Genetic Algorithms

The Traveling Salesman Problem
Genetic Algorithms
detour: Google Maps API, web services

Reading: NTO 16 (and 41)

Traveling Salesman

- The traveling salesman problem is a classic problem in computer science
  - suppose you have a list of \( n \) cities
  - the goal is to define a tour that visits all \( n \) cities and returns to the starting place
  - you can visit each city only one time

- What is the lowest cost tour?
  - at right: a tour of 13,500 US cities

Cost can be defined as driving time, distance, air fare, ...
assume the cost of going from X to Y is the same as going from Y to X
It’s easy to write a program that works for a small number of cities
but the complexity is \( O(2^n) \)
each time a city is added to the list the time to find a tour doubles
for a tour of 20 cities the program might have to check 1 billion combinations

Some figures from TSPlib++ (an “industrial strength”) solver:
- 5.5 hours for 10,000 cities
- estimate over 6,000 years for 25,000 cities

Real-Life Applications

- The idea that anyone would really plan a road trip to 13,000 cities is a bit silly
- But this problem is identical to several important “real world” problems:
  - transportation: school bus routes, service calls, delivering meals, ...
  - manufacturing: an industrial robot that drills holes in printed circuit boards used in computers, video and stereo, almost anything electronic
  - communication: planning new telecommunication networks
  - biology: genetic markers on chromosomes / reassembly
History

- The traveling salesman problem has been studied by mathematicians for over 100 years
  - originally posed as a puzzle in the 1800s
  - started attracting serious attention in the 1930s
  - one of the most widely studied problems in applied math and operations research

Optimization

- The TSP is an example of an optimization problem
  - we saw another optimization problem earlier this term: sequence alignment
  - The goal is to find the order of cities that has the lowest cost
  - A plot like the one shown here is helpful to visualize the process
    - assume there is a method for ordering the solutions along the x axis

Brute Force

- The “obvious” way to solve the problem is to check every ordering
  - Start with tour 0
    - evaluate the cost
    - modify the tour and repeat
    - keep track of the tour number that gives the lowest cost
  - The problem: way too many tours to check
    - for \( n \) cities: up to \( n! \) orderings
    - but there is a way to “reduce” the problem to \( 2^n \) steps

Hill Climbing

- A slightly better approach is known as hill climbing
  - Pick a starting spot
    - may be random, or it may be based on some extra information (e.g. “the tour must go through X, Y, and Z at the start”)
    - check neighboring points
    - move toward the neighbor with the better cost and repeat
  - “Hill climbing” refers to a search for maximum, but the idea still applies when searching for minimum
Local Minima

- The problem with hill climbing is that it might find a local minimum.
- Ideally we want an algorithm that will find the global minimum.
- We also want to avoid an exhaustive search of all possible tours.

Genetic Algorithm

- Today's talk is about a novel approach called a genetic algorithm.
- The basic idea:
  - Create a “population” of possible solutions.
  - The “fitness” of a solution depends on the cost.
    - Lower cost = better fitness.
  - Use “natural selection” to keep good solutions.
  - Replace bad solutions with new ones derived from the survivors.

Genetic Algorithm (cont’d)

- The key idea in a GA is that “individuals” represent problem solutions.
- Generation of new solutions happens by:
  - Mutation: make a copy of an existing solution and make a small change.
  - Cross-over: select two existing solutions, combine elements at random to produce a new solution.
- In both cases the result must a complete solution.

Genetic Changes

- The idea of using point mutations and cross-overs was inspired by genetics.
GA for TSP: Solutions

- The first step in designing a genetic algorithm to solve the TSP is to figure out how to represent a solution.
- The problem is often described in terms of a **graph**.
  - Graphs are similar to trees -- they are collections of nodes.
  - A graph does not have a root.
  - There can be any number of connections between nodes.

A graph with 7 nodes

Two examples of complete tours based on the graph

Solutions (cont’d)

- For many applications there is a “road” between every city.
  - Example: the robot arm drilling holes in a PC board can move freely from any point to any other point.
- For the remainder of these slides we’ll assume that’s the case.
- The graph for this problem is “fully connected.”

