Assignment_5_Traveling_Salesman_[netid]

September 27, 2021

1 Assignment 4: The Traveling Salesman Problem

In our last assignment, we added crossover to the mix, completing the last major piece of our evolutionary algorithm framework! Let's use this basic pipeline to explore a real-world applied problem to help gain an understanding of the complexity of using respresnetations beyond basic bit-strings.

In this assignment, we'll explore the Traveling Salesman Problem (TSP). This problem specifies that a solution should visit each of a list of locations exactly one time, and do so by traveling the shortest distance possible.

```
[ ]: # imports
     import numpy as np
     import copy
     import matplotlib.pyplot as plt
     plt.style.use('seaborn')

     import scikits.bootstrap as bootstrap
     import warnings
     warnings.filterwarnings('ignore') # Danger, Will Robinson! (not a scalable␣
      ↪hack, and may surpress other helpful warning other than for ill-conditioned␣
      ↪bootstrapped CI distributions)

     import scipy.stats # for finding statistical significance

     from IPython.display import clear_output # for real-time displays
     import time # in case you want to analyze runtime of your code
```

Most of my family's road trips are based around getting outside a seeing national parks, so let's optimize a TSP to visit the list of US National Parks. This list of all parks, and their coordinates, is included in the zip folder for this assignment. The below block of code will load this data, ignoring the parks outside the continental US (so we can pretend to drive, instead of flying to Alaska or Hawaii). Ignoring these, we're left with a list of 50 parks (what counts as a "National Park" is a bit fuzzy, but this is the list I scraped from Wikipedia). For the most part we'll actually be ignoring the `park_names` and working with `park_lat_long`, which is an 2-D array that provides the coordiantes of each park (indexed in alphabetical order to match the names list – not that this matters much for our purposes in this assignment).

[ ]:
```
file = open("parks_list.csv")
park_names = []
park_lat_long = []
for line in file: # get name, latitute, and longitude of park from file
    park_name = line.split(",")[0].strip()
    park_lat = float(line.split(",")[1].strip())
    park_long = float(line.split(",")[2].strip())

    if park_lat > -125 and park_lat < -65 and park_long > 25 and park_lat < 50:␣
 ↪# just look at continental US
        park_names.append(park_name)
        park_lat_long.append([park_lat,park_long])

file.close()
park_lat_long = np.array(park_lat_long) # convert to numpy array for easier␣
 ↪indexing/slicing

print("Number of National Parks in Continental US:",len(park_names))
print(park_names)

print(park_lat_long)
```

Given that we know the coordinates of each park, a fun dataviz we can do is to superimpose the parks on a map of the United States.

[ ]:
```
fig, ax = plt.subplots(figsize=(16, 8)) # generate figure and axes
ax.imshow(plt.imread("us_map.png"), extent=[-125, -65, 25, 50]) # import␣
 ↪backbround
ax.scatter(park_lat_long[:,0],park_lat_long[:,1],color='m',s=100) # plot each␣
 ↪park at its coordinates
ax.set_xlim([-125,-65]);
ax.set_ylim([25,50]);
ax.grid(False);
```

Since we'll be opitmizing routes between these parks, it would also be good to be able to draw routes between two parks. For the sake of simplicity (i.e. so you don't need to interface with the Google Maps API), we'll just consider straight-line distance between parks as a proxy for driving distance. An example of how to draw a line showing a path between parks might look something like this.

[ ]:
```
fig, ax = plt.subplots(figsize=(16, 8)) # generate figure and axes
ax.imshow(plt.imread("us_map.png"), extent=[-125, -65, 25, 50]) # import␣
 ↪backbround
ax.scatter(park_lat_long[:,0],park_lat_long[:,1],color='m',s=100) # plot each␣
 ↪park at its coordinates
ax.set_xlim([-125,-65]);
ax.set_ylim([25,50]);
ax.grid(False);

park1_index = 0 # pick to arbitrary parks for this demo example
park2_index = 23
ax.
 ↪plot([park_lat_long[park1_index,0],park_lat_long[park2_index,0]],[park_lat_long[park1_index
 ↪ # draw a line between them
```

1.0.1 Q1: TSP Ftiness Function

Define a fitness function that takes a genome of park indexes and calculates the total round trip distance from the starting point (first park) back around a full cycle to end up at the park you started in.

