A Motivating Example Problem for Teaching Adaptive Systems Design

Lutz Prechelt (prechelt@ira.uka.de)
Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe, D-76128 Karlsruhe, Germany
Phone: ++49/721/608-4068, Fax: ++49/721/694092

Abstract

There are some general lessons to be learned about the design of adaptive systems and the best method to learn them is an appropriate exercise. This paper lists these lessons, discusses why it is difficult to use examples from real applications for the exercise, and suggests a game to be used as an alternative example problem.

1 Adaptive Systems

The term *adaptive system* is being used in various areas of computer science for quite a while now and adaptivity has been considered to be one of the most important properties for information systems of the future. Adaptive systems are roughly defined to be *"systems that react sensibly even in situations not foreseen by their designers"*. Of course we are not satisfied if such behavior is purely accidental — we want to *design* adaptive systems. Incorporating this aspect into the definition above shows that the definition is highly problematic, because then it reads: *"Adaptive systems are systems that are designed to react sensibly even in situations not foreseen by their designers"*, which is almost self-contradictory. Weaker definitions do not help: The definition *"An adaptive system can react sensibly to a set of situations not explicitly considered completely during the design of the system"* allows any heuristic algorithm to qualify as an adaptive system.

However, although good definitions are difficult, a computer scientist has an intuitive notion of what should be called an adaptive system and what is just an algorithm containing an `IF`. We thus do not want to participate in the debate about the proper definition of this term, nor in the debate whether it is justified to use it in contexts such as connectionist systems. Instead, this paper contributes an example that is useful in teaching the explicit design of adaptive systems.

Terminology: In the following we will talk about *(adaptive) systems* (i.e., programs) that interact with *environments* (i.e., parts of the world outside the programs but relevant to them) in order to implement *applications* (i.e., useful information systems).

The questions of concern for teaching are

1. How can students develop a notion of what an adaptive system is ?
2. How can we teach the *design* of adaptive systems ?

2 Teaching Adaptive Systems Design

To answer both questions, we will first discuss the general lessons to be learned by the students about adaptive systems design, then list some of the "real" applications of adaptive systems and their problems for teaching, and finally suggest a game as a good example for teaching and exercising adaptive systems design.

The following lessons must be learned about adaptive systems design:

1. Before an adaptive system can be designed, the state space and dynamics of the environment it will be exposed to must be analyzed thoroughly. Hasty designs will often result very weird behavior.
2. It is difficult to make an adaptive system complete, i.e., *surprising situations* (which the system cannot handle well) often arise in a real environment.

3. Once an adaptive system is created, its dynamics are often difficult to understand, i.e., the behavior of the adaptive system itself may also be surprising.
4. Not only must systems react to their environment but also the environment reacts to the actions taken by the system; this complicates the dynamics of an adaptive system.
5. In most cases there is no single best solution for the design of an adaptive system; different approaches have different strengths and weaknesses.

All of these issues can be addressed using example tasks taken from real applications such as manufacturing control, traffic control, computer-aided learning, intelligent cache (or stock) management, etc. These applications, however, all exhibit one or more of the following problems when used as examples to teach adaptive systems design

1. The application domains in themselves are quite complex and require a lot of knowledge acquisition work on the part of the student to be understood well enough.
2. There are many channels (parameters) through which the system to be used influences its environment. This makes the adaptation problem multi-dimensional and thus very difficult.
3. Within the bounds of the *simplified* form of the problem that can reasonably be expected to be tackled in a course, real surprises in environment behavior tend to be rare for these problems. The environments are relatively simple and students understand them pretty well.
4. It is often quite difficult to evaluate how good a particular adaptive system for some application is. Of course there are objective functions whose values can be recorded and compared — but which environment situations should be selected to test the system? In most cases it will be impossible to perform a long-enough real world test, which would be the only way to avoid this problem.

What we need to find in order to teach adaptive systems design is a good example that is simple enough to be understood and implemented in the context of a single course, is difficult enough to let us learn the lessons listed above, and allows to evaluate solutions easily.

I propose a game, called Knobeln, for this purpose. The corresponding adaptive systems design task is "write a program that implements a successful strategy for this game". Using a game has the additional advantage that the evaluation takes the form of a contest and is thus very motivating for the students.

