Lecture 22 - Assignment 3 Background

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

March 5, 2012

Midterm

- How did it go?
- Anything I should be aware of before starting grading?
- Should be handed back on Wednesday

- Also, we'll do Course Critiques on Wednesday, **please attend and tell everyone else to attend**

Assignment Problem

Solving the travelling salesman problem

- Given a list of cities and their pairwise distances
- What is the shortest tour that visits each city exactly one?
- The tour begins and ends at the first city

It is an NP-hard problem

- We'll use a genetic algorithm for this assignment

Introduction

- Basically it applies heuristics from genetics to make a bunch of random operations improve the answer
- Only has a few operations, allows it to solve other problems
    - Course scheduling, a host of other NP-hard problems
    - Seen it used to play games

Terms for Our Problem:

- **Individual** - a tour of cities (1, 4, 3, 2)
- **Population** - a collection of individuals

Pseudocode

```
create an initial population $pop(0) = X_1, ..., X_N$
$t \leftarrow 0$
repeat
    assign each $X_i$ in $pop(t)$ a probability $f(X_i)/(\sum f(X_i))$
    for i $\leftarrow$ 1 to N do
       a $\leftarrow$ random selection of individual from $pop(t)$
       b $\leftarrow$ random selection of individual from $pop(t)$
       child $\leftarrow$ reproduce(a, b)
       with small probability mutate child
       add child to $pop(t+1)$
   $t \leftarrow t + 1$
until stopping criteria
return most fit individual
```

Pseudocode Explained

That's it, the only operations we have are

- Creating an initial population
- Creating a fitness function (higher is better)
- Creating a selection function (this is not problem specific)
- Creating a crossover function (or reproduce)
- Creating a mutation function

We have two parameters, which we will keep constant

- **Population size** - 100
- **Mutation probability** - 1%

Initial Population

Note: we represent our tours as a sequence of the city indexes

- We just randomly shuffle the tours around for each individual

Example If we have a tour of 5 cities, we can randomly shuffle the base tour of 1, 2, 3, 4, 5

- 1, 3, 2, 5, 4
- 1, 5, 2, 4, 3
- 1, 4, 5, 2, 3

Remember, our tour begins and ends at the first city

Fitness Function

Basically, an evaluation of an individual in the population

- Most obvious metric is to just use the distance of the tour
    - In this case, lower is better
    - We need higher is better
- Therefore, we just subtract the **maximum distance in the population** by the individual's distance to get an individual's fitness

This means the individual with the highest distance in the population will have a fitness of 0

Selection Function

Intuition: have a better chance of picking more fit individuals

- Find the fitness of each indivudal, and normalize the values
    - The sum of all the normalized fitness values should equal 1
- Sort the population by descending fitness values
- Accumlate the normalized fitness values (cumulative values)
- Pick a random value, R, between 0 and 1
- Select the individual whose accumulated normalized value is greater than R

(also on Wikipedia and linked in the Assignment handout)

Selection Function Example

Tours (Distance) [Fitness]
P0: 1, 5, 2, 4, 3 (100) [0]
P1: 1, 4, 5, 2, 3 (80) [20]
P2: 1, 3, 2, 5, 4 (50) [50]
P3: 1, 2, 4, 5, 3 (70) [30]

- Normalize the values
  - P0: 0, P1: 0.2, P2: 0.5, P3: 0.3
- Sort the population by descending fitness values
  - P2: 0.5, P3: 0.3, P1: 0.2, P0: 0
- Accumlate the normalized fitness values (cumulative values)
  - P2: 0.5, P3: 0.8, P1: 1, P0: 1
- Pick a random value, R, between 0 and 1, then select the individual whose accumulated normalized value is greater than R
  - 0.4, therefore we pick P2

We can repeat the last point as many times as required

Crossover Function

This is how we combine two individuals to create another individual

- We will use a simple ordered
  - Select a random subtour from the first parent and copy it into the child (in order, all the cities at the same spot in the tour)
  - Copy the remaining cities, not already in the child, in the order they appear in the second parent

Exmaple:

P2: 1, 3, 2, 5, 4 (first parent) and P3: 1, 2, 4, 5, 3 (second parent)

- We select a subtour (elements 2 to 3) to copy to the child
  - 1, ?, 2, 5, ?
- Fill in the values from the second parent
  - 1, 4, 2, 5, 3

Mutation Function

This is how Xavier's School for Gifted Youngsters got started
... or not, we're just going to randomly change around a tour

- Select a random subtour
- Reverse the order of the subtour

Example:
1, 4, 2, 5, 3

- We pick elements 1 to 3 to reverse
  - 1, 5, 2, 4, 3

Pseudocode Again

```
create an initial population $pop(0) = X_1, ..., X_N$
$t \leftarrow 0$
repeat
    assign each $X_i$ in $pop(t)$ a probability $f(X_i)/(\sum f(X_i))$
    for i $\leftarrow$ 1 to N do
        a $\leftarrow$ random selection of individual from $pop(t)$
        b $\leftarrow$ random selection of individual from $pop(t)$
        child $\leftarrow$ reproduce(a, b)
        with small probability mutate child
        add child to $pop(t+1)$
    $t \leftarrow t + 1$
until stopping criteria
return most fit individual
```

