# Experiences with simulated robot soccer as a teaching tool

Rhys Hill Anton van den Hengel

School of Computer Science
University of Adelaide,
Adelaide, South Australia 5005,
Email: {rhys,anton}@cs.adelaide.edu.au

## Abstract

*The development of assignments for undergraduate teaching typically requires a compromise between what is achievable by an average student and what will engage the interest of a more advanced member of the class. Selecting a suitable compromise is particularly problematic for undergraduate Artificial Intelligence (AI) courses which typically attempt to cover a very broad range of topics, without delving too deeply into the details. Ideally, a single problem would be selected whose solution could be approached with more than one technique covered in the course, enabling students to carry out a comparative analysis of performance.*

*Robot soccer simulation has provided an interesting platform for Artificial Intelligence research and is increasingly being used as a teaching apparatus. There are a number of limitations with existing simulation methodologies for this purpose. Current robot soccer simulators are aimed at research groups where accuracy is paramount and all facets of the real system must be emulated. However, many of the intricacies of a real robot soccer player are inappropriate for a teaching environment, as they detract from desired learning outcomes. Consequently, there is a need for a simulation that employs a simplified set of game rules and dynamics. This paper describes the design and implementation of such a framework and presents experiences gained from its use as a third year practical.*

## 1. Introduction

Developing artifical intelligence practicals for undergraduate students can be difficult. Interesting problems generally require a large amount of base software be provided to the student, before they can begin working. Simpler problems often fail to engage the student, reducing participation and learning.

Recently, many Computer Science Schools have started drawing upon simulated environments or activities as the basis for student practicals[1]. Obvious areas for such simulations include sport, war games and gambling. Many of these cannot be used at a University due to possible cultural sensitivities. Of the three examples provided, sport is the most inclusive.

Separately, robot soccer is a burgeoning research area, with many Universities participating in international tournaments. Part of the reason for this growth is the excitement associated with competition and the hands-on nature of the area. These aspects of robot soccer make it an ideal basis for undergraduate practicals. However, supplying a large number of students with access to real robot soccer players is intractable, instead a simulation is more appropriate.

A number of robot soccer simulators are currently available [4, 2, 3]. Unfortunately the majority of these simulators tend to be unsuitable for the purpose of teaching artificial intelligence. One major problem is the existence of example players or 'brains' available for download. It becomes very difficult to determine what is original student work if a large number of example solutions are available. Moreover, existing simulators are often complicated to learn and to use [1], thus deflecting students from the pedagogical aims towards technical problems.

This paper reports on the design and implementation of a robot soccer simulation framework and communicates some of the lessons learned. A simplified model for simulated robot soccer is presented, along with the details of its implementation. By combining this model and careful software design, the resultant system provided fertile ground for student exploration.

## 2. Requirements

Artificial Intelligence is a third year subject at the University of Adelaide, with a typical enrolment of 180 students. The assessment for the subject consists of two practicals and an exam. The framework presented in this paper formed the basis of both practicals in 2004. Section 3 provides details on these practicals, while the remainder of this section discusses design and implementation.

### 2.1. The Soccer Model

The model employed by the framework is based loosely on RoboCup rules [5]. By limiting particular behaviours of the field, ball and players, many of the more difficult aspects of playing soccer are removed.

The field is a discrete plane; the ball and players may only be positioned at integer locations (See figure 1). Each player can move in any one of eight directions from their current position. Player can stand on the edge of the field, but the ball cannot. This prevents situations where the ball might get stuck. The ball bounces off the edges of the field, rather than going out of bounds. If this occurs in the goal area, a point is scored. The size of the field is variable; ball and player speeds are fixed at one unit per turn. The simulation proceeds as a turn based game. Each player takes their turn in order, with control switching between teams at each turn. Section 2.4 provides more detail on the exact mechanism employed to controls turns. Each player can execute one of ten moves per turn. They may move from their current position to any neighbouring position, stay still or kick the ball. Kicking the ball only succeeds if the player is within one unit of its current position. The balls' direction will then be determined by its position relative to the player. After each player kicks the ball, it goes into the air for one unit, and will remain there until it moves over a clear patch of the field. Once it lands, the ball will continue to move for 3 units, unless it hits an obstacle. When a goalie kicks the ball, it stays in the air for a minimum of three units. If the ball hits a player, it will stop.