Recall that we are using the straight-line distance between parks (and you may find it useful to make a helper function which calculated the distance between any two indidivual parks in calculating the total distance traveled).

To minimize the modifications necessary to your existing evolutionary algorithm code, which maximized fitness, return the negative of the distance traveled (i.e. fitness will always be less than zero, with longer trips more negative), so we can continue to maximize fitness values when finding the shortest part for the TSP.

Hint: Even though the fitness function definition comes first, it may be easier to complete after defining the genome representation below

```
[ ]: def distance_between_parks(park1_index,park2_index):
         """ Optional helper function to calculate straight-line distance between␣
      ↪two parks

             parameters:
             park1_index: (int) location of first park in genome
             park2_index: (int) location of second park in genome

             returns:
             distance: (float) straight-line distance between parks
         """


         ...

     def traveling_salesman_fitness_function(genome):
         """ Fitness function for TSP.  Round trip (i.e. full cycle) distance␣
      ↪traveled by the "traveling salesman"

             parameters:
             genome: (numpy array) individual's genome containing ordered list of␣
      ↪parks to visit
```

```
        returns:
        distance: (float) straight-line total distance for visiting all parks
    """
    ...
```

1.0.2 Q2: Adapting the Evolutionary Algorithm for the TSP

Copy over your evolutionary algorithm code (both the `Individual` class and the `evolutionary_algorithm` function) from your last assignemnt (or the posted solutions) to provide a framework to get started with. Now please adopt this code to sovle the TSP. Let's begin with the simple case of just mutation and not crossover. Let's also focus on the simpler case of a hard constraint on visiting each loaction just once.

We mentioned a number of potential representations in class last week, please pick one that enforces each location be visited only once as part of the encoding.

Let's also begin with a mutation opperator we discussed in class, randomly selecting two locations and swapping their place in the visitation order.

Additionally, for visualizations later on, please save the timeseries of solutions over time (in addition to the fitness values over time you were saving last assignment).

Hint: If your genome is a set of indexes, don't forget that they need to be ints (not floats).

Hint: If you're fitness values are all negative, keep this in mind when initializing your best-solution-so-far value.

[ ]:
```
class Individual:

    def __init__(self, fitness_function, genome_length):
        ...

    def eval_fitness(self):
        ...
```

[ ]:
```
def evolutionary_algorithm(fitness_function=None, total_generations=100,␣
↪num_parents=10, num_children=10, genome_length=10, num_elements_to_mutate=1,␣
↪crossover=True, tournament_size=4, num_tournament_winners=2):
    """ Evolutinary Algorithm (copied from the basic hillclimber in our last␣
↪assignment)

        parameters:
        fitness_funciton: (callable function) that return the fitness of a␣
↪genome
                          given the genome as an input parameter (e.g. as␣
↪defined in Landscape)
```

```
        total_generations: (int) number of total iterations for stopping␣
↪condition
        num_parents: (int) the number of parents we downselect to at each␣
↪generation (mu)
        num_childre: (int) the number of children (note: parents not included␣
↪in this count) that we baloon to each generation (lambda)
        genome_length: (int) length of the genome to be evoloved
        num_elements_to_mutate: (int) number of alleles to modify during␣
↪mutation (0 = no mutation)
        crossover: (bool) whether to perform crossover when generating children
        tournament_size: (int) number of individuals competing in each␣
↪tournament
        num_tournament_winners: (int) number of individuals selected as future␣
↪parents from each tournament (must be less than tournament_size)

        returns:
        fitness_over_time: (numpy array) track record of the top fitness value␣
↪at each generation
        solutions_over_time: (numpy array) track record of the top genome value␣
↪at each generation
    """

    # initialize record keeping
    ...

    # the initialization proceedure
    ...

    # get population fitness
    ...

    ... # repeat

        # the modification procedure
        ...

            # inheretance
            ...

            # crossover
            # N/A

            # mutation
            ...

            ... # add children to the new_children list
        # the assessement procedure
        ...

        # selection procedure
       ...

        # tournament selection
        ...

        # record keeping
        ...

    return fitness_over_time, solutions_over_time
```

As usual, let's store our results for later plotting

[ ]:
```
experiment_results = {}
solutions = {}
```

1.0.3 Q2: Collect and Analyze Results

Similar to last week, let's run multiple trials to systematically test our algorithm. To keep compute times down, let's start with a smaller subset of the problem, using just the first half of the parks in our dataset.

