`TSearch` Class Documentation

Version 1.1 – 2/15/08

Randall D. Beer
Cognitive Science Program
1910 E. 10th St. – 840 Eigenmann
Indiana University
Bloomington, IN 47406
rdbeer@indiana.edu
http://mypage.iu.edu/~rdbeer/

1. Introduction

Evolutionary algorithms (EVAs) have become increasingly widely used in adaptive behavior research since the pioneering work in the late 1980s and early 1990s. General introductions to genetic algorithms include (Goldberg, 1989; Mitchell, 1996; Jacob, 2001), while Nolfi and Floreano (2000) provide an overview of the application of EVAs to autonomous agents. The purpose of the `TSearch` class is to provide an easy-to-use, efficient C++ implementation of a simple, real-valued, generational EVA for use in evolutionary agent simulations. It depends on the files `VectorMatrix.h`, `random.h` and `random.cpp`. Versions of this software have been in continuous use in my research group since 1995, but this represents its first public release.

2. Class Overview

A search is an instance of the class `TSearch`. In order to carry out an evolutionary search, you must create a search object, configure it, execute the search, and then examine the results. The configuration and execution of a search object are described in detail in the next two sections. Here, I will briefly describe how to create a search and query its status. A simple example search program is given in Section 6.

From the point of view of agent evolution, the most important thing you need to specify about a search is how it will interact with your agent model. This requires two pieces of

information, both of which can either be supplied to the `TSearch` constructor or set later using the appropriate member functions. First, you must specify the number of real parameters that you will be searching. The relevant member functions for accessing this information are `(Set)VectorSize()`. Second, you must provide an evaluation function that accepts a vector of parameters and returns a real number representing the performance of that parameter vector on the task you have set. The relevant member function for setting a search's evaluation function is `SetEvaluationFunction()`.

An evaluation function needs to take a search vector and a `RandomState` as input. The random state should be used for all of the random number generation that your evaluation function does (see the file `random.h` for supported operations). The type of a search vector is `TVector<double>`, where a `TVector` is a C++ vector class that supports a large variety of operations. For our purposes here, the most important operations are querying the size of a `TVector` (using the member function `Size()`) and accessing or modifying an element of a `TVector` (using the fast but unsafe operator `[]` or the slower but safe `()` operator). Each element of a parameter vector is a double-precision floating-point number, nominally in the range [-1, 1] (although it may move outside of this range during a search, see the `(Set)SearchConstraint()` member function).

Your evaluation function must decode to elements of the search vector into the domain parameters necessary for evaluation (e.g., in the weights, biases and time constants for a CTRNN). The utility function `MapSearchParameter()` is provided for this purpose. It linearly maps a number in the nominal range [-1, 1] to a given range. Optionally, you can also specify minimum and maximum values to clip the mapped value to. This can be important for parameters such as CTRNN time constants that must never be mapped below a certain value. Alternately, you can use your own exponential mapping for such parameters. Note that there is also a utility function `InverseMapSearchParameter()` that performs the inverse linear map.

Once the search vector has been decoded, your evaluation function must perform the actions necessary to evaluate the performance of that parameter vector on your task and return a non-negative double-precision floating-point number characterizing that performance. Note that a performance of 0 may result in that individual never being selected to serve as a parent, depending upon the selection mode that you chose (see `(Set)SelectionMode()` in Section 3 below).

A number of member functions for querying the status of a search object are available. The `Generation()` member function returns the current generation. The member functions `Individual()`, `Performance()` and `Fitness()` return the search vector, performance and fitness of a given individual in the population (Note that the first individual in the population has an index of 1, not 0). Finally, the member functions `BestIndividual()`, and `BestPerformance()` return the parameter vector and the performance, respectively, of the best agent found so far during a search.

3. Configuring a Search

3.1 Basic Configuration

The `TSearch` class supports a wide variety of configuration options. As already mentioned in Section 2 above, the number of parameters and the evaluation function can either be specified in the constructor or set later using the `SetVectorSize()` and `SetEvaluation Function()` member functions. In addition, the population size and the maximum number of generations that the search should run can be set and accessed by the member functions `(Set)PopulationSize()` and `(Set)MaxGenerations()`, respectively.

