To appear in *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems* (AAMAS), Valencia, Spain, June 2012.

UT Austin Villa 2011: A Champion Agent in the RoboCup 3D Soccer Simulation Competition

Patrick MacAlpine, Daniel Urieli, Samuel Barrett, Shivaram Kalyanakrishnan*, Francisco Barrera, Adrian Lopez-Mobilia, Nicolae Ştiurcă†, Victor Vu, and Peter Stone
Department of Computer Science, The University of Texas at Austin, Austin, TX 78701, USA
{patmac, urieli, sbarrett, shivaram, tank225, alomo01, nstiurca, diragjie, pstone}@cs.utexas.edu

ABSTRACT

This paper presents the architecture and key components of a simulated humanoid robot soccer team, UT Austin Villa, which was designed to compete in the RoboCup 3D simulation competition. These key components include (1) an omnidirectional walk engine and associated walk parameter optimization framework, (2) an inverse kinematics based kicking architecture, and (3) a dynamic role assignment and positioning system. UT Austin Villa won the RoboCup 2011 3D simulation competition in convincing fashion by winning all 24 games it played. During the course of the competition the team scored 136 goals while conceding none. We analyze the effect of each component in isolation and show through extensive experiments that the complete team significantly outperforms all the other teams from the competition.

Categories and Subject Descriptors

I.2.9 [**Computing Methodologies**]: Artificial Intelligence—*Robotics*

General Terms

Algorithms, Design, Experimentation

Keywords

Humanoid robotics, Robot soccer, Machine learning

1. INTRODUCTION

Robot Soccer [3] has served as an excellent research domain for autonomous agents and multi-agent systems over the past decade and a half. In this domain, teams of autonomous robots compete with each other in a complex, real-time, noisy and dynamic environment, in a setting that is both collaborative and adversarial. Robot soccer has spread over several popular platforms, each having its own advantages. For example, the real robot competitions, including the humanoid robot league, have typically emphasized low-level robot control challenges. On the other hand, the RoboCup 2D simulation platform has emphasized high-level team strategy challenges. In this paper, we focus on the RoboCup 3D simulation platform, which integrates both these low-level and high-level challenges under one umbrella.

In the 3D simulation league teams of nine simulated humanoids play in a simulation environment with realistic physics, state-noise, multidimensional actions and real-time control. One advantage of the 3D simulation domain over real robots is avoiding the high cost of errors, and the relatively slow feedback loop, that happens when testing new skills in the real world. An advantage over the 2D simulator is the ability to test high-level team strategies under the constraints of humanoid locomotion. Due to the complexity of the environment, parts of the agent are hard to design by hand. For instance, it is a significant challenge to design a walk that is both fast and stable. The 3D simulation platform allows for designing and investigating general methodologies for skill and strategy acquisition in a complex, challenging domain, using machine learning.

In this paper, we present UT Austin Villa, the winning agent of the 3D simulation league in RoboCup 2011. Each of UT Austin Villa's field players is controlled by (a separate instance of) the same program. The players continually estimate the world state from noisy observations, reason about position assignments, and then quickly and robustly move on the field using a learned walk. In this paper, we describe the complete agent, but focus particularly on the most novel components that were key contributors to our success. Specifically, we focus on (1) an omnidirectional walk agent and an associated walk parameter optimization framework, (2) an automatically optimized inverse kinematics based kicking architecture, and (3) a dynamic role assignment and positioning system. We analyze the individual components and the complete team's performance both in competition and in controlled experiments.[1]

The rest of the paper is structured as follows. Section 2 gives a domain description. Section 3 describes our agent's architecture. Section 4, 5 and 6 describe the three key components of our agent, respectively. Results are given in Section 7, and Section 8 summarizes.

2. DOMAIN DESCRIPTION

Robot soccer has served as an excellent platform for testing learning scenarios in which multiple skills, decisions, and

*S. Kalyanakrishnan is currently at Yahoo! Labs.
†N. Ştiurcă is currently at the University of Pennsylvania.

[1]Videos of these components in action can be found online at `http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/AustinVilla3DSimulationFiles/2011/html/components.html`

Appears in: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, Conitzer, Winikoff, Padgham, and van der Hoek (eds.), 4-8 June 2012, Valencia, Spain.

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 [6, 12], to high-level decision-making problems [10, 13].

