Dependency-Tree Estimation of Distribution Algorithm Reference Manual

Generated by Doxygen 1.4.7

Wed Sep 13 19:46:49 2006

Contents

Chapter 1

Dependency-Tree Estimation of Distribution Algorithm Main Page

This program implements an estimation of distribution algorithm with dependency tree probabilistic models. The algorithm does not assume that solutions are binary strings, every string position can obtain any number of values, wich is specified by the user.

For references and more information on estimation of distribution algorithms or dependency tree models, please see the MEDAL report published with this code at http://medal.cs.umsl.edu/files/2006010.pdf

All user-defined functions are located in obj-function.cpp, which defines the objective function, the function to set the numbers of values in different string positions, and the function to verify optimality.

Usage: dt-eda [parameter file name] [–help] [–version]

Author:

Martin Pelikan

Date:

2006

Version:

1.0

1.1 License Information

Feel free to use, modify and distribute the code for academic purposes with an appropriate acknowledgment of the source, but in all resulting publications please include a citation to the following publication:

Martin Pelikan (2006). Implementation of the Dependency-Tree Estimation of Distribution Algorithm in C++. MEDAL Report No. 2006010, Missouri Estimation of Distribution Algorithms Laboratory, University of Missouri-St. Louis, MO.

Chapter 2

Dependency-Tree Estimation of Distribution Algorithm Data Structure Index

2.1 Dependency-Tree Estimation of Distribution Algorithm Data Structures

Here are the data structures with brief descriptions:

Chapter 3

Dependency-Tree Estimation of Distribution Algorithm File Index

3.1 Dependency-Tree Estimation of Distribution Algorithm File List

Here is a list of all files with brief descriptions:

Chapter 4

Dependency-Tree Estimation of Distribution Algorithm Data Structure Documentation

4.1 AveragePopulationStatistics Struct Reference

Population statistics for a set of runs.

`#include <stats.hpp>`

Data Fields

- double avg_minF
- double avg_maxF
- double avg_avgF
- double min_minF
- double max_maxF
- double avg_num_evals
- double p_success
- int population_size
- int problem_size
- int num_runs

4.1.1 Detailed Description

Population statistics for a set of runs.

Definition at line 32 of file stats.hpp.

4.1.2 Field Documentation

4.1.2.1 double AveragePopulationStatistics::avg_avgF

Definition at line 36 of file stats.hpp.

Referenced by average_population_statistics().

4.1.2.2 double AveragePopulationStatistics::avg_maxF

Definition at line 35 of file stats.hpp.

Referenced by average_population_statistics(), and print_bisection_summary().

4.1.2.3 double AveragePopulationStatistics::avg_minF

Definition at line 34 of file stats.hpp.

Referenced by average_population_statistics().

4.1.2.4 double AveragePopulationStatistics::avg_num_evals

Definition at line 42 of file stats.hpp.

Referenced by average_population_statistics(), and print_bisection_summary().

4.1.2.5 double AveragePopulationStatistics::max_maxF

Definition at line 39 of file stats.hpp.

Referenced by average_population_statistics().

4.1.2.6 double AveragePopulationStatistics::min_minF

Definition at line 38 of file stats.hpp.

Referenced by average_population_statistics().

4.1.2.7 int AveragePopulationStatistics::num_runs

Definition at line 54 of file stats.hpp.

Referenced by average_population_statistics(), and print_bisection_summary().

4.1.2.8 double AveragePopulationStatistics::p_success

Definition at line 45 of file stats.hpp.

4.1.2.9 int AveragePopulationStatistics::population_size

Definition at line 48 of file stats.hpp.

Referenced by average_population_statistics(), bisection(), and print_bisection_summary().

4.1.2.10 int AveragePopulationStatistics::problem_size

Definition at line 51 of file stats.hpp.

Referenced by average_population_statistics().

The documentation for this struct was generated from the following file:

- stats.hpp

4.2 Parameters Struct Reference

Parameters for the decision-tree EDA (can be set using parameter files).

`#include <eda.hpp>`

Data Fields

- int population_size
  Population size.
- int problem_size
  Number of variables (string positions).
- int max_generations
  Maximum number of iterations.
- int tournament_size
  Size of tournaments.
- int replacement
  Replacement method (RTR=0, full=1).
- int bisection
  Optimize population size using bisection (must run many runs)?
- int num_bisection_runs
  Number of successful runs for optimal population sizing with bisection.
- int quiet_mode
  Quiet mode?
- int verbose_mode
  Verbose mode?

4.2.1 Detailed Description

Parameters for the decision-tree EDA (can be set using parameter files).

Definition at line 15 of file eda.hpp.

4.2.2 Field Documentation

4.2.2.1 int Parameters::bisection

Optimize population size using bisection (must run many runs)?

Definition at line 21 of file eda.hpp.

Referenced by main(), one_run(), parse_input_file(), and print_parameters().

4.2.2.2 int Parameters::max_generations

Maximum number of iterations.

Definition at line 18 of file eda.hpp.

Referenced by main(), one_run(), parse_input_file(), and print_parameters().

4.2.2.3 int Parameters::num_bisection_runs

Number of successful runs for optimal population sizing with bisection.

Definition at line 22 of file eda.hpp.

Referenced by do_runs(), main(), parse_input_file(), and print_parameters().

4.2.2.4 int Parameters::population_size

Population size.

Definition at line 16 of file eda.hpp.

Referenced by bisection(), do_runs(), main(), one_run(), parse_input_file(), print_parameters(), and variation().

4.2.2.5 int Parameters::problem_size

Number of variables (string positions).

Definition at line 17 of file eda.hpp.

Referenced by main(), one_run(), parse_input_file(), print_parameters(), and variation().

4.2.2.6 int Parameters::quiet_mode

Quiet mode?

Definition at line 23 of file eda.hpp.

Referenced by bisection(), main(), one_run(), parse_input_file(), and print_parameters().

4.2.2.7 int Parameters::replacement

Replacement method (RTR=0, full=1).

Definition at line 20 of file eda.hpp.

Referenced by main(), one_run(), parse_input_file(), and print_parameters().

4.2.2.8 int Parameters::tournament_size

Size of tournaments.

Definition at line 19 of file eda.hpp.

Referenced by main(), one_run(), parse_input_file(), and print_parameters().

4.2.2.9 int Parameters::verbose_mode

Verbose mode?

Definition at line 24 of file eda.hpp.

Referenced by bisection(), main(), one_run(), parse_input_file(), print_parameters(), and variation().

The documentation for this struct was generated from the following file:

- eda.hpp

4.3 PopulationStatistics Struct Reference

Basic population statistics (single run).

`#include <stats.hpp>`

Data Fields

- double minF
- int idx_minF
- double maxF
- int idx_maxF
- double avgF
- long int num_evals
- int success
- int population_size
- int problem_size

4.3.1 Detailed Description

Basic population statistics (single run).

Definition at line 11 of file stats.hpp.

4.3.2 Field Documentation

4.3.2.1 double PopulationStatistics::avgF

Definition at line 15 of file stats.hpp.

Referenced by average_population_statistics(), compute_population_statistics(), init_population_statistics(), and print_status().

4.3.2.2 int PopulationStatistics::idx_maxF

Definition at line 14 of file stats.hpp.

Referenced by compute_population_statistics(), and init_population_statistics().

4.3.2.3 int PopulationStatistics::idx_minF

Definition at line 13 of file stats.hpp.

Referenced by compute_population_statistics(), and init_population_statistics().

4.3.2.4 double PopulationStatistics::maxF

Definition at line 14 of file stats.hpp.

Referenced by average_population_statistics(), compute_population_statistics(), init_population_statistics(), print_status(), and print_summary().

4.3.2.5 double PopulationStatistics::minF

Definition at line 13 of file stats.hpp.

Referenced by average_population_statistics(), compute_population_statistics(), init_population_statistics(), and print_status().

4.3.2.6 long int PopulationStatistics::num_evals

Definition at line 18 of file stats.hpp.

Referenced by average_population_statistics(), init_population_statistics(), one_run(), and print_summary().

4.3.2.7 int PopulationStatistics::population_size

Definition at line 24 of file stats.hpp.

Referenced by average_population_statistics(), compute_population_statistics(), and print_summary().

4.3.2.8 int PopulationStatistics::problem_size

Definition at line 27 of file stats.hpp.

Referenced by average_population_statistics(), compute_population_statistics(), and print_summary().

4.3.2.9 int PopulationStatistics::success

Definition at line 21 of file stats.hpp.

Referenced by init_population_statistics(), one_run(), and print_summary().

The documentation for this struct was generated from the following file:

- stats.hpp

4.4 Status Class Reference

Text-mode status bar used in verbose mode in learning algorithms.

`#include <status.hpp>`

Public Member Functions

- Status (int n=55)
- void update (double p)
- ~Status ()
- void reset ()

4.4.1 Detailed Description

Text-mode status bar used in verbose mode in learning algorithms.

Definition at line 8 of file status.hpp.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 Status::Status (int n = 55)

Definition at line 10 of file status.cpp.

```
11 {
12   last=-1;
13   this->n=n;
14 }
```

4.4.2.2 Status::~Status ()

Definition at line 16 of file status.cpp.

```
17 {
18 }
```

4.4.3 Member Function Documentation

4.4.3.1 void Status::reset ()

Definition at line 44 of file status.cpp.

Referenced by TreeModel::learnStructure().

```
45 {
46   last=-1;
47 }
```

4.4.3.2 void Status::update (double *p*)

Definition at line 20 of file status.cpp.

Referenced by TreeModel::learnProbabilities(), and TreeModel::learnStructure().

```
21 {
22   int i,k;
23
24   k=(int)((double)p*n);
25
26   if (last!=k)
27     {
28       if (last>=0)
29         for (i=0; i<n+2; i++)
30           printf("\b");
31
32       printf("[");
33       for (i=0; i<k; i++)
34         printf("=");
35       for (i=n-k; i>0; i--)
36         printf("-");
37       printf("]");
38       fflush(stdout);
39
40       last=k;
41     };
42 }
```

The documentation for this class was generated from the following files:

- status.hpp
- status.cpp

4.5 TreeModel Class Reference

Class for storing, creating, and sampling the tree models.

`#include <tree-model.hpp>`

Public Member Functions

- TreeModel (int n)
  Constructor (assumes binary representation).
- TreeModel (int n, int *num_vals)
  Another constructor (assumes arbitrary finite alphabets).
- ~TreeModel ()
  Destructor.
- void learnStructure (int **x, int N, int loud=1)
  Learn the structure of the model.
- void learnProbabilities (int **x, int N, int loud=1)
  Learn the probabilities for a given structure.
- void sampleModel (int **x, int N)
  Sample the tree model to generate new candidate solutions.
- void printModel (FILE *f)
  Print the model into a stream.

4.5.1 Detailed Description

Class for storing, creating, and sampling the tree models.

Definition at line 13 of file tree-model.hpp.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 TreeModel::TreeModel (int *n*)

Constructor (assumes binary representation).

Definition at line 20 of file tree-model.cpp.

```
21 {
22   this->n=n;
23
24   // assume all variables are binary
25
26   num_vals=new int[n];
27   for (int i=0; i<n; i++)
28     num_vals[i]=2;
29
```

```
30   // initialize some variables
31
32   max_num_vals=2;
33
34   mi_num = NULL;
35   mi_p = NULL;
36   mi_pi = NULL;
37   mi_pj = NULL;
38   succ = NULL;
39   pred = NULL;
40   num_succ = NULL;
41   anc_order = NULL;
42   prob = NULL;
43   num_prob = NULL;
44 }
```

4.5.2.2 TreeModel::TreeModel (int *n*, int * *num_vals*)

Another constructor (assumes arbitrary finite alphabets).

Definition at line 50 of file tree-model.cpp.

```
51 {
52   this->n=n;
53
54   // store range of all variables
55
56   this->num_vals=new int[n];
57   max_num_vals=0;
58   for (int i=0; i<n; i++)
59     {
60       this->num_vals[i]=num_vals[i];
61       if (this->num_vals[i]>max_num_vals)
62         max_num_vals=this->num_vals[i];
63     }
64
65   // initialize some variables
66
67   mi_num = NULL;
68   mi_p = NULL;
69   mi_pi = NULL;
70   mi_pj = NULL;
71   succ = NULL;
72   pred = NULL;
73   num_succ = NULL;
74   anc_order = NULL;
75   prob = NULL;
76 }
```

4.5.2.3 TreeModel::~TreeModel ()

Destructor.

Definition at line 82 of file tree-model.cpp.

```
83 {
84   // free memory used by the range of all variables
85
86   delete[] num_vals;
87
88   if (mi_num)
```

```
89     delete[] mi_num;
90   if (mi_p)
91     delete[] mi_p;
92   if (mi_pi)
93     delete[] mi_pi;
94   if (mi_pj)
95     delete[] mi_pj;
96   if (anc_order)
97     delete[] anc_order;
98   if (prob)
99     {
100       for (int i=0; i<n; i++)
101         delete[] prob[i];
102       delete[] prob;
103       delete[] num_prob;
104     };
105
106   if (succ)
107     {
108        for (int i=0; i<n; i++)
109          delete[] succ[i];
110       delete[] succ;
111       delete[] num_succ;
112     }
113
114   if (pred)
115     delete[] pred;
116 }
```

4.5.3 Member Function Documentation

4.5.3.1 void TreeModel::learnProbabilities (int ** *x*, int *N*, int *loud* = 1)

Learn the probabilities for a given structure.

Definition at line 441 of file tree-model.cpp.

References Status::update().

Referenced by variation().

```
442 {
443   int i,j;
444   Status *status=new Status();
445
446   if (loud)
447     printf("   learning model parameters\n");
448
449   if (prob==NULL)
450     {
451       prob=new double*[n];
452       num_prob=new int[n];
453     }
454   else
455     for (i=0; i<n; i++)
456       if (prob[i])
457         delete[] prob[i];
458
459   if (loud)
460     printf("      phase 1/1: ");
461
462   for (i=0; i<n; i++)
463     {
464       if (loud)
465         status->update(((double)i)/n);
```

```
466
467       if (pred[i]<0)
468         {
469           int k=num_vals[i];
470           prob[i]=new double[k];
471           num_prob[i]=k;
472           for (j=0; j<k; j++)
473             prob[i][j]=0;
474
475           for (j=0; j<N; j++)
476             prob[i][x[j][i]]+=1;
477           for (j=0; j<k; j++)
478             prob[i][j]=prob[i][j]/N;
479         }
480       else
481         {
482           int k=num_vals[i]*num_vals[pred[i]];
483           prob[i]=new double[k];
484           num_prob[i]=k;
485           for (j=0; j<k; j++)
486             prob[i][j]=0;
487
488           int *total=new int[num_vals[pred[i]]];
489           for (j=0; j<num_vals[pred[i]]; j++)
490             total[j]=0;
491
492           for (j=0; j<N; j++)
493             {
494               int idx=num_vals[pred[i]]*x[j][i]+x[j][pred[i]];
495               prob[i][idx]+=1;
496               total[x[j][pred[i]]]++;
497             };
498
499           for (j=0; j<k; j++)
500             {
501               int pred_value=j%num_vals[pred[i]];
502               prob[i][j]=prob[i][j]/total[pred_value];
503             };
504
505           delete[] total;
506         }
507     }
508
509   if (loud)
510     {
511       status->update(1);
512       printf("\n");
513     };
514
515   delete status;
516 }
```

4.5.3.2 void TreeModel::learnStructure (int ** *x*, int *N*, int *loud* = 1)

Learn the structure of the model.

Definition at line 200 of file tree-model.cpp.

References build_max_heap(), HEAP_MINUS_INFINITY, increase_key_max_heap(), intRand(), pop_max_heap(), Status::reset(), and Status::update().