[ ]:
```
park_names = park_names[:25]
park_lat_long = park_lat_long[:25]
```

In this smaller problem, let's run for `100` generations, `50` parents + `50` childeren in a mu+alpha evolutionary strategies, a tournament selection of tournaments of size `10` with `2` winners selected at each tournament. Let's just run this for `20` independent trials.

For reference (as there were runtime length questions about the last assignment), in my implementation of this, each run taks a little over a second, so all 20 runs finish in under 30 seconds.

[ ]:
```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
genome_length = len(park_names)
num_parents = 50
num_children = 50
tournament_size = 10
num_tournament_winners = 2

...
```

Again pulling from the previous assignment, please plot the mean and boostrapped confidence interval of your experiements. Again, you may find it convenient to make this into a function (and if you do, FYI, later use cases may involve plotting just the mean without the CI, since this boostrapping proceedure can be computationally expensive).

```
[ ]: def plot_mean_and_bootstrapped_ci_over_time(input_data = None, name = "change␣
     ↪me", x_label = "change me", y_label="change me", y_limit = None,␣
     ↪plot_bootstrap = True):
         """

         parameters:
         input_data: (numpy array of shape (max_k, num_repitions)) solution metric␣
     ↪to plot
         name: (string) name for legend
         x_label: (string) x axis label
         y_label: (string) y axis label

         returns:
         None
         """

         ...
```

```
[ ]: # plotting
     ...
```

One of the most fun parts of working in machine learning (for me) is to see the solutions take shape over time. Here's I've written a function that takes in your solutions_over_time from a single trial run above, and visualizes its optimization over time.

```
[ ]: def show_solution_evolution(solutions_over_time, final_solution_only= False):
         """ Show animation of evolutionary optimization for TSP.

         parameters:
         solutions_over_time: (numpy array) track record of the top genome value at␣
     ↪each generation
         final_solution_only: (bool) flag to skip animation

         returns:
         None
         """

         solutions_over_time = solutions_over_time.astype(int) # in case you forgot␣
     ↪to cast array type

         last_fitness = 0
         for i in range(total_generations):
```

```
        if final_solution_only: i = total_generations-1 # skip to end if not␣
↪showing full animation

        genome = solutions_over_time[i]
        fitness = traveling_salesman_fitness_function(genome)

        if fitness != last_fitness: # only show new solution, if different from␣
↪the last generation
            last_fitness = fitness
            print("Generation:",i,"\nFitness:",fitness)
            clear_output(wait=True) # erase prior figure to enable animation
            fig, ax = plt.subplots(figsize=(16, 8)) # generate figure and axes
            ax.grid(False)

            ax.imshow(plt.imread("us_map.png"), extent=[-125, -65, 25, 50]) #␣
↪plot map background
            ax.scatter(park_lat_long[:,0],park_lat_long[:,1],color='m',s=100) #␣
↪plot park points
            ax.set_xlim([-125,-65]);
            ax.set_ylim([25,50]);

            for park_index in range(len(genome)): # for each park in the␣
↪solution
                ax.
↪plot([park_lat_long[genome[park_index],0],park_lat_long[genome[(park_index+1)%len(genome)],
↪# draw a line from it to the following park (looping back around to start␣
↪when at end)
            plt.show()

    last_fitness = fitness
    print("Generation:",i,"\nFitness:",fitness)
```

1.0.4 Q3: Inspecting Results

Write a quick script to find the best performing single run that you've found so far, and show its evolution over time using the `show_solution_evolution` funciton given above.