The RoboCup 3D simulation environment is based on SimSpark [4], a generic physical multiagent system simulator. SimSpark uses the Open Dynamics Engine [2] (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 [1], 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, agents can communicate with each other 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. AGENT ARCHITECTURE

The UT Austin Villa agent receives visual sensory information from the environment which provides distances and angles to different objects on the field. It is relatively straightforward to build a world model by converting this information about the objects into Cartesian coordinates. This of course requires the robot to be able to localize itself for which the agent uses a particle filter. In addition to the vision perceptor, the agent also uses its accelerometer readings to determine if it has fallen and employs its auditory channels for communication.

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

Once a world model is built, the agent's control module is invoked. At the lowest level, the humanoid is controlled by specifying torques to each of its joints. This is implemented through PID controllers for each joint, which take as input the desired angle of the joint and compute the appropriate torque. Further, the agent uses routines describing inverse kinematics for the arms and legs. Given a target position and pose for the hand or the foot, the inverse kinematics routine uses trigonometry to calculate the target angles for the different joints along the arm or the leg to achieve the specified target, if possible.

The PID control and inverse kinematics routines are used as primitives to describe the agent's skills. In order to determine the appropriate joint angle sequences for walking and turning, the agent utilizes an omnidirectional walk engine which is described in Section 4. When invoking the kicking skill, the agent uses inverse kinematics to control the trajectory of the kicking foot as discussed in Section 5. Two other useful skills for the robot are falling (for instance, by the goalie to block a ball) and rising from a fallen position.

It is worth mentioning that some of the agent's skills, like diving, rising from a fall, and kicking, are defined using a flexible text-file-based skill description language, which was used by our team in RoboCup 2010 [15], and which allows to quickly create new skills, while leaving some of the skills' parameters opened for optimization using machine learning.

Because the team's emphasis was mainly on learning robust and stable low-level skills, the high-level team strategy is relatively straightforward. The player closest to the ball is instructed to go to it while other field player agents dynamically choose target positions on the field as explained in Section 6. The goalie is instructed to stand a little in front of its goal and, using a Kalman filter to track the ball, attempts to dive and stop the ball if it comes near.

4. OMNIDIRECTIONAL WALK ENGINE AND OPTIMIZATION

The primary key to UT Austin Villa's success in the 2011 RoboCup 3D simulation competition was its development and optimization of a stable and robust fully omnidirectional walk. The team used an omnidirectional walk engine based on the research performed by Graf et al. [8]. The main advantage of an omnidirectional walk is that it allows the robot to request continuous velocities in the forward, side, and turn directions, permitting it to approach its destination more quickly. In addition, the robustness of this engine allowed the robots to quickly change directions, adapting to the changing situations encountered during soccer games.

4.1 Walk Engine Implementation

The walk engine uses a simple set of sinusoidal functions to create the motions of the limbs with limited feedback control. It processes desired walk velocities given as input, chooses destinations for the feet and torso, and then inverse kinematics are used to determine the joint positions required. Finally, PID controllers for each joint convert these positions into torque commands that are sent to the joints.

The walk engine first selects a trajectory for the torso to follow, and then determines where the feet should be with respect to the torso location. 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 [8], the simplifying assumption that there is no double support phase is used, 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. Further details of the walk can be found in [11].

The walk engine is parameterized using more than 40 parameters, ranging from intuitive quantities, like the step size and height, to less intuitive quantities like the maximum acceptable center of mass error. These parameters are initialized based on an understanding of the system and also testing them out on an actual Nao robot. This initialization resulted in a stable walk. However, the walk was extremely slow compared to speeds required during a competition. We refer to the agent that uses this walk as the *Initial* agent.

4.2 Walk Engine Parameter Optimization

The slow speed of the *Initial* agent calls for using machine learning to obtain better walk parameter values. Parameters are optimized using the CMA-ES algorithm [9], which has been successfully applied in [15]. CMA-ES is a policy search algorithm that successively generates and evaluates sets of candidates. 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 next set of candidates is generated by sampling with probability that is biased towards directions of previously successful search steps.

As optimizing 40 real-valued parameters, can be impractical, a carefully chosen subset of 14 parameters was selected for optimization while keeping all the other parameters fixed. The chosen parameters are those that have the highest potential impact on the speed and stability of the robot, and are mainly: rotation and height; the robot's center of mass height, shift amount, and default position; the fraction of time a leg is on the ground and the time allocated for one step phase; the step size PID controller; center of mass normal error and maximum acceptable errors; and the robot's forward offset.

Similarly to a conclusion from [15], we have found that optimization works better when the robot'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 of walking straight. Therefore, the robot's in-game behavior is broken down into a set of smaller tasks, and the parameters for each one of these tasks is sequentially optimized. When optimizing for a specific task, the performance of the robot on the task is used as CMA-ES's fitness value for the current candidate parameter set values.

