Assignment 6: Evolution In Action (Pair)
Due: All Written and Coding 1–2: 11:59pm, Tuesday, February 28
Final Design.txt and Coding 3–4: 11:59pm, Tuesday March 7

Questions about this assignment can be sent to cs70help@cs.hmc.edu.

Background — Genetic Algorithms

Genetic algorithms attempt to solve difficult problems by “evolving” a good solution. In real life, natural selection is the process by which members of a species that are well-suited to their environment tend to survive and breed and those that are ill-suited to that environment fare less well and are less likely to procreate. Future generations are made from the genes of those members of the species that are most fit for the environment in which that species lives. A genetic algorithm applies the mechanism of natural selection using a “fitness function” chosen by the programmer. For example, a simple fitness function might interpret the genes of an organism as the value of $x$ in a complicated equation. The natural-selection process could then be tuned to prefer organisms that generate an output near zero, so that the survivors would eventually produce a solution to the equation.

Genetic algorithms were the first step in the current research area called “artificial life”, and they have been used to successfully solve many otherwise intractable problems.

There are three basic processes in evolution: crossover, mutation and selection. Crossover involves taking copies of the genes of two parent organisms and splicing them together to form a child organism. Usually a randomly chosen genetic subsequence from one parent replaces the equivalent subsequence from the other parent. The crossover function lies at the heart of a genetic algorithm—a properly written crossover function will have some possibility of capturing good traits from both parents, but it does so in complete ignorance of whether one gene sequence or subsequence is good or bad. It is quite likely that a child will actually be less fit than either of its parents.

After crossover has produced a new organism, there is a random chance that this new organism will undergo a mutation. Mutation involves selecting a gene site and modifying it in some fashion, usually by replacing it with another gene. Mutation is very rare in real life, and in a genetic algorithms it should also occur rarely, otherwise too many fit organisms will be lost to harmful mutations.

The final step, selection, involves evaluating the organisms according to some criterion (the “fitness function”) and choosing the ones that satisfy this criterion most successfully. In real life, selection is the harsh process of “survival of the fittest”. In a genetic algorithm, the same method is used: the least fit organisms are discarded (i.e., “killed”) without being allowed to reproduce. The fit organisms get a chance to reproduce before they...
As in real life, there is some randomness, so that a somewhat unfit organism has a chance of surviving to reproduce even though that may mean that a more fit organism is discarded. This randomness turns out to be important to the success of the method, because any two slightly unfit parents might (through crossover) generate an extremely fit child.

Scenario

To pay some bills over winter break, you’re working at Lamarkian Enterprises, a bold new startup company whose eventual goal is to make programmers redundant by evolving algorithms to solve all the hardest problems in computer science. Their lead programmer, Dakota Winter, began exploring the problem space by writing a genetic algorithm to solve some “classic” graph problems, including the Traveling Salesrep Problem. However, Dakota is no longer with the company, due to an unfortunate incident better not talked about.

Management looked into what Dakota had left behind, and although there was no documentation file with the code, there were two printed pages in Dakota’s cubicle (which have been reproduced on the following two pages) that give some description of the program, natselpath, that Dakota was working on.

Management took a look at Dakota’s code and discovered it had no Makefile and that almost all of Dakota’s code resided in a single file. Worse, the code defines very few C++ classes, choosing instead to implement most of its functionality via top-level functions, even though there are some obvious ways in which the code could be broken into classes.

In addition, one particularly pointy-haired manager expressed concern at Dakota’s extensive use of the C++ Standard Template Library (STL); he had heard a rumor from his cousin-in-law’s auto mechanic that the STL is inefficient, especially its list class template. According to this manager, list is wasteful because it uses templates and, worse, list is implemented as a doubly linked list while all natselpath needs is a singly-linked list.

Your manager has handed you Dakota’s code to clean up. You will have to

- Create a Makefile for the program
- Write an IntList class (storing a singly linked list of integers) that can replace the uses of list<int> in the code
- Factor out logical components of the code into separate classes
The natselpath Program

The natselpath program attempts to solve two common graph problems, shortest Hamiltonian path (SHP) and the shortest tour problem (also known as the traveling salesrep problem (TSP)).