Referenced by variation().

```
201 {
202   int i,j,k;
```

```
203
204   // initialize the complete graph with I(i,j) weight on each edge (i,j)
205
206   int num_edges=n*(n-1)/2;
207
208   int *a=new int[num_edges];
209   int *b=new int[num_edges];
210   int *index=new int[num_edges];
211   int *rev_index=new int[num_edges];
212   int *selected=new int[n];
213
214   Status *status = new Status();
215
216   if (pred==NULL)
217     pred=new int[n];
218
219   double *weights = new double[num_edges];
220   double *I = new double[num_edges];
221
222   if (num_succ==NULL)
223     num_succ=new int[n];
224
225   int num_selected=0;
226   int last_selected=-1;
227   int last_predecessor=-1;
228   int heap_size=0;
229
230   if (loud)
231     printf("   learning model structure\n");
232
233   // initialize variables
234
235   for (i=0; i<n; i++)
236     {
237       selected[i]=0;
238       num_succ[i]=0;
239     }
240
241   // create the tree
242
243   if (loud)
244     {
245       printf("      phase 1/2: ");
246       status->reset();
247       status->update(0);
248     };
249
250   for (i=1,k=0; i<n; i++)
251     {
252       for (j=0; j<i; j++,k++)
253         {
254           a[k]=j;
255           b[k]=i;
256           weights[k]=HEAP_MINUS_INFINITY;
257           I[k]=mutualInformation(j,i,x,N);
258           index[k]=k;
259           rev_index[k]=k;
260         };
261
262       if (loud)
263         status->update(((double)1.0*k)/num_edges);
264     };
265
266   if (loud)
267     printf("\n");
268
269   // build the initial heap
```

```
270
271   build_max_heap(a,b,weights,num_edges,index,rev_index);
272   heap_size=num_edges;
273
274   // a random node is first selected (last_selected stores its index)
275
276   last_selected = intRand(n);
277   last_predecessor = -1;
278   pred[last_selected]=-1;
279   root = last_selected;
280
281   // initialize the status bar
282
283   if (loud)
284     {
285       printf("      phase 2/2: ");
286       status->reset();
287       status->update(0);
288     };
289
290   // we continue until all nodes have been selected
291
292   while (num_selected<n)
293     {
294       selected[last_selected]=1;
295       pred[last_selected]=last_predecessor;
296       num_selected++;
297       if (loud)
298         status->update(((double) 1.0*num_selected)/n);
299
300       if (last_predecessor>=0)
301         num_succ[last_predecessor]++;
302
303       // relax neighbors and find a new node to select (if not done)
304
305       if (num_selected<n)
306         {
307
308           for (i=0; i<last_selected; i++)
309             {
310               int this_index=last_selected*(last_selected-1)/2+i;
311               if (I[this_index]>weights[this_index])
312                 increase_key_max_heap(weights,this_index,I[this_index],heap_size,index,rev_index);
313             };
314
315           for (i=last_selected+1; i<n; i++)
316             {
317               int this_index=i*(i-1)/2+last_selected;
318               if (I[this_index]>weights[this_index])
319                 increase_key_max_heap(weights,this_index,I[this_index],heap_size,index,rev_index);
320             };
321
322           do {
323             i=pop_max_heap(weights,heap_size,index,rev_index);
324           } while ((selected[a[i]])&&(selected[b[i]]));
325
326           if (!selected[a[i]])
327             {
328               last_selected=a[i];
329               last_predecessor=b[i];
330             }
331           else
332             {
333               last_selected=b[i];
334               last_predecessor=a[i];
335             };
336         };
```

```
337     };
338   if (loud)
339     printf("\n");
340
341   // create the actual graph
342
343   if (succ==NULL)
344     succ=new int*[n];
345   else
346     for (i=0; i<n; i++)
347       if (succ[i])
348         delete[] succ[i];
349
350   for (int i=0; i<n; i++)
351     if (num_succ[i]>0)
352       {
353         succ[i]=new int[num_succ[i]];
354         num_succ[i]=0;
355       }
356     else
357       succ[i]=NULL;
358
359   for (int i=0; i<n; i++)
360     if (pred[i]>=0)
361       {
362         int p=pred[i];
363
364         succ[p][num_succ[p]++]=i;
365       };
366
367   // compute the ancestral ordering
368
369   if (anc_order==NULL)
370     anc_order=new int[n];
371
372   int count=0;
373   anc_order[count]=root;
374   ancestralOrderingRec(count);
375
376   // free memory
377
378   delete[] a;
379   delete[] b;
380
381   delete[] weights;
382   delete[] selected;
383   delete[] index;
384   delete[] rev_index;
385
386   delete[] I;
387   delete status;
388 };
```

4.5.3.3 void TreeModel::printModel (FILE * *f*)

Print the model into a stream.

Definition at line 394 of file tree-model.cpp.

```
395 {
396   printf("\n-------------------------------------------------------------------------------\n");
397   printf("Structure:\n");
398   for (int i=0; i<n; i++)
399     {
400       printf("   %u",i);
```

```
401       if (num_succ[i]>0)
402         {
403           printf(" ->");
404           for (int j=0; j<num_succ[i]; j++)
405             printf(" %u",succ[i][j]);
406         };
407       printf("\n");
408     };
409
410   printf("Probabilities:\n");
411   for (int i=0; i<n; i++)
412     {
413       printf("   %2u: ",i);
414       for (int j=0; j<num_prob[i]; j++)
415         printf("%4.2f ",prob[i][j]);
416       printf("\n");
417     };
418   printf("\n-------------------------------------------------------------------------------\n");
419 }
```

4.5.3.4 void TreeModel::sampleModel (int ** *x*, int *N*)

Sample the tree model to generate new candidate solutions.

Definition at line 522 of file tree-model.cpp.

Referenced by variation().

```
523 {
524   int i,j;
525
526   for (i=0; i<N; i++)
527     {
528       int idx=anc_order[0];
529       x[i][idx]=generate_marginal(num_vals[idx],prob[idx]);
530       for (j=1; j<n; j++)
531       {
532         int idx=anc_order[j];
533
534         // printf("can use max i=%u\n",num_prob[idx]);
535
536         x[i][idx]=generate_conditional(num_vals[idx],num_vals[pred[idx]],prob[idx],x[i][pred[idx]]);
537       }
538     }
539 }
```

The documentation for this class was generated from the following files:

- tree-model.hpp
- tree-model.cpp

Chapter 5

Dependency-Tree Estimation of Distribution Algorithm File Documentation

5.1 bisection.cpp File Reference

Bisection method that computes near optimal population size.

```
#include "bisection.hpp"
#include "eda.hpp"
#include "stats.hpp"
```

Functions

- void bisection (Parameters *params, AveragePopulationStatistics *avg_stats)

  Bisection method for determining optimal population size.

- int do_runs (Parameters *params, AveragePopulationStatistics *avg_stats, int quiet)

  Run a number of successful runs, terminate on a failure.

- void print_bisection_summary (FILE *f, AveragePopulationStatistics *stats)

  Print summary of a set of runs (represented by average statistics).

5.1.1 Detailed Description

Bisection method that computes near optimal population size.

Definition in file bisection.cpp.

5.1.2 Function Documentation

5.1.2.1 void bisection (Parameters * *params*, AveragePopulationStatistics * *avg_stats*)

Bisection method for determining optimal population size.

Examples:

example_input, example_input_big, and example_input_bisection.

Definition at line 15 of file bisection.cpp.

References do_runs(), Parameters::population_size, AveragePopulationStatistics::population_size, Parameters::quiet_mode, separator(), and Parameters::verbose_mode.

Referenced by main().

```
16 {
17   AveragePopulationStatistics current_avg_stats;
18   Parameters *current_params = new Parameters;
19   int result;
20   int quiet=params->quiet_mode;
21
22   params->quiet_mode=1;
23   params->verbose_mode=0;
24
25   // make the first set of runs
26
27   *current_params = *params;
28   result=do_runs(current_params,&current_avg_stats,quiet);
29
30   int minN=0;
31   int maxN=0;
32
33   if (result)
34     {
35       // if successful, try to halve population size until failure
36
37       *avg_stats=current_avg_stats;
38
39       while (result)
40         {
41           current_params->population_size/=2;
42           result=do_runs(current_params,&current_avg_stats,quiet);
43
44           if (result)
45             *avg_stats=current_avg_stats;
46         };
47
48       // set initial population size bounds
49
50       minN=current_params->population_size;
51       maxN=minN*2;
52     }
53   else
54     {
55       // if failure, double population size until success
56
57       while (!result)
58         {
59           current_params->population_size*=2;
60           result=do_runs(current_params,&current_avg_stats,quiet);
61
```

```
62           if (result)
63             *avg_stats=current_avg_stats;
64         };
65
66       // set initial population size bounds
67
68       maxN=current_params->population_size;
69       minN=maxN/2;
70     }
71
72   // perform bisection until interval width is at most 5% of the
73   // lower bound
74
75   while ((((double)maxN-minN)/minN)>0.05)
76     {
77       // try to make a set of runs in the middle of the interval
78
79       current_params->population_size=(minN+maxN)/2;
80       result=do_runs(current_params,&current_avg_stats,quiet);
81
82       if (result)
83         *avg_stats=current_avg_stats;
84
85       // success goes halves the interval downwards, failure upwards
86
87       if (result)
88         maxN=current_params->population_size;
89       else
90         minN=current_params->population_size;
91     }
92
93   if (quiet==0)
94     {
95       printf("Finished bisection with N=%u\n",avg_stats->population_size);
96       separator(stdout);
97     };
98
99   delete current_params;
100 }
```

5.1.2.2 int do_runs (Parameters * *params*, AveragePopulationStatistics * *avg_stats*, int *quiet*)

Run a number of successful runs, terminate on a failure.

Definition at line 107 of file bisection.cpp.

References average_population_statistics(), Parameters::num_bisection_runs, one_run(), and Parameters::population_size.

Referenced by bisection().

```
108 {
109   PopulationStatistics *stats = new PopulationStatistics[params->num_bisection_runs];
110   int run;
111   int failed=0;
112
113   for (run=0; (run<params->num_bisection_runs)&&(!failed); run++)
114     {
115       int result=one_run(params,&stats[run]);
116       failed=!result;
117     };
118
119   if (quiet==0)
```

```
120     printf("bisection -> %2u / %2u successes with N=%u\n",
121            run,params->num_bisection_runs,params->population_size);
122
123   average_population_statistics(avg_stats,stats,params->num_bisection_runs);
124
125   delete[] stats;
126
127   if (failed)
128     return 0;
129   else
130     return 1;
131 }
```

5.1.2.3 void print_bisection_summary (FILE * *f*, AveragePopulationStatistics * *stats*)

Print summary of a set of runs (represented by average statistics).

Definition at line 138 of file bisection.cpp.

References AveragePopulationStatistics::avg_maxF, AveragePopulationStatistics::avg_num_evals, AveragePopulationStatistics::num_runs, and AveragePopulationStatistics::population_size.

Referenced by main().

```
139 {
140   fprintf(f,"Bisection summary output:\n");
141   fprintf(f,"   num_runs        = %u\n",stats->num_runs);
142   fprintf(f,"   avg_best_found  = %f\n",stats->avg_maxF);
143   fprintf(f,"   avg_num_evals   = %f\n",stats->avg_num_evals);
144   fprintf(f,"   pop_size        = %u\n",stats->population_size);
145 }
```

5.2 bisection.hpp File Reference

Header file for bisection.cpp.

```
#include "eda.hpp"
```

Functions

- void bisection (Parameters *params, AveragePopulationStatistics *avg_stats)
  Bisection method for determining optimal population size.
- int do_runs (Parameters *params, AveragePopulationStatistics *avg_stats, int quiet=1)
  Run a number of successful runs, terminate on a failure.
- void print_bisection_summary (FILE *f, AveragePopulationStatistics *stats)
  Print summary of a set of runs (represented by average statistics).

5.2.1 Detailed Description

Header file for bisection.cpp.

Definition in file bisection.hpp.

5.2.2 Function Documentation

5.2.2.1 void bisection (Parameters * *params*, AveragePopulationStatistics * *avg_stats*)

Bisection method for determining optimal population size.

Definition at line 15 of file bisection.cpp.

References do_runs(), AveragePopulationStatistics::population_size, Parameters::population_size, Parameters::quiet_mode, separator(), and Parameters::verbose_mode.

Referenced by main().

```
16 {
17   AveragePopulationStatistics current_avg_stats;
18   Parameters *current_params = new Parameters;
19   int result;
20   int quiet=params->quiet_mode;
21
22   params->quiet_mode=1;
23   params->verbose_mode=0;
24
25   // make the first set of runs
26
27   *current_params = *params;
28   result=do_runs(current_params,&current_avg_stats,quiet);
29
30   int minN=0;
31   int maxN=0;
32
33   if (result)
34     {
```

```
35      // if successful, try to halve population size until failure
36
37      *avg_stats=current_avg_stats;
38
39      while (result)
40        {
41          current_params->population_size/=2;
42          result=do_runs(current_params,&current_avg_stats,quiet);
43
44          if (result)
45            *avg_stats=current_avg_stats;
46        };
47
48      // set initial population size bounds
49
50      minN=current_params->population_size;
51      maxN=minN*2;
52    }
53  else
54    {
55      // if failure, double population size until success
56
57      while (!result)
58        {
59          current_params->population_size*=2;
60          result=do_runs(current_params,&current_avg_stats,quiet);
61
62          if (result)
63            *avg_stats=current_avg_stats;
64        };
65
66      // set initial population size bounds
67
68      maxN=current_params->population_size;
69      minN=maxN/2;
70    }
71
72  // perform bisection until interval width is at most 5% of the
73  // lower bound
74
75  while ((((double)maxN-minN)/minN)>0.05)
76    {
77      // try to make a set of runs in the middle of the interval
78
79      current_params->population_size=(minN+maxN)/2;
80      result=do_runs(current_params,&current_avg_stats,quiet);
81
82      if (result)
83        *avg_stats=current_avg_stats;
84
85      // success goes halves the interval downwards, failure upwards
86
87      if (result)
88        maxN=current_params->population_size;
89      else
90        minN=current_params->population_size;
91    }
92
93  if (quiet==0)
94    {
95      printf("Finished bisection with N=%u\n",avg_stats->population_size);
96      separator(stdout);
97    };
98
99  delete current_params;
100 }
```

5.2.2.2 int do_runs (Parameters * *params*, AveragePopulationStatistics * *avg_stats*, int *quiet* = 1)

Run a number of successful runs, terminate on a failure.

Definition at line 107 of file bisection.cpp.

References average_population_statistics(), Parameters::num_bisection_runs, one_run(), and Parameters::population_size.

Referenced by bisection().

```
108 {
109   PopulationStatistics *stats = new PopulationStatistics[params->num_bisection_runs];
110   int run;
111   int failed=0;
112
113   for (run=0; (run<params->num_bisection_runs)&&(!failed); run++)
114     {
115       int result=one_run(params,&stats[run]);
116       failed=!result;
117     };
118
119   if (quiet==0)
120     printf("bisection -> %2u / %2u successes with N=%u\n",
121            run,params->num_bisection_runs,params->population_size);
122
123   average_population_statistics(avg_stats,stats,params->num_bisection_runs);
124
125   delete[] stats;
126
127   if (failed)
128     return 0;
129   else
130     return 1;
131 }
```

5.2.2.3 void print_bisection_summary (FILE * *f*, AveragePopulationStatistics * *stats*)

Print summary of a set of runs (represented by average statistics).

Definition at line 138 of file bisection.cpp.

References AveragePopulationStatistics::avg_maxF, AveragePopulationStatistics::avg_num_evals, AveragePopulationStatistics::num_runs, and AveragePopulationStatistics::population_size.