In order to simulate common situations encountered in gameplay, the walk engine parameters are optimized for a goToTarget subtask. This consists of an obstacle course in which the agent tries to navigate to a variety of target positions on the field. The goToTarget optimization[2] includes quick changes of target/direction for focusing on the reaction speed of the agent as well as holding targets for longer durations to improve the straight line speed of the agent. Additionally the agent is instructed to stop at different times during the optimization to ensure that it is stable and does not fall over when doing so. In order to encourage the agent to learn both quick turning behavior and a fast forward walk, the agent always walks and turns toward its designated target at the same time. This allows for the agent to swiftly adjust and switch its orientation to face its target, thereby emphasizing the amount of time during the optimization that it is walking forward. Optimizing the walk engine parameters in this way resulted in a significant improvement in performance with the *GoToTarget* agent able to quickly turn and walk in any direction without falling over. This improvement also showed itself in actual game performance as when the *GoToTarget* agent played 100 games against the *Initial* agent, the *GoToTarget* agent won on average by 8.82 goals with a standard error of .11.

To further improve the forward speed of the agent, a second walk engine parameter set is optimized for walking straight forward. This is accomplished by running the goToTarget subtask optimization again, but this time the *goToTarget* parameter set is held fixed while a new parameter set, called the *sprint* parameter set, is learned. The *sprint* parameter set is used when the agent's orientation is within 15° of its target. By learning the *sprint* parameter set in conjunction with the *goToTarget* parameter set, the new *Sprint* agent remains stable while switching between the two walk parameter sets, and the agent's speed increases from .64 m/s to .71 m/s as timed when walking forward for ten seconds after starting from a stand still.

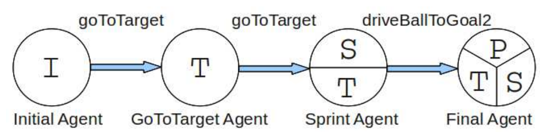

Figure 2: UT Austin Villa's walk parameter optimization progression. Circles represent the set(s) of parameters used by each agent during the optimization progression while the arrows and associated labels above them indicate the optimization tasks used in learning. Parameter sets are the following: I = *initial*, T = *goToTarget*, S = *sprint*, P = *positioning*.

In the next step we further optimize the agent to quickly position near the ball. While the goToTarget optimization emphasizes quick turns and forward walking speed, positioning around the ball involves more side-stepping to circle the ball. To account for this discrepancy, the agent learns a third parameter set called the *positioning* parameter set. To learn this new parameter set a driveBallToGoal2[3] optimization task is created, 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. Whenever the agent enters a radius of .8 meters from the ball, it transitions to using the *positioning* parameter set. During the optimization, both the *goToTarget* and *sprint* parameter sets are held fixed. As the optimization naturally includes

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

[3]The '2' at the end of the name driveBallToGoal2 is used to differentiate it from a driveBallToGoal optimization that was used in [15].

transitions between all three parameter sets, this constrains all parameter sets to be compatible with each other. Adding both the *positioning* and *sprint* parameter sets further improves the agent's performance such that the resulting *Final* agent, is able to beat the *GoToTarget* agent by an average of .24 goals with a standard error of .08 across 100 games. A summary of the progression in optimizing the three different walk parameter sets can be seen in Figure 2.

5. KICK ENGINE

While the learned walk described in Section 4 is by far the aspect of UT Austin Villa that is most responsible for its success, as is affirmed in Section 7, robust and accurate kicking is another skill that is essential for playing soccer at a high level.

To motivate some of the design decisions in our kick engine which we discuss in depth later in this section, we first present the desired qualities of the engine. For a kick to be broadly applicable, it needs to be agile, robust, versatile, and easily and concisely parameterizable. *Agility* refers to taking shots quickly. *Robustness* entails taking accurate and powerful shots in spite of positioning errors (e.g., without the agent being perfectly lined up with the ball). *Versatility* refers to being able to kick in multiple directions from multiple ball starting locations. The parameterization criterion serves to facilitate learning optimized kicks.

5.1 Kick Engine Implementation

To achieve these criteria, our kick engine employs a system of defining and dynamically computing smooth curves which guide the foot's trajectory through the ball at high speed and in the desired direction. We use Cubic Hermite Splines to define the foot trajectories. Agility and robustness are achieved by defining the kick trajectory relative to the ball in Cartesian space. Unlike our previous year's team which used fixed joint angle skills exclusively, the current agents do not have to tip-toe eg., directly behind the ball at a set distance in order to kick the ball eg., forward. Instead, the kick engine dynamically computes the trajectory of the foot once the agent is close enough to the ball, regardless of whether the agent finished positioning or whether the agent was able to position itself precisely relative to the ball. Versatility is achieved because multiple directional kicks can be defined and used at will. Learning and optimization of kicks is facilitated by the parameterization of the foot trajectories in terms of a sparse set of control (way-) points. The flow of the kick engine follows.