Q: How many possible tours are there in this graph?

A simple way to represent the tour is to use a string.

- If there are \( n \) cities there are \( n \) letters in the string.
- Tours of more than 26 cities would use arrays of integers, but strings are useful for small demos (easy to understand, easy to display).
- For the small graph shown below strings would have the letters “A” through “G.”

Any string that is a permutation of these letters is a valid solution.

“ACBDFEG” “ACDFEGB”

GA for TSP: Mutations

- The role of mutations in a genetic algorithm is to make small changes to one of the current solutions.
- This allows the algorithm to close in on the final solution.
- The example at right shows a population of 5 (blue).
- New solutions (green) are only slightly different than current ones.
- Not every mutation leads to a better solution.
Mutations (cont’d)

- One technique for defining mutations on paths:
  - pick two links at random
  - swap their endpoints
  - reverse the direction of the loop between the endpoints
- Example: swap ends of $C \to D$ and $H \to I$

![Diagram of a graph with nodes A to J and edges connecting them, showing an example of mutations.

GA for TSP: Cross-Overs

- Cross-overs define larger changes
- The idea is that combining the best parts of two very different solutions has a chance of "hopping out" of the region near a local minimum
- Not every cross will lead to an improvement

![Graph showing the cost over tour number with a downward trend indicating improvement.

GA for TSP: Main Loop

- Create a string $S$ with one letter per city ("ABCD...")
- Create an initial population using $n$ random permutations of $S$
- Repeat:
  - natural selection -- remove individual $i$ with $p(\text{fitness}(i))$
  - rebuild the population to size $n$:
    - copy random individuals, apply point mutation
    - apply cross-over to random pairs
- Stop when the best solution does not improve, or after a maximum number of steps

![Graph showing the cost over tour number with a downward trend indicating improvement.

Cross-overs (cont’d)

- Defining cross-overs for strings is a little more difficult
  - pick a cross-over point $x$
  - in one string select left of $x$, copy to the new string
  - in the other string select from the right of $x$
  - stop when selecting a letter already copied
  - finish up by selecting the remaining chars at random
- Example: $\text{HIECBDAFGJ} \times \text{BAICHEGDFJ}$ with $D$ as the crossover point

![Diagram of a string with nodes E and F connected, showing the process of crossing over.

GA for TSP: Main Loop

- Create a string $S$ with one letter per city ("ABCD...")
- Create an initial population using $n$ random permutations of $S$
- Repeat:
  - natural selection -- remove individual $i$ with $p(\text{fitness}(i))$
  - rebuild the population to size $n$:
    - copy random individuals, apply point mutation
    - apply cross-over to random pairs
- Stop when the best solution does not improve, or after a maximum number of steps
Parameters and Variations

- There are lots of small adjustments one can make to the basic algorithm
- **Selection:**
  - what percentage of the population should be kept?
  - larger proportions lead to more stability but may be too slow to evolve
  - always remove the least fit? or use p(fitness)?
  - keeping random poor solutions adds to variability in the population
- **Mutation:**
  - how many to apply at each round?
  - mutate only new solutions?
- **Cross-over:**
  - how often should cross-overs happen?
  - should there always be fewer cross-overs than point mutations?

Honor Diversity

- The first time a GA is tested the developer often finds selection is “too effective”
- The algorithm zeroes in on a few local minima
- The trick is to ensure enough variability in the population so cross-overs eventually find the valley with the global minimum
- One idea: delete solutions that are too similar to other solutions

Road Trip

- To test this algorithm I wrote a small Ruby program
  - data set: cities of the Pac-10 universities
  - distances between each city supplied by maps.google.com
  - I decided to 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 \((10 \times 9) / 2\) 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 Map API (cont’d)

- Note that adding a map is not a matter of cutting-and-pasting the map image.
- We’re not adding a static graphic to the page.
- When an http request comes in to the company server:
  - The server builds an outline of the page.
  - It uses the API to fetch a map from Google.
  - The map returned by Google is inserted into the page.
- The page is a dynamic page.
  - The same methods can add news articles, weather, and a wide variety of other information.