Shortest Hamiltonian path takes a set of nodes and internode distances (see the example in Figure 1(a) and (b)), and tries to find the shortest path that includes all the nodes exactly once (we do not require any particular start or end point). Figure 1(c) shows the shortest Hamiltonian path for our example, and, for contrast, Figure 1(d) shows the longest Hamiltonian path. The shortest tour problem is analogous, except that we return to our starting point.

The examples shown in Figure 1 are easy to solve using a “brute force” method where we evaluate every possible path through the graph and find the best one. In the example, there are only 6!/2 (i.e., 360) possible Hamiltonian paths, so it would be easy to check each one. But checking every possible path has exponential time complexity and quickly becomes intractable. For 36 nodes, there are 36!/2 possible paths (i.e., 185,996,663,394,950,608,733,999,724,075,417,600,000,000—if we could check one possibility every nanosecond, it would take about six septillion years to check them all).

These problems are NP-complete, which, in essence, means that every algorithm that guarantees to always find the correct answer to this problem is an exponential-time algorithm like the one above. Genetic algorithms like natselpath do not guarantee to always find the optimal solution, but they usually do converge on a good solution. Moreover, as a probabilistic algorithm, each run is different, and so even if you don’t get an optimal solution on the first try, you may do better if you run the program again.

In natselpath, different paths through the graph constitute the different organisms in the ecosystem. The organisms’ “DNA” is the path through the graph represented as a list of integers: an organism representing the winning path E, B, F, A, D, C (Figure 1(c)) would be represented as a list 4, 1, 5, 0, 3, 2.

Initially, the colony of organisms have DNA that corresponds to random paths, but their fitness is based on how short they are. Organisms representing shorter paths survive and breed with other “shorter” organisms. Over time the traits that make an organism good are reinforced and bad traits eliminated.
Figure 1: Path problems solved by natselpath

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>66</td>
<td>87</td>
<td>76</td>
<td>98</td>
<td>40</td>
</tr>
<tr>
<td>B</td>
<td>66</td>
<td>0</td>
<td>144</td>
<td>100</td>
<td>45</td>
<td>44</td>
</tr>
<tr>
<td>C</td>
<td>87</td>
<td>144</td>
<td>0</td>
<td>70</td>
<td>158</td>
<td>101</td>
</tr>
<tr>
<td>D</td>
<td>76</td>
<td>100</td>
<td>70</td>
<td>0</td>
<td>97</td>
<td>58</td>
</tr>
<tr>
<td>E</td>
<td>98</td>
<td>45</td>
<td>158</td>
<td>97</td>
<td>0</td>
<td>61</td>
</tr>
<tr>
<td>F</td>
<td>40</td>
<td>44</td>
<td>101</td>
<td>58</td>
<td>61</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) Nodes & distances (graphically)  (a) Nodes & distances (matrix)

(c) Shortest Hamiltonian path
(d) Longest Hamiltonian path
(e) Shortest tour
(f) Longest tour

Current Data Structures

The program currently uses the following data structures:
Organism

An organism will be represented entirely by its gene sequence. Each element in the sequence will contain only a single integer from 0 to 9. In the current implementation of the code the gene sequence is represented using the type list<int>. The Organism type is simply a synonym for list<int> to make the code more readable.

Organism::iterator and Organism::const_iterator

At several places in the code, the program needs to cycle through the genes of the organism. The code uses Organism::iterator and Organism::const_iterator to perform these iteration tasks. In the current code, these types are equivalent to list<int>::iterator and list<int>::const_iterator, respectively.

Colony

The Colony type is used to represent the population of organisms. It is simply a container type that holds Organism objects—in the current code, Colony is a synonym for vector<Organism>. Like the organism type, it currently has iterators.

Before You Begin

Your tsp subdirectory will contain the following files:

- Answers.txt
- Design.txt
- natselpath-private.hpp
- natselpath.cpp
- data/abcdef.labels
- data/abcdef.matrix
- data/ca_cities.labels
- data/ca_cities.matrix
- data/uk_cities.labels
- data/uk_cities.matrix
- data/us_cities.labels
- data/us_cities.matrix

Written Component

When answering the following questions, explain your answers clearly. Your answers should be placed in the file Answers.txt.

