Genetic Algorithm in Ruby

Solving the Traveling Salesman Problem with Ruby
review: Genetic Algorithms
detour: Google Maps API, web services
overview of this week’s lab project

Review -- GA and TSP

- Genetic algorithms, inspired by concepts from population biology, are useful for optimization problems
  - make a population of solutions
  - apply “selection” to toss out poor solutions
  - make new solutions from survivors
  - use a combination of small changes (point mutations to a single solution) and larger changes (cross-overs that take large chunks from two solutions)
  - after several generations good solutions begin to emerge
- Genetic algorithms are an alternative to brute-force search when the “search space” is very large

Review (cont’d)

- The traveling salesman problem is an example of an optimization problem
- A genetic algorithm for the TSP:
  - select several potential tours at random
  - gradually improve the tours via point mutations and crossovers
  - depending on the problem domain and evolution parameters the process will eventually converge on a good (maybe best) solution
- Today:
  - a Ruby implementation of TSP
  - lab projects will use this program

Pac 10 Road Trip

- To data set for this lab is based on the cities of the Pac 10 universities
  - distances between each city supplied by maps.google.com
  - we’ll use driving time as the measure of cost

To data set for this lab is based on the cities of the Pac 10 universities
distances between each city supplied by maps.google.com
we’ll use driving time as the measure of cost
Google Map API

- I'm way too lazy and impatient to do 45 Google queries by hand
  - the complete graph for 10 nodes has \( \frac{10 \times 9}{2} = 45 \) links
- Google allows / encourages people to include maps on their own web sites
  - example: a business can put a map to its store on its web page
- API = “application program interface”
  - a protocol for getting information from Google
- Could I use the Google Maps API in a program to get the 45 distances automatically?

Google Maps URL

- After poking around on the web a bit I found the URL for driving instructions:
  - maps.google.com/maps?saddr=SRC\&daddr=DEST\&hl=en
  - where SRC is the starting point and DEST is the ending point
- Some things to note:
  - everything after ? in a URL is a parameter
  - \& separates parameters
  - this is just like calling a method in Ruby and passing parameters to it
    - e.g. s.append(“xxx”), or s.insert(3,”abc”)
  - change spaces to +, e.g. “Eugene,+OR”
- The URL for fetching instructions to get to Pullman from Eugene:
  - maps.google.com/maps?saddr=Eugene,+OR\&daddr=Pullman,+WA\&hl=en

Screen Scraping

- We can use Ruby’s HTTP library to fetch a map:
  - google = “maps.google.com”
  - path = “maps?saddr=Eugene,+OR\&daddr=Pullman,+WA\&hl=en”
  - resp = Net::HTTP.get_response(google,path)
- What comes back is the complete map page, but we can use regular expressions to extract the estimated driving time:
  - hours = resp.body[/\d+ hours/]  
  - mins = resp.body[/\d+ mins/] 
- Getting information from an HTML document that was intended to be viewed in a browser is called screen scraping
  - as the name implies it's not a very elegant solution...
Using Google to Get Distances

- If you’d like to see the code here is the main loop of the program that gets distances between cities:

  ```ruby
  cities = ["Seattle, WA", "Pullman, WA", "Corvallis, OR", ...]
  base = "maps.google.com/maps?saddr=SRC&daddr=DEST&hl=en"
  for i in 0..cities.length-2
    for j in i+1..cities.length-1
      path = base.sub("SRC",cities[i]).sub("DEST",cities[j])
      path.gsub!(" ", "+")
      resp = Net::HTTP.get_response(google,path)
      days = resp.body[/\d+ days/]
      hours = resp.body[/\d+ hours/]
      mins = resp.body[/\d+ mins/]
      puts ["cities[i],cities[j],days,hours,mins].join("\t")
    end
  end
  ```

- tsp.rb

  - The Ruby program that implements the genetic algorithm is in a file named tsp.rb
  - The next few slides will go over the methods defined in this file and how you can use them to explore the genetic algorithm used to solve the TSP

    ```ruby
    >> load "tsp.rb"
    >> t = bestTour(m,:popsize => 50)
    17 generations
    => #1140: CBADHGJIFE / 3971
    >> t.path.each_byte { |c| puts m[c] }
    Corvallis, OR
    Pullman, WA
    Seattle, WA
    Eugene, OR
    ...
    ```