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...

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"
- 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

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", "Eugene, OR", ...]
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(" ") 
  end
end
```
### Results

- Here are the outputs from some test runs:
  - Parameters: keep best 50% of each generation, use cross-overs to build back to 100%, apply mutations to 50%

<table>
<thead>
<tr>
<th>population size</th>
<th>#generations</th>
<th>best path (min)</th>
<th>population size</th>
<th>#generations</th>
<th>best path (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>3980</td>
<td>50</td>
<td>10</td>
<td>3948</td>
</tr>
<tr>
<td>100</td>
<td>3960</td>
<td></td>
<td>100</td>
<td>3960</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>3960</td>
<td>3828</td>
<td>500</td>
<td>3960</td>
<td>3828</td>
</tr>
<tr>
<td>1000</td>
<td>3960</td>
<td>3828</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Population too small -- caught in a local minimum
Best result seen from all tests -- probable global minimum

### Best Path

- A run that found the best path:
  ```
  $ tspga cities.txt -g 50 --ps .25 --pp .25 -n 50
  time = 3828
  Shortest route: 2 days 15 hours 48 minutes
  Eugene -> Corvallis -> Seattle -> Pullman -> Tempe -> Tucson -> Pasadena -> Los Angeles -> Stanford -> Berkeley -> Eugene
  ```

- One that came close:
  ```
  $ tspga cities.txt -g 50 --ps .25 --pp .25 -n 50
  time = 3960
  Shortest route: 2 days 18 hours 0 minutes
  Eugene -> Los Angeles -> Tempe -> Tucson -> Pasadena -> Stanford -> Berkeley -> Corvallis -> Pullman -> Seattle -> Eugene
  ```

- Random starting tours had values as high as 6148 (4 days 6 hours 28 mins)

### Ruby Code

- Some highlights of the Ruby program:
  - Method to read file with distances between cities:
    ```ruby
    def readCities(fn,m)
      raise "specify city file on command line" unless fn != nil
      File.open(fn).each do |line|
        rec = line.split(/\t/)  # extract city names and distances
        i = CityKey.getKey(rec[0])
        j = CityKey.getKey(rec[1])
        dist = ((24 * getValue(rec[2]) + getValue(rec[3])) * 60) + getValue(rec[4])
        m.assign(i,j,dist)
        m.assign(j,i,dist)
      end
    end
    ```

- Method to compute the cost of a path defined by a string S:
  ```ruby
  def pathcost(s)
    sum = $dist[s[0]][s[-1]]  # link from end to start
    for i in 0...s.length-2
      sum += $dist[s[i]][s[i+1]]
    end
    return sum
  end
  ```

- The method that computes the cost of a path defined by a string S:
  ```ruby
  def pathcost(s)
    sum = $dist[s[0]][s[-1]]  # link from end to start
    for i in 0...s.length-2
      sum += $dist[s[i]][s[i+1]]
    end
    return sum
  end
  ```

- Distance from city S[i] to city S[i+1] (uses matrix read by method on last slide)
**Ruby Code (cont’d)**

- **Method to make an initial population (array of individuals) where each individual is defined by a random string:**

```ruby
def initPopulation(n, s)
  a = Array.new
  n.times do
    i = 0..s.length-2
    r = rand(s.length-i) + i
    # i <= r < s.length
    s[i], s[r] = s[r], s[i]
  end
  a.push(Individual.new(s))
  return a
end
```

- **Overview of the main loop:**

```ruby
def evolve(p)
  $gmax.times do
    p.sort! { |x,y| x.fitness <=> y.fitness }
    p.slice!(nk..-1)
    ...
    i = rand(nk)
    j = rand(nk)
    p.push(Individual.cross(p[i], p[j]))
    ...
    i = rand(p.length)
    p[i].mutate()
  end
end
```

---

**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
    - bioinformatic and other scientific data
    - news, weather, sports scores
  - On the horizon: the “Global Grid”
    - information resource analogous to the power grid...