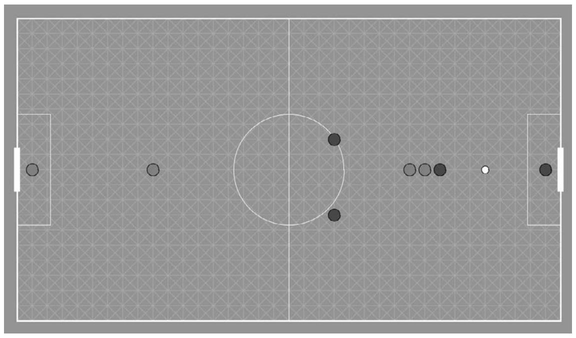

**Figure 1. Game play screen shot**

### 2.2. Implementation Requirements

The majority of students enrolled in Artificial Intelligence have been taught to program in `Java`, making it the language of choice for the framework. `Java`'s cross-platform capabilities are also appealing, as students are required to use both Unix and Macintosh systems with the School, with many using Microsoft Windows at home.

The framework was constructed such that it could be reused many times and extended to fit new requirements. In particular, it was hoped that the framework could form the core of all practicals for the next few years. A second desirable feature was the ability to change the decisions making mechanism the students were required to employ. To this end, the simulator engine was carefully structured to be as flexible as possible and provides numerous options for configuring its behaviour.

In the 2004 course, students were required to implement two decision making algorithms; minimax and a solution driven by a genetic algorithm. The prevention of cheating was also a key concern, which meant preventing access to particular data and careful structuring of important sections of the framework. The overall layout of the framework is shown in Figure 2. Note that all players execute their code in a separate thread, along with the core of the simulator itself. Section 2.4 will discuss this in more detail.

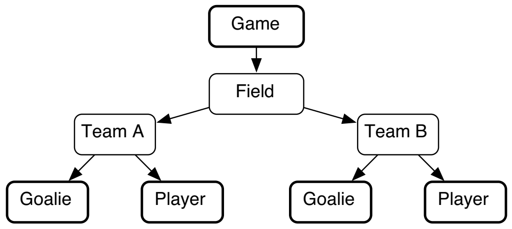

**Figure 2. This figure shows the dependence of various framework structures on one-another. Those entities shown with thick outlines are executed in their own thread.**

### 2.3. Player Hierarchy

To run within the simulator, student solutions must fit into the Player class hierarchy. A simple base class, `Player`, provides the minimal amount of code required to be a player. Students develop players that may compete in a soccer match by extending this class. A key aspect of `Player` is an abstract method called *haveTurn*, in which the students' decision making code must be written. However, for some decision making mechanisms, a simple base class is unsuitable. Section 3 provides some examples of this scheme.

### 2.4. Execution Modes

Real-time control is an important facet of robot soccer, requiring timely completion of searches and decision making algorithms. The framework satisfies this need by allowing the specification of execution constraints, in particular enforcing a time limit on each players' decision making mechanism.

Executing decision making code in a thread is a simple solution to this problem. In theory, each thread can be killed or suspended after the allowed time has elapsed. Unfortunately, `Java` does not provide these mechanisms, except via deprecated methods. Moreover, `Java`'s implementation uses exceptions, meaning player code could still prevent thread suspension, thus avoiding time restrictions. Instead, each player is given a certain amount of time to evaluate their decision. If this time is exceeded, their decision will not be executed during the current turn, and control is passed immediately to the next player. Figure 3 shows an example of this scheme. This scheme is implemented in the

