Introduction

The examples included with this file – GAPBILExample, and GPEaxmple – are designed to illustrate how different types of optimisation algorithms can be used to directly adapt an agent's behaviour. The four algorithms that are included (a perturbation search, a genetic algorithm (GA), a population based incremental learner (PBIL), and genetic programming (GP)) were chosen to emphasise the wide range of techniques that can be applied and are not necessarily the most effective in this or any other application. In fact, there is a formal proof that no one optimisation algorithm can perform better than all other on all problems.

Eat Food or Eat Poison?

To provide a framework within which the reader can experiment with the algorithms, a single agent is placed in an environment containing three types of objects, walls (black squares), food (green squares), and poison (red squares). The agent can perform three actions, move forward, turn left or turn right, and can also sense the type of the object directly in front of it. Associated with the agent is a "health" parameter which is incremented each time the agent consumes a food object and decremented each time it consumes a poison object.

The performance of the agent is measured by its health after 5,000 time steps in the simulated environment. Although the problem of maximising the agent's health appears to be very simple to solve (and indeed, the simplicity of the problem is a bonus in terms of understanding the agent's behaviour), there are some subtleties. For example, although the agent has no memory, it must learn to wander around the environment in such a way that it does not become caught in a cycle.

GAPBILExample

This file shows how perturbation search, GA, and PBIL optimisation algorithms can be used to learn (or evolve in the terminology of GAs) rules for controlling the agent in such a way as to maximise its health. The details of the implementations are explained in the comments of the source files and will not be repeated here. The perturbation search is the most basic of the three techniques and simply involves testing random modifications of the best behaviour discovered thus far to see if any of them improve upon it. As soon as one is found, it becomes the new best behaviour and the process is repeated. Unlike the GA and PBIL, the perturbation search is "greedy" and frequently becomes stuck in locally optimal behaviours.

GAs are described in detail in this book in the article "Genetic Algorithms: Evolving the Perfect Troll", and good internet resources (including an excellent tutorial) can be found at http://cs.felk.cvut.cz/~xobitko/ga/, or by searching for "GA" at http://www.researchindex.com. PBILs are not as well known as GAs, but can be used as drop-in replacements, and have been shown to offer superior performance in a number of applications. More information on PBILs can be found at http://www.researchindex.com by using "PBIL" as a search term.

The active optimisation algorithm can easily be changed by modifying lines 44, 96, 104, and 515 in the file `GAPBILExampleDoc.cpp`. Line 104 controls whether the program loads a pre-trained optimisation algorithm, such as those provided on the CD (in `PS.txt`, `GA.txt` and `PBIL.txt`), while line 515 periodically saves the operational algorithm's status. To maximise the speed at which behaviours can be evaluated, the program does not animate the agent or its environment during the optimisation process. To actually see the agent in action, the program needs to be recompiled with the `nSlowMotion` flag (initialised in line 72) set to one, and line 104 set to load a saved optimisation algorithm. No learning or adaptation occurs when the program is running in slow motion.

GPExample

This file demonstrates how GP can be used to evolve programs for controlling the agent. GP represents programs as tree structures and applies specialised genetic operators (i.e. crossover and mutation) to derive novel "child" programs from parents. The GP example is more open-ended than the perturbation search, GA, and PBIL ones in the sense that there is no pre-defined upper limit on the complexity of the behaviour that can evolve. The GP example automatically stops adapting and switches to slow motion when the agent's health after 5,000 iterations reaches the value defined on line 417 in `GPExampleDoc.cpp` (set to 250 by default). The GP implementation provided is quite simple and the interested reader is encouraged to read more about GP from sources such as http://www.genetic-programming.org and by searching for "GP" at http://www.researchindex.com.

John Manslow
02/10/2001