Several of the questions require you to make small changes to the code. In the coding component you will have to create a Makefile, but if you do not wish to do that yet, you can compile the code manually without a Makefile by executing

```
g++ -g -Wall -Wextra -pedantic -O2 -o natselpath natselpath.cpp
```

1The -O2 is “oh two”, not “zero two”.

5
(This command line includes the compiler option -O2 to turn on optimization, which is useful for doing the performance comparison questions. In general, you should not use optimization while you are actively doing development—it makes compilation slower and debugging with gdb more sketchy.)

W1. On Knuth, compile the code (with optimization) and then run the command

```
/usr/bin/time ./natselpath -d -S 1 data/us_cities
```

(a) What does this command line mean? (You can find out about the time command by typing man time, and about the arguments to the natselpath program by looking at the initial comments in natselpath.cpp.)

(b) What output does it produce?

W2. Comment out line 71 in natselpath.cpp, which reads

```cpp
typedef list<int> Organism;
```

and replace it by each of the following (in turn):

(a) `typedef list<char> Organism;`

(b) `typedef vector<int> Organism;`

(c) `typedef vector<char> Organism;`

(d) `typedef string Organism;`

For each of the parts above, recompile the code (with optimization) and report the results from running

```
time ./natselpath -d -S 1 data/us_cities
```

(When you are done, revert the definition of Organism to list<int>.)

W3. Lists and arrays have quite different representations. Why can we change Organism from being a synonym for list<int> to a synonym for vector<char> or string without causing compiler errors?

W4. The code uses a functor. What is a functor? Where is it created? (See Stroustrup, Section 18.4 for a discussion of functors.)

W5. Explain how findBest works (i.e., find out what the STL algorithm min_element is used for and explain why it is the right tool for this problem).

W6. There is a comment at the top of the naturalSelection function explaining that early versions of the code used sort, later versions used partial_sort, and the current version uses nth_element.
(a) Highlight the similarities and differences between the STL algorithms sort, partial_sort, and nth_element. (You will almost certainly need to look these functions up, either in Chapter 18 of Stroustrup or on the web—see the resources page on the class website.)

(b) Explain why the code evolved from sort to partial_sort to nth_element, including why the code is still correct and whether any efficiency gains are likely.

(c) Modify the code to use each of the other methods in turn and run the code to determine whether each change improves performance in practice. Summarize your findings.

W7. In the separate file Design.txt, list the two or more new classes you would recommend adding to the program — see Question C3 below. Describe the operations that each class will support. For the first deadline, make your best attempt; you can (and should!) revise it for the second deadline. the March 28 submission, (do not write code or header files for these classes for the first deadline, unless everything else is completely done!)

**Coding Component**

These coding questions are designed so that you can submit your code after completing each question. You will, however, only be graded on your final submission.

C1. Create a Makefile for your code, and add it to your CS 70 repository. (You will need to keep the Makefile up to date as you make changes.)

C2. Create the files intlist.hpp and intlist.cpp, implementing an IntList class and add them to your CS 70 repository. You should write your IntList class such that if the definition of the Organism type is changed to

```cpp
#include "intlist.hpp"

typedef IntList Organism;
```

the program will still compile, run, and produce the same output as before.

The overall structure of your IntList class should be similar to the StringStack class seen during lectures (which you may use as a starting point). Thus, there will be three classes involved, IntList, IntList::Node, and IntList::Iterator. However, unlike the stack class, you will need both a head and a tail pointer in the header (in order to support push_back efficiently).

Your linked-list class must be named IntList and must support the following operations:

- A default constructor.
• A swap operation.
• A copy constructor.
• A destructor. The destructor must clean up properly; in other words, it must empty the list.
• An assignment operator.
• A push_back function that inserts a single integer at the tail of the list. This function should be declared as
  
  void push_back(int value);

Your push_back function must take constant time (e.g., not have its running time increase for longer lists).
• Two typedefs, defining the types iterator and const_iterator as synonyms for Iterator.²
• Two inner classes, Iterator and Node, where Node is private.
• A begin function that returns an iterator that refers to the start of the list. This function should be declared as
  
  iterator begin() const;

• An end function that returns an invalid/past-the-end iterator,
  
  iterator end() const;

• An equality test (operator==) and an inequality test (operator!=).

