Oblig 1 The Traveling Salesman Problem

INF4490

Joseph Knutson
`github.com/mathhat`

September 25, 2017

Contents

1 Exhaustive search **2**
1.1 Creating the Code 2
1.2 Timetables 2
1.3 Shortest Path Result 4

2 Hillclimber **5**
2.1 Code 5
2.2 Results 6

3 Genetic Algorithm **8**
3.1 PMX code 8
3.2 results 9

4 Hybrid **9**

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:

```
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.

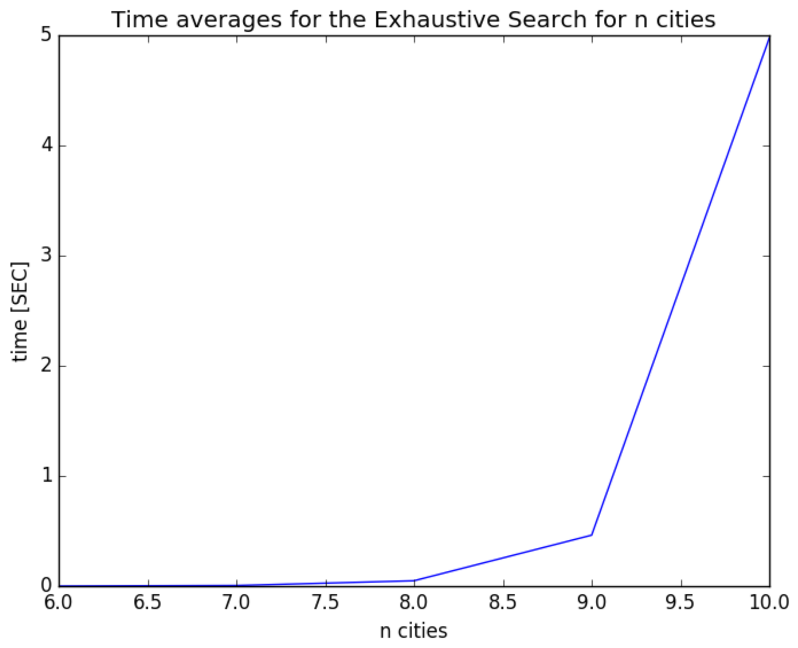

Figure 1: Time to calculate shortest path as a function of the number of cities.

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!):

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

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]]

            #Now we can compare the old distance with the new path
            #- created by the hillclimbing operation above

            if newDist < distanceTravelled:
                distanceTravelled = newDist
                sequence = posSeq
    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

n_cities = 10 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 standard deviation.

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

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

```
 for i in range(n_sims):                        #n_sims = 20
    seq, dist =  hillclimb(distances, n_cities,i) #i is also used
    as a seed
    lengths[i]=dist

lengths=sorted(lengths) #sorting the results

File = open("dist_hillclimber%scities.txt" % n_cities, "w")

for i in range(len(lengths)):
    File.write("%.2f"%lengths[i])
    File.write("\n"%lengths[i])

File.write("\n"%lengths[i])
File.write("standard dev = %.2f"%standard_dev)
```

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

```
$ cat dist_hillclimber10cities.txt
7486.31
7486.31
7486.31
7486.31
7503.10
7503.10
7503.10
7503.10
7503.10
7503.10
7503.10
7737.95
7737.95
```

```
7737.95
7737.95
7737.95
7737.95
7737.95
8349.94
8349.94

standard dev = 209.85

$ cat dist_hillclimber24cities.txt
13456.51
13650.46
13665.88
13806.32
13817.47
14119.10
14190.63
14209.14
14305.25
14314.38
14394.55
14410.91
14665.60
15329.68
15575.97
15695.83
16062.43
16256.35
16317.54
16381.76

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:

```
Mean = np.mean(lengths)
Mean_sq = np.mean(lengths*lengths)
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 an amount of offspring of a lesser or equal amount of the parents. From the offspring and parents, I choose elitism, and only pick the fittest. The population is held constant and my population values 10, 100 and 1000.

3.1 PMX code

The code was written just now and I haven't had the time to comment properly on it, but I hope it is sligthly readable:

```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)
        #Choosing interval of chromosome to cross over

        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[parent1] = parents[parent1]
            offspring[parent2] = parents[parent2]


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


            #inserting genomes
            offspring[parent1][index1:index2] = genome2 #offspring 1 gets sequence from parent 2
            offspring[parent2][index1:index2] = genome1 #offspring 2 gets sequence from parent 1
            #crossover from parents to offspring 1
            for i in range(len(genome2)):
                if genome1[i] in genome2:
                    pass
                else:
                    gene = genome2[i]
                    success = 0
                    while success == False:

                        if gene == genome2[np.where(genome1==gene)]:
                            success = True
                            pass
                        if gene in genome1:
                            gene = genome2[np.where(genome1==gene)]
                            #genomeswitch = genome1
                            #genome1 = genome2
                            #genome2 = genomeswitch
                        else:
                            offspring[parent1][np.where(parents[parent1]==gene)] = genome1[i]
                            success = True
    return offspring
```

After the code creates offspring n_pop times by crossing over parents, the offspring is returned, assembled with the parents, sorted and then checked for elimination. The population is held static by elimination. The mating is then resumed for another generation to go by.

3.2 results

to be continued

4 Hybrid