Travelling Salesman Problem - Brute Force

John Murray

Computer Science

Speed School of Engineering

jmmurr02@louisville.edu

1. Introduction

The Travelling Salesman Problem is a problem in which the goal is the find the most efficient path between all of the nodes such that the path creates a Hamiltonian path. This paper discusses the results of brute-forcing the problem and the methods used to generate all possible paths.

2. Approach

Generating All Possible Paths

My original attempt was to write a function to eagerly generate all possible paths (the permutation) and then loop through the resulting set, calculating the cost of each path. However, I quickly realized that, for larger data-sets, this was not going to work. With only 12 data-points, the program struggled to complete. From this, it was obvious that the program would be better off if it could lazily load permutations either in manageable sections or one at a time.

The target-language (Ruby) supported a way to generate permutations via an Enumerable object. This means that the permutations are calculated and accessed one at a time resulting in a (much) smaller memory footprint, but possibly more CPU time (more function calls). The resulting code can now be expressed succinctly as:

```
# points: set of all nodes (eg. [1, 2, 3, 4, …, n])
perm_enum = points.permutation
perm_enum.each do |perm|
   # … processing …
end
```

Each time `each` is called on `perm_enum`, `next` is called on the enumerable object. This results in the calculation of the next permutation. Within the Ruby VM, this is completed by wrapping the computing function into a Fiber (Ruby's co-routine) and starting/stopping the fiber each time a result is requested/yielded. The function that does the heavily lifting within the Ruby VM for

generating permutations is as follows:

```
static void
permute0(long n, long r, long *p, long index, char *used, VALUE values)
{
    long i,j;
    for (i = 0; i < n; i++) {
       if (used[i] == 0) {
           p[index] = i;
           if (index < r-1) {            /* if not done yet */
              used[i] = 1;               /* mark index used */
              permute0(n, r, p, index+1, /* recurse */
                     used, values);
              used[i] = 0;               /* index unused */
           }
           else {
              /* We have a complete permutation of array indexes */
              /* Build a ruby array of the corresponding values */
              /* And yield it to the associated block */
              VALUE result = rb_ary_new2(r);
              VALUE *result_array = RARRAY_PTR(result);
              const VALUE *values_array = RARRAY_PTR(values);

              for (j = 0; j < r; j++) result_array[j] = values_array[p[j]];
              ARY_SET_LEN(result, r);
              rb_yield(result);
              if (RBASIC(values)->klass) {
                  rb_raise(rb_eRuntimeError, "permute reentered");
              }
           }
       }
    }
}
```

You can see that this operates recursively, yielding each permutation to the associated block (lambda/proc). Using this method of generating permutations allows utilization of an existing framework, but also offers a reference if a more feature-rich permutation generation scheme is needed in the future.

In my tests, it took between 200 and 220 seconds to iterate through a permutation of 12 data-points (running on MRI Ruby 1.9.3). However, running in JRuby (1.7.0-p2) allows a faster execution time (because of hot-spot optimization) and takes between 120 to 140 seconds to run the same bit of code.
Note: The above tests to not include any computation on each iteration

If I were to push this further, I could parallelize the work done on each enumeration. However, since the current computation is CPU bound and is merely performing many small (quick)

operations, the time to spin up and shutdown threads would probably cost more than multiple threads would save.

Computing Each Permutation

After generating the permutations, it was simply a matter of computing the cost for each permutation. This was done using the distance formula. As the methods are fairly simple and short in nature, I have not included them in this paper. They can be found in `tsp.rb` as two functions, `compute_path_total` and `compute_distance`.

3. The Results

Data

The TSP problem was solved for 9 sample-problems that were generated from the Concorde software. Of the 9 sample problems, each contain a unique number of data-points between 4 and 12 data-points. While sets with a small number of data-points is easier to solve, the file with 12 data-points yields 479,001,600 permutations which must be checked. Each data-point in the file is given an ID or index. These are referenced below when discussing the cheapest path.

Results

The results below show the path and cost of each solution:

File: Random4.tsp
```
cheapest path:        1 4 2 3 1
cheapest path costs:  215.08553303209044
```

File: Random5.tsp
```
cheapest path:       1 2 5 3 4 1
cheapest path costs: 139.1335417499496
```

File: Random6.tsp
```
cheapest path:       1 2 3 4 5 6 1
cheapest path costs: 118.96891407553862
```

File: Random7.tsp
```
cheapest path:       1 2 7 3 6 5 4 1
cheapest path costs: 63.863031874767636
```

File: Random8.tsp
```
cheapest path:       4 5 2 3 7 1 6 8 4
cheapest path costs: 310.9820797442316
```

File: Random9.tsp
```
cheapest path:       6 7 1 8 4 9 2 5 3 6
cheapest path costs: 131.02836613987674
```

File: Random10.tsp
```
cheapest path:       1 2 7 6 8 5 9 10 4 3 1
cheapest path costs: 106.78582021866472
```

File: Random11.tsp

```
cheapest path:       1 6 10 11 8 9 7 5 3 4 2 1
cheapest path costs: 252.6844344550543
```

File: Random12.tsp

```
cheapest path:       1 8 2 3 12 4 9 5 10 6 7 11 1
cheapest path costs: 66.08484401133855
```

4. Discussion

Issues in Scale

While this is a fun problem, the current implementation will not scale far beyond 12 data-points. Since the size of the permutation *set* grows exponentially, and since are current algorithm operates at *O*(n), this solution only applicable to small data-sets.

Visualizing the Problem

To aid in testing and (visual) verification of the results, it was useful to visualize the problem. Given that Ruby has poor UI libraries, but great web libraries, it is easy to create a web-api that allows the creation of a web-based UI.

As a result, the web-server is available online and on GitHub. All of the test-files have been included and can be chosen for viewing on the web-based UI. Do note that only the data-points are currently shown, no solutions (yet).

```
URL:       http://tsp-web.herokuapp.com
Clone URI: git://github.com/JohnMurray/tsp-web.git
GitHub:    https://github.com/JohnMurray/tsp-web
```

Assuming that we will be working with the Travelling Salesman Problem more extensively, I imagine that this UI will evolve and include more features to aid in visualization and verification.