ASSIGNMENT 3 BACKGROUND

ECE 459: Programming for Performance

Jon Eyolfson

March 6, 2015

University of Waterloo

Travelling Salesman Problem

PROBLEM STATEMENT

Given: a list of cities, with distances between each pair of cities

Goal: find the shortest route that visits each city exactly once and returns to the first city

GENETIC ALGORITHM

HIGH-LEVEL OVERVIEW

Generalizes heuristics from genetics

Describes how to apply different random operations to improve our answer

We only need to define a small number of operations that describe the problem

APPLICATIONS

- Course scheduling
- Many other NP-hard problems
- Playing games

TERMS

Individual	A tour of cities, e.g. 1, 4, 3, 2
Population	A collection of individuals
Distance	The total units of distance to complete a tour

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
```

OUR JOB

That’s it, the operations we have to define are:

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

There are also two parameters, which we will keep constant:

Population size	100
Mutation probability	1%

INITIAL POPULATION

We represent our tours as a sequence of the city indexes

- Start with a base tour which is all the cities in order, e.g. 1, 2, 3, 4, 5
- Randomly shuffle the tours around for each individual

ID	Tour
P0	1, 5, 2, 4, 3
P1	1, 4, 5, 2, 3
P2	1, 3, 2, 5, 4
P3	1, 2, 4, 5, 3

For consistency our tour begins and ends at the first city

FITNESS FUNCTION

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 and we need higher is better

1. Find the maximum distance in the population
2. Subtract a individual’s distance from that value to obtain the fitness

Therefore the individual with the highest distance has a fitness of 0

SELECTION FUNCTION

Intuition is to have a better chance of picking more fit individuals

Find the fitness of each individual, and normalize the values

- The sum of all the normalized fitness values should equal 1

Sort, accumulate, and pick random values, R, between 0 and 1

Select the individual whose accumulated normalized value is greater than R

SELECTION FUNCTION EXAMPLE

ID	Tour	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

1. Normalize the fitness values (P0 = 0, P1 = 0.2, P2 = 0.5, P3 = 0.3)
2. Sort population by descending values (P2 = 0.5, P3 = 0.3, P1 = 0.2, P0 = 0)
3. Accumulate the values (P2 = 0.5, P3 = 0.8, P1 = 1, P0 = 1)
4. Pick a random value, R, and select individual (if R = 0.4, pick P2)

Repeat step 4 for as many selections as you need

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

CROSSOVER FUNCTION EXAMPLE

ID	Tour
P2	1, 3, 2, 5, 4
P3	1, 2, 4, 5, 3

Suppose we chose P2 as the first parent and P3 as the second

1. Create a new child (C = 1, ?, ?, ?, ?)
2. Select a subtour from first parent to copy to child
   Suppose we chose indexes 2 to 3 (C = 1, ?, 2, 5, ?)
3. Fill in the missing values from the second parent in order (C = 1, 4, 2, 5, 3)

MUTATION FUNCTION

How we may change an individual based off no other factors

In our case we randomly change around a tour

MUTATION FUNCTION EXAMPLE

ID	Tour
C	1, 4, 2, 5, 3

1. Select a subtour and reverse the order
   Suppose we chose indexes 1 to 3 (C = 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 all the operations we need for our genetic algorithm

YOU MAY NOT CHANGE THEIR HIGH-LEVEL OPERATION

The interface to your algorithm is the following:

- A constructor call (will have the initial population)
- An iteration call, repeated (selection, crossover, mutate)
- A call to get the best individual found

YOU MAY NOT CHANGE THIS INTERFACE

Can't parallelize calls to iteration function, parallelize the iteration function itself

IMPLEMENTATION

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

- The language should not be the main hurdle, ask if anything is confusing

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

- `string` indexes, etc.

BUILT-IN DATA STRUCTURES

`vector`	basically an array
`unordered_map`	basically a hash table, key and associated value Hint: `unordered_set` (no associated values) may be useful
`pair`	class with two fields (`first` and `second`) with associated types
`iterator`	abstraction for pointers on containers

OUR DATA STRUCTURES

`distance_map` field `distances`, a lookup table for distances between cities, implemented with a 2D hash map (distances calculated in constructor)

`individual` field `best_individual`, consists of a tour and a metadata which may represent distance, fitness, normalized fitness or accumulated fitness

- `tour_container` field `tour`, a vector of indexes
  Note, does not include the first index, that is defined by field `first_index`
- `union` field `metadata`, a union to all doubles, since we don't need distance/fitness/etc. values all at the same time

`population_container` field `population`, a vector of individuals

OUR 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

BUILT-IN ALGORITHMS

All these standard C++ algorithms work with anything using the 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

TIPS

SOME HURDLES

Confusing C++ error messages involving templates

- Use `clang++` to compile your program for clearer error messages

For the profiler reports, it might get pretty bad

- Look for one of the main functions (selection, crossover, etc.)
- If the name is very weird, look where it's called from
- If you run into mangled names use `c++filt`

EXAMPLE PROFILER 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 implementation, so there should be a lot you can do

- Introduce threads, using pthreads, C++11 threads, OpenMP, etc.
- Experiment around with compiler options
- Use better algorithms or data structures

THINGS YOU NEED TO DO

Profile!

Keep the number of iterations constant between all profiles so they're comparable

Start with a baseline profile (no changes)

Pick your two best performance changes to add to the report

- Include a profile before and after the change (and only that change!)
- More specific instructions in the handout

You don't have to profile both changes from the baseline

THINGS TO NOTICE

As you increase the number of iterations, your best answer should get better

The faster your program, the more iterations you can do in the time limit

The optimal answer is 7542

Your program will be run on an unloaded equivalent of `ece459-1`

The scores on the leaderboard are how many iterations you can do in 10 seconds

GLHF