Referenced by main().

```
139 {
140   fprintf(f,"Bisection summary output:\n");
141   fprintf(f,"   num_runs        = %u\n",stats->num_runs);
142   fprintf(f,"   avg_best_found  = %f\n",stats->avg_maxF);
143   fprintf(f,"   avg_num_evals   = %f\n",stats->avg_num_evals);
144   fprintf(f,"   pop_size        = %u\n",stats->population_size);
145 }
```

5.3 eda.cpp File Reference

EDA-specific functions (except for the model-related ones).

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "eda.hpp"
#include "random.hpp"
#include "obj-function.hpp"
#include "stats.hpp"
#include "tree-model.hpp"
```

Typedefs

- typedef int ReplacementMethod (int **x, int **y, int n, int N, double *fx, double *fy)

Functions

- int generate_population (int **x, int n, int N, int *num_vals=NULL)
  Generate a random population of individuals (uniform distribution).
- int generate_BB_population (int **x, int n, int N, int k)
  Generate a poulation full of blocks of 0s and 1s. Only for debugging.
- int ** allocate_population (int n, int N)
  Allocate memory for a new population of specified parameters.
- void free_population (int **x, int n, int N)
  Free memory occupied by a population of specified parameters.
- int evaluate_population (int **x, double *f, int n, int N, long &num_evals)
  Evaluate a population using the user-specified objective function.
- int tournament_selection (int **y, int **x, double *f, int n, int N, int k)
  Tournament selection with replacement (k-ary tournaments).
- void print_population (FILE *f, int **x, int n, int N)
  Print the population to the specified file.
- int restricted_tournament_replacement (int **x, int **y, int n, int N, double *fx, double *fy)
  Restricted tournament replacement (Harik, 1995) for niching.
- int full_replacement (int **x, int **y, int n, int N, double *fx, double *fy)
  Replace the entire old population with the new population.

- int individual_distance (int *x, int *y, int n)

  Compute a distance between two individuals (number of non-matching chars).

- int one_run (Parameters *params, PopulationStatistics *stats)

  Execute one run of a decision-tree EDA with the specified parameters.

- int variation (int **sampled_population, int **selected_population, Parameters *params, int *num_vals)

  Variation operator of the decision-tree EDA (learns and samples model).

- void print_status (int t, PopulationStatistics *stats)

  Print the status of the algorithm.

- void separator (FILE *f, int type)

  Print a sequence of dashes to separate text output.

- void print_summary (PopulationStatistics *stats)

  Print summary of a run.

5.3.1 Detailed Description

EDA-specific functions (except for the model-related ones).

Definition in file eda.cpp.

5.3.2 Typedef Documentation

5.3.2.1 typedef int ReplacementMethod(int **x, int **y, int n, int N, double *fx, double *fy)

Definition at line 19 of file eda.cpp.

5.3.3 Function Documentation

5.3.3.1 int** allocate_population (int *n*, int *N*)

Allocate memory for a new population of specified parameters.

Definition at line 84 of file eda.cpp.

Referenced by one_run().

```
85 {
86   int i;
87   int **x;
88 
89   // allocate a population as a 2-dimensional int-array
90 
91   x=new int*[N];
92   for (i=0; i<N; i++)
93     x[i]=new int[n];
94 
```

```
95   // return the allocated array
96
97   return x;
98 }
```

5.3.3.2 int evaluate_population (int ** *x*, double * *f*, int *n*, int *N*, long & *num_evals*)

Evaluate a population using the user-specified objective function.

Definition at line 122 of file eda.cpp.

References objective_function().

Referenced by one_run().

```
123 {
124   // evaluate all individuals, one by one
125
126   for (int i=0; i<N; i++)
127     f[i]=objective_function(x[i],n);
128
129   // increase the number of evaluations
130
131   num_evals+=N;
132
133   // return the number of evaluated individuals
134
135   return N;
136 }
```

5.3.3.3 void free_population (int ** *x*, int *n*, int *N*)

Free memory occupied by a population of specified parameters.

Definition at line 105 of file eda.cpp.

Referenced by one_run().

```
106 {
107   int i;
108
109   // free memory used by the population
110
111   for (i=0; i<N; i++)
112     delete[] x[i];
113
114   delete[] x;
115 }
```

5.3.3.4 int full_replacement (int ** *x*, int ** *y*, int *n*, int *N*, double * *fx*, double * *fy*)

Replace the entire old population with the new population.

Definition at line 243 of file eda.cpp.

Referenced by one_run().

```
244 {
245   int i;
246
247   // just replace entire old population with the new guys
248
249   for (i=0; i<N; i++)
250     {
251       memcpy(x[i],y[i],sizeof(x[i][0])*n);
252       fx[i]=fy[i];
253     }
254
255   // return the number of processed new individuals
256
257   return N;
258 }
```

5.3.3.5 int generate_BB_population (int ** x, int n, int N, int k)

Generate a poulation full of blocks of 0s and 1s. Only for debugging.

Definition at line 57 of file eda.cpp.

References drand().

```
58 {
59   int i,j,l;
60
61   // generate a population full of 000..0 and 111..1 building blocks
62   // (used for testing the model building procedure, unnecessary for
63   // practical applications)
64
65   for (i=0; i<N; i++)
66     for (j=0; j<n;)
67       {
68         int val=(drand()<0.5)? 0:1;
69
70         for (l=0; l<k; l++)
71           x[i][j++]=val;
72     };
73
74   // return the number of generated individuals
75
76   return N;
77 }
```

5.3.3.6 int generate_population (int ** x, int n, int N, int * num_vals = NULL)

Generate a random population of individuals (uniform distribution).

Definition at line 31 of file eda.cpp.

References drand(), and intRand().

Referenced by one_run().

```
32 {
33   int i,j;
34
35   // general all variables uniformly randomly
36   // (if the numbers of values is not supplied, assumes binary)
37
38   if (num_vals==NULL)
```

```
39     for (i=0; i<n; i++)
40       for (j=0; j<N; j++)
41         x[j][i]=(drand()<0.5)? 0:1;
42   else
43     for (i=0; i<n; i++)
44       for (j=0; j<N; j++)
45         x[j][i]=intRand(num_vals[i]);
46
47   // return the number of generated individuals
48
49   return N;
50 }
```

5.3.3.7 int individual_distance (int * *x*, int * *y*, int *n*)

Compute a distance between two individuals (number of non-matching chars).

Definition at line 265 of file eda.cpp.

Referenced by restricted_tournament_replacement().

```
266 {
267   int i;
268   int d=0;
269
270   for (i=0; i<n; i++)
271     if (x[i]!=y[i])
272       d++;
273
274   return d;
275 }
```

5.3.3.8 int one_run (Parameters * *params*, PopulationStatistics * *stats*)

Execute one run of a decision-tree EDA with the specified parameters.

Definition at line 282 of file eda.cpp.

References allocate_population(), Parameters::bisection, compute_population_statistics(), evaluate_population(), free_population(), full_replacement(), generate_population(), init_population_statistics(), Parameters::max_generations, N, PopulationStatistics::num_evals, Parameters::population_size, print_status(), Parameters::problem_size, Parameters::quiet_mode, Parameters::replacement, restricted_tournament_replacement(), set_num_vals(), PopulationStatistics::success, tournament_selection(), Parameters::tournament_size, variation(), and Parameters::verbose_mode.

Referenced by do_runs(), and main().

```
283 {
284   int N=params->population_size;
285   int n=params->problem_size;
286   int max_generations=params->max_generations;
287
288   int **current_population;
289   int **selected_population;
290   int **sampled_population;
291
292   double *current_f;
293   double *sampled_f;
294
```

```
295   int *num_vals = new int[n];
296
297   // allocate populations and necessary fitness arrays
298
299   current_population = allocate_population(n,N);
300   selected_population = allocate_population(n,N);
301   sampled_population = allocate_population(n,N);
302
303   current_f = new double[N];
304   sampled_f = new double[N];
305
306   // initialize the replacement method
307
308   ReplacementMethod *replacement;
309   if (params->replacement==0)
310     replacement=restricted_tournament_replacement;
311   else
312     if (params->replacement==1)
313       replacement=full_replacement;
314     else
315       {
316         printf("ERROR: Unknown replacement method (%u)\n",params->replacement);
317         exit(-2);
318       };
319
320   // initialize the number of values
321
322   set_num_vals(num_vals,n);
323
324   // generate initial population
325
326   generate_population(current_population,n,N,num_vals);
327
328   // evaluate initial population
329
330   evaluate_population(current_population,current_f,n,N,stats->num_evals);
331
332   // initialize some main-loop variables
333
334   int done=0;
335   int t=0;
336
337   // init the statistics and print the initial status information
338
339   init_population_statistics(stats);
340   compute_population_statistics(current_population,current_f,n,N,num_vals,stats);
341
342   if (params->quiet_mode==0)
343     print_status(t,stats);
344   else
345     params->verbose_mode=0;
346
347   // main loop
348
349   while (!done)
350     {
351       // increment generation counter
352
353       t++;
354
355       // selection
356
357       tournament_selection(selected_population,
358                            current_population,
359                            current_f,
360                            n,
361                            N,
```

```
362                           params->tournament_size);
363
364       // variation
365
366       variation(sampled_population,selected_population,params,num_vals);
367
368       // evaluation of new candidates
369
370       evaluate_population(sampled_population,sampled_f,n,N,stats->num_evals);
371
372       // replacement
373
374       replacement(current_population,
375                   sampled_population,
376                   n,
377                   N,
378                   current_f,
379                   sampled_f);
380
381       // statistics
382
383       compute_population_statistics(current_population,current_f,n,N,num_vals,stats);
384
385       // print current status
386
387       if ((params->quiet_mode==0)&&(params->bisection==0))
388         print_status(t,stats);
389
390       // should terminate?
391
392       if ((t>max_generations)||(stats->success==1))
393         done=1;
394     }
395
396   // free memory
397
398   delete[] num_vals;
399   delete[] sampled_f;
400   delete[] current_f;
401
402   free_population(current_population,n,N);
403   free_population(selected_population,n,N);
404   free_population(sampled_population,n,N);
405
406   // return with success/failure (0/1)
407
408   return stats->success;
409 }
```

5.3.3.9 void print_population (FILE * *f*, int ** *x*, int *n*, int *N*)

Print the population to the specified file.

Definition at line 175 of file eda.cpp.

References objective_function().

```
176 {
177   int i,j;
178
179   for (i=0; i<N; i++)
180     {
181       for (j=0; j<n; j++)
182         fprintf(f,"%u",x[i][j]);
183       fprintf(f,"   %f\n",objective_function(x[i],n));
```

```
184     };
185 }
```

5.3.3.10 void print_status (int *t*, PopulationStatistics * *stats*)

Print the status of the algorithm.

Definition at line 451 of file eda.cpp.

References PopulationStatistics::avgF, PopulationStatistics::maxF, PopulationStatistics::minF, and separator().

Referenced by one_run().

```
452 {
453   printf("Generation: %u\n",t);
454   printf("   min:  %f\n",stats->minF);
455   printf("   max:  %f\n",stats->maxF);
456   printf("   mean: %f\n",stats->avgF);
457   separator(stdout);
458 }
```

5.3.3.11 void print_summary (PopulationStatistics * *stats*)

Print summary of a run.

Definition at line 478 of file eda.cpp.

References PopulationStatistics::maxF, PopulationStatistics::num_evals, PopulationStatistics::population_size, PopulationStatistics::problem_size, and PopulationStatistics::success.

Referenced by main().

```
479 {
480   printf("Summary:\n");
481   printf("   status       = %s\n",(stats->success)? "success":"failure");
482   printf("   best_found   = %f\n",stats->maxF);
483   printf("   num_evals    = %lu\n",stats->num_evals);
484   printf("   pop_size     = %u\n",stats->population_size);
485   printf("   problem_size = %u\n",stats->problem_size);
486 }
```

5.3.3.12 int restricted_tournament_replacement (int ** *x*, int ** *y*, int *n*, int *N*, double * *fx*, double * *fy*)

Restricted tournament replacement (Harik, 1995) for niching.

Definition at line 192 of file eda.cpp.

References individual_distance(), and intRand().

Referenced by one_run().

```
193 {
194   int i,j;
195
196   // use default value for window size
197
```

```
198   int windowSize=(n<N/20)? n:(N/20);
199
200   // for every individual, do the same
201
202   for (i=0; i<N; i++)
203     {
204       // select a random subset from the original population (window) and
205       // find the most similar guy to the new individual in this window
206       // (string Hamming distance)
207
208       int pick=intRand(N);
209       int dist=individual_distance(x[i],y[pick],n);
210
211       for (j=1; j<windowSize; j++)
212         {
213           int pick2=intRand(N);
214           int dist2=individual_distance(x[i],y[pick2],n);
215
216           if (dist2<dist)
217             {
218               pick=pick2;
219               dist=dist2;
220             }
221         }
222
223       // if the most similar guy from the window is better than the new guy,
224       // the new guy replaces it
225
226       if (fx[i]<fy[pick])
227         {
228           memcpy(x[i],y[pick],sizeof(x[i][0])*n);
229           fx[i]=fy[pick];
230         }
231     }
232
233   // return the number of processed new individuals
234
235   return N;
236 }
```

5.3.3.13 void separator (FILE * *f*, int *type*)

Print a sequence of dashes to separate text output.

Definition at line 465 of file eda.cpp.

Referenced by bisection(), print_parameters(), print_status(), and variation().

```
466 {
467   if (type==0)
468     fprintf(f,"- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");
469   else
470     fprintf(f,"-------------------------------------------------------------------------------\n");
471 }
```

5.3.3.14 int tournament_selection (int ** *y*, int ** *x*, double * *f*, int *n*, int *N*, int *k*)

Tournament selection with replacement (k-ary tournaments).

Definition at line 143 of file eda.cpp.

References intRand().

Referenced by one_run().

```
144 {
145   int i,j;
146
147   for (i=0; i<N; i++)
148     {
149       // select a winner of k tournaments (with replacement)
150
151       int winner=intRand(N);
152       for (j=1; j<k; j++)
153         {
154           int l=intRand(N);
155           if (f[l]>f[winner])
156             winner=l;
157         };
158
159       // the winner takes the next spot in the selected population
160
161       for (int ii=0; ii<n; ii++)
162         y[i][ii]=x[winner][ii];
163     };
164
165   // return the number of selected individuals
166
167   return N;
168 }
```

5.3.3.15 int variation (int ** *sampled_population*, int ** *selected_population*, Parameters * *params*, int * *num_vals*)

Variation operator of the decision-tree EDA (learns and samples model).

Definition at line 416 of file eda.cpp.

References TreeModel::learnProbabilities(), TreeModel::learnStructure(), N, Parameters::population_size, Parameters::problem_size, TreeModel::sampleModel(), separator(), and Parameters::verbose_mode.

Referenced by one_run().

```
417 {
418   int N=params->population_size;
419   int n=params->problem_size;
420   int loud=params->verbose_mode;
421
422   if (loud)
423     separator(stdout,0);
424
425   TreeModel *t = new TreeModel(n, num_vals);
426
427   // learn the model structure and model parameters
428
429   t->learnStructure(selected_population,N,loud);
430   t->learnProbabilities(selected_population,N,loud);
431
432   // sample the learned model to generate new candidate solutions
433
434   t->sampleModel(sampled_population,N);
435
436   // free memory
437
438   delete t;
```

```
439
440   // return the number of generated individuals
441
442   return N;
443 }
```

5.4 eda.hpp File Reference

Header file for eda.cpp.