3.2 Configuring Selection

Two selection modes are supported. In `FITNESS_PROPORTIONATE` mode, individuals are selected to serve as parents with a probability proportional to their fitness (normalized performance), using linear fitness scaling to maintain a constant selection pressure (Goldberg,

1989). In `RANK_BASED` mode, individuals are sorted by fitness and assigned an offspring count based solely on their rank using Baker's linear ranking method (Baker, 1985). In both cases, Baker's stochastic universal sampling algorithm is used to perform the actual selection (Baker, 1987). These selection modes can be set and accessed by `(Set)SelectionMode()`. The selection mode defaults to `RANK_BASED`. Elitist selection (De Jong, 1975), in which some fraction of the best individuals are copied to the new population, is also supported through the `(Set)ElitistFraction()` member functions. The elitist fraction defaults to 0.0. Finally, the selection pressure can be controlled by specifying how many offspring the best individual in the population should have on average using the member functions `(Set)MaxExpectedOffspring()`. This value must by greater than 1 (so that fraction of highly fit individuals actually increases in the population over time) and less then or equal to 2 (so that the linear fitness scaling formula makes sense). The expected offspring of the best individual defaults to 1.1.

3.3 Configuring Reproduction

Once parents have been selected, children must be produced to fill the new population. `TSearch` provides both mutation and crossover genetic operators. The mutation operator is Gaussian in nature (Bäck, 1996), perturbing a parent with a mutation vector whose direction is uniformly distributed on the unit hypersphere and whose size is normally-distributed with 0 mean and a variance that can be set or accessed by `(Set)MutationVariance()`. The mutation variance defaults to 1.0. The crossover operator can function in either UNIFORM or TWO_POINT (the default) mode, specified by `(Set)CrossoverMode()`. Crossover occurs with a probability specified by `(Set)CrossoverProbability()`, which defaults to 0.0.

Support is also provided for modular crossover, in which sets of parameters are treated as a unit for the purposes of crossover. For example, you might want to treat the bias, time constant and incoming weights of each neuron in a CTRNN as a unit for crossover. There are two ways to specify crossover modules. A crossover template is a `TVector<int>` that contains numbered

runs that specify the different crossover units. For example, the template vector (1, 1, 1, 2, 2, 3, 3, 3, 3, 3) specifies three crossover units: (1) parameters 1-3, (2) parameters 4 and 5, (3) parameters 6-10. A crossover template can be specified by `(Set)CrossoverTemplate()`. Note that these numbered runs must be contiguous; a template vector such as (1, 1, 1, 2, 2, 1) is not allowed. Alternately, you can specify a `TVector<int>` of crossover points. For example, the vector (1, 4, 6) of crossover points specifies the same crossover units as the earlier crossover template example. Crossover points can be specified by `(Set)CrossoverPoints()`. By default, no crossover modules are present, so each parameter is treated as its own unit for the purposes of crossover.

There are two different modes in which reproduction can occur, specified by the `(Set)ReproductionMode()` member functions. In `GENETIC_ALGORITHM` mode, children always replace their parents in the new population. In `HILL_CLIMBING` mode, children only replace their parents if they have higher fitness. The reproduction mode defaults to `GENETIC_ALGORITHM`.

3.4 Miscellaneous Configuration

Search vectors are always initialized to random values in the range [-1, 1], which are then mapped to parameters in your evaluation function. A *constrained search* is one in which the elements of a search vector are clipped to these ranges throughout a search, while an *unconstrained search* allows these values to subsequently leave the range [-1, 1] during the search if higher fitness can be obtained by doing so. The member functions `(Set)SearchConstraint()` can be used to specify the search constraint, with an argument of 1 meaning the search is constrained and an argument of 0 meaning the search is unconstrained. It is also possible to pass in a `TVector<int>` of 0s and 1s which specifies for each parameter individually whether or not it is constrained. By default, all search parameters are constrained.

Under some conditions, parents may be retained in the new population. This is true both for elitist selection and for parents whose fitness is higher than their children in `HILL_CLIMBING` mode. Normally, the fitness of a given search vector is evaluated only once, so a retained parent will also retain its fitness value from the first time it was evaluated. However, in some cases (especially when there is randomization in your fitness evaluation), it is important to re-evaluate the parent each time. In this case, you must call the `(Set)ReEvaluationFlag()` member function with an integer flag (1 = re-evaluate, 0 = do not re-evaluate). The re-evaluation flag defaults to 0.

