Grid Soccer Simulator 1.0
User's Manual

Sina Iravanian and Sahar Araghi

April 2011

Contents

1 About Grid-Soccer

Grid-Soccer Simulator, is a grid-world environment in which two teams of agents play soccer together. The main intention behind developing this environment, is to provide a test-bed for reinforcement learning, or other multi-agent learning algorithms. Grid-Soccer provides test-bed for researching about learning in cooperative, competitive, and mixed environments. The communication between the environment and the clients is done through a text-based network protocol; thus, one can create a client in any desired programming language as long as that language supports networking. Currently there are several agents developed in C# and sample agents in C++ and Java, which can be easily extended to fit one's experiments' needs. There's also a text-client in C# which can be used to test and practice the communication protocol.

The developers believe this environment can be useful for experiments in reinforcement learning, and multi-agent systems. It is a very simplified form of a soccer server which can be used to learn team policies without having to be concerned about low-level skills and environmental non-determinism of the currently existing sophisticated soccer simulators such as RoboCup Soccer Server [2]. It is useful also for teaching and demonstrating related fields in the class, or defining proper assignments for the students.

Currently the experiments for masters thesis of the developers are performed and evaluated in this environment. The source code for agents of the first thesis, titled "*Using Data-Mining methods for Agent Learning in Multi-agent Environments*" [1] are located in the directory: "Source/RLAgents/MRLDM". The source code for agents of the second thesis, titled "*Evaluation of NeuroEvolution of Augmenting Topologies in Cooperative Multi-agent Learning*" [3] are located in "Source/NEATAgents".

If you are using this environment in your research, please cite this manual in your related publications as:

Sina Iravanian and Sahar Araghi. Grid Soccer Simulator 1.0: User's Manual, April 2011. http://gridsoccer.codeplex.com.

Or use the following BibTeX entry:

```
@Manual{gsman11,
  title = {Grid Soccer Simulator 1.0: User's Manual},
  author = {Sina Iravanian and Sahar Araghi},
  month = {April},
  year = {2011},
  note = "http://gridsoccer.codeplex.com",
  url = "http://gridsoccer.codeplex.com"
}
```

2 Requirements

In order to build and run the source code of Grid-Soccer, .NET Framework 3.0 or higher, or mono is required. In order to open the solution file, you can use Visual Studio 2010, or SharpDevelop 4.0. The simulator and several C# sample clients have been tested in Microsoft Windows 7, and openSuSE 11.0 (with mono). It is expected that they should successfully work in other operating systems provided that the appropriate version of .NET Framework or mono is installed.

3 The Source Package

The most recent version of Grid-Soccer Simulator can be downloaded from http://gridsoccer.codeplex.com.

After you download and decompress the archive file, you see the following files and folders in the root folder:

Name	Description
Bin	Contains the output for all libraries and executables
ExternalLibs	Here the 3rd party libraries are located
Images	This folder contains the images used in the simulator and the clients
Manual	The LaTeX source for this manual is kept here
Scripts	In this folder the scripts that run simulations are kept. These scripts are simply batch files which run proper programs from the Bin folder with proper arguments. The log files generated from the simulation are also kept in this folder.
Source	This is the root folder for the source files and projects of the simulator and the clients.
Copying.txt	A copy of the MIT license

Among these folders, the Source folder needs more inspection. In the root of the Source folder, there exists the "AllInOne2010.sln" file, which is the solution file created by "Visual Studio 2010" and can be opened by "Visual Studio 2010" or "SharpDevelop 4.0" or higher. This solution files includes all the projects including the simulator and the clients except the "JavaSam-

pleClient" which needs a Java IDE to be opened. "JavaSampleClient" is created using the Eclipse IDE.