```
#include <stdio.h>
```

```
#include "stats.hpp"
```

Data Structures

- struct Parameters

  Parameters for the decision-tree EDA (can be set using parameter files).

Functions

- int generate_population (int **x, int n, int N)
- int ** allocate_population (int n, int N)

  Allocate memory for a new population of specified parameters.

- void free_population (int **x, int n, int N)

  Free memory occupied by a population of specified parameters.

- int evaluate_population (int **x, double *f, int n, int N)
- int tournament_selection (int **y, int **x, double *f, int n, int N, int k)

  Tournament selection with replacement (k-ary tournaments).

- void print_population (FILE *f, int **x, int n, int N)

  Print the population to the specified file.

- int restricted_tournament_replacement (int **y, int **x, int n, int N, double *fx, double *fy)

  Restricted tournament replacement (Harik, 1995) for niching.

- int full_replacement (int **y, int **x, int n, int N, double *fx, double *fy)

  Replace the entire old population with the new population.

- int individual_distance (int *x, int *y, int n)

  Compute a distance between two individuals (number of non-matching chars).

- int variation (int **sampled_population, int **selected_population, Parameters *params, int *num_vals)

  Variation operator of the decision-tree EDA (learns and samples model).

- void print_status (int t, PopulationStatistics *stats)

  Print the status of the algorithm.

- void print_summary (PopulationStatistics *stats)

  Print summary of a run.

- int one_run (Parameters *params, PopulationStatistics *stats)

  Execute one run of a decision-tree EDA with the specified parameters.

- int generate_BB_population (int **x, int n, int N, int k)

  Generate a poulation full of blocks of 0s and 1s. Only for debugging.

- void separator (FILE *f, int type=1)

  Print a sequence of dashes to separate text output.

5.4.1 Detailed Description

Header file for eda.cpp.

Definition in file eda.hpp.

5.4.2 Function Documentation

5.4.2.1 int** allocate_population (int *n*, int *N*)

Allocate memory for a new population of specified parameters.

Definition at line 84 of file eda.cpp.

Referenced by one_run().

```
85 {
86   int i;
87   int **x;
88 
89   // allocate a population as a 2-dimensional int-array
90 
91   x=new int*[N];
92   for (i=0; i<N; i++)
93     x[i]=new int[n];
94 
95   // return the allocated array
96 
97   return x;
98 }
```

5.4.2.2 int evaluate_population (int ** *x*, double * *f*, int *n*, int *N*)

5.4.2.3 void free_population (int ** *x*, int *n*, int *N*)

Free memory occupied by a population of specified parameters.

Definition at line 105 of file eda.cpp.

Referenced by one_run().

```
106 {
107   int i;
108 
109   // free memory used by the population
110 
```

```
111   for (i=0; i<N; i++)
112     delete[] x[i];
113
114   delete[] x;
115 }
```

5.4.2.4 int full_replacement (int ** *y*, int ** *x*, int *n*, int *N*, double * *fx*, double * *fy*)

Replace the entire old population with the new population.

Definition at line 243 of file eda.cpp.

Referenced by one_run().

```
244 {
245   int i;
246
247   // just replace entire old population with the new guys
248
249   for (i=0; i<N; i++)
250     {
251       memcpy(x[i],y[i],sizeof(x[i][0])*n);
252       fx[i]=fy[i];
253     }
254
255   // return the number of processed new individuals
256
257   return N;
258 }
```

5.4.2.5 int generate_BB_population (int ** *x*, int *n*, int *N*, int *k*)

Generate a poulation full of blocks of 0s and 1s. Only for debugging.

Definition at line 57 of file eda.cpp.

References drand().

```
58 {
59   int i,j,l;
60
61   // generate a population full of 000..0 and 111..1 building blocks
62   // (used for testing the model building procedure, unnecessary for
63   // practical applications)
64
65   for (i=0; i<N; i++)
66     for (j=0; j<n;)
67       {
68         int val=(drand()<0.5)? 0:1;
69
70         for (l=0; l<k; l++)
71           x[i][j++]=val;
72     };
73
74   // return the number of generated individuals
75
76   return N;
77 }
```

5.4.2.6 int generate_population (int ** *x*, int *n*, int *N*)

5.4.2.7 int individual_distance (int * *x*, int * *y*, int *n*)

Compute a distance between two individuals (number of non-matching chars).

Definition at line 265 of file eda.cpp.

Referenced by restricted_tournament_replacement().

```
266 {
267   int i;
268   int d=0;
269
270   for (i=0; i<n; i++)
271     if (x[i]!=y[i])
272       d++;
273
274   return d;
275 }
```

5.4.2.8 int one_run (Parameters * *params*, PopulationStatistics * *stats*)

Execute one run of a decision-tree EDA with the specified parameters.

Definition at line 282 of file eda.cpp.

References allocate_population(), Parameters::bisection, compute_population_statistics(), evaluate_population(), free_population(), full_replacement(), generate_population(), init_population_statistics(), Parameters::max_generations, N, PopulationStatistics::num_evals, Parameters::population_size, print_status(), Parameters::problem_size, Parameters::quiet_mode, Parameters::replacement, restricted_tournament_replacement(), set_num_vals(), PopulationStatistics::success, tournament_selection(), Parameters::tournament_size, variation(), and Parameters::verbose_mode.

Referenced by do_runs(), and main().

```
283 {
284   int N=params->population_size;
285   int n=params->problem_size;
286   int max_generations=params->max_generations;
287
288   int **current_population;
289   int **selected_population;
290   int **sampled_population;
291
292   double *current_f;
293   double *sampled_f;
294
295   int *num_vals = new int[n];
296
297   // allocate populations and necessary fitness arrays
298
299   current_population = allocate_population(n,N);
300   selected_population = allocate_population(n,N);
301   sampled_population = allocate_population(n,N);
302
303   current_f = new double[N];
304   sampled_f = new double[N];
305
306   // initialize the replacement method
307
```

```
308   ReplacementMethod *replacement;
309   if (params->replacement==0)
310     replacement=restricted_tournament_replacement;
311   else
312     if (params->replacement==1)
313       replacement=full_replacement;
314     else
315       {
316         printf("ERROR: Unknown replacement method (%u)\n",params->replacement);
317         exit(-2);
318       };
319
320   // initialize the number of values
321
322   set_num_vals(num_vals,n);
323
324   // generate initial population
325
326   generate_population(current_population,n,N,num_vals);
327
328   // evaluate initial population
329
330   evaluate_population(current_population,current_f,n,N,stats->num_evals);
331
332   // initialize some main-loop variables
333
334   int done=0;
335   int t=0;
336
337   // init the statistics and print the initial status information
338
339   init_population_statistics(stats);
340   compute_population_statistics(current_population,current_f,n,N,num_vals,stats);
341
342   if (params->quiet_mode==0)
343     print_status(t,stats);
344   else
345     params->verbose_mode=0;
346
347   // main loop
348
349   while (!done)
350     {
351       // increment generation counter
352
353       t++;
354
355       // selection
356
357       tournament_selection(selected_population,
358                            current_population,
359                            current_f,
360                            n,
361                            N,
362                            params->tournament_size);
363
364       // variation
365
366       variation(sampled_population,selected_population,params,num_vals);
367
368       // evaluation of new candidates
369
370       evaluate_population(sampled_population,sampled_f,n,N,stats->num_evals);
371
372       // replacement
373
374       replacement(current_population,
```

```
375                   sampled_population,
376                   n,
377                   N,
378                   current_f,
379                   sampled_f);
380 
381       // statistics
382 
383       compute_population_statistics(current_population,current_f,n,N,num_vals,stats);
384 
385       // print current status
386 
387       if ((params->quiet_mode==0)&&(params->bisection==0))
388         print_status(t,stats);
389 
390       // should terminate?
391 
392       if ((t>max_generations)||(stats->success==1))
393         done=1;
394     }
395 
396   // free memory
397 
398   delete[] num_vals;
399   delete[] sampled_f;
400   delete[] current_f;
401 
402   free_population(current_population,n,N);
403   free_population(selected_population,n,N);
404   free_population(sampled_population,n,N);
405 
406   // return with success/failure (0/1)
407 
408   return stats->success;
409 }
```

5.4.2.9 void print_population (FILE * *f*, int ** *x*, int *n*, int *N*)

Print the population to the specified file.

Definition at line 175 of file eda.cpp.

References objective_function().

```
176 {
177   int i,j;
178 
179   for (i=0; i<N; i++)
180     {
181       for (j=0; j<n; j++)
182         fprintf(f,"%u",x[i][j]);
183       fprintf(f,"   %f\n",objective_function(x[i],n));
184     };
185 }
```

5.4.2.10 void print_status (int *t*, PopulationStatistics * *stats*)

Print the status of the algorithm.

Definition at line 451 of file eda.cpp.

References PopulationStatistics::avgF, PopulationStatistics::maxF, PopulationStatistics::minF, and separator().

Referenced by one_run().

```
452 {
453   printf("Generation: %u\n",t);
454   printf("   min:  %f\n",stats->minF);
455   printf("   max:  %f\n",stats->maxF);
456   printf("   mean: %f\n",stats->avgF);
457   separator(stdout);
458 }
```

5.4.2.11 void print_summary (PopulationStatistics * *stats*)

Print summary of a run.

Definition at line 478 of file eda.cpp.

References PopulationStatistics::maxF, PopulationStatistics::num_evals, PopulationStatistics::population_size, PopulationStatistics::problem_size, and PopulationStatistics::success.

Referenced by main().

```
479 {
480   printf("Summary:\n");
481   printf("   status       = %s\n",(stats->success)? "success":"failure");
482   printf("   best_found   = %f\n",stats->maxF);
483   printf("   num_evals    = %lu\n",stats->num_evals);
484   printf("   pop_size     = %u\n",stats->population_size);
485   printf("   problem_size = %u\n",stats->problem_size);
486 }
```

5.4.2.12 int restricted_tournament_replacement (int ** *y*, int ** *x*, int *n*, int *N*, double * *fx*, double * *fy*)

Restricted tournament replacement (Harik, 1995) for niching.

Definition at line 192 of file eda.cpp.

References individual_distance(), and intRand().

Referenced by one_run().

```
193 {
194   int i,j;
195
196   // use default value for window size
197
198   int windowSize=(n<N/20)? n:(N/20);
199
200   // for every individual, do the same
201
202   for (i=0; i<N; i++)
203     {
204       // select a random subset from the original population (window) and
205       // find the most similar guy to the new individual in this window
206       // (string Hamming distance)
207
208       int pick=intRand(N);
209       int dist=individual_distance(x[i],y[pick],n);
210
211       for (j=1; j<windowSize; j++)
```

```
212         {
213           int pick2=intRand(N);
214           int dist2=individual_distance(x[i],y[pick2],n);
215 
216           if (dist2<dist)
217             {
218               pick=pick2;
219               dist=dist2;
220             }
221         }
222 
223       // if the most similar guy from the window is better than the new guy,
224       // the new guy replaces it
225 
226       if (fx[i]<fy[pick])
227         {
228           memcpy(x[i],y[pick],sizeof(x[i][0])*n);
229           fx[i]=fy[pick];
230         }
231     }
232 
233   // return the number of processed new individuals
234 
235   return N;
236 }
```

5.4.2.13 void separator (FILE * *f*, int *type* = 1)

Print a sequence of dashes to separate text output.

Definition at line 465 of file eda.cpp.

Referenced by bisection(), print_parameters(), print_status(), and variation().

```
466 {
467   if (type==0)
468     fprintf(f,"- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");
469   else
470     fprintf(f,"--------------------------------------------------------------------------------\n");
471 }
```

5.4.2.14 int tournament_selection (int ** *y*, int ** *x*, double * *f*, int *n*, int *N*, int *k*)

Tournament selection with replacement (k-ary tournaments).

Definition at line 143 of file eda.cpp.

References intRand().

Referenced by one_run().

```
144 {
145   int i,j;
146 
147   for (i=0; i<N; i++)
148     {
149       // select a winner of k tournaments (with replacement)
150 
151       int winner=intRand(N);
152       for (j=1; j<k; j++)
153         {
154           int l=intRand(N);
```

```
155           if (f[l]>f[winner])
156             winner=l;
157         };
158
159       // the winner takes the next spot in the selected population
160
161       for (int ii=0; ii<n; ii++)
162         y[i][ii]=x[winner][ii];
163     };
164
165   // return the number of selected individuals
166
167   return N;
168 }
```

5.4.2.15 int variation (int ** *sampled_population*, int ** *selected_population*, Parameters * *params*, int * *num_vals*)

Variation operator of the decision-tree EDA (learns and samples model).

Definition at line 416 of file eda.cpp.

References TreeModel::learnProbabilities(), TreeModel::learnStructure(), N, Parameters::population_size, Parameters::problem_size, TreeModel::sampleModel(), separator(), and Parameters::verbose_mode.

Referenced by one_run().

```
417 {
418   int N=params->population_size;
419   int n=params->problem_size;
420   int loud=params->verbose_mode;
421
422   if (loud)
423     separator(stdout,0);
424
425   TreeModel *t = new TreeModel(n, num_vals);
426
427   // learn the model structure and model parameters
428
429   t->learnStructure(selected_population,N,loud);
430   t->learnProbabilities(selected_population,N,loud);
431
432   // sample the learned model to generate new candidate solutions
433
434   t->sampleModel(sampled_population,N);
435
436   // free memory
437
438   delete t;
439
440   // return the number of generated individuals
441
442   return N;
443 }
```

5.5 heap.cpp File Reference

Maximum heap used to store edges in Prim's algorithm for (maximum) spanning trees.

```
#include <stdio.h>
#include "heap.hpp"
```

Defines

- #define parent_index(i) ((i-1)/2)
- #define left_child_index(i) (1+2*i)

Functions

- int build_max_heap (int *a, int *b, double *x, int n, int *index, int *rev_index)
  Build a max-heap for a given array.
- int max_heapify (double *x, int i, int n, int *index, int *rev_index)
  Run max-heapify on a specified element of a heap (push down).
- int increase_key_max_heap (double *x, int i, double val, int n, int *index, int *rev_index)
  Increase a given key in the heap.
- int float_up_max_heap (double *x, int i, int n, int *index, int *rev_index)
  Float up an element up the heap (after increasing its value).
- int pop_max_heap (double *x, int &n, int *index, int *rev_index)
  Pop the maximum (includes the removal of maximum).
- void print_max_heap (double *x, int n, int *index, int *rev_index)
  Print the max heap (used mostly for debugging).
- int check_max_heap (double *x, int n, int *index, int *rev_index)
  Check the max heap for errors (used mostly in debugging).

5.5.1 Detailed Description

Maximum heap used to store edges in Prim's algorithm for (maximum) spanning trees.

Definition in file heap.cpp.

5.5.2 Define Documentation

5.5.2.1 #define left_child_index(i) (1+2*i)

Definition at line 12 of file heap.cpp.

Referenced by max_heapify().

5.5.2.2 #define parent_index(i) ((i-1)/2)

Definition at line 11 of file heap.cpp.

Referenced by float_up_max_heap().

5.5.3 Function Documentation

5.5.3.1 int build_max_heap (int * *a*, int * *b*, double * *x*, int *n*, int * *index*, int * *rev_index*)

Build a max-heap for a given array.

Definition at line 19 of file heap.cpp.

References max_heapify().

Referenced by TreeModel::learnStructure().

```
20 {
21   // builds a max-heap by max-heapifying all nodes with some children
22
23   for (int i=(n>>1)-1; i>=0; i--)
24     max_heapify(x,i,n,index,rev_index);
25
26   // get back
27
28   return n;
29 };
```

5.5.3.2 int check_max_heap (double * *x*, int *n*, int * *index*, int * *rev_index*)

Check the max heap for errors (used mostly in debugging).

Definition at line 156 of file heap.cpp.

