Oblig 1 The Traveling Salesman Problem
INF4490

Joseph Knutson
github.com/mathhat

October 2, 2017

Contents

1 Exhaustive search

This section guides you through the code, efficiency and result of the exhaustive search method for the traveling salesman problem.

1.1 Creating the Code

Making my Exhaustive Search code began with using the example.py file on the course site which imports the city grid. From there I followed the advice of the assignment regarding the itertools module's permutations function. Looping over every sequence and summing the distances for each sequence, the final Exhaustive search function looked something like this:

```python
start = time.time()                    #start clock
for sequence in Permutations:          #exhaustive search begins
    dist = 0
    for index in range(cities -1):
        dist += distances[sequence[index]][sequence[index+1]]
    dist += distances[sequence[cities -1]][sequence[0]]
    if dist < best:                    #save shortest distance yet
        best=dist
        best_sequence = sequence

end = time.time()                      #end clock
Time = (end-start)                     #sum time
return(best, best_sequence, Time)
```

You can observe on the last line that the function returns the shortest path's distance, the path sequence and the time it took to iterate over all the permutations.

1.2 Timetables

The time it takes for the program to run varies with the amount of cities we add. However, it is not sufficiently accurate to measure only once. Calling the Exhaustive Search function 10 times helped create an approximate time average for its execution time, and I did so for the first 6 to 10 cities in the european_cities.csv file. The result can be seen in figure 1. The code calculating the timeaverages and producing the plot lies in the exhaustive_time.py file.

If you go into the time_exhaustive.txt file (created by running exhaustive_time.py), you can quickly find how long the calculations took (below). Let's use these numbers to predict how long it takes to use Exhaustive Search with 24 cities.

```
Time average for 7 cities = 0.005085 seconds

Time average for 8 cities = 0.042169 seconds

Time average for 9 cities = 0.420994 seconds

Time average for 10 cities = 4.878382 seconds
```

To find out how long simulations of a higher city number will take, we try to make a model based on the amount of flops (+ - * /) that are executed.
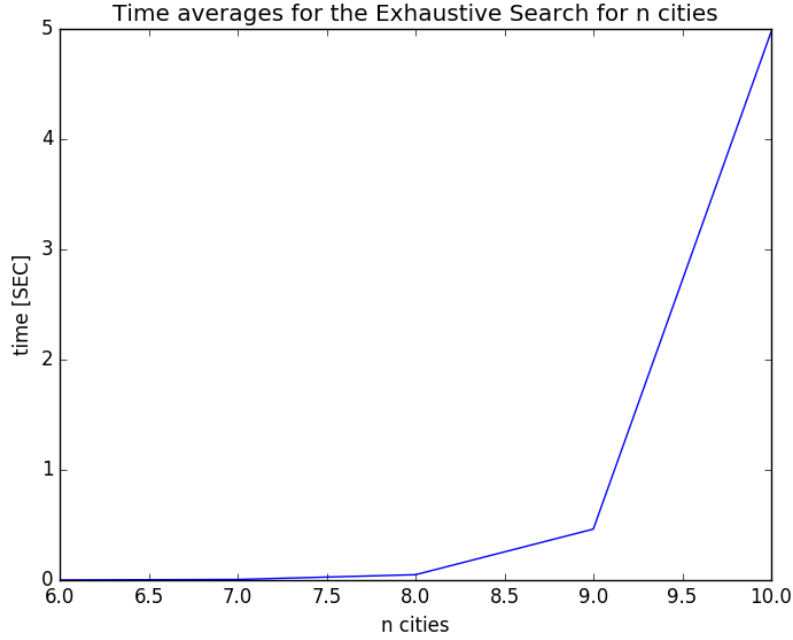
Figure 1: Time to calculate shortest path as a function of the number of cities.

The number of permutations for 10 cities are equal to N!= 3628800 and for each permutation (or path) one has to add the distances between each city to calculate the total distance of the path. This leaves us with N*N!= 36288000 floating point operations (flops). If we divide the amount of flops the program used on the runtime of the program, we get an approximation of how many flops (additions) occur per second:

$$\text{flops\_per\_sec} = \frac{36288000}{4.878382\text{sec}} = 7438531.87389\text{flops/sec}$$

This approximation of a constant can give us an idea of how long it takes to run an exhaustive search for a larger amount of cities. To find the runtime of a larger path with 24 cities, we need to find the amount of flops the program will require (N*N!):

$$\text{flops}(24 \text{ cities}) = 24 * 24! = 1.4890762 \ 10^{25}\text{flops}$$

Now we can calculate the seconds this program will need to finish running (not to mention the insane amount of RAM):

$$\text{runtime of ES for 24 cities} = \frac{1.4890762 \ 10^{25}\text{flops}}{\text{flops\_per\_sec}} \approx 63.4 \ 10^{9}\text{years}$$

The calculations make it obvious that Exhaustive Search is useless once the amount of cities pass 10.