First, a kick is selected, and the agent approaches the ball (Section 5.1.1). Once close enough to the ball, it shifts its weight onto the support foot and computes the kicking foot trajectory necessary to perform the desired kick (Section 5.1.2). At each time step during the kick, the kick engine interpolates the control (way-) points defined in the kick skill file (Section 5.1.5) to produce a target pose for the foot in Cartesian space (Section 5.1.3). Finally, an IK solver computes the necessary joint angles of the kicking leg, and these angles are fed to the joint PID controllers (Section 5.1.4). Figure 3 illustrates the program flow of the kick engine.

5.1.1 Kick Choice and Ball Approach

As the agent approaches the ball, it must decide which type of kick to attempt (Section 5.1.6 describes the options)

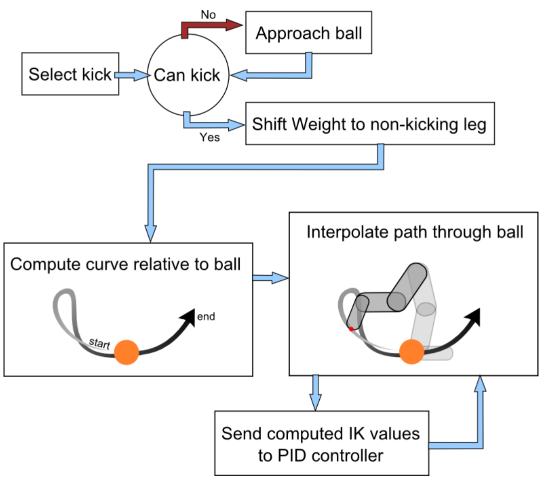

Figure 3: The flow of the agent deciding when to kick the ball and how to interpolate the curve created relative to the ball.

and whether to use the left or right foot. Each kick skill definition includes a target offset of the agent relative to the ball. Choosing a kick reduces to choosing the target with the lowest cost for the agent to move to. We calculate the cost of each target through the following variables and formula:

$$\begin{aligned}
distCost &= |\text{agentPosition} - \text{targetOffsetPosition}|\,/m \\
turnCost &= \frac{|\text{agentOrientation} - \text{targetOrientation}|}{360°} \\
ballPenalty &= \begin{cases} .5 & \text{if ball is in path to target offset} \\ 0 & \text{otherwise} \end{cases} \\
kickCost &= \text{distCost} + \text{turnCost} + \text{ballPenalty}
\end{aligned}$$

The chosen target is approached using the walk engine. During approach, the kick engine continuously checks if the agent is close enough to kick by using the IK solver to determine if the foot can reach most ($> 90\%$) of the points along the trajectory for the chosen kick.

5.1.2 Dynamically Compute Kick Trajectory

Once the agent has shifted its weight in preparation for a kick, it notes the ball's position with respect to itself (specifically its torso, the root of the leg kinematic chains). This offset is added to the control points in the kick skill file to dynamically compute the exact curve of the foot with respect to the agent's torso.

5.1.3 Interpolate Kick Trajectory

The control points defined in the kick skill files are used to compute a smooth 3D curve. We use the Cubic Hermite Spline formulation to interpolate the control points because Hermite Splines yield curves with C^1 continuity which pass through all control points [5]. The time offset from the start of the kick is normalized to the range $[0 - 1]$ (0 is the start of the kick; 1 is the end), and the normalized offset is used to sample the Hermite Spline. The kick skill files also define the Euler angles (roll, pitch, and yaw) of the foot at each control point. These angles are linearly interpolated.

5.1.4 Kick Inverse Kinematics

For the inverse kinematics calculations, we used Open-

RAVE's [7] analytic inverse kinematics solver. The OpenRAVE IK solver can process arbitrary forward kinematic chains defined in XML and produce fast C++ source code that solves the inverse kinematics. Note that the time-consuming analytic processing is done offline, and the fast C++ code can be queried hundreds of times at each time step without a significant computational cost.

5.1.5 Kick Skill Definition

Extending the skill definition files to allow Cartesian coordinate plus Euler angle waypoints for each foot, we predefine all six degree of freedom positions of the foot for a given curve at any linear time through the curve.