```
157 {
158   int ok=1;
159
160   for (int i=0; (i<n)&&(ok); i++)
161     {
162       if (i*2+1<n)
163         if (x[index[i]]<x[index[2*i+1]])
164           ok=0;
165
166       if (i*2+2<n)
167         if (x[index[i]]<x[index[2*i+2]])
168           ok=0;
169     };
170
171   if (ok==0)
172     printf("not OK\n");
173   else
174     printf("OK\n");
175
176   return ok;
177 }
```

5.5.3.3 int float_up_max_heap (double * *x*, int *i*, int *n*, int * *index*, int * *rev_index*)

Float up an element up the heap (after increasing its value).

Definition at line 94 of file heap.cpp.

References parent_index.

Referenced by increase_key_max_heap().

```
95 {
96   double this_x=x[index[i]];
97   int old_ii=index[i];
98
99   int parent_idx=parent_index(i);
100
101   while ((i>0)&&(this_x>x[index[parent_idx]]))
102     {
103       index[i]=index[parent_idx];
104       rev_index[index[i]]=i;
105       i=parent_idx;
106       if (i>0)
107         parent_idx=parent_index(i);
108     };
109
110   index[i]=old_ii;
111   rev_index[index[i]]=i;
112
113   return i;
114 };
```

5.5.3.4 int increase_key_max_heap (double * *x*, int *i*, double *val*, int *n*, int * *index*, int * *rev_index*)

Increase a given key in the heap.

Definition at line 83 of file heap.cpp.

References float_up_max_heap().

Referenced by TreeModel::learnStructure().

```
84 {
85   x[i]=val;
86   return float_up_max_heap(x,rev_index[i],n,index,rev_index);
87 };
```

5.5.3.5 int max_heapify (double * *x*, int *i*, int *n*, int * *index*, int * *rev_index*)

Run max-heapify on a specified element of a heap (push down).

Definition at line 36 of file heap.cpp.

References left_child_index.

Referenced by build_max_heap(), and pop_max_heap().

```
37 {
38   double this_x=x[index[i]];
```

```
39   int old_ii=index[i];
40
41   int done=0;
42   do {
43     int c1=left_child_index(i);
44     int c2=c1+1;
45
46     int ic1=index[c1];
47
48     if (c2<n)
49       {
50         int ic2=index[c2];
51
52         if (x[ic1]<x[ic2])
53           {
54             ic1=ic2;
55             c1=c2;
56           };
57       };
58
59     if (x[ic1]>this_x)
60       {
61         index[i]=index[c1];
62         rev_index[index[i]]=i;
63         i=c1;
64         if (left_child_index(i)>=n)
65           done=1;
66       }
67     else
68       done=1;
69
70   } while (!done);
71
72   index[i]=old_ii;
73   rev_index[index[i]]=i;
74
75   return i;
76 };
```

5.5.3.6 int pop_max_heap (double * *x*, int & *n*, int * *index*, int * *rev_index*)

Pop the maximum (includes the removal of maximum).

Definition at line 121 of file heap.cpp.

References max_heapify().

Referenced by TreeModel::learnStructure().

```
122 {
123   n--;
124   int val=index[0];
125
126   index[0]=index[n];
127   rev_index[index[n]]=0;
128   max_heapify(x,0,n,index,rev_index);
129
130   return val;
131 }
```

5.5.3.7 void print_max_heap (double * *x*, int *n*, int * *index*, int * *rev_index*)

Print the max heap (used mostly for debugging).

Definition at line 138 of file heap.cpp.

```
139 {
140   for (int i=0; i<n; i++)
141     {
142       printf("x[%u] = %5.3f (",i,x[index[i]]);
143       if (i*2+1<n)
144         printf("%5.3f ",x[index[2*i+1]]);
145       if (i*2+2<n)
146         printf("%5.3f ",x[index[2*i+2]]);
147       printf(")\n");
148     };
149 }
```

5.6 heap.hpp File Reference

Header file for heap.cpp.

Defines

- #define HEAP_MINUS_INFINITY -1

Functions

- int max_heapify (double *x, int i, int n, int *index, int *rev_index)

  Run max-heapify on a specified element of a heap (push down).

- int build_max_heap (int *a, int *b, double *x, int n, int *index, int *rev_index)

  Build a max-heap for a given array.

- int float_up_max_heap (double *x, int i, int n, int *index, int *rev_index)

  Float up an element up the heap (after increasing its value).

- int increase_key_max_heap (double *x, int i, double val, int n, int *index, int *rev_index)

  Increase a given key in the heap.

- int pop_max_heap (double *x, int &n, int *index, int *rev_index)

  Pop the maximum (includes the removal of maximum).

- void print_max_heap (double *x, int n, int *index, int *rev_index)

  Print the max heap (used mostly for debugging).

- int check_max_heap (double *x, int n, int *index, int *rev_index)

  Check the max heap for errors (used mostly in debugging).

5.6.1 Detailed Description

Header file for heap.cpp.

Definition in file heap.hpp.

5.6.2 Define Documentation

5.6.2.1 #define HEAP_MINUS_INFINITY -1

Definition at line 9 of file heap.hpp.

Referenced by TreeModel::learnStructure().

5.6.3 Function Documentation

5.6.3.1 int build_max_heap (int * a, int * b, double * x, int n, int * index, int * rev_index)

Build a max-heap for a given array.

Definition at line 19 of file heap.cpp.

References max_heapify().

Referenced by TreeModel::learnStructure().

```
20 {
21   // builds a max-heap by max-heapifying all nodes with some children
22
23   for (int i=(n>>1)-1; i>=0; i--)
24     max_heapify(x,i,n,index,rev_index);
25
26   // get back
27
28   return n;
29 };
```

5.6.3.2 int check_max_heap (double * x, int n, int * index, int * rev_index)

Check the max heap for errors (used mostly in debugging).

Definition at line 156 of file heap.cpp.

```
157 {
158   int ok=1;
159
160   for (int i=0; (i<n)&&(ok); i++)
161     {
162       if (i*2+1<n)
163         if (x[index[i]]<x[index[2*i+1]])
164           ok=0;
165
166       if (i*2+2<n)
167         if (x[index[i]]<x[index[2*i+2]])
168           ok=0;
169     };
170
171   if (ok==0)
172     printf("not OK\n");
173   else
174     printf("OK\n");
175
176   return ok;
177 }
```

5.6.3.3 int float_up_max_heap (double * x, int i, int n, int * index, int * rev_index)

Float up an element up the heap (after increasing its value).

Definition at line 94 of file heap.cpp.

References parent_index.

Referenced by increase_key_max_heap().

```
95 {
96   double this_x=x[index[i]];
97   int old_ii=index[i];
98
99   int parent_idx=parent_index(i);
100
101   while ((i>0)&&(this_x>x[index[parent_idx]]))
102     {
103       index[i]=index[parent_idx];
104       rev_index[index[i]]=i;
105       i=parent_idx;
106       if (i>0)
107         parent_idx=parent_index(i);
108     };
109
110   index[i]=old_ii;
111   rev_index[index[i]]=i;
112
113   return i;
114 };
```

5.6.3.4 int increase_key_max_heap (double * *x*, int *i*, double *val*, int *n*, int * *index*, int * *rev_index*)

Increase a given key in the heap.

Definition at line 83 of file heap.cpp.

References float_up_max_heap().

Referenced by TreeModel::learnStructure().

```
84 {
85   x[i]=val;
86   return float_up_max_heap(x,rev_index[i],n,index,rev_index);
87 };
```

5.6.3.5 int max_heapify (double * *x*, int *i*, int *n*, int * *index*, int * *rev_index*)

Run max-heapify on a specified element of a heap (push down).

Definition at line 36 of file heap.cpp.

References left_child_index.

Referenced by build_max_heap(), and pop_max_heap().

```
37 {
38   double this_x=x[index[i]];
39   int old_ii=index[i];
40
41   int done=0;
42   do {
43     int c1=left_child_index(i);
44     int c2=c1+1;
45
46     int ic1=index[c1];
47
48     if (c2<n)
```

```
49       {
50         int ic2=index[c2];
51
52         if (x[ic1]<x[ic2])
53           {
54             ic1=ic2;
55             c1=c2;
56           };
57       };
58
59     if (x[ic1]>this_x)
60       {
61         index[i]=index[c1];
62         rev_index[index[i]]=i;
63         i=c1;
64         if (left_child_index(i)>=n)
65           done=1;
66       }
67     else
68       done=1;
69
70   } while (!done);
71
72   index[i]=old_ii;
73   rev_index[index[i]]=i;
74
75   return i;
76 };
```

5.6.3.6 int pop_max_heap (double * x, int & n, int * index, int * rev_index)

Pop the maximum (includes the removal of maximum).

Definition at line 121 of file heap.cpp.

References max_heapify().

Referenced by TreeModel::learnStructure().

```
122 {
123   n--;
124   int val=index[0];
125
126   index[0]=index[n];
127   rev_index[index[n]]=0;
128   max_heapify(x,0,n,index,rev_index);
129
130   return val;
131 }
```

5.6.3.7 void print_max_heap (double * x, int n, int * index, int * rev_index)

Print the max heap (used mostly for debugging).

Definition at line 138 of file heap.cpp.

```
139 {
140   for (int i=0; i<n; i++)
141     {
142       printf("x[%u] = %5.3f (",i,x[index[i]]);
143       if (i*2+1<n)
```

```
144         printf("%5.3f ",x[index[2*i+1]]);
145       if (i*2+2<n)
146         printf("%5.3f ",x[index[2*i+2]]);
147       printf(")\n");
148     };
149 }
```

5.7 main.cpp File Reference

Main function.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "bisection.hpp"
#include "eda.hpp"
#include "heap.hpp"
#include "parse-input.hpp"
#include "random.hpp"
#include "tree-model.hpp"
```

Functions

- int main (int argc, char **argv)
  The main function.

5.7.1 Detailed Description

Main function.

Definition in file main.cpp.

5.7.2 Function Documentation

5.7.2.1 int main (int *argc*, char ** *argv*)

The main function.

Definition at line 62 of file main.cpp.

References bisection(), Parameters::bisection, Parameters::max_generations, Parameters::num_bisection_runs, one_run(), param_help(), parse_input_file(), Parameters::population_size, print_bisection_summary(), print_parameters(), print_summary(), Parameters::problem_size, Parameters::quiet_mode, Parameters::replacement, Parameters::tournament_size, and Parameters::verbose_mode.

```
63 {
64   Parameters params;
65
66   // set default parameter values
67
68   params.population_size=300;
69   params.problem_size=25;
70   params.max_generations=50;
71   params.tournament_size=2;
72   params.replacement=0;
73   params.verbose_mode=0;
```

```
74   params.quiet_mode=0;
75   params.bisection=0;
76   params.num_bisection_runs=20;
77
78   // process input arguments
79
80   if (argc>2)
81     {
82       printf("Expected only one command-line argument.\n");
83       exit(-10);
84     }
85   else
86   if (argc==2)
87   {
88     if ((strcmp(argv[1],"--help")==0)||
89         (strcmp(argv[1],"-h")==0)||
90         (strcmp(argv[1],"/help")==0)||
91         (strcmp(argv[1],"/h")==0)||
92         (strcmp(argv[1],"-?")==0)||
93         (strcmp(argv[1],"/?")==0))
94       {
95         // help
96
97         printf("Usage: dt-eda [parameter file name] [--help] [--version]\n\n");
98         param_help();
99         exit(0);
100       }
101     else
102     if ((strcmp(argv[1],"--version")==0)||
103         (strcmp(argv[1],"-v")==0))
104       {
105         // version info
106
107         printf("dt-eda-1.0\n");
108         exit(0);
109       }
110     else
111       {
112         // parameter file
113
114         FILE *f=fopen(argv[1],"r");
115         if (f==NULL)
116           {
117             printf("ERROR: Could not open parameter file %s\n",argv[1]);
118             exit(-1);
119           }
120
121         parse_input_file(f,&params);
122         printf("Parameter file: %s\n",argv[1]);
123       };
124   }
125   else
126     printf("No parameter file: Using default parameters\n");
127
128   // print the parameters
129
130   print_parameters(&params);
131
132   // perform one run of the dependency-tree EDA with the current parameters
133   // and user defined functions (parameters from user-defined parameter files
134   // go first)
135
136   if (params.bisection)
137     {
138       AveragePopulationStatistics avg_stats;
139
140       bisection(&params,&avg_stats);
```

```
141
142       print_bisection_summary(stdout,&avg_stats);
143     }
144   else
145     {
146       PopulationStatistics stats;
147
148       one_run(&params,&stats);
149
150       print_summary(&stats);
151     };
152
153   // get out
154
155   return 0;
156 }
```

5.8 MT.cpp File Reference

Mersenne Twister random number generator.

Defines

- #define N 624
- #define M 397
- #define MATRIX_A 0x9908b0dfUL
- #define UPPER_MASK 0x80000000UL
- #define LOWER_MASK 0x7fffffffUL

Functions

- void init_genrand (unsigned long s)
- void init_by_array (unsigned long init_key[], int key_length)
- unsigned long genrand_int32 (void)
- long genrand_int31 (void)
- double genrand_real1 (void)
- double genrand_real2 (void)
- double genrand_real3 (void)
- double genrand_res53 (void)

5.8.1 Detailed Description

Mersenne Twister random number generator.

Definition in file MT.cpp.

5.8.2 Define Documentation

5.8.2.1 #define LOWER_MASK 0x7fffffffUL

Definition at line 55 of file MT.cpp.

Referenced by genrand_int32().

5.8.2.2 #define M 397

Definition at line 52 of file MT.cpp.

Referenced by genrand_int32().

5.8.2.3 #define MATRIX_A 0x9908b0dfUL

Definition at line 53 of file MT.cpp.

Referenced by genrand_int32().

5.8.2.4 #define N 624

Definition at line 51 of file MT.cpp.

Referenced by genrand_int32(), init_by_array(), init_genrand(), one_run(), and variation().

5.8.2.5 #define UPPER_MASK 0x80000000UL

Definition at line 54 of file MT.cpp.

Referenced by genrand_int32().

5.8.3 Function Documentation

5.8.3.1 long genrand_int31 (void)

Definition at line 144 of file MT.cpp.

References genrand_int32().

```
145 {
146     return (long)(genrand_int32()>>1);
147 }
```

5.8.3.2 unsigned long genrand_int32 (void)

Definition at line 106 of file MT.cpp.

References init_genrand(), LOWER_MASK, M, MATRIX_A, N, and UPPER_MASK.

Referenced by genrand_int31(), genrand_real1(), genrand_real2(), genrand_real3(), and genrand_res53().

```
107 {
108     unsigned long y;
109     static unsigned long mag01[2]={0x0UL, MATRIX_A};
110     /* mag01[x] = x * MATRIX_A  for x=0,1 */
111
112     if (mti >= N) { /* generate N words at one time */
113         int kk;
114
115         if (mti == N+1)   /* if init_genrand() has not been called, */
116             init_genrand(5489UL); /* a default initial seed is used */
117
118         for (kk=0;kk<N-M;kk++) {
119             y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
120             mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
121         }
122         for (;kk<N-1;kk++) {
123             y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
124             mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
125         }
126         y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
127         mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];
128
129         mti = 0;
130     }
131
132     y = mt[mti++];
```

```
133
134     /* Tempering */
135     y ^= (y >> 11);
136     y ^= (y << 7) & 0x9d2c5680UL;
137     y ^= (y << 15) & 0xefc60000UL;
138     y ^= (y >> 18);
139
140     return y;
141 }
```

5.8.3.3 double genrand_real1 (void)

Definition at line 150 of file MT.cpp.

References genrand_int32().

```
151 {
152     return genrand_int32()*(1.0/4294967295.0);
153     /* divided by 2^32-1 */
154 }
```

5.8.3.4 double genrand_real2 (void)

Definition at line 157 of file MT.cpp.

References genrand_int32().

Referenced by drand().

```
158 {
159     return genrand_int32()*(1.0/4294967296.0);
160     /* divided by 2^32 */
161 }
```

5.8.3.5 double genrand_real3 (void)

Definition at line 164 of file MT.cpp.