1.3 Shortest Path Result

Our Exhaustive Search function returns not only time, but also the distance, *best*, and the sequence of cities traveled along the shortest path, *sequence*.

```
'Best' = ', 7486.309999999999
'Best sequence = ', (6, 8, 3, 7, 0, 1, 9, 4, 5, 2)

Barcelona, Belgrade, Berlin, Brussels, Bucharest, Budapest,
    Copenhagen, Dublin, Hamburg, Istanbul
```

Above is the result for $n_{cities} = 10$. I've written the cities in alphabetic order. The sequence has translated each city into a number from 0 to 9. If we place the 10 cities' names in the order that creates the shortest path, we get:

```
Copenhagen Hamburg Brussels Dublin Barcelona Belgrade Istanbul
    Bucharest Budapest Berlin
```

2 Hillclimber

This section compares my Hillclimber method with the Exhaustive Search method from the previous assignment. The method I've written is heavily inspired by the course book's (*Machine Learning*) example. The Hillclimber method tries to find a maxima or a minima of a fitness function by slightly tweaking the parameters of the problem in random ways, for our problem the tweaks involve reordering the sequence of cities we travel to. Like the example in the book, I pick any two random cities in an initially created path and switch their order of travel.

2.1 Code

```python
#Inspired by the book example

def hillclimb(distances, n_cities, seed):
    np.random.seed(seed)
    #updating seed gives different initial sequences per run

                 #order of cities we visit
    sequence = np.asarray(range(n_cities))

    np.random.shuffle(sequence)  #create an initial path, from
                                 #this order make small changes

    distanceTravelled = np.inf   #variable updated to shortest path

    i = 0                        #loop variable to signify 1000 changes

    while i < 1000:
        newDist = 0

        #declaring variable to compare a new-
        #path to the previously shortest path.

        #Choose 2 random integers representing cities and change
        #the initial path by switching/reordering their position
        city1 = np.random.randint(n_cities)
        city2 = np.random.randint(n_cities)

        if city1 != city2:
            i += 1

            #If the cities are not the same, the we switch
            #their position and count this  hillclimbing operation

            posSeq = sequence.copy()
            posSeq = np.where(posSeq==city1,-1, posSeq)
            posSeq = np.where(posSeq==city2,city1, posSeq)
            posSeq = np.where(posSeq==-1,city2, posSeq)

            #Here I simply sum up the distance of the path like in
    exhaustive search

            for j in range(n_cities-1):
                newDist += distances[posSeq[j]][posSeq[j+1]]
            newDist += distances[posSeq[-1]][posSeq[0]]
```

```
45                #Now we can compare  the  old  distance  with  the  new  path
46                #— created  by  the  hillclimbing  operation  above
47
48                if  newDist  <  distanceTravelled :
49                    distanceTravelled  =  newDist
50                    sequence  =  posSeq
51        return  sequence ,  distanceTravelled            #returns  both  path
         distance  and  which  order  the  cities  are  traveled  to
```

2.2   Results

For the hillclimbing method, we measured the distance traveled instead of the
time it takes to run the program (almost instantaneous). As you can see from
the while loop in the code above, I use the hillclimbing operator 1000 times for
each run, and I run the program 20 times for both

$$\text{n\_cities} = 10 \text{ and n\_cities} = 24.$$

This produces a heap of results from which we can pick the longest distance
(worst result) and the shortest distance (solution) and even calculate the stan-
dard deviation.

To do distance measurements and write them to file i simply import the hill-
climber function and call it 20 times.

The following code can be found in *hillclimbing_dist.py* and the results are pre-
calculated in the text files: *dist_hillclimber10cities.txt* and *dist_hillclimber10cities.txt*.

```
1   for  i  in  range ( n_sims ) :                        #n_sims  =  20
2       seq ,  dist  =   hillclimb ( distances ,  n_cities , i )  #i  is  also  used
        as  a  seed
3       lengths [ i]= dist
4
5   lengths= sorted ( lengths )  #sorting  the  results
6
7   File  =  open ( " dist_hillclimber%scities . txt "  %  n_cities ,  "w" )
8
9   for  i  in  range ( len ( lengths ) ) :
10      File . write ( "%.2 f"%lengths [ i ])
11      File . write ( "\n"%lengths [ i ])
12
13  File . write ( "\n"%lengths [ i ])
14  File . write ( " standard  dev  =  %.2 f"%standard_dev )
```