5.1.6 Directional Kicks

We defined five kicks that assume that the ball is in front of the agent such that it can kick directly forward and at 45° and 90° angles either outward or inward, depending on which leg is used. We also created directional kicks which assume that the ball is to the side of or behind one of the legs. See Figure 4.

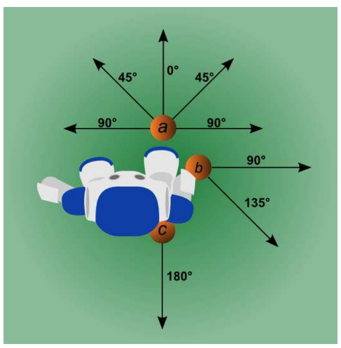

Figure 4: The agent can dynamically kick the ball in varied directions with respect to the placement of the ball at *a, b,* and *c*.

5.2 Kick Optimization

We can then optimize the waypoints (three to five per kick) for kicked distance and speed through CMA-ES. This then allows us to have multiple directional kicks defined through simple curves as we do not have to dedicate large amounts of time tweaking each one and can create rough paths to guide the initial seed of the agent's kick.

In order to learn the parameters for a kick we set up an optimization task where the agent approaches the ball from ten different angles along a half circle arc around the ball and attempts to kick the ball toward a specific target. The parameters being optimized are the XYZ and RPY values of the waypoints that define the curve of the kick, how quickly the kicking foot moves through the curve, and also the target offset from the ball to move toward during the kick approach. The fitness of an agent is measured by the average distance the ball travels toward the target across all kick attempts. The agent is given a penalty fitness of -1 for every kick during which it falls over, runs into the ball, or isn't able to kick the ball after ten seconds have passed. Penalizing the agent for taking too long to kick encourages kicking agility while having the agent approach the ball from multiple angles and penalizing for falling promote kicking robustness.

5.3 Kick Performance

While our kicking system shows a lot of promise, we found out after the competition that our agent does slightly better without kicking turned on during self play. A version of our agent with the kicking system turned off was able to beat our agent that does kick by an average of .15 goals per game across 100 games with a standard error of .07. This resulted in a tally of 27 wins for the agent that does not kick, 12 wins for that agent that does kick, and 61 ties. We believe the reason for this slight degradation in performance when kicking is due to our kicking agent needing to slow down a little when approaching the ball to kick it, instead of maintaining a full speed walk while dribbling the ball, so as to not accidentally run into the ball. Additionally we have yet to implement a strategy for passing and only kick in the direction we want to dribble if an opponent agent is approaching to take the ball away. We therefore include a description of the kick in this paper as a key component of the overall agent, even though it was not necessary for winning this year's competition.

With better tuning such that the agent can approach the ball without needing to slow down, and the addition of a strategy to take full advantage of the ability for kicking to quickly move the ball, we expect our kick system to provide a substantial gain in the performance of the agent. The kicking system has already shown some promise when used with walks that are not as effective at dribbling as our current walk. When playing kicking and non-kicking versions of our agent with slow *initial* walk parameters, as described in Section 4.1, against each other the kicking agent scored 8 goals while the non-kicking agent failed to score.

6. DYNAMIC ROLE ASSIGNMENT AND POSITIONING SYSTEM

While low level skills such as walking and kicking are vitally important for having a successful soccer playing agent, the agents must work together as a team in order to maximize their game performance. One often thinks of the soccer teamwork challenge as being about where the player with the ball should pass or dribble, but at least as important is where the agents position themselves when they *do not* have the ball [10]. Positioning the players in a formation requires the agents to coordinate with each other and determine where each agent should position itself on the field. In our team, players' roles are determined in three steps. First, a full team formation is computed; second, each player computes the best assignment of players to roles in this formation according to its own view of the world; and third, a coordination mechanism is used to communicate and choose among all players' suggestions. In this section, we use the terms (player) position and (player) role interchangeably.

6.1 Formation

In general, the team formation is determined by the ball position on the field. As an example, Figure 5 depicts the different role positions of the formation and their relative offsets when the ball is at the center of the field. As can be seen in the figure, the formation can be broken up into two separate groups, an offensive and a defensive group. Within the offensive group, the role positions on the field are determined by adding a specific offset to the ball's coordinates. The *onBall* role, assigned to the player closest to the ball,

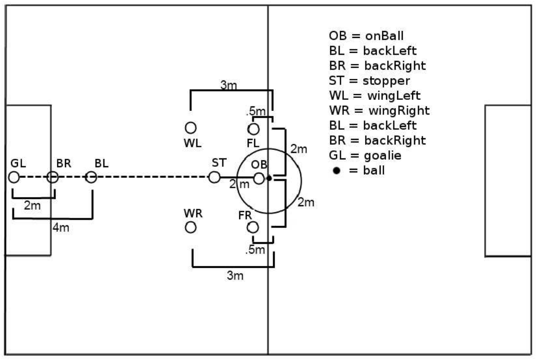

Figure 5: Formation role positions.