References genrand_int32().

```
165 {
166     return (((double)genrand_int32()) + 0.5)*(1.0/4294967296.0);
167     /* divided by 2^32 */
168 }
```

5.8.3.6 double genrand_res53 (void)

Definition at line 171 of file MT.cpp.

References genrand_int32().

```
172 {
173     unsigned long a=genrand_int32()>>5, b=genrand_int32()>>6;
174     return(a*67108864.0+b)*(1.0/9007199254740992.0);
175 }
```

5.8.3.7 void init_by_array (unsigned long *init_key*[], int *key_length*)

Definition at line 80 of file MT.cpp.

References init_genrand(), and N.

```
81 {
82     int i, j, k;
83     init_genrand(19650218UL);
84     i=1; j=0;
85     k = (N>key_length ? N : key_length);
86     for (; k; k--) {
87         mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1664525UL))
88           + init_key[j] + j; /* non linear */
89         mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
90         i++; j++;
91         if (i>=N) { mt[0] = mt[N-1]; i=1; }
92         if (j>=key_length) j=0;
93     }
94     for (k=N-1; k; k--) {
95         mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1566083941UL))
96           - i; /* non linear */
97         mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
98         i++;
99         if (i>=N) { mt[0] = mt[N-1]; i=1; }
100     }
101
102     mt[0] = 0x80000000UL; /* MSB is 1; assuring non-zero initial array */
103 }
```

5.8.3.8 void init_genrand (unsigned long *s*)

Definition at line 61 of file MT.cpp.

References N.

Referenced by genrand_int32(), init_by_array(), and setSeed().

```
62 {
63     mt[0]= s & 0xffffffffUL;
64     for (mti=1; mti<N; mti++) {
65         mt[mti] =
66             (1812433253UL * (mt[mti-1] ^ (mt[mti-1] >> 30)) + mti);
67         /* See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier. */
68         /* In the previous versions, MSBs of the seed affect   */
69         /* only MSBs of the array mt[].                        */
70         /* 2002/01/09 modified by Makoto Matsumoto             */
71         mt[mti] &= 0xffffffffUL;
72         /* for >32 bit machines */
73     }
74 }
```

5.9 MT.hpp File Reference

Header file for MT.cpp.

Functions

- void init_genrand (unsigned long s)
- void init_by_array (unsigned long init_key[], int key_length)
- unsigned long genrand_int32 (void)
- long genrand_int31 (void)
- double genrand_real1 (void)
- double genrand_real2 (void)
- double genrand_real3 (void)
- double genrand_res53 (void)

5.9.1 Detailed Description

Header file for MT.cpp.

Definition in file MT.hpp.

5.9.2 Function Documentation

5.9.2.1 long genrand_int31 (void)

Definition at line 144 of file MT.cpp.

References genrand_int32().

```
145 {
146     return (long)(genrand_int32()>>1);
147 }
```

5.9.2.2 unsigned long genrand_int32 (void)

Definition at line 106 of file MT.cpp.

References init_genrand(), LOWER_MASK, M, MATRIX_A, N, and UPPER_MASK.

Referenced by genrand_int31(), genrand_real1(), genrand_real2(), genrand_real3(), and genrand_res53().

```
107 {
108     unsigned long y;
109     static unsigned long mag01[2]={0x0UL, MATRIX_A};
110     /* mag01[x] = x * MATRIX_A  for x=0,1 */
111
112     if (mti >= N) { /* generate N words at one time */
113         int kk;
114
115         if (mti == N+1)   /* if init_genrand() has not been called, */
116             init_genrand(5489UL); /* a default initial seed is used */
117
118         for (kk=0;kk<N-M;kk++) {
```

```
119             y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
120             mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
121         }
122         for (;kk<N-1;kk++) {
123             y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
124             mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
125         }
126         y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
127         mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];
128 
129         mti = 0;
130     }
131 
132     y = mt[mti++];
133 
134     /* Tempering */
135     y ^= (y >> 11);
136     y ^= (y << 7) & 0x9d2c5680UL;
137     y ^= (y << 15) & 0xefc60000UL;
138     y ^= (y >> 18);
139 
140     return y;
141 }
```

5.9.2.3 double genrand_real1 (void)

Definition at line 150 of file MT.cpp.

References genrand_int32().

```
151 {
152     return genrand_int32()*(1.0/4294967295.0);
153     /* divided by 2^32-1 */
154 }
```

5.9.2.4 double genrand_real2 (void)

Definition at line 157 of file MT.cpp.

References genrand_int32().

Referenced by drand().

```
158 {
159     return genrand_int32()*(1.0/4294967296.0);
160     /* divided by 2^32 */
161 }
```

5.9.2.5 double genrand_real3 (void)

Definition at line 164 of file MT.cpp.

References genrand_int32().

```
165 {
166     return (((double)genrand_int32()) + 0.5)*(1.0/4294967296.0);
167     /* divided by 2^32 */
168 }
```

5.9.2.6 double genrand_res53 (void)

Definition at line 171 of file MT.cpp.

References genrand_int32().

```
172 {
173     unsigned long a=genrand_int32()>>5, b=genrand_int32()>>6;
174     return(a*67108864.0+b)*(1.0/9007199254740992.0);
175 }
```

5.9.2.7 void init_by_array (unsigned long *init_key*[], int *key_length*)

Definition at line 80 of file MT.cpp.

References init_genrand(), and N.

```
81 {
82     int i, j, k;
83     init_genrand(19650218UL);
84     i=1; j=0;
85     k = (N>key_length ? N : key_length);
86     for (; k; k--) {
87         mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1664525UL))
88           + init_key[j] + j; /* non linear */
89         mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
90         i++; j++;
91         if (i>=N) { mt[0] = mt[N-1]; i=1; }
92         if (j>=key_length) j=0;
93     }
94     for (k=N-1; k; k--) {
95         mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1566083941UL))
96           - i; /* non linear */
97         mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
98         i++;
99         if (i>=N) { mt[0] = mt[N-1]; i=1; }
100     }
101
102     mt[0] = 0x80000000UL; /* MSB is 1; assuring non-zero initial array */
103 }
```

5.9.2.8 void init_genrand (unsigned long *s*)

Definition at line 61 of file MT.cpp.

References N.

Referenced by genrand_int32(), init_by_array(), and setSeed().

```
62 {
63     mt[0]= s & 0xffffffffUL;
64     for (mti=1; mti<N; mti++) {
65         mt[mti] =
66             (1812433253UL * (mt[mti-1] ^ (mt[mti-1] >> 30)) + mti);
67         /* See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier. */
68         /* In the previous versions, MSBs of the seed affect   */
69         /* only MSBs of the array mt[].                        */
70         /* 2002/01/09 modified by Makoto Matsumoto             */
71         mt[mti] &= 0xffffffffUL;
72         /* for >32 bit machines */
73     }
74 }
```

5.10 obj-function.cpp File Reference

User-defined functions that define the problem.

`#include <stdlib.h>`

`#include <stdio.h>`

Functions

- double objective_function (int *x, int n)

  The example included by default is a simple onemax problem, which is defined as the sum of all characters in a string (usually binary but it can really be used with any alphabet represented by integers like here).

- void set_num_vals (int *num_vals, int n)

  Each string position i has values from 0 to num_values[i]-1.

- int is_optimal (int *x, int n, int *num_vals)

  Verify whether the given solution is the global optimum.

5.10.1 Detailed Description

User-defined functions that define the problem.

See also:

Example input parameter files: example_input, example_input_bisection, example_input_big
Example objective function (concatenated trap of order 5): example-trap5.cpp
Example function for setting the number of characters in each string position: example-num-values.cpp

Definition in file obj-function.cpp.

5.10.2 Function Documentation

5.10.2.1 int is_optimal (int * *x*, int *n*, int * *num_vals*)

Verify whether the given solution is the global optimum.

If there is no such function, just return 0 by default. The default function works for the default objective function, that is, onemax.

Parameters:

x input string (candidate solution)

n string length

num_vals array of the numbers of values for all string positions

Returns:

0 if the solution is non-optimal, something else if the solution is optimal

Definition at line 117 of file obj-function.cpp.

```
118 {
119   int ok=1;
120
121   for (int i=0; (i<n)&&(ok); i++)
122     if (x[i]!=num_vals[i]-1)
123       ok=0;
124
125   return ok;
126 };
```

5.10.2.2 double objective_function (int * *x*, int *n*)

The example included by default is a simple onemax problem, which is defined as the sum of all characters in a string (usually binary but it can really be used with any alphabet represented by integers like here).

The objective function is coupled with the function that specifies the number of values of all string positions, set_num_vals(int *, int). Both these functions can be found in obj-function.cpp.

Parameters:

x input string (candidate solution)

n string length

Returns:

Value of the objective function for the given candidate solution.

Examples:

example-trap5.cpp.

Definition at line 36 of file obj-function.cpp.

Referenced by evaluate_population(), and print_population().

```
37 {
38   int val=0;
39
40   // compute the sum of all values in the given string
41   // (works also for non-binary representations)
42
43   for (int i=0; i<n; i++)
44     val+=x[i];
45
46   // return the final value
47
48   return val;
49 }
```

5.10.2.3 void set_num_vals (int * *num_vals*, int *n*)

Each string position i has values from 0 to num_values[i]-1.

By default, the code makes all string positions ternary (values 0 to 2) but alternative assignments can be used.

Parameters:

num_vals array to store the numbers of values for all string positions

n string length

Returns:

No return value.

Examples:

example-num-values.cpp.

Definition at line 80 of file obj-function.cpp.

Referenced by one_run().

```
81 {
82   // set the number of values for each string position (variable),
83   // use 2 for binary, 3 or higher for higher candinality alphabets
84
85   for (int i=0; i<n; i++)
86     num_vals[i]=3;
87 }
```

5.11 obj-function.hpp File Reference

Header file for obj-function.cpp.

Functions

- double objective_function (int *x, int n)

  The example included by default is a simple onemax problem, which is defined as the sum of all characters in a string (usually binary but it can really be used with any alphabet represented by integers like here).

- void set_num_vals (int *num_vals, int n)

  Each string position i has values from 0 to num_values[i]-1.

- int is_optimal (int *x, int n, int *num_vals)

  Verify whether the given solution is the global optimum.

5.11.1 Detailed Description

Header file for obj-function.cpp.

Definition in file obj-function.hpp.

5.11.2 Function Documentation

5.11.2.1 int is_optimal (int * *x*, int *n*, int * *num_vals*)

Verify whether the given solution is the global optimum.

If there is no such function, just return 0 by default. The default function works for the default objective function, that is, onemax.

Parameters:

x input string (candidate solution)

n string length

num_vals array of the numbers of values for all string positions

Returns:

0 if the solution is non-optimal, something else if the solution is optimal

Definition at line 117 of file obj-function.cpp.

```
118 {
119   int ok=1;
120
121   for (int i=0; (i<n)&&(ok); i++)
122     if (x[i]!=num_vals[i]-1)
123       ok=0;
124
125   return ok;
126 };
```

5.11.2.2 double objective_function (int * x, int n)

The example included by default is a simple onemax problem, which is defined as the sum of all characters in a string (usually binary but it can really be used with any alphabet represented by integers like here).

The objective function is coupled with the function that specifies the number of values of all string positions, set_num_vals(int *, int). Both these functions can be found in obj-function.cpp.

Parameters:

x input string (candidate solution)

n string length

Returns:

Value of the objective function for the given candidate solution.

Definition at line 36 of file obj-function.cpp.

Referenced by evaluate_population(), and print_population().

```
37 {
38   int val=0;
39
40   // compute the sum of all values in the given string
41   // (works also for non-binary representations)
42
43   for (int i=0; i<n; i++)
44     val+=x[i];
45
46   // return the final value
47
48   return val;
49 }
```

5.11.2.3 void set_num_vals (int * num_vals, int n)

Each string position i has values from 0 to num_values[i]-1.

By default, the code makes all string positions ternary (values 0 to 2) but alternative assignments can be used.

Parameters:

num_vals array to store the numbers of values for all string positions

n string length

Returns:

No return value.

Definition at line 80 of file obj-function.cpp.

Referenced by one_run().

```
81 {
82   // set the number of values for each string position (variable),
```

```
83   // use 2 for binary, 3 or higher for higher candinality alphabets
84
85   for (int i=0; i<n; i++)
86     num_vals[i]=3;
87 }
```

5.12 parse-input.cpp File Reference

Necessary functions for parsing input parameter files.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "parse-input.hpp"
#include "random.hpp"
```

Defines

- #define MAX_LINE_WIDTH 1000
  Maximum width of a line to read from parameter file.

Functions

- void parse_input_file (FILE *f, Parameters *params)
  Parse the input file and store the results in a structure Parameters.
- void param_help ()
  Print a short description of parameters.
- int get_new_identifier (FILE *f, char *s)
  Get a new identifier from the file and scan until after next '='.
- int get_int_value (FILE *f, int &val)
  Read an integer value from a file.
- int print_parameters (Parameters *params)
  Print the parameters to stdout.

5.12.1 Detailed Description

Necessary functions for parsing input parameter files.

Definition in file parse-input.cpp.

5.12.2 Define Documentation

5.12.2.1 #define MAX_LINE_WIDTH 1000

Maximum width of a line to read from parameter file.

Definition at line 15 of file parse-input.cpp.

Referenced by get_int_value(), and parse_input_file().

5.12.3 Function Documentation

5.12.3.1 int get_int_value (FILE * *f*, int & *val*)

Read an integer value from a file.

Definition at line 166 of file parse-input.cpp.

References MAX_LINE_WIDTH.

Referenced by parse_input_file().

```
167 {
168   char line[MAX_LINE_WIDTH];
169   fgets(line,MAX_LINE_WIDTH,f);
170   sscanf(line,"%u",&val);
171
172   return 1;
173 }
```

5.12.3.2 int get_new_identifier (FILE * *f*, char * *s*)

Get a new identifier from the file and scan until after next '='.

Definition at line 105 of file parse-input.cpp.

Referenced by parse_input_file().

```
106 {
107   char c=0;
108   int n=0;
109   int done=0;
110
111   while ((!feof(f))&&(!done))
112     {
113       c=fgetc(f);
114       if (((c>='a')&&(c<='z'))||
115           ((c>='A')&&(c<='Z'))||
116           (c=='_'))
117         done=1;
118     }
119
120   if (done)
121     {
122       s[0]=c;
123       n=1;
124       done=0;
125       while ((!feof(f))&&(!done))
126         {
127           c=fgetc(f);
128           if (((c>='a')&&(c<='z'))||
129               ((c>='A')&&(c<='Z'))||
130               (c=='_'))
131             s[n++]=c;
132           else
133             done=1;
134         };
135       s[n]='\0';
136
137       if (c=='=')
138         done=1;
139       else
140         done=0;
```

```
141
142       while ((!feof(f))&&(!done))
143         {
144           c=fgetc(f);
145           if (c=='=')
146             done=1;
147         };
148
149       if (!done)
150         {
151           printf("Expected '=' after identifier %s\n",s);
152           exit(-1);
153         }
154
155       return 1;
156     }
157
158   return 0;
159 }
```

5.12.3.3 void param_help ()

Print a short description of parameters.

Definition at line 83 of file parse-input.cpp.

Referenced by main().