After writing the distances to file, we can print them in the terminal:

```
1   $ cat  dist_hillclimber10cities . txt
2   7486.31
3   7486.31
4   7486.31
5   7486.31
6   7503.10
7   7503.10
8   7503.10
9   7503.10
10  7503.10
11  7503.10
12  7503.10
13  7737.95
14  7737.95
```

```
15  7737.95
16  7737.95
17  7737.95
18  7737.95
19  7737.95
20  8349.94
21  8349.94
22
23  standard dev = 209.85
24
25  $ cat dist_hillclimber24cities.txt
26  13456.51
27  13650.46
28  13665.88
29  13806.32
30  13817.47
31  14119.10
32  14190.63
33  14209.14
34  14305.25
35  14314.38
36  14394.55
37  14410.91
38  14665.60
39  15329.68
40  15575.97
41  15695.83
42  16062.43
43  16256.35
44  16317.54
45  16381.76
46
47  standard dev = 961.10
```

The standard deviation from these results can be expressed as

$$\sigma = \sqrt{[E(x^2) - (E(x))^2]}$$

where E is the mean operator, x is the array of the distance of the paths found and x^2 is an array of these distances squared. Implemented, the standard deviation of our results look like this:

```
1  Mean = np.mean(lengths)
2  Mean_sq = np.mean(lengths*lengths)
3  standard_dev = np.sqrt(Mean_sq-Mean**2)
```

These expressions return the standard deviation values:
$\sigma(\text{n\_sims} = 20, \ \text{n\_cities} = 10) = 209.85$ and
$\sigma(\text{n\_sims} = 20, \ \text{n\_cities} = 24) = 961.10$

One unique seed is used for each of the 20 simulations/hillclimb calls, from seed = 1 to seed = 20.

Compared to the exhaustive search method, we easily solve the n_city = 10 problem by achieving a distance of 7486.31. We need a smarter algorithm, or more than 20 unique initial paths to solve the problem for all 24 cities when using hillclimbing, however. Since I've implemented a hillclimber which only has 1 populant and 1 offspring, I am stuck in a local maximum.

3 Genetic Algorithm

To solve the TSP, we're going to use partially mapped crossover with two parents and one offspring. After creating offspring by using the PMX operations as many times as there are parents, I am left with as many offspring as parents. From the offspring and parents, I choose elitism, and only pick the fittest. This defines 1 generation, but since my last delivery I now run 10 generations per run instead of 1. The population is held constant and my 3 tested population values are 10, 100 and 1000.

3.1 PMX code

The partially_mapped function defines most of what happens in my ga.py script, but the function is found in functions.py: this is how it looks

```python
def partially_mapped(parents, distances, n_cities, n_pop):

    offspring = np.zeros((n_pop,n_cities),int)      #matrix
      n_population x n_cities

    for iterations in range(n_pop):
        #Choosing parents to mate
        parent12 = np.random.randint(0,n_pop,2,int)
        parent1 = min(parent12)
        parent2 = max(parent12)
        index1 = np.random.randint(0,n_cities)
        index2 = index1+2
        if (index1 != index2) and (parent1 != parent2) :  #Making
      sure parents are different and -
            #Here starts the partially mapped crosseover

            #initial kids are identical to parents

            offspring[iterations] = parents[parent1]


            #crossover genomes/sequences
            genome1 = parents[parent1][index1:index2]
            genome2 = parents[parent2][index1:index2]

            #inserting genomes
            offspring[iterations][index1:index2] = genome2
            #offspring 1 gets sequence from parent 2
            #crossover from parents to offspring:
            for i in range(len(genome2)):
                if genome1[i] in genome2:
                    #instance where gene in crossover
                    #- sequence must be ignored
                    pass
                else:
                    #Here the fun pinball dynamics take place
                    gene = genome2[i]
                    success = 0
                    while success == False:
                        if gene == genome2[np.where(genome1==gene)
      ]:
                            success = True
```

```
41                            pass
42                        if gene in genome1: #This thing removes
     infinite loops
43                            gene = genome2[np.where(genome1==gene)]
44                        else:
45                            offspring[iterations][np.where(parents[
     parent1]==gene)] = genome1[i]
46                            success = True
47     return offspring
48
```

