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 projects

Review -- GA and TSP

- One way to implement a search: train an algorithm to recognize examples of the pattern you want it to find
  - SPAM, handwriting samples, images of craters, ...
- Genetic algorithms, inspired by concepts from population biology, are widely used in machine learning
  - make a population of solutions
  - new solutions are derived from existing solutions (using point mutations and crossovers)
  - after several generations good solutions begin to emerge
- Programs that learn are very complex
  - to understand how genetic algorithms work we looked at a much simpler application: optimization
  - “learning” in this case is just “you’re getting warmer...”
  - no generalization

Review (cont’d)

- The traveling salesman problem is an example of an optimization problem
- A genetic algorithm is a way of converging on the best solution
  - select several potential tours at random
  - gradually improve the tours via point mutations
  - every now and then introduce a substantial change
  - 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
I’m too lazy and impatient to do 45 Google queries by hand
the complete graph for 10 nodes has \((10 \times 9) / 2 = 45\) 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 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?

Suppose you want to add a map to your office as part of a company web site
One method: make a map, insert it as a graphic image in the HTML
Another approach: make a dynamic page
When an http request comes in to the company server:
• the server builds an outline of the page
• it fetches the latest map from Google
• the map returned by Google is inserted into the page
The same methods can add news articles, weather, and a wide variety of other information to your company page.

After poking around on the web a bit I found the URL for driving instructions:
\(\text{maps.google.com/maps}\)?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. \(\text{s.append(“xxx”)}, \text{o.insert(3,”abc”)}\)
• change spaces to \(+\), e.g. \(\text{“Eugene,+OR”}\)
The URL for fetching instructions to get to Pullman from Eugene:
\(\text{maps.google.com/maps}\)?saddr=Eugene,+OR\&daddr=Pullman,+WA\&hl=en

We can use Ruby’s HTTP library to fetch a map:
\(\text{google = ”maps.google.com”}\)
\(\text{path = ”maps?}\text{saddr=Eugene,}\text{+OR\&daddr=Pullman,}\text{+WA\&hl=en”}\)
\(\text{resp = Net::HTTP.get_response(google,path)}\)
What comes back is the complete map page, but we can use regular expressions to extract the estimated driving time:
\(\text{hours = resp.body[/\d+ hours/]}\)
\(\text{mins = resp.body[/\d+ mins/]}\)
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...
Screen Scraping (cont’d)

- Screen scraping is awkward, error-prone, and very hard to test
  - what happens if the time is one minute more than an hour? does Google say “1 mins” or “1 min”?
  - if we have “mins” in the regular expression it will fail on “1 min”
- Our program will almost certainly break in the near future when Google decides to change the format of the maps page
- Note a very solid foundation for roadtrip.com, our fledgling e-business...

From the Jargon File:

Nowadays [screen scraping] often refers to parsing the HTML in generated web pages with programs designed to mine out particular patterns of content. In either guise screen-scraping is an ugly, ad-hoc, last-resort technique that is very likely to break on even minor changes to the format of the data being snooped.

Web Services

- A far more robust way to get information like this is to use a web service
- Web services use standard protocols for exchanging information
  - SOAP
  - XML-RPC
  - Restful
  - many others
- Ruby and many other languages have extensive libraries for creating new web services and for connecting to existing web services
- Using a web service is a lot like using a web server
  - send a request, get a response
  - the difference is the response contains only the data you want -- no extra fluff used by browsers to format the data (tables, graphics, ...)

Web Services (cont’d)

- A service called a “geocoder” returns the location of a city
  - geocoder.us is a free service
  - uses data provided by the US census
- Using irb to test a call to geocoder.us:
  >> require 'net/http'
  >> resp = Net::HTTP.get_response("geocoder.us",
   "service/csv/geocode?city=eugene&state=or")
  => #<Net::HTTPOK 200 OK readbody=true>
  >> resp.body
  => "44.0522222222222, -123.085555555556, Lane, Eugene, OR\n"
- Note the response does not include any HTML -- it’s just a string with commas separating the fields (“CSV” format)
- The API for this service is defined by the parameters to include in the URL and the format of the strings it returns

XML

- Most web services define their API using XML
  - XML = “extensible markup language”
  - syntactically it looks a lot like HTML -- use < and > to indicate structures
- XML-RPC (“remote procedure call”) is one of the standards for web services
- Ruby’s XML-RPC library will transform an XML response into a hash
  >> require "xmlrpc/client"
  >> server = XMLRPC::Client.new("geocoder.us",
   "/service/xmlrpc")
  >> deschutes = "1377 E. 13th Ave, Eugene, OR"
  >> loc = server.proxy("geocode").geocode(deschutes)
  >> loc[0]["zip"]
  => 97401
Using Google to Get Distances

- Google has XML-based APIs for many of its services, but I couldn’t find one for the driving instructions.
- 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
```

The Ruby program that implements the genetic algorithm is in a file named tsp.rb.
- TSP = traveling salesman problem
- 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.

