2100 games played to generate the data for Table 3, our agent won all but 21 of them which ended in ties (no losses). The few ties were all against three of the better teams: apollo3d, boldhearts, and robocanes. We can therefore conclude that UT Austin Villa was the rightful champion of the competition.

While there were multiple factors and components that contributed to the success of the UT Austin Villa team in winning the competition, its omnidirectional walk was the one which proved to be the most crucial. When switching out the omnidirectional walk developed for the 2011 competition with the fixed directional walk used in the 2010 competition, and described in (Urieli et al. 2011), the team did not fare nearly as well. The agent with the previous year's walk had a negative average goal differential against nine of the teams from the 2011 competition, suggesting a probable tenth place finish. Also this agent lost to our *Final* agent by an average of 6.32 goals across 100 games with a standard error of .13.

7 Summary and Discussion

We have presented an optimization framework and methodology for learning multiple parameter sets for an omnidirec-

Table 3: Full game results, averaged over 100 games. Each row corresponds to an agent from the RoboCup 2011 competition, with its rank therein achieved. Entries show the goal difference from 10 minute games versus our final optimized agent. Values in parentheses are the standard error.

Rank	Team	Goal Difference
3	apollo3d	1.45 (0.11)
5-8	boldhearts	2.00 (0.11)
5-8	robocanes	2.40 (0.10)
2	cit3d	3.33 (0.12)
5-8	fcportugal3d	3.75 (0.11)
9-12	magmaoffenburg	4.77 (0.12)
9-12	oxblue	4.83 (0.10)
4	kylinsky	5.52 (0.14)
9-12	dreamwing3d	6.22 (0.13)
5-8	seuredsun	6.79 (0.13)
13-18	karachikoalas	6.79 (0.09)
9-12	beestanbul	7.12 (0.11)
13-18	nexus3d	7.35 (0.13)
13-18	hfutengine3d	7.37 (0.13)
13-18	futk3d	7.90 (0.10)
13-18	naoteamhumboldt	8.13 (0.12)
19-22	nomofc	10.14 (0.09)
13-18	kaveh/rail	10.25 (0.10)
19-22	bahia3d	11.01 (0.11)
19-22	l3msim	11.16 (0.11)
19-22	farzanegan	11.23 (0.12)

tional walk engine. The key to our optimization method is learning different parameters in tandem, as opposed to in isolation, for representative subtasks of the scenarios encountered in gameplay. This learned walk was crucial to UT Austin Villa winning the 2011 RoboCup 3D simulation competition.

Our ongoing research agenda includes applying what we have learned in simulation to the actual Nao robots which we use to compete in the Standard Platform league of RoboCup. Additionally, we would like to learn and add further parameter sets to our team's walk engine for important subtasks such as goalie positioning to get ready to block a shot.

More information about the UT Austin Villa team, as well as video highlights from the competition, can be found online at the team's website.[8]

Acknowledgements

This work has taken place in the Learning Agents Research Group (LARG) at UT Austin. Thanks especially to UT Austin Villa 2011 team members Shivaram Kalyanakrishnan, Michael Quinlan, Francisco Barrera, Nick Collins, Adrian Lopez-Mobilia, Art Richards, and Nicolae Ştiurcă. Also thanks to Yinon Bentor and Suyog Dutt Jain for contributions to early versions of the optimization framework employed by the team. LARG research is supported in part by NSF (IIS-0917122), ONR (N00014-09-1-0658), and the FHWA (DTFH61-07-H-00030). Patrick MacAlpine and Samuel Barrett are supported by NDSEG fellowships.

[8]`www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/`

References

Allen, P.; Timcenko, A.; Yoshimi, B.; and Michelman, P. 1993. Automated tracking and grasping of a moving object with a robotic hand-eye system. *Robotics and Automation, IEEE Transactions on* 9(2):152 –165.

Behnke, S.; Schreiber, M.; Stückler, J.; Renner, R.; and Strasdat, H. 2006. See, walk, and kick: Humanoid robots start to play soccer. In *Proceedings of the Sixth IEEE-RAS International Conference on Humanoid Robots (Humanoids 2006)*, 497–503. IEEE.

Graf, C.; Härtl, A.; Röfer, T.; and Laue, T. 2009. A robust closed-loop gait for the standard platform league humanoid. In Zhou, C.; Pagello, E.; Menegatti, E.; Behnke, S.; and Röfer, T., eds., *Proceedings of the Fourth Workshop on Humanoid Soccer Robots in conjunction with the 2009 IEEE-RAS International Conference on Humanoid Robots*, 30 – 37.

Hansen, N. 2009. *The CMA Evolution Strategy: A Tutorial.* `http://www.lri.fr/~hansen/cmatutorial.pdf`.

Kalyanakrishnan, S., and Stone, P. 2010. Learning complementary multiagent behaviors: A case study. In *RoboCup 2009: Robot Soccer World Cup XIII*, 153–165. Springer.

Riedmiller, M.; Gabel, T.; Hafner, R.; and Lange, S. 2009. Reinforcement learning for robot soccer. *Autonomous Robots* 27(1):55–73.

Shih, C.-L. 1999. Ascending and descending stairs for a biped robot. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on* 29(3):255 –268.

Stone, P. 1998. *Layered Learning in Multi-Agent Systems*. Ph.D. Dissertation, School of Computer Science, Carnegie Mellon Univerity, Pittsburgh, PA, USA.

Urieli, D.; MacAlpine, P.; Kalyanakrishnan, S.; Bentor, Y.; and Stone, P. 2011. On optimizing interdependent skills: A case study in simulated 3d humanoid robot soccer. In Tumer, K.; Yolum, P.; Sonenberg, L.; and Stone, P., eds., *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, volume 2, 769–776. IFAAMAS.