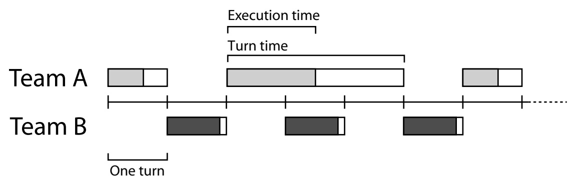

**Figure 3. Execution model diagram**

following fashion. Each player is given allocated a game thread, which contains a simple loop, shown in Algorithm 1. Each thread has its own semaphore, initialised to zero. By incrementing the semaphore, the main thread can control the time at which each player's turn begins. The main thread then sleeps until the time quanta has expired and checks if the player has completed their turn. If not, control moves to the next player, and the current player misses their opportunity to move. In this case, the first player is at a large disadvantage, as they are stationary for one turn. Once they complete the required computation, the resultant move will be executed. However, it will be based on stale state information, as most players will take a snapshot of the board state at the start of their turn. If the player has successfully completed their move, the state of the field is updated, and the next player begins their turn.

**Algorithm 1** Player thread execution

```
while Game not over do
  Decrement semaphore
  Make decision (Execute the students code)
  Apply decision at end of turn
end while
```

An obvious fault in this scheme is that if the current player does not complete their turn, the next player does not have full control of the CPU, reducing the amount of work they can do in their allocated time. Solving this problem is difficult, but may be addressed in the future. The effect on game results is negligible, as both players are equally effected.

## 3. Applications

In the 2004 offering of AI, the framework was used for two practicals. The first was based on minimax, the second on genetic algorithms. This section explores how the framework was employed and extended for each topic.

### 3.1. Minimax

Minimax is a well known heuristic-based search algorithm[6]. In the context of robot soccer, the right heuristic can produce intelligent behaviour. A well-known disadvantage of minimax that it requires a large number of node expansions - $n^k$, where $k$ is the number of available moves per turn, and $n$ is the level of look-ahead required. The robot soccer framework provides a number of ways to reduce the values of both $n$ and $k$. One of the aims of the practical was for students to develop further minimisation strategies.

The model reduces $k$ by only allowing ten different moves from any position on the board. Students could reduce this number further by careful selection of the moves to expand. Simply not attempting to kick the ball when it is too far away reduces $k$ to 9 in most cases. Similarly, a player need not expand moves in all directions, since some may not be useful. For instance, when the ball is in front of the player there is no need to evaluate moves which take the player backwards.

To look-ahead $t$ turns, $n = t \times 2p$ levels must be expanded, where $p$ is the size of each team. If a particular implementation can expand $n'$ levels within the allotted time, setting $p$ to 1 will clearly produce the furthest look-ahead. However, having one player per team would not provide a very exciting game. Thus, a goalie was added to each team. The goalies were given entirely deterministic behaviour, allowing them to be omitted from the minimax tree.

A second common problem with minimax is that of deadlock. If two players can perfectly predict where the other will move, minimax will always collapse into deadlock, as it has the least-bad outcome for both players. The framework greatly reduces instances of deadlock by insisting that the ball is kicked over players, for at least one unit. Detecting deadlocked situations and restarting the match would be a useful feature of a future version of the framework.

### 3.2. Genetic algorithms

Genetic algorithms formed the basis of the second practical. Constructing a decision making system with a genetic algorithm requires two major components; an optimiser and a cost function. Interpreting the contents of the chromosome and designing a suitable cost function were the key challenges faced by students. Both required careful thought on the part of each student and necessitated the inclusion of debugging and testing tools in the overall framework. Unfortunately, this task proved to be beyond many students, necessitating the provision of a second method of assessment. A driver was conceived which was able to test the optimisation portion of their solution separately. Further, a series of appropriate test functions were also produced, all of which had known solutions. This enabled students to verify the correctness of their GA optimiser.