The following files and folders exist in the Source folder:

Name	Description
BasicAgents	Contains agents that implement basic functionalities, and provide samples for more sophisticated implementations.
CPPSampleAgent	A C++ sample agent that simply plays randomly.
GridSoccerSimulator	Contains the source code for the simulator.
JavaSampleClient	A Java sample agent that simply plays randomly.
NEATAgents	Contains agents that use the NEAT and HyperNEAT algorithms. Also the source code for "HyperSharpNEAT 2.1" is also included in this folder.
RLAgents	Contains some basic and sophisticated Reinforcement Learning (RL) agents. "RLAgentsCommon" is the class library which implements the basic functionalities of RL agents. Using this library one can easily create $Q(\lambda)$, SARSA(λ), and modular RL learning agents.
Tools	Includes miscellaneous tools used by the agents or the simulator, or used as standalone application. These tools include a log-plotter program which plots RL log files, a Neural Network library and its tester application, a plotting library which plots lines and 3D surfaces and its tester application.

The "BasicAgents" folder, contains agents with the most basic functionalities. The source code for these agents provide the best starting-point for learning how to create agents for the Grid-Soccer environment:

Name	Description
ClientBase	A class library which implements all the functionalities of a Grid-Soccer client, and performs all the needed network-based communication.
DPClient	A Dynamic-Programming agent.
HandCodedClient	An agent with a hand-coded policy. When run as an offensive agent, it follows the ball-owner player until it will become the ball-owner player. Having owned the ball, it moves directly towards the opponent's goal to score a goal.
RandomClient	An agent that plays randomly (i.e., chooses random actions)
SampleClient	An agent that simply moves round the field.
TextClient	A client application that lets you send the commands by typing them on the specified field, and see the incoming messages from the server.

4 The Simulator Config File

Having built the source files, the binary for the simulator and the accompanied agents are copied in the *Bin* folder. There you will find "GridSoccerSimulator.exe" and "GridSoccerSimulator.exe.config" files. The ".exe" file is the simulator executable file and the ".exe.config" is its config file. The config file is formatted as an XML document. Having opened the config file with a text editor you can modify the following options as desired:

Option Name	Description
NumCycles	The total number of cycles of each game. The game is stopped after this number of cycles is passed.
CycleDuration	The length of each simulation cycle in milliseconds.
MaxPlayers	The maximum number of players allowed to connect to the simulator in each team. If you need to connect more players, change this option.
MinPlayers	The minimum number of players allowed to connect to the simulator in each team. Currently this option is ignored by the simulator.
VisibleRadius	The distance up to which a player can see other objects. If this value is set to 0 the player can see the whole field.
PassableDistance	The distance up to which a player can pass the ball to another player. If this value is set to 0 the player can pass to any teammate in any distance from it.
NumRows	The number of rows in the grid forming the field.
NumCols	The number of columns in the grid forming the field.
GoalWidth	The width of the goal.
PortNumber	The port number of the local machine on which the simulator accepts incoming connections.
WaitForAllPlayers	*True/False*. If set to *True* (the default) the simulator waits for all players to send their actions before starting the next simulation cycle. This mode ensures that no players miss a cycle. If set to *False*, the simulator starts the next cycle after the cycle length has been passed regardless of the fact that whether the players have sent their actions.
AllowEpisodeTimeout	*True/False*. If set to *True* (the default) the simulator allows the connected players to send *episode-timeout* messages, which is a message specific for trainer players.

5 The Communication Protocol

In Grid-Soccer Simulator, clients and the simulator communicate through a human-readable plain-text network protocol. The syntax of the commands and messages is like LISP expressions. That's mainly because theses messages are inspired from the syntactic form of the messages of the RoboCup Soccer Simulator [2].

The simulator uses the TCP/IP protocol for connections with players. The default port number for the simulator is **5050**. The reason behind choosing TCP/IP rather than UDP/IP is to enable the simulator to run the game in a way that no players miss a cycle. This option is named "Wait-For-All-Players" and can be turned off whenever desired. This option may be required for some experiments in order for their results to be truly verified. This functionality cannot be guaranteed with UDP/IP.

The commands sent must be ASCII encoded strings (1 byte per character), and should be terminated will a null character ('\0'). It happens sometimes that multiple messages are received at once by a process. The receiving process can separate the constituent messages using the null character as delimiter.

5.1 Connecting to the Simulator

The Grid-Soccer simulator listens for incoming connections from the clients in port 5050. The default port number can be changed in the simulator's configuration file. A Grid-Soccer client should connect to the port using TCP/IP. After the connection is established, the client should send an *init* command to introduce a player to the simulator. The format of the *init* command is as follows:

```
(init <team-name> <uniform-number>)
```

The *team-name* parameter is a one-word string for the name of the team that the player belongs to. The *uniform-number* parameter is an integer between 1 and *MAX-PLAYERS*, which is the maximum number of players that can be connected to the simulator in one team. Its default value is 11, but can be modified in the simulator config file.