The rand Method

- In previous projects we used a method named rand
- If you pass it an integer argument \( n \) it returns an integer between 0 and \( n-1 \):
  ```ruby
  >> rand(6)
  => 3
  >> rand(6)
  => 0
  ```
- When you call it with no arguments it returns a real number between 0 and 1:
  ```ruby
  >> rand
  => 0.875517493469671
  >> rand
  => 0.515481694941054
  >> rand
  => 0.6878418751571
  ```

Permutations

- The first place tsp.rb uses random numbers is in making random tours at the start of the program
- Recall tours are represented by strings
  - one letter for each city
  - e.g. Seattle is “A”, Pullman is “B”, Corvallis is “C”, etc
  - a tour of the Pac 10 cities is a permutation of the string “ABCDEFGHIJ”
Permutations

- Ruby has a useful type of statement known as **parallel assignment**
  - it allows us to do two assignments in one expression:

  ```ruby
  >> x, y = 2, 6
  => [2, 6]
  >> x
  => 2
  >> y
  => 6
  
  Values on the right side are copied to the corresponding variables named on the left side.

- Parallel assignment comes in handy when we want to exchange two letters in a string when making permutations:

  ```ruby
  >> s = "apple"
  => "apple"
  >> s.permute
  => "terucpom"
  >> s.permute
  => "reptcmuo"
  >> s.permute
  => "upmeotrc"
  
  Looks OK....
  
  The current value of s[4] is copied to s[0] at the same time the current value of s[0] is copied to s[4].

The permute Method

- Here is the pseudocode for a method that permutes a string
  - if you're interested see the complete definition in tsp.rb:

  ```ruby
  def permute
    for i in 0..length-2
      j = random index to the right of i
      s[i], s[j] = s[j], s[i]
    end
  end
  ```

  *Exchange the letter at location i with the letter at a random location between i and the end of the string.*
The CityList Class

- The program needs to keep track of the distances between pairs of cities
  - a convenient data structure is a table like the one found in maps
- The Ruby program uses a class named CityList
  - when you make a new object, pass it the name of a file with pairs of distances
  - the new object stores the city names and gives each one a unique letter

![CityList Table](image)

The Tour Class

- Another new class defined in tsp.rb is named Tour
- A Tour object has a unique id (so we can keep track of different tours), a string representing the path between cities, and the cost of that path
- To make an object, just call new, passing it the matrix of distances:
  ```ruby
  >> t1 = Tour.new(m)
  => #0: AFDICEHJGB / 7651
  >> t2 = Tour.new(m)
  => #1: DHABJICFE / 8016
  >> t3 = Tour.new(m)
  => #2: GHAEDICFJB / 9684
  ``
  - The path is a random permutation of the letters corresponding to cities
  - The cost is the sum of the costs of each leg of the tour
  - Note: the cost includes the link from the last city back to the first!

The CityList Class

- This example shows how to make a CityList object for the cities in the Pac-10 data set
  ```ruby
  >> m = CityList.new("pac10.txt")
  => List of 10 cities
  >> m["A"]
  => "Seattle, WA"
  >> m["B"]
  => "Pullman, WA"
  ``
  - Use the index operator to find out which city belongs to a particular label
  - The distance method returns the driving time between two cities:
    ```ruby
    >> m.distance("A","B")
    => 320
    ``
    - The time the number of minutes (according to Google maps)

The Tour Class

- Another way to make a new Tour object is to call a method named reproduce
- If you pass this method a single Tour object, you get back a copy of that tour with a random point mutation added to it
  ```ruby
  >> t0 = Tour.new(m)
  => #3: EBCFDAGJHI / 6354
  >> t1 = Tour.reproduce(t0)
  => #4: EBCFADGJHI / 6352
  ``
  - A method that creates a new object is called a constructor
    Every class has at least one constructor, named new
    The reproduce method is a second constructor for the Tour class
  - Note the new object has a different ID and a slightly different cost

Can you find where the mutation occurred?
**The Tour Class**

- If you pass the `reproduce` method two Tour objects, the new object is a cross between the two
  - the method copies a random chunk of the first parameter
  - it then adds the missing cities by getting them in order from the second parameter

