The Traveling Salesman Problem

Search Algorithms
Optimization
The Traveling Salesman Problem

Reading: SoC Chapter 14 (tutorial project only)

Review: Search

- Earlier in the term we looked at algorithms that searched through lists
  - linear search compares each item until it finds what is looking for (the key)
  - binary search is much more efficient (if the list is sorted)
  - a search for the largest item in an unsorted list compares each item

<table>
<thead>
<tr>
<th>x: 3</th>
<th>3</th>
<th>7</th>
<th>6</th>
<th>2</th>
<th>9</th>
<th>2</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 7</td>
<td>3</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>x: 7</td>
<td>3</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Review: Search

- More recently we looked at techniques for searching in strings
  - exact match: use the index method
  - pattern-based search using regular expressions

```python
>> s = "Now is the time for all good men..."
>> s.index("time")
=> 11
>> s.index(/t\w+/)
=> 7
```

Often written as:
```
s =~ /t\w+/`
```

Inexact Match

- Another type of string search looks for things that are “almost like” the key
- Example: search a dictionary for words like “cow”
- An inexact match algorithm would find three-letter words with one difference
  - “how”, “now”, “row”, “con”, “cop”, ...
- It might also return words that are slightly longer or shorter
  - “ow”, “cowl”, “chow”, ...

These algorithms require us to define similarity
- the algorithm needs some way to determine when words are “too different”
- should it return “scowl”? 
- what about “crawl”?
An example of an algorithm that searches based on similarity: **BLAST**
- Basic Local Alignment Search Tool
- widely used by molecular biologists and evolutionary biologists to search databases of gene and protein sequences

DNA molecules are **polymers**
- long chains of smaller molecules called **nucleotides**
- there are four different nucleotides
- a convenient way to describe a DNA molecule is to write a string using a single letter to represent each nucleotide

```
ATGGTGCTGTCT...
```

**Example: search for DNA sequences similar to the human gene that produces hemoglobin**

Enter DNA sequence in a text box in a web browser

The result is a list of similar sequences found in a database at NCBI

The results are sorted, with the most similar sequence at the top of the list

**Bioinformatics**
- BLAST and other algorithms that work with gene sequence data are part of a field known as **bioinformatics**
- the term “informatics” is used in many other areas as well
- research in these fields makes use of concepts from computer science to help manage vast quantities of information
- learn more about bioinformatics from the NCBI web site

We won’t get a chance to explore inexact match algorithms this term
- but this week’s project does use ideas inspired by the study of genes -- we’ll return to gene sequences later in this lecture
Optimization

- The topic for this week’s project is another type of search known as **optimization**
- The goal is to find a combination of parameters that give the highest or lowest value for a function

![Graph for a function](image)

**What values of x and y give the highest value of f?**

Optimization Problems

- Many important problems can be described as optimization problems
- Example: designing a fuel-efficient car
- The best design requires trade-offs involving
  - materials
  - overall weight
  - fuel type
  - engine size
  - air resistance
  - many more variables

Another example: **bin packing**
- the goal: choose a set of packages that will fill the most space on a truck
- a “greedy” algorithm puts the big boxes on first
- a better choice might include lots of smaller boxes
- an algorithm potentially has to examine all combinations

![Bin packing](image)

\[ n \text{ boxes} \rightarrow 2^n \text{ combinations} \]

The Traveling Salesman

- This week’s lab explores a well-known problem known as the **traveling salesman** problem (TSP)
- suppose we have a set of \( n \) cities
- the goal is to define a tour that visits all \( n \) cities and returns to the starting place
- we can visit each city only once
- What is the lowest cost tour?

![Traveling salesman](image)

Images and examples from the Traveling Salesman Page at Georgia Tech:
http://www.tsp.gatech.edu/
The Traveling Salesman

- 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
- Although it sounds simple this is a very hard problem to solve
  - it’s easy to write a program that works for a small number of cities
  - but each time a city is added to the list the possible number of tours increases dramatically
  - the total number of tours of \( n \) cities:
    \[
    \frac{(n - 1)!}{2}
    \]
- Extra Credit: can you explain why there are this many tours?

The TSP Game

- One of the pages at the Georgia Tech lab has an interactive game
  - solve the simple tour and they give you another one, slightly bigger
  - it’s addicting if you like puzzles....