A key aspect of the testing and debugging facilities was the inclusion of a series of players, against whom the students could train their own player. During the early stages of optimisation, it is likely that most chromosomes would generate poor strategy, making it impossible to win against a sophisticated opponent. The provision of progressively more capable opponents allowed the training to proceed in a gradual fashion, helping to drive the optimisation process.

## 4. Outcomes

The primary objective for the practical components was to imbue each student with an understanding of the required decision making algorithms. However, by adding some extra aspects to the assignment such as time limited decision making and using a GA for the second assignment, other outcomes were also achieved.

### 4.1. Nightly Tournaments and Competition Ladder

Each students' player participated in nightly tournaments against all other members of the class. This provided timely feedback about their performance relative to the rest of the class. As part of the tournament, a ladder was established, charting the performance of each student. The ladder encouraged a competitive atmosphere to develop, which drove students to improve their solution and thus their ranking. In 2004, a round-robin tournament system was selected, in which each player plays against every other player, $n^2 - n$ matches in total. This method was chosen in the interests of fairness, to provide a completely objective view of performance. However, the run-time of the tournament was significant. Each match had a 5 minute time limit set and around 40 students participated in the initial tournaments. The total run time for this tournament was thus 8000 hours, clearly far too long. Section 5 discusses alternate models for the tournament.

Along with the tournaments and the ladder, a bulletin board system was also employed during 2004. In combination with the ladder, the bulletin board hosted a large amount of discussion between students, debating various approaches to each stage of the practical. This discussion gave rise to clever and insightful solutions to each stage of the practicals.

### 4.2. Efficiency

The minimax assignment posed several efficiency challenges, as the search is order $n^k$, where $n$ is the level of look ahead, and $k$ is the number of available moves per turn. The primary approach to timely search completion is to limit the look ahead to a fairly conservative value. However, the students discovered that having the maximum possible look-ahead would greatly increase their chances of winning. Several discussions were held on the bulletin boards on the best mechanisms for reducing the time taken by the search. One optimisation can be made by realising that `Java` applications slow down as more objects are allocated and freed, principally due to garbage collection. By reducing the number of object allocations, minimax can be made significantly faster. A naive implementation allocates one object per node expansion in the tree, resulting in $O(n^k)$ objects being allocated and destroyed in total. However, it is possible to reduce this number to $n$, allowing the level of look-ahead to be increased.

## 5. Conclusion and future work

This paper has presented a soccer simulation framework using a simplified game model. Key implementation issues were discussed, along with experiences from its use in an undergraduate artificial intelligence course.

Several areas of the frameworks could be improved in future versions. Improving the mechanism for controlling player execution would make the framework fairer and more predictable. The tournament proves the most important area for improvement. The current round-robin scheme has many positive aspects; it is fair and very simple to implement. However, its execution time is far too long. There are two solutions to this problem. The first is to retain the current scheme and distribute the computation. This would provide a significant decrease in execution time, but would not improve scalability. The second is to move to a division based scheme, such as that employed in many amateur sports. In this model, the class is broken up into a series of small groups, consisting of around 10 students, and in each group a round-robin tournament is held. If a player manages to defeat a significant number of their peers, they will be moved up into the next division. If they lose against too many opponents, they will drop. A third option is to treat the tournament as a sorting problem, where ordering two players consists of playing a match between them. A simple sorting method, such as bubble-sort, lends itself to this application.

## References

[1] S. Coradeschi and J. Malec. How to make a challenging AI course enjoyable using the RoboCup soccer simulation system. *RoboCup'98: Robot Soccer World Cup II, Lecture Notes in Artificial Intelligence*, (1604), 1999.

[2] T. B. et al. TeamBots. Technical report, 2004.

[3] W. Fikkert and H. Dollen. MiS20 - Robot Soccer Simulator. Technical report, 2003.

[4] R. S. L. Members. The RoboCup Soccer Simulator. Technical report, 2004.

[5] M. Technical Committee. *Middle Size Robot League Rules and Regulations for 2004, Draft Version pre-8.2*. 2004.

[6] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, $3^{rd}$ edition edition, 1993.