Note: You may not change the function and type names given above. You may, however, write additional member functions such as push_front, pop_front, front, back, and reverse if you wish. (If you have written your list class correctly, it will not be not possible to write pop_back efficiently, so there is little point in supplying this function.)

The IntList::Iterator class must provide at least the following operations:

• A copy constructor
• An assignment operator
• A destructor
• An equality test (operator==) and an inequality test (operator!=)
• A preincrement operator (operator++)

²In an industrial-strength implementation, we would define separate types for iterator and const_iterator, but in this assignment doing so is more trouble than it is worth. As a result, people will be able modify objects even when they only have a const_iterator.
• An operator* that returns an int& (so that the integer in the current position can be modified if necessary)
• The five required typedefs to make the iterator STL-friendly.

Compile and test your code thoroughly. Your IntList class will be tested separately, so it is important that it works correctly.\(^3\)

After you have created and tested your IntList class, you can either continue to use it in the genetic-algorithm code or switch that code back to using the STL’s list<int> type. Using your IntList may result in more comprehensible error messages while you are working on coding the next part, whereas using list<int> will insulate your later code from any bugs in your list class.

C3. The code that has been provided to you uses many top-level functions and defines almost no classes. It could be argued that the code only provides the NatSelEnv class because it is very convenient to pass a function object to the STL algorithms.

Examine the code to discover the logical relationships between the functions and then break the code up into separate files and classes. You should create at least two new classes reflecting the logical structure of the program. (A solution involving four new classes is quite possible.)

Any new header and code files must be added to the Makefile and added to the subversion repository using svn add.

In your final code, you should find that the overall code looks simpler and is easier to follow. If you have done things properly many of the functions will have fewer arguments.

Note that the execution behavior of your code must be exactly the same as the existing code—this requirement means that you must take care when making changes involving the random-number generator to make sure that the same numbers are generated. (The random-number functions used by the code are described in the UNIX manual pages—man random).

C4. Extend your Design.txt file to accurately and fully describe your final design. Guidance on writing an adequate Design.txt file will be available on the course website at least a week before the due date.

---

\(^3\)You may find that your IntList class seems slower than the STL’s list<int> class, even if you have written clean, elegant, and efficient code. The reasons behind this slowdown are curious indeed—a small reward is available to any student who can discover why the slowdown occurs and whether it can be prevented.
Testing

Testing is your responsibility; we will not look at your test code.

In its default condition, the program is nondeterministic (i.e., two successive runs may produce different results, especially if you use the -d switch to see the program’s debugging output). To make testing easier, the program accepts a switch that makes it deterministic. If you use -S n, where n is an integer, the random seed will be set to that value. Specifying the random seed will allow you to control the program’s behavior so that you can reproduce bugs.

You will also find it instructive to run the program with the -d switch, and to run it for many different values of the -g, -m, -p, -r, and -s switches. Judicious reading of the comments, together with experimentation, will reveal the purpose of these switches and how they interact.

Sample Runs

To make it clearer how the program is used, here are some sample runs from executing natselpath on Turing (if you run the code on Mac OS X or Linux-based systems you will get similar but not identical results, because each operating system uses a different random number generator). Below are the results of planning a round trip through eighteen prominent US cities:

```
unix% ./natselpath -g 30 -S 1 data/us_cities
2 Boston (197)
12 New York City (235)
17 Washington D.C. (375)
4 Cleveland (173)
7 Detroit (288)
3 Chicago (410)
10 Minneapolis (625)
13 St. Louis (559)
1 Atlanta (653)
9 Miami (857)
11 New Orleans (509)
5 Dallas (654)
0 Albuquerque (457)
6 Denver (527)
14 Salt Lake City (703)
8 Los Angeles (384)
15 San Francisco (808)
16 Seattle

Total Distance: 8414
```
If we start with a different random seed, we get a different (worse) result:

```
unix% ./natselpath -g 30 -S 11 data/us_cities
2   Boston (197)
12  New York City (235)
17  Washington D.C. (525)
7   Detroit (173)
4   Cleveland (345)
3   Chicago (410)
10  Minneapolis (625)
13  St. Louis (559)
1   Atlanta (653)
9   Miami (857)
11  New Orleans (509)
5   Dallas (654)
0   Albuquerque (457)
6   Denver (527)
14  Salt Lake City (703)
8   Los Angeles (384)
15  San Francisco (808)
16  Seattle
```

Total Distance: 8621

(although if we removed the -g 30 option—and thus ran with the default of 50 generations—we would again get an answer of 8414 miles).