is always based on where the ball is and is therefore never given an offset. On either side of the ball we have two forward roles, *forwardRight* and *forwardLeft*. Directly behind the ball we have a *stopper* role as well as two additional roles, *wingLeft* and *wingRight*, located behind and to either side of the ball. When the ball is near the edge of the field we adjust some of the roles' offsets from the ball so as to prevent them from moving outside the field of play.

Within the defensive group there are two roles, *backLeft* and *backRight*. To determine their position on the field a line is calculated between the center of our goal and the ball. Both backs are placed along that line at specific offsets from the end line. The goalie positions itself independently of its teammates in order to always be in the best position to dive and stop a shot on goal. If the goalie assumes the *onBall* role, however, a third role is included within the defensive group, the *goalie* role. A field player assigned to the *goalie* role is told to stand in front of the center of the goal to cover for the goalie going to the ball.

6.2 Assigning Agents to Roles

Given a desired team formation, we need to map players to roles (target positions on the field). A naïve mapping having each player permanently mapped to one of the roles performs poorly due to the dynamic nature of the game. With such static roles an agent assigned to a defensive role may end up out of position and, without being able to switch roles with a teammate in a better position to defend, allow for the opponent to have a clear path to the goal. In this section, we present a dynamic role assignment algorithm. A role assignment algorithm can be thought of as implementing a role assignment *function*, which takes as input the state of the world, and outputs a one-to-one mapping of players to roles. We start by defining three properties that a role assignment function must satisfy (Section 6.2.1). We then construct a role assignment function that satisfies these properties (Section 6.2.2). Finally, we present a dynamic programming algorithm implementing this function (Section 6.2.3).

6.2.1 Desired Properties of a Valid Role Assignment Function

Before listing desired properties of a role assignment function we make a couple of assumptions. The first of these is that no two agents and no two role positions occupy the same position on the field. Secondly we assume that all agents move toward fixed role positions along a straight line

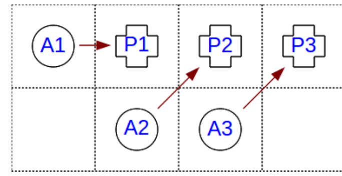

Figure 6: Lowest lexicographical cost (shown with arrows) to highest cost ordering of mappings from agents (A1,A2,A3) to role positions (P1,P2,P3). Each row represents the cost of a single mapping.

1:	$\sqrt{2}$ (A2→P2),	$\sqrt{2}$ (A3→P3),	1 (A1→P1)
2:	2 (A1→P2),	$\sqrt{2}$ (A3→P3),	1 (A2→P1)
3:	$\sqrt{5}$ (A2→P3),	1 (A1→P1),	1 (A3→P2)
4:	$\sqrt{5}$ (A2→P3),	2 (A1→P2),	$\sqrt{2}$ (A3→P1)
5:	3 (A1→P3),	1 (A2→P1),	1 (A3→P2)
6:	3 (A1→P3),	$\sqrt{2}$ (A2→P2),	$\sqrt{2}$ (A3→P1)

at the same constant speed. While this assumption is not always completely accurate, the omnidirectional walk described in Section 4 gives a fair approximation of constant speed movement along a straight line.

We call a role assignment function *valid* if it satisfies the following three properties:

1. *Minimizing longest distance* - it minimizes the maximum distance from a player to target, with respect to all possible mappings.
2. *Avoiding collisions* - agents do not collide with each other as they move to their assigned positions.
3. *Dynamically consistent* - a role assignment function f is dynamically consistent if, given a *fixed* set of target positions, if f outputs a mapping m of players to targets at time T, and the players are moving towards these targets, f would output m for every time $t > T$.

Based on our two given assumptions, the first two properties guarantee that the chosen role assignment is one that minimizes the time to its completion, and the third property guarantees that once a role assignment is decided, it is unchanged as long as the target positions are not changed.

6.2.2 Constructing a Valid Role Assignment Function

Let M be the set of all one-to-one mappings between players and roles. If the number of players is n, then there are $n!$ possible such mappings. Given a state of the world, specifically n player positions and n target positions, let the *cost* of a mapping m be the n-tuple of distances from each player to its target, sorted in decreasing order. We can then sort all the $n!$ possible mappings based on their costs, where comparing two costs is done lexicographically. Sorted costs of mappings from agents to role positions for a small example are shown in Figure 6.