```
>> load("tsp.rb")
>> bestTour (m, { :verbose => true, :popsize => 50 })
97 generations
best cost: 3828
best tour:
Los Angeles, CA
Pasadena, CA
Tucson, AZ
...
```

Random Numbers

- Genetic algorithms need random numbers
  - make random permutations of paths to make initial tours
  - decide which tours survive to make the next generation
  - decide whether to apply a point mutation or a cross-over
- Making random numbers is harder than might seem at first
  - how can an algorithm, which has a well-defined sequence of steps, generate different results each time it is called?

```
>> rand(10)
=> 6
>> rand(10)
=> 9
>> rand(10)
=> 3
```

Pseudo-Random Numbers

- The algorithm generates the same sequence of numbers, but we just use a small part of the sequence so it looks random
  - the algorithm defines a recurrence \( x_{i+1} = f(x_i) \)
  - Ruby uses an object to hold the current value of \( x \)
    - when \( f \) is called, it uses the current value to compute the next value
- The values of \( x \) are very big -- range from 0 to \( 10^{18} \)
- When we ask for a random number between 0 and 9, Ruby computes the next value of \( x \), then gives us the remainder of \( x / 10 \)
- Since we just want a few numbers and they are small they appear to be random
- Random-looking values generated by algorithms are called **pseudo-random** numbers
The \texttt{rand} Method

- As we saw on a previous slide, the Ruby method is named \texttt{rand}.
- When you call it with no arguments it returns a real number between 0 and 1:
  \begin{verbatim}
  >> rand
  => 0.875517493469671
  >> rand
  => 0.515481694941054
  >> rand
  => 0.6878418751571
  \end{verbatim}
- If you pass it an integer argument \textit{n} it returns an integer between 0 and \textit{n}-1:
  \begin{verbatim}
  >> rand(6)
  => 3
  >> rand(6)
  => 0
  \end{verbatim}

The \texttt{permute} Method

- The first place \texttt{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”

The \texttt{permute} Method (cont’d)

- Here is the method implemented in \texttt{tsp.rb}:
  \begin{verbatim}
  class String
  def permute
    for i in 0..length-2
      r = rand(length-i) + i
      self[i],self[r] = self[r],self[i]
    end
    self
  end
  end
  \end{verbatim}

- Let’s see if it works:
  \begin{verbatim}
  s = ”Hello, I’m a Mac”
  => ”Hello, I’m a Mac”
  s.permute
  => ”clao BM’ea ,1m”
  \end{verbatim}

The \texttt{permute} Method (cont’d)

- Looks OK….
The CityList Class

- The program needs to keep track of the names of cities and the distances between pairs of cities.
- To do this there is a new class named CityList.
  - A CityList is a lot like a Hash (associative array).
  - When you make a new object, pass it the name of a file with pairs of distances.
  - The new object stores the cities and assigns each one a unique letter.

```ruby
m = CityList.new("pac10.txt")
=> List of 10 cities
=> m["A"]
=> "Seattle, WA"
=> m["B"]
=> "Pullman, WA"
```

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 list of cities:

```ruby
t1 = Tour.new(m)
=> #0: AFDICEHJGB / 7651

\*\* The initial path is a random permutation of the letters corresponding to cities \*\*

t2 = Tour.new(m)
=> #1: DHABJCIGFE / 8016

t3 = Tour.new(m)
=> #2: GHAEDICFJB / 9684
```

The CityList Class (cont’d)

- As with a Hash, you can ask for the keys and values stored in the object:
  ```ruby
  m.keys
  
  m.values
  
  There is a method that will return the distance (estimated driving time, in minutes) between two cities:
  ```ruby
  m.distance("A","B")
  => 320
  
  m.distance("A","J")
  => 1800
  ```
The Tour Class (cont’d)

- Cross-overs (large changes) are implemented by the cross method
- A cross-over combines parts of two tours
- Call \( x\.cross(y) \) to replace a chunk of tour \( x \) with a chunk from tour \( y \):

\[
\begin{align*}
&>> \text{t1} \\
&=> #0: \text{HECIDFAJGB} / 10082 \\
&>> \text{t2} \\
&=> #1: \text{DHABJCIGFE} / 8016 \\
&>> \text{t1.cross(t2)} \\
&=> #0: \text{BHECIGFE} / 8655
\end{align*}
\]