Finally, in addition to controlling the number of generations (-g), we can control the mutation rate (-m), the population size (-p), the percentage survival rate (-s, which should be smaller than the population size), and the percentage of the survivors that survive through luck rather fitness (-r), and run with debugging (-d). The command line

```
./natselpath -S 7 -d -s 2.5 -r 1 data/uk_cities
```

finds the shortest path through 36 British cities. These parameters (the tiny percentage survival rate) lead the algorithm to converge quickly on an answer, but the colony is prone to lacking genetic diversity, different seed values reveal that most runs do not converge on the best path of 2190 miles.

**Note:**

You can think of the running time of the program as being proportional to the population and to the number of generations. Don’t use huge numbers or you’ll wait all day! If you don’t specify the -S switch, you will get different results every time you run the program. That’s a feature, not a bug.
Tricky Stuff

As usual, there are some tricky parts to this assignment. Some of them are

- Be sure to read the code in natselpath.cpp before you start, so that you understand the requirements placed on the IntList and IntList::Iterator classes.

- Before you try to write the iterator, it would be wise to debug the list code itself. To help with that task, you will probably want to write test code. (Your test code will not be graded.)

- Be sure your IntList destructor, copy constructor, and assignment operator are working before you try to run the main program. If you don’t debug them in isolation, you will experience strange bugs that will be hard to find.

- Remember that the iterator access operator (operator*) must return an integer by reference (int&). Otherwise you won’t be able to get the mutation operator to work.

- Remember that push_back must run in constant time. You will be penalized if your code must traverse the entire list.

- Refactoring the code (breaking it up into separate files/classes) may seem daunting at first, but there are some fairly obvious lines you can draw between different parts of the code. Remember that you can plan things out in Design.txt or on paper first—you don’t have to try to keep all the details in your head.

- If you save a copy of the original executable, you’ll be able to test whether your revised version of the program always behaves the same way. Better yet, keep a copy of the entire original source.

- Work incrementally. It is hard to maintain exactly the same behavior. If you restructure the entire program and you get different output, it will be nearly impossible to track down the cause of the problem. If you make smaller changes and retest after each step, you’re much more likely to pinpoint problems.

- The behavior of the program is extremely sensitive to the random numbers it gets from the randomInt function. These random numbers in turn depend on the number of times randomInt has been called. If you have trouble matching the sample output or the behavior of the original code, check to see whether you’re creating scratch organisms that cause extra calls to randomInt.

If you’re still having problems, add extra code to print out a large random number (e.g., randomInt(10000)) at the same key points in both your code and the original version. If both programs have the same random seed, initially they’ll both print the same numbers. When they diverge, you’ll know that your code did something different from original code.
• If you believe there may be bugs in your IntList, you can temporarily use the STL’s list<int> and see whether the bug persists.

• The new classes you write need not exactly match the behavior of the types used in the current code. The operations you provide for each class need to be meaningful for that class. If, for example, you decided you needed a Television class, I would not be pleased to see it having a push_front() member function. Televisions don’t “push front”. However it is so very idiomatic to use iterators that televisions might supply begin and end if iterating over all channels is to be supported.

Similarly, if you were refactoring code that had the typedef

    typedef vector<Food> Cupboard;

and you had decided that Cupboard would be better implemented as a class, you would then need to separate the code that used Cupboards into two kinds, external code and internal code. External users of Cupboards don’t care about how Cupboards are represented internally and don’t try to do things like use array subscripting on them. Internal code implements the class, and does need to know where and what the underlying representation is (in this case, perhaps some data member vector<Food> shelves_.

When you refactor the code, the different uses of Cupboard need different operations. Code external to the class will use Cupboard operations, and only code internal to the class will be doing vector<Food> operations directly on the internal vector.

• The purpose of a C++ class is to generate objects, which encapsulate data with the appropriate operations. If you find yourself writing a class with no (non-constant) data members, it probably shouldn’t be a class! It’s just a bunch of related global functions that would be better grouped in their own file or namespace.