```
1  /*
2  This file is part of Ionlib.  Copyright (C) 2016  Tim Sweet
3
4  Ionlib is free software : you can redistribute it and / or modify
5  it under the terms of the GNU General Public License as published by
6  the Free Software Foundation, either version 3 of the License, or
7  (at your option) any later version.
8
9  Ionlib is distributed in the hope that it will be useful,
10 but WITHOUT ANY WARRANTY; without even the implied warranty of
11 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.See the
12 GNU General Public License for more details.
13
14 You should have received a copy of the GNU General Public License
15 along with Ionlib.If not, see <http://www.gnu.org/licenses/>.
16 */
17 #ifndef ION_GENETIC_ALGORITHM_H_
18 #define ION_GENETIC_ALGORITHM_H_
19 #include <vector>
20 #include <algorithm>
21 #include <numeric>
22 #include "ionlib/math.h"
23 #include "ionlib/log.h"
24 namespace ion
25 {
26     class GeneticAlgorithm
27     {
28     public:
29         GeneticAlgorithm(size_t num_members, size_t chromosome_length, double
              mutation_probability, double crossover_probability)
30         {
31             if (chromosome_length > RAND_MAX || num_members > RAND_MAX)
32             {
33                 LOGFATAL("Your chromosome or population size is greater than
                      RAND_MAX, so the random number generator will never select some
                       members");
34             }
35             mutation_probability_ = mutation_probability;
36             crossover_probability_ = crossover_probability;
37             population_.reserve(num_members);
38             fitness_.resize(num_members);
39             num_evaluations_ = 0;
40             for (uint32_t member_index = 0; member_index < num_members; +
                  +member_index)
41             {
42                 std::vector<bool> member;
43                 member.reserve(chromosome_length);
44                 for (uint32_t gene_index = 0; gene_index < chromosome_length; +
                      +gene_index)
45                 {
46                     member.push_back(ion::randlf(0.0, 1.0) > 0.5);
47                 }
```

```
48                  population_.push_back(member);
49              }
50          }
51          void Mutate()
52          {
53              //we start with the second element because we are doing elite selection
54              for (std::vector<std::vector<bool>>::iterator member_it = population_.begin()+1; member_it != population_.end(); ++member_it)
55              {
56                  for (std::vector<bool>::iterator gene_it = member_it->begin(); gene_it != member_it->end(); ++gene_it)
57                  {
58                      //generate a random number between 0 and 1
59                      double random_number = ion::randlf(0.0, 1.0);
60                      if (random_number < mutation_probability_)
61                      {
62                          *gene_it = !*gene_it;
63                      }
64                  }
65              }
66          }
67          void Select()
68          {
69              //note that the fitnesses must already be set
70              //get the total fitness
71              double fitness_sum = std::accumulate(fitness_.begin(), fitness_.end(), 0.0);
72              //create a temporary population
73              std::vector<std::vector<bool>> temp_population;
74              temp_population.reserve(population_.size());
75              //since we are using elite selection, push the elite member
76              temp_population.push_back(GetEliteMember());
77              //start selecting elements by treating the fitness as cumulative density function
78              for (uint32_t member_index = 1; member_index < population_.size(); ++member_index)
79              {
80                  //get a number between 0 and fitness_sum
81                  double selected_individual = ion::randlf(0.0, fitness_sum);
82                  //traverse the CDF until selected_individual is found
83                  std::vector<double>::iterator parent_it;
84                  uint32_t parent_index = 0;
85                  for (parent_it = fitness_.begin(); parent_it != fitness_.end(); ++parent_it,++parent_index)
86                  {
87                      selected_individual -= *parent_it;
88                      if (selected_individual < 0.0000000001)
89                      {
90                          break;
91                      }
92                  }
93                  LOGASSERT(selected_individual <= 0.0000000001);
```

```
94                  //now parent_it is the member that is getting propogated to the
                      next generation
95                  temp_population.push_back(*(population_.begin()+parent_index));
96                  //if this iteration is an odd number (that is, we have pushed an
                      even number of elements onto the queue) attempt crossover on
                      these two members
97                  if (member_index % 1 == 1)
98                  {
99                      double random_number = ion::randlf(0.0, 1.0);
100                     if (random_number < crossover_probability_)
101                     {
102                         //we will do crossover
103
104                         //select a point to start the crossover at
105                         uint32_t crossover_location = (uint32_t)ion::randull(0,
                        (uint32_t)population_.begin()->size() - 1);
106                         //perform the crossover
107                         //get the last two members
108                         std::vector<std::vector<bool>>::reverse_iterator mate1 =
                        temp_population.rbegin();
109                         std::vector<std::vector<bool>>::reverse_iterator mate2 =
                        mate1 + 1;
110                         std::swap_ranges(mate1->begin(), mate1->begin() +
                        crossover_location, mate2->begin());
111                     }
112                 }
113             }
114             population_.swap(temp_population);
115         }
116         void NextGeneration()
117         {
118             //do fitness proportional selection
119             Select();
120             Mutate();
121             EvaluateMembers();
122         }
123         double GetAverageFitness()
124         {
125             return std::accumulate(fitness_.begin(),fitness_.end(),0.0)/
                  fitness_.size();
126         }
127         double GetMaxFitness()
128         {
129             return *(std::max_element(fitness_.begin(), fitness_.end()));
130         }
131         std::vector<bool> GetEliteMember()
132         {
133             std::vector<double>::iterator elite_it = std::max_element
                  (fitness_.begin(), fitness_.end());
134             return population_[elite_it - fitness_.begin()];
135         }
136         double GetMinFitness()
137         {
```

```
138             return *(std::min_element(fitness_.begin(), fitness_.end()));
139         }
140         std::vector<bool> GetWorstMember()
141         {
142             size_t worst_index = std::min_element(fitness_.begin(), fitness_.end
                  ()) - fitness_.begin();
143             return population_[worst_index];
144 
145         }
146         uint32_t GetNumEvals()
147         {
148             return num_evaluations_;
149         }
150         virtual void EvaluateMembers() = 0;
151     protected:
152         std::vector<std::vector<bool>> population_;
153         std::vector<double> fitness_;
154         double mutation_probability_;
155         double crossover_probability_;
156         uint32_t num_evaluations_;
157     };
158 
159 };
160 #endif //ION_FILE_H_ION_GENETIC_ALGORITHM_H_
```