[ ]:
```
# plot best solution found thus far as interactive plot
```

1.0.5 Q3b: Applying Intuition

One of the nice parts about viewing the solutions coming together over time is that you can see the stepping stones taken to get to good solutions, or where your algorithm may have gotten stuck. Do the results that you see make sense? Do you notice anything in particular about the intermediate solutions or search strategy that might inform further algorithmic ideas for the TSP?

insert text here

1.0.6 Q4: Alternative Mutation Opperators

What's a different mutation opperator that you might wanto to implement?

One that comes to mind for me would be the idea of randomly moving a single entry (gene/park) to another random location in the genome (vs. swapping two).

What are the potential implication that you could see stemming from this change (or whatever other alternative mutation opperator you want to implement below)?

insert text here

1.0.7 Q4b: Implementation

Implement this mutation opperator in the cell below

```
[ ]: def evolutionary_algorithm(fitness_function=None, total_generations=100,␣
  ↪num_parents=10, num_children=10, genome_length=10, num_elements_to_mutate=1,␣
  ↪crossover=True, tournament_size=4, num_tournament_winners=2):
        """ Evolutinary Algorithm (copied from the basic hillclimber in our last␣
  ↪assignment)

            parameters:
            fitness_funciton: (callable function) that return the fitness of a␣
  ↪genome
                                  given the genome as an input parameter (e.g. as␣
  ↪defined in Landscape)
            total_generations: (int) number of total iterations for stopping␣
  ↪condition
            num_parents: (int) the number of parents we downselect to at each␣
  ↪generation (mu)
            num_childre: (int) the number of children (note: parents not included␣
  ↪in this count) that we baloon to each generation (lambda)
            genome_length: (int) length of the genome to be evoloved
            num_elements_to_mutate: (int) number of alleles to modify during␣
  ↪mutation (0 = no mutation)
            crossover: (bool) whether to perform crossover when generating children
            tournament_size: (int) number of individuals competing in each␣
  ↪tournament
            num_tournament_winners: (int) number of individuals selected as future␣
  ↪parents from each tournament (must be less than tournament_size)

            returns:
```

```
        fitness_over_time: (numpy array) track record of the top fitness value␣
↪at each generation
    """

    ...

    return fitness_over_time, solutions_over_time # for simplicity, return just␣
↪the fitness_over_time record
```

1.0.8 Q5: Run and Plot Experiment

Yeah, do that.

[ ]:
```
num_runs = 20
total_generations = 100
num_elements_to_mutate = 1
genome_length = len(park_names)
num_parents = 50
num_children = 50
tournament_size = 10
num_tournament_winners = 2
crossover = False

...
```

[ ]:
```
# plotting
...
```

It may also be helpful (or just fun) to visualize the evolution over time as well.

[ ]:
```
# interactive viz
...
```

1.0.9 Q5b: Analysis

Did this approach work better? Worse? Indistinguishable from the first? Was this what you expected? What does the difference in mutation opperator performance suggest to you about the methods or benefits of crossover?

insert text here

1.0.10 Q6: Crossover

Let's also implement a crossover for the TSP. There are many approaches to crossover that may be appropriate, and you should feel free to choose any (e.g. the simplest) you want. I would encourage you to think about crossover approaches that maintain the visit-each-location-once hard constraint as part of the crossover mechanism (but as dicussed in class, you're welcome to also explore approaches that visit each location multiple times, then prune down to a single complete cycle before sending the resulting genome off for evaluation).

If you're short on ideas, this paper has a few examples that might be fun to implement.

For simplicity's sake, feel free to just add crossover in addition to your favorite of the already implemented mutation opperators (rather than having to explore every permutation of mutation and/or crossover approaches). I'll just tell you that crossover on its own (given the hyperparameters we have here) doesn't work particularly well – as you might expect and we discussed in class.