```ruby
>> t0 = Tour.new(m, "ABCDEFGHIJ")  # A fake tour with all cities in order (so we can more easily see how the crossover works)
=> #5: ABCDEFGHIJ / 4112
>> t1 = Tour.new(m, "AEIBCDFGHJ")  # The second fake tour has all the vowels first, then the consonants
=> #6: AEIBCDFGHJ / 6938
>> Tour.reproduce(t0, t1)
=> #8: EFGHAIBCDJ / 7528
```

Can you see how the crossover was created?

---

**Populations**

- To make a “population” -- a collection of tours -- call `initPopulation`
- Pass it the number of tours you want and the `CityList` object
- You’ll get back an array of tours made from those cities:

```ruby
>> p = initPopulation(10, m)
```

---

**Evolution**

- A method named `evolve` will modify a population
  - It sorts the population by fitness, so best tour is `p[0]` and worst is `p[n-1]`
  - “Roll the dice” -- delete tour `i` with probability `i/n`
    - `i = 0:  pr = 0`
    - `i = 1:  pr = .1`
    - `i = 2:  pr = .2`
  - Rebuild the population by making new Tour objects with calls to `Tour.reproduce`
  - The return value is the best tour in the updated population:

```ruby
>> evolve(p)
=> #10: JHCBDAGFEI / 5165
```

---

**Evolution (cont’d)**

- You can pass an optional parameter to tell the program to print lots of output as it runs:

```ruby
>> evolve( p, {:trace => true} )
keep #14: IGDAEHCBFJ / 6269
keep #10: GECBAJHIFD / 6562
keep #12: IBJHEDCAGF / 6731
zap #19: GIDAEHCBFJ / 7000
zap #21: HJEGCBADFI / 7128
keep #22: BAFHJECGD / 7197
keep #17: BAFHJAGCDB / 7197
keep #24: DAFHJECDGB / 7557
keep #23: GECBHAIFD / 8262
keep #20: BFAHJECGID / 8819
#20: BFAHJECGID / 8819 => #25: BFAHJECGID / 8943
#23: GECBHAIFD / 8626 x #14: IGDAEHCBFJ / 6269 => #26: GECBHAJIDF / 8626
```

By throwing out random tours the algorithm maintains "genetic diversity"

Some high-cost tours will be left after each round

Don’t worry about the details of how tours are chosen for deletion -- just know that this method throws out some tours and replaces them
The Complete Genetic Algorithm

- The `bestTour` method shown at the beginning of these slides simply calls `evolve` until there is no improvement in the cost of the best tour:

  ```
  >>> bestTour(m)
  24 generations
  => #64: BJIFEGHDCA / 4529
  ```

- Calling it again will start with a new initial population, and may end up with a different result:

  ```
  >>> bestTour(m)
  24 generations
  => #107: HGEIJFCADB / 5313
  >>> bestTour(m)
  20 generations
  => #151: FIJEHGCBAD / 4710
  ```

Tour Parameters

- Several options control the simulation:
  - `popsize` determines the number of tours in a population
  - `maxgen` specifies how long to keep searching for a better tour
  - `maxstatic` is the maximum number of generations to continue without a change in the best tour

- The defaults are pretty small -- experiment with higher values to get lower cost tours:

  ```
  >>> bestTour(m, { :maxgen => 100, :popsize => 20 })
  100 generations
  => #1812: EFHIJGDABC / 3961
  ```

Some Results

- I ran the program with an option that printed the average tour cost at each generation:
  - the plot at right shows the mean cost starts out high
  - it drops pretty quickly
  - eventually there is little change -- the collection of tours is about as good as it will get

  ![Mean Tour Cost](image)

- The plot at right shows the same data, with "error bars" added to the drawing:
  - the vertical line through each point shows the range of tour values
  - lots of interesting info here

  ![Mean Tour Cost with Error Bars](image)

- Use a high enough value of "maxstatic" so the algorithm is patient
Summary

- The traveling salesman problem is an example of an **optimization** problem
- A **genetic algorithm** is a way of estimating the best solution
  - select several potential solutions at random
  - gradually improve the solutions
  - every now and then introduce a substantial change
  - depending on the problem domain and evolution parameters the process will eventually converge on a good (if not best) solution
- The TSP is another example of **abstraction**
  - find the essential parts of a problem description
  - define an algorithm as operations on abstract representation of the problem
  - many real problems reduce to this same abstract problem

What You Should Know

- Understand the general idea behind a genetic algorithm
  - each item in a “population” is a potential solution
  - the best solution gradually evolves
- Understand how an object of the Tour class represents a solution of the TSP
- Be able to show how a point mutation or crossover would create a new Tour object from one or two existing tours