After the code creates offspring n_pop times by crossing over parents, the off-spring is returned, assembled with the parents, sorted and then checked for elimination. My sorting function takes a matrix filled with the parents and offspring combined and arranges the solutions by shortest path length:

```
1  def sort(pop_matrix, distances):
2      population = []      #best solutions
3      path_lengths = []    #way of finding best solutions
4      pop_matrix = pop_matrix[0]
5
6      n_cities = len(pop_matrix[0]) #city number
7      for sequence in pop_matrix:    #first calculate path lenths
8          dist = 0
9          if sum(sequence)>0:          #ignores sequences like
     [0,0,0,0,0]
10             for index in range(n_cities-1):
11                 dist += distances[sequence[index]][sequence[index
     +1]]
12             dist += distances[sequence[n_cities-1]][sequence[0]]
13             population.append(sequence)
14             path_lengths.append(dist)
15
16      population = np.asarray(population)
17      path_lengths = np.asarray(path_lengths)
18
19      ranked_paths = []
20      ranked_sequences = []
21      for i in range(len(population)): #then arrange them in
     diminishing order
22          ranked_paths.append(np.amin(path_lengths))
23          ranked_sequences.append(population[np.argmin(path_lengths)
     ])
24          path_lengths = np.delete(path_lengths, np.argmin(
     path_lengths))
25
26      return np.asarray(ranked_paths), np.asarray(ranked_sequences)
```

The population is held static by elimination. The mating is then resumed for another generation to go by. In my program *ga.py* the number of generations per run is set to 10. This means n_pop pmx operations followed by 1 filtering will happen 10 times per run.

3.2 Results

By running my ga.py script, three files are produced which return all sequences and their lengths in an ordered fashion. These files are already produced in my

folder and possess names like *GA_result_of_10cities_and_1000populants.txt*:

```
1  These are the results for a poplation of size 1000 who 20 times
       have jumped 10 generations.
2  For each generation, 1000 PMX operations create as many offspring
       as parents.
3  After each 1000 PMX operation, an elitist filter is applied to keep
       the population static and only the best solutions available.
4  The sequences are printed to terminal since I'm having trouble
       writing them to file.
5  path lengths in increasing order:
6  path lengths    sequences:
7    7486.31        9452683701   #1st/best
8    7486.31        9783102546
9    7486.31        8127039465
10   7486.31        8179345260
11   7486.31        3125798046
12   7486.31        1073862549
13   7663.51        9452687301
14   7663.70        9452863701
15   7729.01        9452867301
16   7775.10        3086795241
17       [...]
18   12818.33         9841052673 #1000th / worst
```

Here's the final result of the GA using 24 cities and 1000 populants with 10
generations per run, for 20 runs. *GA_result_of_24cities_and_1000populants.txt*:

```
1  These are the results for a poplation of size 1000 who 20 times
       have jumped 10 generations.
2  [...] path lengths in increasing order:
3  13915.25 #1st
4  13915.25
5  14746.64
6  14746.64
7  16138.36
8  16138.36
9  16138.36
10 16138.36
11 16138.36
12 [...]
13 30347.27 #1000th
```

From these results we can see that the best solution for the TSP is reachable.

4 Results Across Methods

As I've understood from the assignment, I am to pick the best individuals of the population for every run (20) and present the best, worst and their standard deviation. For low populations, the best individual tends to stay the same, which is why the standard deviation goes towards zero.

Here's data from the GA:

```
10 cities , 10 populants
('best of the 20 individuals ', 9737.3999999999996)
('worst of the 20 individuals ', 9737.3999999999996)
('standard dev of 20 best individuals ', 0.00017263349150062197)
10 cities , 100 populants
('best of the 20 individuals ', 7503.1000000000004)
('worst of the 20 individuals ', 7745.8000000000011)
('standard dev of 20 best individuals ', 52.89523867990529)
10 cities , 1000 populants
('best of the 20 individuals ', 7486.3099999999995)
('worst of the 20 individuals ', 7767.1600000000008)
('standard dev of 20 best individuals ', 59.641132207667056)


24 cities , 10 populants
('best of the 20 individuals ', 27431.400000000005)
('worst of the 20 individuals ', 27431.400000000005)
('standard dev of 20 best individuals ', 0.00034526698300124393)
24 cities , 100 populants
('best of the 20 individuals ', 18899.630000000001)
('worst of the 20 individuals ', 21879.710000000003)
('standard dev of 20 best individuals ', 750.81412341537668)
24 cities , 1000 populants
('best of the 20 individuals ', 13915.25)
('worst of the 20 individuals ', 21663.470000000001)
('standard dev of 20 best individuals ', 2109.1238597016604)
```

From what we see here, we can tell that a larger population tend to sprout the best individuals, but the standard deviation increases since the worst individuals differ largely from the best.