Denote the role assignment function that always outputs the mapping with the lexicographically smallest cost as f_v. Here we provide an informal proof sketch that f_v is a valid role assignment; we provide a longer, more thorough derivation in a technical report [11].

THEOREM 1. *f_v is a valid role assignment function.*

It is trivial to see that f_v minimizes the longest distance traveled by any agent (Property 1) as the lexicographical ordering of distance tuples sorted in descending order ensures this. If two agents in a mapping are to collide (Property 2) it can be shown, through the triangle inequality, that f_v will find a lower cost mapping as switching the two agents' targets reduces the maximum distance either must travel. Finally, as we assume all agents move toward their targets at the same constant rate, the distance between any agent and target will not decrease any faster than the distance between an agent and the target it is assigned to. This serves to preserve the lowest cost lexicographical ordering of the chosen mapping by f_v across all timesteps thereby providing dynamic consistency (Property 3). The next section presents an algorithm that implements f_v.

6.2.3 *Dynamic Programming Algorithm for Role Assignment*

Clearly f_v could be calculated using a brute force method to compare all possible mappings. As there are 8 field players, this would require creating $8! = 40,320$ mappings, then computing the cost of each of the mappings, and finally sorting them lexicographically and choosing the smallest one. However, as our agent acts in real time, and f_v needs to be computed during a decision cycle (0.02 seconds), a brute force method is too computationally expensive. Therefore, we present a dynamic programming implementation shown in Algorithm 1 that is able to compute f_v within the time constraints imposed by the decision cycle's length.

Algorithm 1 Dynamic programming implementation

```
1:  HashMap bestRoleMap = ∅
2:  Agents = {a_1, ..., a_n}
3:  Positions = {p_1, ..., p_n}
4:  for k = 1 to n do
5:     for each a in Agents do
6:        S = C(n-1, k-1) sets of k-1 agents from Agents - {a}
7:        for each s in S do
8:           Mapping m_0 = bestRoleMap[s]
9:           Mapping m = (a → p_k) ∪ m_o
10:          bestRoleMap[a ∪ s] = mincost(m, bestRoleMap[a ∪ s])
11: return bestRoleMap[Agents]
```

THEOREM 2. *Let A and P be sets of n agents and positions respectively. Denote the mapping $m := f_v(A, P)$. Let m_0 be a subset of m that maps a subset of agents $A_0 \subset A$ to a subset of positions $P_0 \subset P$. Then m_0 is also the mapping returned by $f_v(A_0, P_0)$.*

A key recursive property of f_v that allows us to exploit dynamic programming is expressed in Theorem 2. This property stems from the fact that if within any subset of a mapping a lower cost mapping is found, then the cost of the complete mapping can be reduced by augmenting the complete mapping with that of the subset's lower cost mapping. The savings from using dynamic programming comes from only evaluating mappings whose subset mappings are returned by f_v. This is accomplished in Algorithm 1 by iteratively building up optimal mappings for position sets from $\{p_1\}$ to $\{p_1, ..., p_n\}$, and using optimal mappings of $k-1$ agents to positions $\{p_1, ..., p_{k-1}\}$ (line 8) as a base when constructing each new mapping of k agents to positions $\{p_1, ..., p_k\}$ (line 9), before saving the lowest cost mapping for the current set of k agents to positions $\{p_1, ..., p_k\}$ (line 10).

As $\binom{n-1}{k-1}$ agent subset mapping combinations are evaluated for mappings of each agent assigned to the *kth* position, the total number of mappings computed for each of the n agents is thus equivalent to the sum of the $n-1$ binomial coefficients. That is,

$$\sum_{k=1}^{n} \binom{n-1}{k-1} = \sum_{k=0}^{n-1} \binom{n-1}{k} = 2^{n-1}$$

Therefore the total number of mappings that must be evaluated using our dynamic programming approach is $n2^{n-1}$. For $n = 8$ we thus only have to evaluate 1024 mappings which is very manageable.

Table 1: Full game results, averaged over 100 games. Each row corresponds to an agent with varying formation and positioning systems as described in Section 6.3. Entries show the goal difference from 10 minute games versus our agent using the dynamic role positioning system and formation described in Section 6. Values in parentheses are the standard error.

Team	Goal Difference
Defense	.29 (.06)
Static	.32 (.07)
AllBall	.43 (.09)
Boxes	1.26 (.10)

6.3 Formation Evaluation

To test how our formation and role positioning system affects the team's performance we created a number of teams to play against by modifying the positioning system of UT Austin Villa that was used in the competition.