If *init* succeeds, the server responds with:

```
(init l|r ok)
```

If the response is (init l ok), it means that the player is successfully connected to the server and its side is left. Conversely if the player's side is

right the message would be (init r ok). If there's any kind of problem with the player's *init* message, or if the player is trying to connect after the game is started, then the simulator responds with an (error) message.

Write after that a player is successfully *init*ed, the simulator sends the player the settings of the game and a description of the environment in the form of a *settings* message. The format of the settings message is as follows:

```
(settings (<setting>)+)
<setting> ::  rows <number>
           |  cols <number>
           |  goal-width <number>
           |  pass-dist <number>
           |  visible-dist <number>
           |  min-players <number>
           |  max-players <number>
           .
```

In the *settings* message, the simulator sends the player information about the dimensions of the field, the goal width, passabe and visible distance around each player, and the minimum and maximum number of players allowed to connect to the simulator.

5.2 Specifying a Player's Home Position

A player's home position is the location where it is situated before start of the game, and after a goal is scored. A player can specify its home position right before the game is started by using the command *home*:

```
(home <row> <column>)
```

If it is not successful (e.g., if the new position is already occupied by another player) then the simulator responds by:

```
(error could-not-set-home)
```

The left team always start a game. After a goal is scored the team that received the goal will start the game. Among the game-starting team's players, the player whose home-position is nearest to the center of the field will start the game.

5.3 Players' Actions

These are the commands that the players can send during the game:

(hold)
(move east)
(move south)
(move west)
(move north)
(move north-east)
(move south-east)
(move south-west)
(move north-west)
(pass <uniform-number>)

The above commands enable a player to move in the eight main directions, hold, or pass the ball to a teammate. The *pass* command needs an option which is the uniform number of the passee.

5.4 Players' Sensory Input

A player gets aware of its environment through the *see* message which is sent from the simulator at the end of each cycle. The *see* message tells the player about the current time-cycle of the game, the location of the players with its visible distance, and the location of the ball (if it is within the visible distance of the player). The format of the *see* message is as follows:

```
(see <cycle-number> (<see-item>)+)
<see-item> :: self <row> <column>
            | b    <row> <column>
            | l <unum> <row> <column>
            | r <unum> <row> <column>
            .
```

In the above message format, *self* is used for informing the player of its own location, *b* is used for informing the player about the ball location, provided that it is in the player's visibility distance. The location of players in the left-side team are reported in an expression starting with *l*. Likewise the location of players in the right-side team are reported in an expression starting with *r*. Within *self* and *b* expressions the coordinates (i.e., row and

column pair) of the referred object is reported. Along with the coordinates, the r and l expressions include the uniform number of the player too. In each cycle, the players get informed of their own locations, but they will get informed about the location of the ball and other players only if they are located within the player's visibility distance.

5.5 Messages Sent by a Trainer

A trainer is a player who can declare end of an episode without necessarily a goal being scored. This is ideal for training players seeking a special goal. For example, we may want to train players who can score within 30 cycles. If the players fail to score a goal after 30 cycles the trainer calls an end to the episode, and adds to the score of the opponent team.

In Grid-Soccer, there are no special ways of *init*ing a player as a trainer. Any player can send trainer specific messages to the simulator. The *episode-timeout* message is the message used by a trainer to call an end to an episode:

```
(episode-timeout <stat>)
<stat> :: our-fail
        | our-pass
        | opp-fail
        | opp-pass
        .
```

Whenever the simulator receives an *episode-timeout* message from any of the players, it will restart the game as if a goal has been scored. If a player of a team sends an *our-fail* message, the simulator increments the score of the other team. If a player sends an *our-pass* message, the simulator increments the score of the player's team (penalizing the other team). Likewise if a player sends an *opp-fail* message, the score of the player's team is incremented and if a player sends an *opp-pass* message, the score of the player's opponent team is incremented.