For long-running evolutionary searches, it can be very unpleasant to have a machine go down just as your search is nearing its end. To help with such problems, `TSearch` provides a very simple checkpointing facility that can be used to save a later resume an ongoing search. When checkpointing is enabled, every `CheckpointInterval()` generations, `TSearch` dumps a binary file named `"search.cpt"` containing an almost complete snapshot of the state of the search. By "almost complete", I mean the file contains everything about the state of the search except the function pointers (to, for example, the evaluation function), which cannot meaningfully be stored in a file. Checkpoint intervals are specified by the `(Set)CheckpointInterval()` member functions, with a value of 0 meaning no checkpointing is done (the default). This checkpoint file can later be resumed using the `ResumeSearch()` member function (see Section 4). When you resume a checkpointed search, it is the responsibility of your code to ensure that all function pointers are set up as they were in the original search.

Aside from the evaluation function, there are several other features of `TSearch` that can be configured by providing pointers to user-defined functions. The `SetBest ActionFunction()` member function allows you to specify a function that will be called each time a new individual is found whose performance is better than all previously encountered individuals. Note that the best action function is called at most once per generation. A common use of this function is to dump the parameters of the new best individual to a file. By default,

there is no best action function. The `SetPopulationStatisticsDisplay Function()` member function allows you to specify a function that will be called each generation to display information about how a search is progressing. By default, the generation number, best performance, average performance and performance variance are printed each generation. However, it is sometimes useful to be able to customize the display of search progress. For example, you might want to call a function that graphically displays this information each generation. Finally, the `SetSearchResultsDisplayFunction()` member function allows you to specify a function that will be called at the end of a search. For example, you could provide a function that sends an e-mail message when a search completes. By default, there is no search results display function.

4. Executing a Search

Once a search has been configured, it can be executed simply by calling the member function `ExecuteSearch()`. This will initialize the search if it has not already been initialized (set the generation to 0, randomly generate an initial population), and then enter an evaluation/selection/reproduction loop until the search completes. It is sometimes useful to separately initialize and execute a search (see Section 5). For this purpose, the member function `InitializeSearch()` is provided. If `ExecuteSearch()` is invoked on a search that has already been initialized, then evaluation/selection/reproduction loop will simply begin without reinitialization. Furthermore, `InitializeSearch()` can be used to reinitialize a search object in preparation for another search.

Normally, a search ends once the maximum number of generations has been reached. However, it is sometimes convenient to specify other criteria for terminating a search (see Section 5). The `SetSearchTerminationFunction()` member function can be used to specify a function that will be called every generation and the search will terminate whenever this function returns 1. Note that regardless of your search termination function, a search will always terminate once its maximum number of generations has been reached.

Finally, the member function `ResumeSearch()` can be used to resume a checkpointed search from a file named `"search.cpt"` in the same directory as the executable. `ResumeSearch` essentially configures and initializes a search from the checkpoint file and then executes it. Recall that it is not possible to store function pointers in the checkpoint file, so it is up to your code to make sure that, for example, the same evaluation function is set when you resume a search.

5. Advanced Search Patterns

The design of the `TSearch` class supports several more advanced patterns of search, including staged searches, seeded searches, and anchored searches. Each of these search patterns will be described in this section.

5.1 Staged Searches

In a staged search, you arrange the search as series of steps from simpler to harder problems, first evolving the population on a simpler evaluation function until it exceeds some threshold of performance, then switching to a harder evaluation function, and so on. Staged searches can be very important for evolving complex agent behavior.

Here is a simple example. Suppose that `EasyEvaluationFunction` and `HardEvaluationFunction` are defined and we want to switch from the first to the second when the best individual in the population exceeds a performance of 0.95. First, we need to set up the appropriate search termination function:

```
int MyTerminationFunction(int Generation,
                          double BestPerf,
                          double AvgPerf,
                          double PerfVar)
{
    if (BestPerf > 0.95) return 1;
    else return 0;
}
```

Now we set up and execute the 2-stage search:

```
TSearch s(NumParams);

s.SetMaxGenerations(250);
s.SetMutationVariance(0.1);
/* ... and so on to set up the search object */

/* Stage 1 */
s.SetSearchTerminationFunction(MyTerminationFunction);
s.SetEvaluationFunction(EasyEvaluationFunction);
s.ExecuteSearch();

/* Stage 2 */
s.SetSearchTerminationFunction(NULL);
s.SetEvaluationFunction(HardEvaluationFunction);
s.ExecuteSearch();
```

Note the use of a `NULL` search termination function in stage 2 so that that the search will continue to the maximum number of generations regardless of the best performance in the population during stage 2. This idea obviously generalizes to any number of stages and to more complex conditions for passing from one stage to the next.