Real-Life Applications

- The idea that anyone would really plan a road trip to 13,000 cities is a bit silly
- But the solution of several important “real world” problems is the same as finding a tour of a large number of cities
  - transportation: school bus routes, service calls, delivering meals, ...
  - manufacturing: an industrial robot that drills holes in printed circuit boards
  - VLSI (microchip) layout
  - communication: planning new telecommunication networks

The Traveling Salesman

```python
>> for n in 4..20 do puts fact(n-1) / 2 end
3
12
60
360
2520
20160
181440
19958400
239500800
3113510400
43589145600
653837184000
10461394944000
177843714048000
3201186852864000
60822550204416000
```

This tour of 13,500 US cities was generated by an advanced algorithm that used several “tricks” to limit the number of possible tours

Required 5 “CPU-years”

http://www.tsp.gatech.edu/games/index.html

http://www.tsp.gatech.edu/
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

Brute Force

- The best way to visualize how an optimization algorithm works is to think of how we might find the highest point on a curve defined by a function
  - start at the lowest value of $x$
  - check successively higher values, record where the largest value of $f(x)$ was found

The same strategy can be used to find the minimum value of $f(x)$

A pseudo-code description of this method:

```plaintext
for x in xmin..xmax
  if f(x) > opt
    opt = f(x)
    xopt = x
  end
end
```

The largest value of $f(x)$

The value of $x$ that maximizes $f(x)$

Hill Climbing

- A strategy known as hill climbing will avoid checking all the values of $x$
  - Pick a random starting spot
    - evaluate $f(x)$
    - also evaluate $f$ at the neighboring points
    - move toward the higher value of $f$ and repeat
    - stop when the highest value of $f(x)$ is found
Hill Climbing

- The obvious problem is getting caught at a **local maximum**
- Applications that use this strategy often repeat the search from several different random starting points
  - hope that one search settles on the **global maximum**
- The “hill climbing” image implies a search for maximum, but the term is also used when searching for minimum
  - just change > to < in the code that implements the search

Genetic Algorithms

- The project this week looks at a type of method known as a **genetic algorithm**
- This type of algorithm was inspired by concepts from population biology
- The main idea: instead of examining one point at a time consider a large “population” of potential solutions

Genetic Algorithms

- Each member of the “population” represents one potential solution to the problem being solved
  - the **fitness** of an individual \( x \) is determined by \( f(x) \)
- At each step of the algorithm:
  - throw away bad solutions (lower values of \( f \))
  - rebuild the population by producing slight variations of surviving solutions
- As the algorithm progresses the optimal solution will eventually evolve

Genetic Algorithms and the TSP

- For the TSP the goal is to find the minimum cost tour
  - each member of the population is a complete tour of all the cities
  - start with a bunch of random tours
  - evaluate the costs, toss the most expensive ones
  - rebuild the population by making small variations of the surviving tours
Genetics

- New solutions are added to the population using concepts from genetics.
- Recall from the intro slides that DNA can be thought of as a long string made using letters A, C, G, and T.
- The simplest type of change that occurs from one generation to the next is a **point mutation**.
  - a single letter at one place in the string is replaced by a different letter.

Comparing DNA Strings

- I used Ruby to compare the DNA strings for hemoglobin returned by BLAST:

  ```ruby
  human = "ATGGTGCTGTCTCCTGCCGA... 
  cow = "ATGGTGCTGTCTGCCGCCGA... 
  human.length.times { |i| 
    printf("%d %s %s
", i, human[i].chr, cow[i].chr) 
    if human[i] != cow[i] }
  12 C G
  14 T C
  24 A G
  ...
  ```

  - point mutations that occurred since the most recent common ancestor of cows and humans.

Re-Building the Population

- During each round of the genetic algorithm:
  - throw out the “least fit” solutions
  - for the TSP: delete the ones with the highest cost tours
- Replace the deleted solutions with copies of survivors:
  - make a copy of a random survivor
  - add a point mutation
  - for the TSP: make a small change to the tour, e.g. change A-B-C to A-C-B
- Every now and then make a new tour by making a more drastic change:
  - select two survivors and “cross” them
  - for the TSP: make a new tour that has big pieces from each parent

These operations will be explained in more detail after we see how to represent a tour.
The first step in designing a genetic algorithm to solve the TSP is to figure out how to represent a tour. Leonhard Euler showed one way to do this in 1737:

- Euler’s question: is there a tour of Königsberg that crosses each bridge in town exactly once?
- To reduce this problem to its essential features, make a drawing that has a circle for each piece of a land.
- A line connects two circles if the corresponding land is connected by a bridge.

Is such a tour possible?

The data structure defined by Euler is now known as a **graph**:

- A graph has a collection of **nodes**.
- Connections between nodes are called **edges**.

When we draw pictures of graphs we use circles for nodes and lines for edges.

For the TSP, nodes represent “cities”, and lines represent “roads” between pairs of cities.

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 situation.

- When there is an edge between every pair of nodes the graph is said to be **fully connected**.

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

A simple way to represent a tour is to use a string:

- If there are \( n \) cities there will be \( n \) 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.

These strings are the “DNA” of the population of solutions....
Point Mutations

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

Crossovers

- 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: HIECBDAFGJ × BAICHEGDFJ with D as the crossover point

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 \( n \):
    (a) copy random individuals, apply point mutation
    (b) 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 \( p(\text{fitness}) \)?
    - keeping random poor solutions adds to variability in the population
- Mutation:
  - how many to apply at each round?
  - mutate only new solutions?
- Crossover:
  - how often should crossovers happen?
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

To Be Continued

- Next lecture: the Ruby program we’ll use to experiment with the TSP
  - data set: cities of the Pac-10 universities
  - distances between each city supplied by maps.google.com
  - use driving time as the measure of distance