```
84 {
85   printf("List of parameters for input file (you can specify any subset):\n\n");
86
87   printf("  -> Population size:\n      population_size = <number>\n\n");
88   printf("  -> Problem size (number of characters):\n      problem_size = <number>\n\n");
89   printf("  -> Maximum number of generations:\n      max_generations = <number>\n\n");
90   printf("  -> Replacement (0=restricted tournament repl., 1=full repl.):\n      replacement = <number>\n\n");
91   printf("  -> Tournament size for tournament selection:\n      tournament_size = <number>\n\n");
92   printf("  -> Use bisection to optimize the population size:\n      bisection = <number>\n\n");
93   printf("  -> Number of successful runs for optimal population sizing:\n      num_runs=<number>\n\n");
94   printf("  -> Quiet mode (prints only the end-of-the-run summary):\n      quiet_mode = <number>\n\n");
95   printf("  -> Verbose mode (do not use with redirected output):\n      verbose_mode = <number>\n\n");
96   printf("  -> Random seed:\n      random_seed = <number>\n\n");
97   printf("See example_input, example_input_bisection, and example_input_big for example input files\n");
98 }
```

5.12.3.4 void parse_input_file (FILE * *f*, Parameters * *params*)

Parse the input file and store the results in a structure Parameters.

See also:

Example input parameter file: example_input

Definition at line 23 of file parse-input.cpp.

References Parameters::bisection, get_int_value(), get_new_identifier(), Parameters::max_generations, MAX_LINE_WIDTH, Parameters::num_bisection_runs, Parameters::population_size, Parameters::problem_size, Parameters::quiet_mode, Parameters::replacement, setSeed(), Parameters::tournament_size, and Parameters::verbose_mode.

Referenced by main().

```
24 {
25   if (f==NULL)
26     return;
27
28   char id[MAX_LINE_WIDTH];
29
30   int seed_set=0;
31   while (get_new_identifier(f,id))
32     {
33       if (strcmp(id,"population_size")==0)
34         get_int_value(f,params->population_size);
35       else
36       if (strcmp(id,"max_generations")==0)
37         get_int_value(f,params->max_generations);
38       else
39       if (strcmp(id,"problem_size")==0)
40         get_int_value(f,params->problem_size);
41       else
42       if (strcmp(id,"replacement")==0)
43         get_int_value(f,params->replacement);
44       else
45       if (strcmp(id,"tournament_size")==0)
46         get_int_value(f,params->tournament_size);
47       else
48       if (strcmp(id,"bisection")==0)
49         get_int_value(f,params->bisection);
50       else
51       if (strcmp(id,"num_bisection_runs")==0)
52         get_int_value(f,params->num_bisection_runs);
53       else
54       if (strcmp(id,"quiet_mode")==0)
55         get_int_value(f,params->quiet_mode);
56       else
57       if (strcmp(id,"verbose_mode")==0)
58         get_int_value(f,params->verbose_mode);
59       else
60       if (strcmp(id,"random_seed")==0)
61         {
62           int seed;
63           get_int_value(f,seed);
64           setSeed(seed);
65           seed_set=1;
66         }
67       else
68         {
69           printf("%s is an unknown identifier\n",id);
70           exit(-1);
71         };
72     };
73
74   if (seed_set==0)
75     setSeed(123);
76 }
```

5.12.3.5 int print_parameters (Parameters * *params*)

Print the parameters to stdout.

Definition at line 180 of file parse-input.cpp.

References Parameters::bisection, Parameters::max_generations, Parameters::num_bisection_runs, Parameters::population_size, Parameters::problem_size, Parameters::quiet_mode, Parameters::replacement, separator(), Parameters::tournament_size, and Parameters::verbose_mode.

Referenced by main().

```
181 {
182   printf("Parameters:\n");
183   printf("  population_size    = %u\n",params->population_size);
184   printf("  problem_size       = %u\n",params->problem_size);
185   printf("  max_generations    = %u\n",params->max_generations);
186   printf("  tournament_size    = %u\n",params->tournament_size);
187   printf("  replacement        = %u\n",params->replacement);
188   printf("  bisection          = %u\n",params->bisection);
189   printf("  num_bisection_runs = %u\n",params->num_bisection_runs);
190   printf("  quiet_mode         = %u\n",params->quiet_mode);
191   printf("  verbose_mode       = %u\n",params->verbose_mode);
192   separator(stdout);
193
194   return 0;
195 }
```

5.13 parse-input.hpp File Reference

Header file for parse-input.cpp.

```
#include <stdio.h>
```

```
#include "eda.hpp"
```

Functions

- void parse_input_file (FILE *f, Parameters *params)

  Parse the input file and store the results in a structure Parameters.

- void param_help ()

  Print a short description of parameters.

- int get_new_identifier (FILE *f, char *s)

  Get a new identifier from the file and scan until after next '='.

- int get_int_value (FILE *f, int &val)

  Read an integer value from a file.

- int print_parameters (Parameters *params)

  Print the parameters to stdout.

5.13.1 Detailed Description

Header file for parse-input.cpp.

Definition in file parse-input.hpp.

5.13.2 Function Documentation

5.13.2.1 int get_int_value (FILE * *f*, int & *val*)

Read an integer value from a file.

Definition at line 166 of file parse-input.cpp.

References MAX_LINE_WIDTH.

Referenced by parse_input_file().

```
167 {
168   char line[MAX_LINE_WIDTH];
169   fgets(line,MAX_LINE_WIDTH,f);
170   sscanf(line,"%u",&val);
171
172   return 1;
173 }
```

5.13.2.2 int get_new_identifier (FILE * f, char * s)

Get a new identifier from the file and scan until after next '='.

Definition at line 105 of file parse-input.cpp.

Referenced by parse_input_file().

```
106 {
107   char c=0;
108   int n=0;
109   int done=0;
110
111   while ((!feof(f))&&(!done))
112     {
113       c=fgetc(f);
114       if (((c>='a')&&(c<='z'))||
115           ((c>='A')&&(c<='Z'))||
116           (c=='_'))
117         done=1;
118     }
119
120   if (done)
121     {
122       s[0]=c;
123       n=1;
124       done=0;
125       while ((!feof(f))&&(!done))
126         {
127           c=fgetc(f);
128           if (((c>='a')&&(c<='z'))||
129               ((c>='A')&&(c<='Z'))||
130               (c=='_'))
131             s[n++]=c;
132           else
133             done=1;
134         };
135       s[n]='\0';
136
137       if (c=='=')
138         done=1;
139       else
140         done=0;
141
142       while ((!feof(f))&&(!done))
143         {
144           c=fgetc(f);
145           if (c=='=')
146             done=1;
147         };
148
149       if (!done)
150         {
151           printf("Expected '=' after identifier %s\n",s);
152           exit(-1);
153         }
154
155       return 1;
156     }
157
158   return 0;
159 }
```

5.13.2.3 void param_help ()

Print a short description of parameters.

Definition at line 83 of file parse-input.cpp.

Referenced by main().

```
84 {
85   printf("List of parameters for input file (you can specify any subset):\n\n");
86
87   printf("  -> Population size:\n      population_size = <number>\n\n");
88   printf("  -> Problem size (number of characters):\n      problem_size = <number>\n\n");
89   printf("  -> Maximum number of generations:\n      max_generations = <number>\n\n");
90   printf("  -> Replacement (0=restricted tournament repl., 1=full repl.):\n      replacement = <number>\n\n");
91   printf("  -> Tournament size for tournament selection:\n      tournament_size = <number>\n\n");
92   printf("  -> Use bisection to optimize the population size:\n      bisection = <number>\n\n");
93   printf("  -> Number of successful runs for optimal population sizing:\n      num_runs=<number>\n\n");
94   printf("  -> Quiet mode (prints only the end-of-the-run summary):\n      quiet_mode = <number>\n\n");
95   printf("  -> Verbose mode (do not use with redirected output):\n      verbose_mode = <number>\n\n");
96   printf("  -> Random seed:\n      random_seed = <number>\n\n");
97   printf("See example_input, example_input_bisection, and example_input_big for example input files\n");
98 }
```

5.13.2.4 void parse_input_file (FILE * *f*, Parameters * *params*)

Parse the input file and store the results in a structure Parameters.

See also:

Example input parameter file: example_input

Definition at line 23 of file parse-input.cpp.

References Parameters::bisection, get_int_value(), get_new_identifier(), Parameters::max_generations, MAX_LINE_WIDTH, Parameters::num_bisection_runs, Parameters::population_size, Parameters::problem_size, Parameters::quiet_mode, Parameters::replacement, setSeed(), Parameters::tournament_size, and Parameters::verbose_mode.

Referenced by main().

```
24 {
25   if (f==NULL)
26     return;
27
28   char id[MAX_LINE_WIDTH];
29
30   int seed_set=0;
31   while (get_new_identifier(f,id))
32     {
33       if (strcmp(id,"population_size")==0)
34         get_int_value(f,params->population_size);
35       else
36       if (strcmp(id,"max_generations")==0)
37         get_int_value(f,params->max_generations);
38       else
39       if (strcmp(id,"problem_size")==0)
40         get_int_value(f,params->problem_size);
41       else
42       if (strcmp(id,"replacement")==0)
43         get_int_value(f,params->replacement);
44       else
```

```
45       if (strcmp(id,"tournament_size")==0)
46         get_int_value(f,params->tournament_size);
47       else
48       if (strcmp(id,"bisection")==0)
49         get_int_value(f,params->bisection);
50       else
51       if (strcmp(id,"num_bisection_runs")==0)
52         get_int_value(f,params->num_bisection_runs);
53       else
54       if (strcmp(id,"quiet_mode")==0)
55         get_int_value(f,params->quiet_mode);
56       else
57       if (strcmp(id,"verbose_mode")==0)
58         get_int_value(f,params->verbose_mode);
59       else
60       if (strcmp(id,"random_seed")==0)
61         {
62           int seed;
63           get_int_value(f,seed);
64           setSeed(seed);
65           seed_set=1;
66         }
67       else
68         {
69           printf("%s is an unknown identifier\n",id);
70           exit(-1);
71         };
72     };
73
74   if (seed_set==0)
75     setSeed(123);
76 }
```

5.13.2.5 int print_parameters (Parameters * *params*)

Print the parameters to stdout.

Definition at line 180 of file parse-input.cpp.

References Parameters::bisection, Parameters::max_generations, Parameters::num_bisection_runs, Parameters::population_size, Parameters::problem_size, Parameters::quiet_mode, Parameters::replacement, separator(), Parameters::tournament_size, and Parameters::verbose_mode.

Referenced by main().

```
181 {
182   printf("Parameters:\n");
183   printf("  population_size    = %u\n",params->population_size);
184   printf("  problem_size       = %u\n",params->problem_size);
185   printf("  max_generations    = %u\n",params->max_generations);
186   printf("  tournament_size    = %u\n",params->tournament_size);
187   printf("  replacement        = %u\n",params->replacement);
188   printf("  bisection          = %u\n",params->bisection);
189   printf("  num_bisection_runs = %u\n",params->num_bisection_runs);
190   printf("  quiet_mode         = %u\n",params->quiet_mode);
191   printf("  verbose_mode       = %u\n",params->verbose_mode);
192   separator(stdout);
193
194   return 0;
195 }
```

5.14 random.cpp File Reference

Various random number generator related functions based on the basic generator in MT.cpp.

```
#include <stdio.h>
#include <math.h>
#include "random.hpp"
#include "MT.hpp"
```

Functions

- double drand ()
- int intRand (int max)
- long longRand (long max)
- char flipCoin ()
- unsigned long setSeed (unsigned long newSeed)
- double gaussianRandom (double mean, double stddev)

Variables

- long _Q = _M/_A
- long _R = _M%_A
- long _seed = 123
- char whichGaussian = 0

5.14.1 Detailed Description

Various random number generator related functions based on the basic generator in MT.cpp.

Definition in file random.cpp.

5.14.2 Function Documentation

5.14.2.1 double drand ()

Definition at line 31 of file random.cpp.

References genrand_real2().

Referenced by flipCoin(), gaussianRandom(), generate_BB_population(), generate_population(), intRand(), and longRand().

```
32 {
33 //   long lo,hi,test;
34
35 //   hi   = _seed / _Q;
36 //   lo   = _seed % _Q;
37 //   test = _A*lo - _R*hi;
38
39 //   if (test>0)
40 //     _seed = test;
41 //   else
```

```
42 //     _seed = test+_M;
43
44 //   return double(_seed)/_M;
45   return genrand_real2();
46 }
```

5.14.2.2 char flipCoin ()

Definition at line 96 of file random.cpp.

References drand().

```
97 {
98   if (drand()<0.5)
99       return 1;
100   else
101       return 0;
102 };
```

5.14.2.3 double gaussianRandom (double *mean*, double *stddev*)

Definition at line 139 of file random.cpp.

References drand().

```
140 {
141    double  q,u,v,x,y;
142
143    /*
144       Generate P = (u,v) uniform in rect. enclosing acceptance region
145       Make sure that any random numbers <= 0 are rejected, since
146       gaussian() requires uniforms > 0, but RandomUniform() delivers >= 0.
147    */
148    do {
149       do { u=drand(); } while (u==0);
150       do { v=drand(); } while (v==0);
151
152       v = 1.7156 * (v - 0.5);
153
154       /*  Evaluate the quadratic form */
155       x = u - 0.449871;
156       y = fabs(v) + 0.386595;
157       q = x * x + y * (0.19600 * y - 0.25472 * x);
158
159       /* Accept P if inside inner ellipse */
160       if (q < 0.27597)
161          break;
162
163       /*  Reject P if outside outer ellipse, or outside acceptance region */
164     } while ((q > 0.27846) || (v * v > -4.0 * log(u) * u * u));
165
166     /*  Return ratio of P's coordinates as the normal deviate */
167     return (mean + stddev * v / u);
168 }
```

5.14.2.4 int intRand (int *max*)

Definition at line 60 of file random.cpp.

References drand().

Referenced by generate_population(), TreeModel::learnStructure(), restricted_tournament_replacement(), and tournament_selection().

```
61 {
62 //   double r=drand();
63 //   printf("r=%f\n",r);
64   return (int) (drand()*max);
65 };
```

5.14.2.5 long longRand (long *max*)

Definition at line 79 of file random.cpp.

References drand().

```
80 {
81    return (long) ((double) drand()*max);
82 };
```

5.14.2.6 unsigned long setSeed (unsigned long *newSeed*)

Definition at line 116 of file random.cpp.

References init_genrand().

Referenced by parse_input_file().

```
117 {
118   // set the seed and return the result of the operation
119
120   init_genrand(newSeed);
121
122   return newSeed;
123 };
```

5.14.3 Variable Documentation

5.14.3.1 long _Q = _M/_A

Definition at line 12 of file random.cpp.

5.14.3.2 long _R = _M%_A

Definition at line 13 of file random.cpp.

5.14.3.3 long _seed = 123

Definition at line 14 of file random.cpp.

5.14.3.4 char whichGaussian = 0

Definition at line 16 of file random.cpp.

5.15 random.hpp File Reference

Header file for random.cpp.

Defines

- #define _M 2147483647
- #define _A 16807

Functions

- double drand ()
- int intRand (int max)
- long longRand (long max)
- char flipCoin ()
- double gaussianRandom (double mean, double stddev)
- unsigned long setSeed (unsigned long newSeed)

5.15.1 Detailed Description

Header file for random.cpp.

Definition in file random.hpp.

5.15.2 Define Documentation

5.15.2.1 #define _A 16807

Definition at line 10 of file random.hpp.

5.15.2.2 #define _M 2147483647

Definition at line 9 of file random.hpp.

5.15.3 Function Documentation

5.15.3.1 double drand ()

Definition at line 31 of file random.cpp.

References genrand_real2().

Referenced by flipCoin(), gaussianRandom(), generate_BB_population(), generate_population(), intRand(), and longRand().

```
32 {
33 //   long lo,hi,test;
34
35 //   hi   = _seed / _Q;
36 //   lo   = _seed % _Q;
37 //   test = _A*lo - _R*hi;
```

```
38
39 //   if (test>0)
40 //     _seed = test;
41 //   else
42 //     _seed = test+_M;
43
44 //   return double(_seed)/_M;
45   return genrand_real2();
46 }
```

5.15.3.2 char flipCoin ()

Definition at line 96 of file random.cpp.

References drand().