Summary

That's it for our genetic algorithm, all of the operations here are things you **may not change** for this assignment

- The interface to the solver code is only:
  - Constructor call (will have the initial population)
  - Iteration calls (selection, crossover, mutate)
  - A call to get the best individual found

You may not change this interface, this means no attempting to parallelize calls to the iteration function (although you are free to try to parallelize the function itself)

Introduction

- The code is more or less a direct translation of the high level functions
- Written in C++11, so we have complete control with nice abstractions
    - You'll need the `-std=c++0x` flag for g++
- The language should not be the main hurdle, if there's anything you don't understand about the provided code, feel free to talk to me
- I only used standard library functions, link to documentation is in the handout
- Used typedef's so you should be able to change data structures if you want (you can use a string for an index instead of an unsigned int even)

Built-in Data Structures

- `vector` - basically an array
- `unordered_map` - basically a hash table
  - Other hash table structures like `unordered_set` (no associated values) may be useful
- `pair` - class with two elements, first and second with associated types
- `iterator` - abstraction for pointers with containers

Data Structures

- `distance_map` - a lookup table for distances between cities, implemented with a 2 dimensional hash map (distances calculated in constructor)
- `individual` - consists of a tour and a double, which may represent either distance, fitness, normalized fitness or accumulated fitness
  - `tour` - a `tour_container`, be default a vector of indexes
  - Does not include the first index, that is defined by `first_index`
  - `metadata` - a union to all doubles, since we don't need distance/fitness/etc. values all at the same time
- `population` - a vector of individuals
- `best_individual` - self explanatory

Functions

- `iteration` - performs one iteration of the genetic algorithm, it replaces `population` with a new population
    - Before iteration is called, it is assumed all individuals in the current population have a valid value of `distance` for its metadata
- `distance` - calculates the distance of a `tour`
- `selection` - returns 100 (population size) pairs of iterators to individuals in the current population to use for crossovers
- `crossover` - same as high-level explaination, returns a new individual
- `mutate` - same as high-level explainatoin, modifies an individual

Algorithms

All of the C++ built-in algorithms work with anything using the standard container interface

- `max_element` - returns an iterator the the largest element, you can use your own comparator function
- `min_element` - same as above, but the smallest element
- `sort` - sorts a container, you can use your own comparator function
- `upper_bound` - returns an iterator to the first element greater than the value, only works on a **sorted** container (if the default comparator isn't used, you have to use the same one used to sort the container)
- `random_shuffle` - does `n` random swaps of the elements in the container

Some Hurdles

If you've worked with C++ before, you probably know the awful compiler messages and pages of template expansions

- You can use `clang` if you have a compiler error, and let it show you the error instead `-std=c++11`
- For the profiler messages, it might get pretty bad, look for one of the main functions, or if it's a weird name, look where it's called from
- You can use Google Perf Tools to help break it down, more fine grained

Example Profiler Function Output

```
[32] std::_Hashtable<unsigned int, std::pair<unsigned int
    const, std::unordered_map<unsigned int, double, std::
    hash<unsigned int>, std::equal_to<unsigned int>, std::
    allocator<std::pair<unsigned int const, double> > > >,
    std::allocator<std::pair<unsigned int const, std::
    unordered_map<unsigned int, double, std::hash<unsigned
    int>, std::equal_to<unsigned int>, std::allocator<std::
    pair<unsigned int const, double> > > > >, std::
    _Select1st<std::pair<unsigned int const, std::
    unordered_map<unsigned int, double, std::hash<unsigned
    int>, std::equal_to<unsigned int>, std::allocator<std::
    pair<unsigned int const, double> > > > >, std::equal_to<
    unsigned int>, std::hash<unsigned int>, std::__detail::
    _Mod_range_hashing, std::__detail::_Default_ranged_hash,
     std::__detail::_Prime_rehash_policy, false, false, true
    >::clear()
```

is actually `distance_map.clear()`, which is automatically called by the destructor

Things You Can Do

Well, it's the most basic implemenation, so their should be a lot you can do

- You can introduce threads, using pthreads (or C++11 threads, although they're still missing a bunch), OpenMP, or whatever you want
- Play around with compiler options
- Use better algorithms or data strutures
- The list goes on and on

Things You Need to Do

Profile!

- Keep your number of iterations constant between all profiling results so they're comparable
- Baseline profile with no changes
- You will pick your two best performance changes to add to the report
  - You will include a profiling report before the change and just after the change (and only that change!)
  - More specific instructions in the handout
- There may or may not be overlapping between the baseline and the baseline for each change
- My recommendation: use your initial baseline as the "before" for your first change, and the "after" of the first change for the baseline of your second change
- Whatever you choose, it should be convincing

Things To Notice

- As you increase the number of iterations, your best answer should get better
- Therefore, the faster your program, the more iterations you can do and the better answer you should be able to get in 10 seconds
- The optimal answer is 7542
- Your program will be run on `ece459-1` (or equivalent), probably giving you 10 seconds to do as much work as you can
- We will have some type of leaderboard, so the eariler you have some type of submission, the better

A Word

This assignment should be **enjoyable**