5.6 Other Messages Sent by the Simulator

In the case of important changes occurring in the state of the simulator, the simulator will inform the players about that change, so that they can adapt to that change. The most primitive kind of these changes are the *start* and *stop* of a game. After a game is started the simulator will send all the connected players a *start* message:

```
(start)
```

Likewise when the game is stopped, e.g., after the total number of cycles are passed or when the stop button of the simulator is pressed, all the players receive a *stop* message:

```
(stop)
```

Whenever a user decides to change the duration of simulation cycles, the new amount for the cycle length is passed to all the players in the form of a *cycle* message. This could be important because the players can adapt their performance to the new amount:

```
(cycle <number>) %number is the cycle duration in milliseconds
```

Whenever the turbo mode in the simulator is toggled, the players receive a turbo message with a parameter conveying whether the turbo mode is getting on or off:

```
(turbo on)
(turbo off)
```

6 The Model of the Dynamics of the Environment

Nondeterminism occurs in the simulation where there are different outcomes possible from the set of actions performed by the players (for example when more than one player decide to move to a single cell of the grid). There are different sources of nondeterminism in the grid-soccer environment. Theses sources are conflicts in positioning of the players, deciding the next ball owner player when there are more than one players candidate for ball possession, and deciding the pass receiving player, when there are other players in the way between passer and passee. In this section we discuss about these situations and show how these conflicts are resolved in the Grid-Soccer simulator. The stochastic rules which are used to resolve these situations form the model of the dynamics of the grid-soccer environment. This model is also used to implement the *Dynamic-Programming Agent*, which is located in: "Source/BasicAgents/DPClient".

6.1 Positioning Conflict Resolution Rules

We say $p_1 \in P$ is a player, where P is the set of players. We denote the ball with the literal b. We now define the location function, L,

$$L : P \cup \{b\} \longmapsto \mathbb{N}^2$$

such that $L(p_1) = \langle r, c \rangle$, where r is the row of the player p_1 and c is the column of the same player. By $L(b)$, we mean the location of the ball.

We subscript the L function by 0 and 1, to differ the location of the players in the previous and the current time cycles. By L_0 we mean the location of the players in the previous time cycle, and by L_1 the location of the players in the current time cycle is meant.

We say that the players p_1, and $p_2 \in P$, are in positioning conflict if,

$$L_1(p_1) = L_1(p_2) \tag{1}$$

The above rule states the positioning conflict between two players, and can be easily extended to more than two players.

We will state the rules to resolve the positioning conflict between two players. Note that, by $r \sim U(0,1)$ we mean that r is a real random number of uniform distribution between 0 and 1. Also we take it for granted that the conflict condition (1) holds in all the following cases:

1. if both the players made a move:

$$L_0(p_1) \neq L_1(p_1) \wedge L_0(p_2) \neq L_1(p_2) \Longrightarrow \begin{cases} L_1(p_1) \leftarrow L_0(p_1) & U(0,1) < 0.5 \\ L_1(p_2) \leftarrow L_0(p_2) & \text{otherwise} \end{cases}$$

2. if one (say p_1) held its position, and the other (say p_2) made a move:

$$L_0(p_1) = L_1(p_1) \ \wedge \ L_0(p_2) \neq L_1(p_2) \Longrightarrow L_1(p_2) \leftarrow L_0(p_2)$$

3. Repeat the above steps until there are no conflicts remained.

The ball owning conflict will be resolved later.

Theorem 1 *Each step of the positioning conflict resolution algorithm, reduces one from the number of the moving players by making it stay.*

Proof: If there is a conflict, it is resolved using either step 1 or 2 mentioned above. In step 1, one of the players is chosen randomly to stay in position. In step 2 the moving player is made to stay in position. So in either case, one is reduced from the number of the moving players. □

Theorem 2 *The positioning conflict resolution algorithm finishes.*

Proof: Initially all the players are positioned in different locations. If all of the players stay at their position and make no moves, then there will be no positioning conflicts produced. The number of players is finite, and by theorem 1, each iteration of the above algorithm reduces the number of moving players. Therefore, in the worst case, the algorithm finishes by making all the players stay in their previous location. The number of iterations of the algorithm in the worst case (the case where all the players are in conflict with each other) is equal to the number of players. □

If n is the number of players, then the running time complexity of each step of the above algorithm is $O(n^2)$, and therefore adding up to the full positioning conflict resolution algorithm, the running time complexity will be $O(n^3)$.