5.2 Seeded Searches

Normally, a search begins with a completely random population of individuals. However, sometimes we want to seed a search with our own initial individuals. For example, we might want to initialize a search with random perturbations around some individual parameter set (e.g., one obtained from a previous search). Or we might want to initialize a search with random center-crossing CTRNNs.

As a simple example, consider the following function for initializing 3-parameter individuals with random perturbations in the range [-1,1] from the parameter vector (2, 7, -1), where each parameter is in the range [-10, 10]:

```
void InitializeIndividual(TVector<double> &v)
{
  v[1] = InverseMapSearchParameter(2+UniformRandom(-0.1,0.1),-10,10);
  v[2] = InverseMapSearchParameter(7+UniformRandom(-0.1,0.1),-10,10);
  v[3] = InverseMapSearchParameter(-1+UniformRandom(-0.1,0.1),-10,10);
}
```

Note the use of the utility function `InverseMapSearchParameter()` to translate from domain parameters to search parameters. Now we can seed and execute the search as follows:

```
TSearch s(3);

s.SetEvaluationFunction(Evaluate);
s.SetPopulationSize(100);
/* ... and so on to set up the search object */

/* Initialize and seed the search */
s.InitializeSearch();

for (int i = 1; i <= s.PopulationSize(); i++)
  InitializeIndividual(s.Individual(i));

/* Now we can start the search */
s.ExecuteSearch();
```

5.3 Anchored Searches

Sometimes, we want to not merely seed a search with variations of a particular individual, but to anchor a search on that individual so that it cannot wander too far away. In this case, the domain parameters being searched are not the parameters of the individual directly, but rather perturbations to the parameters of the individual. Supporting anchored searches is accomplished simply by modifying your evaluation function so that, rather than mapping search parameters to domain parameters directly, it maps them to perturbations to domain parameters.

For example, instead of doing something like

```
c.SetConnectionWeight(1, 2, MapSearchParameter(v[1], -10, 10));
```

You instead do something like

```
c.SetConnectionWeight(1, 2, 4.0 + MapSearchParameter(v[1], -1, 1));
```

6. Multithreaded Evaluation

The fitness evaluation of individuals in a population is usually the most computationally-intensive operation in an evolutionary algorithm. For this reason, `TSearch` provides a facility for multithreaded evaluation on multicore/multiprocessor computers that support the `pthread` library. Threaded evaluation can be enabled by uncommenting the line `#define THEADED_SEARCH` in `search.h`. The number of evaluation threads to use can be specified by the `THREAD_COUNT` symbol in the same file. Of course, this number should be less than or equal to the total number of processing cores available on your computer.

When using multithreaded evaluation, it is essential that your evaluation function be thread-safe. Since multiple instances of your evaluation function can be running at the same time, it must not maintain any global state. `TSearch` automatically takes care of the most common source of global state, random number generators, by passing a unique random number generator into your evaluation function for each individual in the population. You should use this random state for all random number generation that your evaluation function does. As long as your evaluation function is thread-safe, `TSearch` guarantees that the same result will be obtained from a given random seed regardless of how many threads are used to evaluate your population.

7. A Simple Example

```
// ************************************************
// A very simple example program for the TSearch class
// ************************************************

#include "Search.h"


// A simple 2D inverted quadratic evaluation function

double Evaluate(TVector<double> &v, RandomState &rs)
{
    double p1 = MapSearchParameter(v[1],-10,10),
           p2 = MapSearchParameter(v[2],-10,10);
    return 200-(p1*p1+p2*p2);
}


// The main program

int main (int argc, const char* argv[]) {
    TSearch s(2);

    // Configure the search
    s.SetRandomSeed(765234);
    s.SetEvaluationFunction(Evaluate);
    s.SetSelectionMode(RANK_BASED);
    s.SetReproductionMode(GENETIC_ALGORITHM);
    s.SetPopulationSize(10);
    s.SetMaxGenerations(25);
    s.SetMutationVariance(0.1);
    s.SetCrossoverProbability(0.5);
    s.SetCrossoverMode(TWO_POINT);
    s.SetMaxExpectedOffspring(1.1);
    s.SetElitistFraction(0.1);
    s.SetSearchConstraint(1);
    s.SetCheckpointInterval(5);

    // Run the search
    s.ExecuteSearch();

    // Display the best individual found
    cout << s.BestIndividual() << endl;

    return 0;
}
```