AllBall No formations and every agent except for the goalie just goes to the ball.

Static Each role is statically assigned to an agent based on its uniform number.

Defense Defensive formation in which only two agents are in the offensive group (one on the ball and the other directly behind the ball)

Boxes Field is divided into fixed boxes and each agent is dynamically assigned to a home position in one of the boxes. Similar to the positioning system used in [14].

Results of UT Austin Villa playing against these modified versions of itself are shown in Table 1. We see that a very defensive formation used by the *Defense* agent hurts performance a little likely because the best defense is a good offense. Dynamically assigning roles is better than statically fixing them as is clear in the degradation in performance of the *Static* agent. Having and maintaining formations is also important which is evident by the positive goal difference recorded when playing against the *AllBall* agent. The poor performance of the *Boxes* agent, in which the positions on the field are somewhat static and not calculated as relative offsets to the ball, underscores the importance of being around the ball and adjusting positions on the field based on the current state of the game.

7. COMPETITION RESULTS

UT Austin Villa 2011 won all 24 of its games during the RoboCup 2011 3D simulation 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. In

Table 2: 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 (.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)

order to validate the results of the competition, in Table 2 we show the performance of our team 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 2, 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 UT Austin Villa 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 [15], 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 2011 agent by an average of 6.32 goals across 100 games with a standard error of .13

8. SUMMARY AND DISCUSSION

We have presented the architecture and key components of the UT Austin Villa 2011 RoboCup 3D simulation league team. These key components include an omnidirectional walk engine and associated walk parameter optimization framework, an inverse kinematics based kicking architecture, and a dynamic role and formation positioning system.

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. For next year's competition we expect to better integrate and utilize our kicking system in order to improve the performance of the team. 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.

Acknowledgments

This work has taken place in the Learning Agents Research Group (LARG) at UT Austin. Thanks especially to UT Austin Villa 2011 team members Michael Quinlan, Nick Collins, and Art Richards. 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.

9. REFERENCES

[1] Aldebaran Humanoid Robot Nao. http://www.aldebaran-robotics.com/eng/.
[2] Open Dynamics Engine. http://www.ode.org/.
[3] RoboCup. http://www.robocup.org/.
[4] SimSpark. http://simspark.sourceforge.net/.
[5] E. Angel. *Interactive Computer Graphics*. Pearson Education, Inc., 5th edition, 2009.
[6] S. Behnke, M. Schreiber, J. Stückler, R. Renner, and H. Strasdat. See, walk, and kick: Humanoid robots start to play soccer. In *Proc. of the 6th IEEE-RAS Int. Conf. on Humanoid Robots (Humanoids 2006)*, pages 497–503. IEEE, 2006.
[7] R. Diankov and J. Kuffner. Openrave: A planning architecture for autonomous robotics. Technical Report CMU-RI-TR-08-34, Robotics Institute, Pittsburgh, PA, July 2008.
[8] C. Graf, A. Härtl, T. Röfer, and T. Laue. A robust closed-loop gait for the standard platform league humanoid. In *Proc. of the 4th Workshop on Humanoid Soccer Robots in conjunction with the 2009 IEEE-RAS Int. Conf. on Humanoid Robots*, pages 30 – 37, 2009.
[9] N. Hansen. *The CMA Evolution Strategy: A Tutorial*, January 2009. http://www.lri.fr/~hansen/cmatutorial.pdf.
[10] S. Kalyanakrishnan and P. Stone. Learning complementary multiagent behaviors: A case study. In *RoboCup 2009: Robot Soccer World Cup XIII*, pages 153–165. Springer, 2010.
[11] P. MacAlpine, D. Urieli, S. Barrett, S. Kalyanakrishnan, F. Barrera, A. Lopez-Mobilia, N. Ştiurcă, V. Vu, and P. Stone. UT Austin Villa 2011 3D Simulation Team report. Technical Report AI11-10, The Univ. of Texas at Austin, Dept. of Computer Science, AI Laboratory, December 2011.
[12] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange. Reinforcement learning for robot soccer. *Autonomous Robots*, 27(1):55–73, 2009.
[13] P. Stone. *Layered Learning in Multi-Agent Systems*. PhD thesis, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, USA, December 1998.
[14] P. Stone and M. Veloso. Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence*, 110(2):241–273, June 1999.
[15] D. Urieli, P. MacAlpine, S. Kalyanakrishnan, Y. Bentor, and P. Stone. On optimizing interdependent skills: A case study in simulated 3D humanoid robot soccer. In *Proc. of the Tenth Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, pages 769–776, May 2011.