DGPC – DAVE'S GENETIC PROGRAMMING CODE IN C (BY DAVID ANDRE)

GOALS OF DGPC
• Use Minimal Memory - 2 (or 1) Bytes per point of program tree
• Constant memory use
• Allow easy use of automatically defined functions
• Written in C
• Have highly portable code. DGPC has been tested on Macs, PCs, Windows, Suns, and Alpha's (64-bit)
• Implement steady state evolution
• Allow non-standard individuals
• Support for priming the population
• Constant tables (easy change)
REPRESENTATION

- Polish notation representation
- Example
  - Function set $F = \{\text{AND, OR, NOT}\}$ (with arity of all functions being fixed and known)
  - Terminal set $T = \{X_1, X_2, X_3\}$
(Or (Not x1) (And x2 (Not x3)))

Lisp Representation (Tree based)  Array based Representation
## FOUR TYPES OF NODES

<table>
<thead>
<tr>
<th>Terminals</th>
<th>These nodes have no arguments and are 0-arity functions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>Nodes for $N$-arity functions have $N$ arguments.</td>
</tr>
<tr>
<td>Constants</td>
<td>These nodes are constants that obtain a value once and only once during the existence of the node over all the different fitness cases.</td>
</tr>
<tr>
<td>Macros</td>
<td>These nodes are 'macros', and thus do not evaluate their arguments prior to being called. Instead a user-defined function is responsible for the evaluation of macros. Macros may be 0-arity (as in the case of ADF's with no arguments).</td>
</tr>
</tbody>
</table>
RANDOM CONSTANTS

• If, for example, there are 50 functions and terminals, there is room for 206 random constants
Evaluation of an node

eval(branch * br)
{
    int     temp;
    int * args[MAXNUMARGS];
    temp = g_index_ptr;
    if (ISCONSTANT(br[g_index_ptr]))
        return(br[g_index_ptr++]->value);
    if (ISMACRO(br[g_index_ptr]))
        return(  GETCodeName(br[g_index_ptr]));
    else
        g_index_ptr++;
    for (i=0; i < ARITY(br[temp]); i++)
        args[i] = eval(br);
    return(  GETCodeName(br[temp])(args));
}
EXAMPLE OF FUNCTIONS

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>0</td>
</tr>
</tbody>
</table>

In function table:

<table>
<thead>
<tr>
<th>#13</th>
<th>A arity</th>
<th>Constant</th>
<th>Macro</th>
<th>Print Name</th>
<th>Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>IfTrue</td>
<td>&quot;IfTrue&quot;</td>
</tr>
<tr>
<td>#14</td>
<td>A arity</td>
<td>Constant</td>
<td>Macro</td>
<td>Print Name</td>
<td>Function Name</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>&quot;%d&quot;</td>
<td>&quot;RandomInt&quot;</td>
</tr>
</tbody>
</table>
STRUCTURE OF AN INDIVIDUAL

<table>
<thead>
<tr>
<th>Individual #17,891</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standardized_fitness</td>
</tr>
<tr>
<td>Normalized_Fitness</td>
</tr>
<tr>
<td>Hits</td>
</tr>
<tr>
<td>Penalties</td>
</tr>
</tbody>
</table>

Branches 0  $ightarrow$ tree

ADFs 0  $ightarrow$ tree

Function to 'Branch+ADF' Mapping

Branch_0
- NumFunctions = 3;
- FunctionAssignment[0][0] = 1;
- FunctionAssignment[0][1] = 2;

... etc.

ADF_0
- NumFunctions = 10;
- FunctionAssignment[2][0] = 9;
- FunctionAssignment[2][1] = 9;
- FunctionAssignment[2][2] = 3;

... etc....
STEPS FOR USING DGPC FOR A PROBLEM

(1) Decide on Architecture of individuals
    change NumBranches and NumADF's (dgp_param.h)

(2) Create Functions
    create function table (problem.c)
    create globals and problem-specific variables (problem.c)
    function headers (problem.h)
    function-branch mapping (problem.c)
    function code (problem.c)

(3) Evaluation of individuals
    eval_fitness_of_dude (problem.c)

(4) Termination conditions (dgpc.c)
    (if s_fitness > 0, continue)

(5) Specification of parameters
    Run Parameters (default.in)
    General Parameters (dgp_param.h)
/*
 * dgpc -- Dave's Genetic Programming in C -- v1.0
 * Copyright © 1994, David Andre
 * All rights reserved.
 *
 * dgp.h
 * Main headers and data structure definitions --
 * Uses only 2 bytes per node, which is
 * 2 bytes x 1000 nodes =
 * 2K per Individual == 500 Individual per MEG == 5,000 individuals
 * per 10 MEG
 */

#if __STDC__ || defined(__cplusplus)
#define P_(s) s
#define ANSI_FUNC
#else
#define P_(s) ()
#endif

#include "dgp_param.h"

#define FULL   1
#define GROWTH 2
#define RAMPED 3
#define TOURNAMENT 4
#define FITNESSPROP 5

#include "dgp_param.h"

#define MaxNumFunctions 256
#define GTYPE int
#define GENERIC int

#define max(x,y) ((x)>(y)?(x):(y))
#define min(x,y) ((x)>(y)?(y):(x))

/* Macros for accessing data structures... */

#define function_arity(n)
gpop.func_table[(n).index].arity
#define function_is_macro(n)
gpop.func_table[(n).index].macro
#define function_is_constant(n)
gpop.func_table[(n).index].constant
#define function_printname(n)
gpop.func_table[(n).index].PrintName
#define function_code(n)
gpop.func_table[(n).index].code
#define MAXSTRING 100

typedef struct {
    unsigned char index;}
typedef struct fte {
    int arity;
    int macro;
    int constant;
    char *PrintName;
    GTYPE (*code) P_(( ));
} function_table_entry;

typedef struct branch {
    short branchnum;
    short NumNodes;
    snode *tree;
} branch;

typedef struct {
    struct branch branches[max(1,NumBranches)];
    struct branch adfs[max(1,NumADFs)];
    float s_fitness;
    float n_fitness;
    float a_fitness;
    short hits;
    short penalties;
} individual; /* Structure for an individual */

typedef struct {
    short pop_size;
    individual * members;
    individual child1;
    individual child2;
    int * sorted_by_fitness;
    individual best_so_far;
    short best_of_run_num;
    float best_fitness;
    function_table_entry func_table[MaxNumFunctions];
    short num_general_functions;
    short NumFunctions[NumBranches+NumADFs];
    short
    FunctionAssignment[NumBranches+NumADFs][MaxNumFunctions];
    char ReadFiles[NumBranches+NumADFs][MAXSTRING];
    FILE * OutFile;
} population; /* Structure for a population */

snode * create_tree(short n);
float random_float(float x);
STEADY STATE EVOLUTION

1. Creation of population
2. Evaluate Fitness of each individual in population
3. If 'perfect' individual or counter > MaxReproductions
   - Yes: Quit
   - No: Select Two Parents
4. Select Crossover or Mutation
   - Mutation: Mutate one branch or ADF in both individuals
   - Crossover: Do crossover between matching ADF's or branches in both individuals
5. Choose two individuals in the population to 'die' and be replaced by the two new children.
6. Evaluate Fitness of Children