8. Class Reference

```
// --- Utility functions for mapping to and from search parameters

double MapSearchParameter(double x, double min, double max,
                          double clipmin = -1.0e99, double clipmax = 1.0e99);

double InverseMapSearchParameter(double x, double min, double max);


// --- Supported selection, reproduction and crossover modes

enum TSelectionMode {FITNESS_PROPORTIONATE, RANK_BASED};
enum TReproductionMode {HILL_CLIMBING, GENETIC_ALGORITHM};
enum TCrossoverMode {UNIFORM, TWO_POINT};


// --- The constructor
TSearch(int vectorSize = 0,
        double (*EvalFn)(TVector<double> &, RandomState &) = NULL);


// --- Basic Accessors
int VectorSize(void);
void SetVectorSize(int NewSize);
void SetRandomSeed(long seed);


// --- Search Mode Accessors
TSelectionMode SelectionMode(void);
void SetSelectionMode(TSelectionMode NewMode);

TReproductionMode ReproductionMode(void);
void SetReproductionMode(TReproductionMode NewMode);

TCrossoverMode CrossoverMode(void);
void SetCrossoverMode(TCrossoverMode NewMode);


// --- Search Parameter Accessors
int PopulationSize(void);
void SetPopulationSize(int NewSize);

int MaxGenerations(void);
void SetMaxGenerations(int NewMax);

double ElitistFraction(void);
void SetElitistFraction(double NewFraction);

double MaxExpectedOffspring(void);
void SetMaxExpectedOffspring(double NewVal);

double MutationVariance(void);
void SetMutationVariance(double NewVariance);

double CrossoverProbability(void);
```

```
void SetCrossoverProbability(double NewProb);

TVector<int> &CrossoverTemplate(void);
void SetCrossoverTemplate(TVector<int> &NewTemplate);

TVector<int> &CrossoverPoints(void);
void SetCrossoverPoints(TVector<int> &NewPoints);

TVector<int> &SearchConstraint(void);
void SetSearchConstraint(TVector<int> &Constraint);
void SetSearchConstraint(int Flag);

int ReEvaluationFlag(void);
void SetReEvaluationFlag(int flag);

double CheckpointInterval(void);
void SetCheckpointInterval(int NewInterval);


// --- Function Pointer Accessors
void SetEvaluationFunction(double (*EvalFn)(TVector<double> &v, RandomState &rs));

void SetBestActionFunction(void (*BestFn)(int Generation, TVector<double> &v));

void SetPopulationStatisticsDisplayFunction
 (void (*DisplayFn)(int Generation, double BestPerf, double AvgPerf, double PerfVar));

void SetSearchTerminationFunction
 (int (*TerminationFn)(int Generation, double BestPerf, double AvgPerf, double PerfVar));

void SetSearchResultsDisplayFunction(void (*DisplayFn)(TSearch &s));


// --- Search Status Accessors
int Generation(void);

TVector<double> &Individual(int i);

double Fitness(int i);

double Performance(int i);

double BestPerformance (void);

TVector<double> &BestIndividual(void);


// --- Search Control
void InitializeSearch(void);

void ExecuteSearch(void);

void ResumeSearch(void);
```

References

Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice*. Oxford University Press.

Baker, J.E. (1985). Adaptive selection methods for genetic algorithms. In J.J. Grefenstette (Ed.), *Proceedings of the First International Conference on Genetic Algorithms and Their Applications* (pp. 101-111). Lawrence Erlbaum.

Baker, J.E. (1987). Reducing bias and inefficiency in the selection algorithm. In J.J.Grefenstette, (Ed.), *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications* (pp. 14-21). Lawrence Erlbaum.

De Jong, K.A. (1975). An Analysis of the Behavior of a Class of Genetic Adaptive Systems. Ph.D. Thesis, University of Michigan, Ann Arbor.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

Jacob, C. (2001). *Illustrating Evolutionary Computation with Mathematica*. Morgan Kaufmann.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. MIT Press.

Nolfi, S. and Floreano, D. (2000). *Evolutionary Robotics*. MIT Press.