The return value is the best tour in the updated population \( p \):

\[
\begin{align*}
&>> \text{evolve}(p) \\
&=> #10: \text{JHCBDAEGFI} / 5165
\end{align*}
\]

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 Tour objects:

\[
>> p = \text{initPopulation}(10,m) \\
=> [\#0: \text{FCHEADJIBG} / 7916, \#1: \text{BDAGFEIHJC} / 5898, \#2: \text{CIFEGDJIHB} / 7085, \#3: \text{EGFHDJAIBC} / 9316, \\
\#4: \text{ECDBHGFJAI} / 7960, \#5: \text{CHABIFEGDJ} / 8699, \#6: \text{EHGDIFJAIIC} / 8903, \#7: \text{HLGCBBIDAF} / 6785, \\
\#8: \text{BFHEGDIJCAI} / 9294, \#9: \text{DFGADICHJ} / 9616]
\]

Evolution

- A method named evolve will modify a population
- Sort 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; either
  - clone one of the survivors, then apply mutate
  - clone one survivor, then cross with another survivor
- The return value is the best tour in the updated population \( p \):

\[
\begin{align*}
&>> \text{evolve}(p) \\
&=> #10: \text{JHCBDAEGFI} / 5165
\end{align*}
\]

Evolution (cont’d)

- The same example, passing an optional parameter to tell the program to print lots of output so we can check it:

\[
\begin{align*}
&>> \text{evolve}(p,(\{\text{debug} => \text{true}\})) \\
&\quad \text{keep} \#1: \text{BDAGFEIHJC} / 5898 \\
&\quad \text{keep} \#7: \text{HLGCBBIDAF} / 6785 \\
&\quad \text{keep} \#2: \text{CIFEGDJIHB} / 7085 \\
&\quad \text{keep} \#0: \text{FCHEADJIBG} / 7916 \\
&\quad \text{keep} \#4: \text{ECDBHGFJAI} / 7960 \\
&\quad \text{keep} \#5: \text{CHABIFEGDJ} / 8699 \\
&\quad \text{zap} \#6: \text{EHGDIFJAIIC} / 8903 \\
&\quad \text{keep} \#8: \text{BFHEGDIJCAI} / 9294 \\
&\quad \text{zap} \#3: \text{EGFHDJAIBC} / 9316 \\
&\quad \text{keep} \#9: \text{DFGADICHJ} / 9616 \\
&\quad \text{#1: BDAGFEIHJC} / 5898 \Rightarrow \#10: \text{JHCBDAEGFI} / 5165 \\
&\quad \text{#1: BDAGFEIHJC} / 5898 \Rightarrow \#1: \text{IEFGADHJCB} / 7443 \\
&\quad \Rightarrow \#10: \text{JHCBDAEGFI} / 5165
\end{align*}
\]
The Complete Genetic Algorithm

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

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

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

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

bestTour (cont’d)

- Pass `verbose => true` as an option to get more information about the tour and a nicer display of the result:

  ```ruby
  >> bestTour(m, { :verbose => true })
  10 generations
  best cost: 4657
  best tour:
  Stanford, CA
  Pasadena, CA
  Seattle, WA
  Pullman, WA
  Corvallis, OR
  Eugene, OR
  Berkeley, CA
  Los Angeles, CA
  Tempe, AZ
  Tucson, AZ
  => #228: FGABCDEHIJ / 4657
  ```

Tour Parameters

- Two options control the simulation:
  - `popsize` determines the number of tours in a population
  - `maxiter` specifies how long to keep searching for a better tour

- How `maxiter` works:
  - Each time a new tour is found that is better than the current best tour: reset a counter to 0
  - When the new tour in the next generation is not better than the current best tour: increment the counter
  - If the counter reaches `maxiter` the method returns the current best tour

- The default `maxiter` is 5 -- the fact that we’re getting such a wide variety of tours probably means we’re not “patient” enough

  ```ruby
  >> bestTour(m, { :verbose => true, :maxiter => 25 })
  101 generations
  best cost: 3960
  ```

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
This lecture also introduced a new technology

A **web service** is defined by a protocol
- service provider publishes a description of the service provided
- description includes definition of parameters to include in the service request
- responses are intended to be used in programs
- much more terse -- very hard for humans to read

Examples:
- credit card validation
- PayPal
- bioinformatics and other scientific data
- news, weather, sports scores

On the horizon: the “Global Grid”
- information resource analogous to the power grid...