```
97 {
98   if (drand()<0.5)
99       return 1;
100   else
101       return 0;
102 };
```

5.15.3.3 double gaussianRandom (double *mean*, double *stddev*)

Definition at line 139 of file random.cpp.

References drand().

```
140 {
141    double  q,u,v,x,y;
142
143    /*
144       Generate P = (u,v) uniform in rect. enclosing acceptance region
145       Make sure that any random numbers <= 0 are rejected, since
146       gaussian() requires uniforms > 0, but RandomUniform() delivers >= 0.
147    */
148    do {
149       do { u=drand(); } while (u==0);
150       do { v=drand(); } while (v==0);
151
152       v = 1.7156 * (v - 0.5);
153
154       /*  Evaluate the quadratic form */
155       x = u - 0.449871;
156       y = fabs(v) + 0.386595;
157       q = x * x + y * (0.19600 * y - 0.25472 * x);
158
159       /* Accept P if inside inner ellipse */
160       if (q < 0.27597)
161          break;
162
163       /*  Reject P if outside outer ellipse, or outside acceptance region */
164     } while ((q > 0.27846) || (v * v > -4.0 * log(u) * u * u));
165
166     /*  Return ratio of P's coordinates as the normal deviate */
167     return (mean + stddev * v / u);
168 }
```

5.15.3.4 int intRand (int *max*)

Definition at line 60 of file random.cpp.

References drand().

Referenced by generate_population(), TreeModel::learnStructure(), restricted_tournament_replacement(), and tournament_selection().

```
61 {
62 //   double r=drand();
63 //   printf("r=%f\n",r);
64   return (int) (drand()*max);
65 };
```

5.15.3.5 long longRand (long *max*)

Definition at line 79 of file random.cpp.

References drand().

```
80 {
81    return (long) ((double) drand()*max);
82 };
```

5.15.3.6 unsigned long setSeed (unsigned long *newSeed*)

Definition at line 116 of file random.cpp.

References init_genrand().

Referenced by parse_input_file().

```
117 {
118   // set the seed and return the result of the operation
119
120   init_genrand(newSeed);
121
122   return newSeed;
123 };
```

5.16 stats.cpp File Reference

Functions for computing basic population statistics.

```
#include "obj-function.hpp"
```

```
#include "stats.hpp"
```

Functions

- int compute_population_statistics (int **x, double *f, int n, int N, int *num_vals, PopulationStatistics *stats)

  Compute basic population statistics like the maximum, mean, etc.

- void init_population_statistics (PopulationStatistics *stats)

  Initialize the population statistics structure.

- int average_population_statistics (AveragePopulationStatistics *avg, PopulationStatistics *stats, int num_runs)

  Compute averages of population statistics over multiple runs.

5.16.1 Detailed Description

Functions for computing basic population statistics.

Definition in file stats.cpp.

5.16.2 Function Documentation

5.16.2.1 int average_population_statistics (AveragePopulationStatistics * *avg*, PopulationStatistics * *stats*, int *num_runs*)

Compute averages of population statistics over multiple runs.

Definition at line 74 of file stats.cpp.

References AveragePopulationStatistics::avg_avgF, AveragePopulationStatistics::avg_maxF, AveragePopulationStatistics::avg_minF, AveragePopulationStatistics::avg_num_evals, PopulationStatistics::avgF, AveragePopulationStatistics::max_maxF, PopulationStatistics::maxF, AveragePopulationStatistics::min_minF, PopulationStatistics::minF, PopulationStatistics::num_evals, AveragePopulationStatistics::num_runs, AveragePopulationStatistics::population_size, PopulationStatistics::population_size, AveragePopulationStatistics::problem_size, and PopulationStatistics::problem_size.

Referenced by do_runs().

```
75 {
76   // initialize the average stats
77
78   avg->avg_minF=stats[0].minF;
79   avg->avg_maxF=stats[0].maxF;
80   avg->avg_avgF=stats[0].avgF;
81   avg->avg_num_evals=stats[0].num_evals;
82
```

```
83   avg->min_minF=stats[0].minF;
84   avg->max_maxF=stats[0].maxF;
85
86   avg->population_size=stats[0].population_size;
87   avg->problem_size=stats[0].problem_size;
88   avg->num_runs=num_runs;
89
90   // do the homework
91
92   for (int run=1; run<num_runs; run++)
93     {
94       avg->avg_minF+=stats[run].minF;
95       avg->avg_maxF+=stats[run].maxF;
96       avg->avg_avgF+=stats[run].avgF;
97       avg->avg_num_evals+=stats[run].num_evals;
98
99       if (stats[run].minF<avg->min_minF)
100         avg->min_minF=stats[run].minF;
101
102       if (stats[run].maxF>avg->max_maxF)
103         avg->max_maxF=stats[run].maxF;
104     }
105
106   avg->avg_minF/=num_runs;
107   avg->avg_maxF/=num_runs;
108   avg->avg_avgF/=num_runs;
109   avg->avg_num_evals/=num_runs;
110
111   // return the number of processed runs
112
113   return num_runs;
114 }
```

5.16.2.2 int compute_population_statistics (int ** *x*, double * *f*, int *n*, int *N*, int * *num_vals*, PopulationStatistics * *stats*)

Compute basic population statistics like the maximum, mean, etc.

Definition at line 15 of file stats.cpp.

References PopulationStatistics::avgF, PopulationStatistics::idx_maxF, PopulationStatistics::idx_minF, PopulationStatistics::maxF, PopulationStatistics::minF, PopulationStatistics::population_size, and PopulationStatistics::problem_size.

Referenced by one_run().

```
16 {
17   // store population size and problem size
18
19   stats->population_size=N;
20   stats->problem_size=n;
21
22   // compute basic fitness statistics
23
24   stats->idx_minF = stats->idx_maxF = 0;
25   stats->minF = stats->maxF = stats->avgF = f[0];
26
27   for (int i=1; i<N; i++)
28     {
29       stats->avgF+=f[i];
30
31       if (f[i]>stats->maxF)
32         {
33           stats->maxF=f[i];
```

```
34           stats->idx_maxF=i;
35         }
36       else
37         if (f[i]<stats->minF)
38           {
39             stats->minF=f[i];
40             stats->idx_minF=i;
41           }
42     }
43
44   stats->avgF/=N;
45
46   // check whether we found the optimum
47
48   if (is_optimal(x[stats->idx_maxF],n,num_vals))
49     stats->success=1;
50
51   // get back
52
53   return 0;
54 }
```

5.16.2.3 void init_population_statistics (PopulationStatistics * *stats*)

Initialize the population statistics structure.

Definition at line 61 of file stats.cpp.

References PopulationStatistics::avgF, PopulationStatistics::idx_maxF, PopulationStatistics::idx_minF, PopulationStatistics::maxF, PopulationStatistics::minF, PopulationStatistics::num_evals, and PopulationStatistics::success.

Referenced by one_run().

```
62 {
63   stats->num_evals=0;
64   stats->minF=stats->maxF=stats->avgF=0;
65   stats->idx_minF=stats->idx_maxF=-1;
66   stats->success=0;
67 }
```

5.17 stats.hpp File Reference

Header file for stats.cpp.

Data Structures

- struct PopulationStatistics

  Basic population statistics (single run).

- struct AveragePopulationStatistics

  Population statistics for a set of runs.

Functions

- void init_population_statistics (PopulationStatistics *stats)

  Initialize the population statistics structure.

- int compute_population_statistics (int **x, double *f, int n, int N, int *num_vals, PopulationStatistics *stats)

  Compute basic population statistics like the maximum, mean, etc.

- int average_population_statistics (AveragePopulationStatistics *avg, PopulationStatistics *stats, int num_runs)

  Compute averages of population statistics over multiple runs.

5.17.1 Detailed Description

Header file for stats.cpp.

Definition in file stats.hpp.

5.17.2 Function Documentation

5.17.2.1 int average_population_statistics (AveragePopulationStatistics * *avg*, PopulationStatistics * *stats*, int *num_runs*)

Compute averages of population statistics over multiple runs.

Definition at line 74 of file stats.cpp.

References AveragePopulationStatistics::avg_avgF, AveragePopulationStatistics::avg_maxF, AveragePopulationStatistics::avg_minF, AveragePopulationStatistics::avg_num_evals, PopulationStatistics::avgF, AveragePopulationStatistics::max_maxF, PopulationStatistics::maxF, AveragePopulationStatistics::min_minF, PopulationStatistics::minF, PopulationStatistics::num_evals, AveragePopulationStatistics::num_runs, PopulationStatistics::population_size, AveragePopulationStatistics::population_size, PopulationStatistics::problem_size, and AveragePopulationStatistics::problem_size.

Referenced by do_runs().

```
75 {
76   // initialize the average stats
77
78   avg->avg_minF=stats[0].minF;
79   avg->avg_maxF=stats[0].maxF;
80   avg->avg_avgF=stats[0].avgF;
81   avg->avg_num_evals=stats[0].num_evals;
82
83   avg->min_minF=stats[0].minF;
84   avg->max_maxF=stats[0].maxF;
85
86   avg->population_size=stats[0].population_size;
87   avg->problem_size=stats[0].problem_size;
88   avg->num_runs=num_runs;
89
90   // do the homework
91
92   for (int run=1; run<num_runs; run++)
93     {
94       avg->avg_minF+=stats[run].minF;
95       avg->avg_maxF+=stats[run].maxF;
96       avg->avg_avgF+=stats[run].avgF;
97       avg->avg_num_evals+=stats[run].num_evals;
98
99       if (stats[run].minF<avg->min_minF)
100         avg->min_minF=stats[run].minF;
101
102       if (stats[run].maxF>avg->max_maxF)
103         avg->max_maxF=stats[run].maxF;
104     }
105
106   avg->avg_minF/=num_runs;
107   avg->avg_maxF/=num_runs;
108   avg->avg_avgF/=num_runs;
109   avg->avg_num_evals/=num_runs;
110
111   // return the number of processed runs
112
113   return num_runs;
114 }
```

5.17.2.2 int compute_population_statistics (int ** *x*, double * *f*, int *n*, int *N*, int * *num_vals*, PopulationStatistics * *stats*)

Compute basic population statistics like the maximum, mean, etc.

Definition at line 15 of file stats.cpp.

References PopulationStatistics::avgF, PopulationStatistics::idx_maxF, PopulationStatistics::idx_minF, PopulationStatistics::maxF, PopulationStatistics::minF, PopulationStatistics::population_size, and PopulationStatistics::problem_size.

Referenced by one_run().

```
16 {
17   // store population size and problem size
18
19   stats->population_size=N;
20   stats->problem_size=n;
21
22   // compute basic fitness statistics
23
24   stats->idx_minF = stats->idx_maxF = 0;
25   stats->minF = stats->maxF = stats->avgF = f[0];
```

```
26
27   for (int i=1; i<N; i++)
28     {
29       stats->avgF+=f[i];
30
31       if (f[i]>stats->maxF)
32         {
33           stats->maxF=f[i];
34           stats->idx_maxF=i;
35         }
36       else
37         if (f[i]<stats->minF)
38           {
39             stats->minF=f[i];
40             stats->idx_minF=i;
41           }
42     }
43
44   stats->avgF/=N;
45
46   // check whether we found the optimum
47
48   if (is_optimal(x[stats->idx_maxF],n,num_vals))
49     stats->success=1;
50
51   // get back
52
53   return 0;
54 }
```

5.17.2.3 void init_population_statistics (PopulationStatistics * *stats*)

Initialize the population statistics structure.

Definition at line 61 of file stats.cpp.

References PopulationStatistics::avgF, PopulationStatistics::idx_maxF, PopulationStatistics::idx_minF, PopulationStatistics::maxF, PopulationStatistics::minF, PopulationStatistics::num_evals, and PopulationStatistics::success.

Referenced by one_run().

```
62 {
63   stats->num_evals=0;
64   stats->minF=stats->maxF=stats->avgF=0;
65   stats->idx_minF=stats->idx_maxF=-1;
66   stats->success=0;
67 }
```

5.18 status.cpp File Reference

Methods of class Status, which displays text-mode status bars in verbose mode.

```
#include "status.hpp"
#include <stdio.h>
```

5.18.1 Detailed Description

Methods of class Status, which displays text-mode status bars in verbose mode.

Definition in file status.cpp.

5.19 status.hpp File Reference

Header file for status.cpp.

Data Structures

- class Status

  Text-mode status bar used in verbose mode in learning algorithms.

5.19.1 Detailed Description

Header file for status.cpp.

Definition in file status.hpp.

5.20 tree-model.cpp File Reference

Methods of class TreeModel for working with tree probabilistic models.

```
#include "heap.hpp"
#include "random.hpp"
#include "status.hpp"
#include "tree-model.hpp"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

5.20.1 Detailed Description

Methods of class TreeModel for working with tree probabilistic models.

Definition in file tree-model.cpp.

5.21 tree-model.hpp File Reference

Header file for tree-model.cpp.

```
#include <stdio.h>
```

Data Structures

- class TreeModel

  Class for storing, creating, and sampling the tree models.

5.21.1 Detailed Description

Header file for tree-model.cpp.

Definition in file tree-model.hpp.

Chapter 6

Dependency-Tree Estimation of Distribution Algorithm Example Documentation

6.1 example-num-values.cpp

An additional example of specifying the alphabets for all string positions. Here the first half of the symbols are binary whereas the remaining symbols can contain values from 0 to (n-1) where n is the string length. By default, a function that specifies all attributes as binary is included in the code.

```
1 void set_num_vals(int *num_vals, int n)
2 {
3   // set the number of values for the first half of the string to 2,
4   // while in the remaining positions the number of symbols is n.
5
6   for (int i=0; i<n/2; i++)
7     num_vals[i]=2;
8   for (int i=n/2+1; i<n; i++)
9     num_vals[i]=n;
10 }
```

6.2 example-trap5.cpp

An additional example of the objective function. Here a concatenated trap of order 5 is used, which is fully deceptive. This function can be used for binary strings where the number of bits is an integer multiple of 5. This function should not be used for non-binary representations.

```
1 double objective_function(int *x, int n)
2 {
3   int val=0;
4
5   // verify the string length constraint
6
7   if (n%5!=0)
8     {
9       printf("Trap function requires string length divisible by 5.\n");
10       exit(-55);
11     }
12
13   // add up all the traps
14
15   for (int i=0; i<n;)
16     {
17       int sum=0;
18
19       // compute the sum of bits in the next 5 bits
20
21       for (int k=0; k<5; k++)
22         sum+=x[i++];
23
24       // compute the trap-5 value for the computed sum
25
26       if (sum==5)
27         val+=5;
28       else
29         val+=4-sum;
30     }
31
32   // return the result
33
34   return val;
35 }
```

6.3 example_input

An example parameter file for the decision tree EDA.

```
1 population_size    = 800
2 problem_size       = 100
3 max_generations    = 100
4 replacement        = 0
5 tournament_size    = 2
6
7 bisection          = 0
8 num_bisection_runs = 0
9
10 quiet_mode         = 0
11 verbose_mode       = 0
```

6.4 example_input_big

An example parameter file that solves a big problem and has verbose output switched on.

```
1 population_size    = 3300
2 problem_size       = 200
3 max_generations    = 125
4 replacement        = 0
5 tournament_size    = 2
6
7 bisection          = 0
8 num_bisection_runs = 0
9
10 quiet_mode         = 0
11 verbose_mode       = 1
```

6.5 example_input_bisection

An example parameter file that uses bisection to determine the optimal population size for 10 successful runs.

```
1 population_size    = 100
2 problem_size       = 40
3 max_generations    = 40
4 replacement        = 0
5 tournament_size    = 2
6
7 bisection          = 1
8 num_bisection_runs = 10
9
10 quiet_mode         = 0
11 verbose_mode       = 0
```