[ ]:
```python
def evolutionary_algorithm(fitness_function=None, total_generations=100,␣
↪num_parents=10, num_children=10, genome_length=10, num_elements_to_mutate=1,␣
↪crossover=True, tournament_size=4, num_tournament_winners=2):
    """ Evolutinary Algorithm (copied from the basic hillclimber in our last␣
↪assignment)

        parameters:
        fitness_funciton: (callable function) that return the fitness of a␣
↪genome
                          given the genome as an input parameter (e.g. as␣
↪defined in Landscape)
        total_generations: (int) number of total iterations for stopping␣
↪condition
        num_parents: (int) the number of parents we downselect to at each␣
↪generation (mu)
        num_childre: (int) the number of children (note: parents not included␣
↪in this count) that we baloon to each generation (lambda)
        genome_length: (int) length of the genome to be evoloved
        num_elements_to_mutate: (int) number of alleles to modify during␣
↪mutation (0 = no mutation)
        crossover: (bool) whether to perform crossover when generating children
        tournament_size: (int) number of individuals competing in each␣
↪tournament
        num_tournament_winners: (int) number of individuals selected as future␣
↪parents from each tournament (must be less than tournament_size)

        returns:
        fitness_over_time: (numpy array) track record of the top fitness value␣
↪at each generation
    """

   ...

    return fitness_over_time, solutions_over_time # for simplicity, return just␣
↪the fitness_over_time record
```

1.0.11 Q7: Run and Plot

Just like always.

```
[ ]: num_runs = 20
     total_generations = 100
     num_elements_to_mutate = 1
     genome_length = len(park_names)
     num_parents = 50
     num_children = 50
     tournament_size = 10
     num_tournament_winners = 2
     crossover = True

     ...
```

```
[ ]: # plotting
     ...
```

1.0.12 Q7b: Analysis

Did crossover help? Any thoughts as to why or why not? Is this what you expected?

insert text here

1.0.13 Q8: Scaling Up

Now that you've tested some approaches on a subset of the data, let's try the best method on the full park list. First let's re-import the complete parks list.

```
[ ]: file = open("parks_list.csv")
     park_names = []
     park_lat_long = []
     for line in file: # get name, latitute, and longitude of park from file
         park_name = line.split(",")[0].strip()
         park_lat = float(line.split(",")[1].strip())
         park_long = float(line.split(",")[2].strip())

         if park_lat > -125 and park_lat < -65 and park_long > 25 and park_lat < 50:␣
      ↪# just look at continental US
             park_names.append(park_name)
             park_lat_long.append([park_lat,park_long])

     file.close()
     park_lat_long = np.array(park_lat_long) # convert to numpy array for easier␣
      ↪indexing/slicing
```

```
print("Number of National Parks in Continental US:",len(park_names))
print(park_names)
```

This larger dataset will need to run for a bit longer, let's say `1000` generations. We can keep all other parameters the same, though let's drop the number of independent trials down to `10`, again to keep computation reasonable. Each run takes a little under 20 seconds on my laptop, so this should still only take a few minutes to run.

Feel free to just run your best/favorite approach (e.g. combination of mutation and/or crossover opperators) from the smaller dataset here. No need to compare multiple implementations right now.

[ ]:
```
num_runs = 10
total_generations = 1000
num_elements_to_mutate = 1
genome_length = len(park_names)
num_parents = 50
num_children = 50
tournament_size = 10
num_tournament_winners = 2
crossover = False

...
```

Please also plot the fitness over time. Though feel free to omit the boostrapped confidence intervals, as they take quite some time to run over 1000 generations.

[ ]:
```
# plotting
...
```

Let's also visualize the optimization over time!

[ ]:
```
# interactive viz
```

1.0.14 Q9: Analysis at Scale

Do you notice anything different viewing these results at scale? Does it work as well? What relationships or patterns do you see between the smaller and larger TSP problems that might help to inform how well of a proxy one is for the other?

insert text here

1.0.15 Q10: Future Work

If any of these results were less than perfect solutions (including this last one at scale), what might still be standing in the way? You might think about our discussions on search landscapes, local optima, and selection pressure, or consider additonal modificaitons to the variation pperators, selction criteria, or viability constraints, among many other ideas of potential future directions.

insert text here

1.0.16 Congratulations, you made it to the end!

Nice work – and hopefully you're starting to get the hang of these!

Please save this file as a .ipynb, and also download it as a .pdf, uploading **both** to blackboard to complete this assignment.

For your submission, please make sure that you have renamed this file (and that the resulting pdf follows suit) to replce `[netid]` with your UVM netid. This will greatly simplify our grading pipeline, and make sure that you receive credit for your work.

Academic Integrity Attribution During this assignment I collaborated with:

insert text here