3 The Knobeln Game

These are the rules:

1. Both players (at the same time) chose an integer number in the interval $a \ldots b$. This selection of two numbers is called a *throw*. The players can watch each throw as it is made, i.e., they know all numbers they and their opponent have thrown up to the current throw.
   For the following let us assume that player P choses number p_1 and player Q choses q_1.
2. If $p_1 = q_1$, nobody wins a point.
3. Otherwise, the player with the higher number wins, unless the number is more than twice as high as that of his/her opponent. Let us assume that $p_1 > q_1$, then P wins if $p_1 \leq 2q_1$ and Q wins if $p_1 > 2q_1$.
4. A player who wins a throw with some number n gets $\lfloor \log_2(n) \rfloor$ points in this throw. The other player gets 0 points in this throw.
   Example: if P wins, he/she gets $\lfloor \log_2(p_1) \rfloor$ points e.g. if $p_1 = 6800$, player P gets 12 points.
5. A game consists of L throws.
6. Both players must throw series of non-decreasing throws. These series must (for each player individually) have a length of at least k throws; longer series are allowed.
   Example: If P throws $(p_1, p_2, p_3, \ldots)$ then $p_1 \leq p_2 \leq p_3 \leq \ldots \leq p_k$ is required. After that, $p_k > p_{k+1}$ is allowed. If $p_k > p_{k+1}$ then $p_{k+1} \leq p_{k+2} \leq \ldots \leq p_{k+k}$ is required; else there exists some smallest number j (with $j > k$) for which $p_j > p_{j+1}$ and then $p_{j+1} \leq p_{j+2} \leq \ldots \leq p_{j+k}$ is required. (And so on through the whole game.) If for instance $k = 3$ then the sequence 1, 2, 3, 1, 4, 5, 6, 8, 2 is allowed, while 1, 2, 3, 4, 1, 2, 1 is not because the last 1 (less than 2) comes too early.
7. The values for the parameters are: $a = 1$, $b = 12288$, $k = 8$, $L = 1000$ (other values could be used). This means the maximum number of points to win in a single throw is 13.
8. The game is always played as a tournament in which each player plays against every other. The

objective of the players is to get as many points as possible. The points made by a player in all his/her games are summed. The player with the most points is the winner of the tournament.

The final rule means that there is no such notion as "winning a particular game": A player is not interested in how many points the opponent gets, but only in how many own points can be achieved. This, together with the logarithmic counting rule, makes cooperation attractive: If I am greedy and try to let my opponent not get many points, the opponent may start throwing small numbers and I cannot get many points as well; a good solution is to arrange with my opponent so that first one of us gets 13 points 8 times in a row, then the other gets 13 points 8 times in a row, and so on and on (or some similar schedule).

The adaptation problems to solve in this game are

1. How can I arrange a cooperation with my opponent ?
2. How can I detect how my opponent tries to arrange a cooperation with me ?
3. How can I detect whether (or when) my opponent is non-cooperative ?
4. How can I maximize my points in the non-cooperation case ?

Another approach would be to chose the non-cooperation case from begin on.

Experimentation with the game showed that these problems are not easy to solve (see below). All the "lessons to learn" mentioned above are addressed by this game: Quick hacks fail miserably; even sophisticated programs encounter situations their designers find surprising or react in surprising ways; since the game is symmetric, there is heavy bidirectional interaction between "system" (player 1) and "environment" (player 2); finally, since the success of a particular strategy depends heavily on the behavior of the other strategies in the tournament, there is clearly no single best solution.

4 The Knobeln Contest

The Knobeln game has not yet been used in an actual computer science course. It was, however, the subject of two small student contests here in Karlsruhe and one larger international contest. The latter, the *First International Knobeln Contest*, was announced in various newsgroups of the Usenet News system and took place in May 1993. 41 teams from 9 different countries sent strategy programs written in C by email. The actual contest was run on local machines.

The surprise of the contest was that despite the clear bias of the game towards cooperative approaches, the highest-scoring strategies were all aggressive, exploitative ones. This clearly shows that the Knobeln game, despite the simplicity of its rules, is sufficiently complex in its dynamics to be challenging: It is quite difficult to defeat exploitative moves of the opponent successfully. This is emphasized further by the fact that aggressive strategies won although the contest was carried out in *two* tournaments which both counted for the final result with a one week pause in between. During this pause, the participants could review their results from the first tournament (delivered to them in the form of throw-by-throw game protocols) and could modify their strategy program — a possibility that about half of the participants (the winner was not one of them) used. The winning strategy was one that had been created by a genetic algorithm (employing decision table learning); a fact that opens an interesting perspective on the use of state-of-the-art computer science techniques in the development of adaptive systems.

The software used to carry out the contest and the strategy programs of the participants are available for anonymous ftp from `i41s10.ira.uka.de` in directory `/pub/knobeln` (please get and read the file `README.FIRST` first).

5 Conclusion

A synthetic example, (here: an N-person game), can avoid many of the problems that other examples of adaptive systems have for normal course situations. The suggested Knobeln game features most of the fundamental problems of adaptive systems design, but is simple enough to be discussed and implemented without simplifications or restrictions in the context of normal computer science courses. In addition, its game character implies direct competition of various solutions, making the example very motivating.