6.2 Candidate List

We use the term *candidate list* to refer to the means that enables the process of selecting an item randomly from a set of items where each of them have a different chance for being selected. For example imagine that we want to choose a player randomly from the set $\{p_1, p_2, p_3\}$, where player p_1 has a weight of 4, and players p_2, and p_3 each have a weight of 1. This process is equivalent to selecting a player from the following list in which elements have equal weights:

p_1	p_1	p_1	p_1	p_2	p_3

In this way we can talk more freely about weights without worrying about the final probability when the number of items in the list varies. We refer to this kind of list as a *candidate list*.

6.3 Deciding about the Ball Owner Player

Many of the conflicts on deciding the ball-owner is resolved using a candidate list. If a player, say p_1, owns the ball and holds its position, and other players, say p_2, and p_3, come into its location, then player p_1 nominated in the list for four times, and p_2, and p_3 nominate once each. The ball owner is selected uniformly random from the ball owner candidate list. The general rule here is that the currently ball-owner player which holds its position has a weight of 4, while other players which move into its location each have a weight of

1 to be the next ball owner. In this way when the number of players that move to the location of the ball owner player increases the chance that the same player remain ball owner reduces.

If p_1 owns the ball and bumps into staying p_2, then p_2 will be the only ball owning candidate (i.e., p_2 will be the next ball owner for sure).

Other source of nondeterminism for deciding about the ball owner player is the *pass* action. If p_1 fires the *pass* action we first resolve the positioning conflicts then apply the following rules:

1. If the pass receiving player is positioned along one of the eight main directions, then the ball owner is chosen deterministically by the nearest player in the line connecting the passer and the passee.

2. If the pass receiving player is not along the given line, then the pasee is nominated for, 4 times, and other players which are in the distance between 0 and 1.5 exclusively are nominated once. If a player is found in between, with the distance 0, then it would be deterministically the next ball owner player.

6.4 Simulating the Actions of the Players

First off note that if players fail to send commands to the simulator then it is assumed that they have performed a "hold" action.

1. Before the update period, the list of pending actions is formed, by refining them first. By refinement we mean that if a player tries to go off the field replace its action with a *hold* action. Likewise if a player tries to pass to a player beyond the passable area, replace its action with a *hold* action. Also find and mark the ball owner player to speed up the update procedure.

2. Consider the ball owner player. Does it perform a Move or Pass or Hold?

   (a) *If it is a* ***Move****.* If it bumps into a holding player, give him the ball. Otherwise nominate the ball owning player for 4 times. nominate other players moving to the empty cell once each, while performing the conflict resolution algorithm.

   (b) *If it is a* ***Pass****.* Determine the type of the pass: along the main eight directions, or not. If it is along the main eight directions, determine the ball owning player deterministically, otherwise, calculate the distance of the players, and give them a weight each.

Give the passee a weight of 4. These should be done while performing the conflict resolution algorithm.

(c) *If it is a **Hold**.* Give it a weight of four. Give other players coming to his house a weight of 1 each.

A License

References

[1] Sahar Araghi. Using data-mining methods for agent learning in multi-agent environments. Master's thesis in intelligent systems and artificial intelligence, Department of Computer Science, Amirkabir University of Technology (Tehran Polytechnic), Tehran, Iran, Jan 2010. In Persian.

[2] Mao Chen, Klaus Dorer, Ehsan Foroughi, Fredrik Heintz, ZhanXiang Huang, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Jan Murray, Itsuki Noda, Oliver Obst, Pat Riley, Timo Steffens, Yi Wang, and Xiang Yin. *RoboCup Soccer Server User Manual: for Soccer Server 7.07 and later.* The RoboCup Federation, February 2003. http://sserver.sourceforge.net.

[3] Sina Iravanian. Evaluation of NeuroEvolution of Augmenting Topologies in cooperative multi-agent learning. Master's thesis in scientific computing, Department of Computer Science, Sharif University of Technology, Tehran, Iran, Jan 2011. In Persian.