03_local_search

April 11, 2019

1 Testat Local Search

1.1 Info

- All Questions answerd in the Document itself.
- Only hill climbing done so far (but as requested in Testat)
- This exercise was done in a Team (Michael Nebroj & Steve Ineichen)
- Code can be found on https://github.com/Inux/aiso

1.2 Local Search Algorithms

We have seen the following local search algorithms in class:

1. Hill Climbing
2. Genetic Algorithm
3. Simulated Annealing

Two test these algorithms, we want to find the shortest path connecting several cities. Here, we can simplify and consider the aerial distance between two cities, so that we don't have to care about how to get from one city to another. This problem is known as the "Traveling Sales(man) Problem".

Implement the algorithms to find the shortest path connecting a list of cities of your choice! You can use the following helper functions to plot your path and to evaluate a path cost.

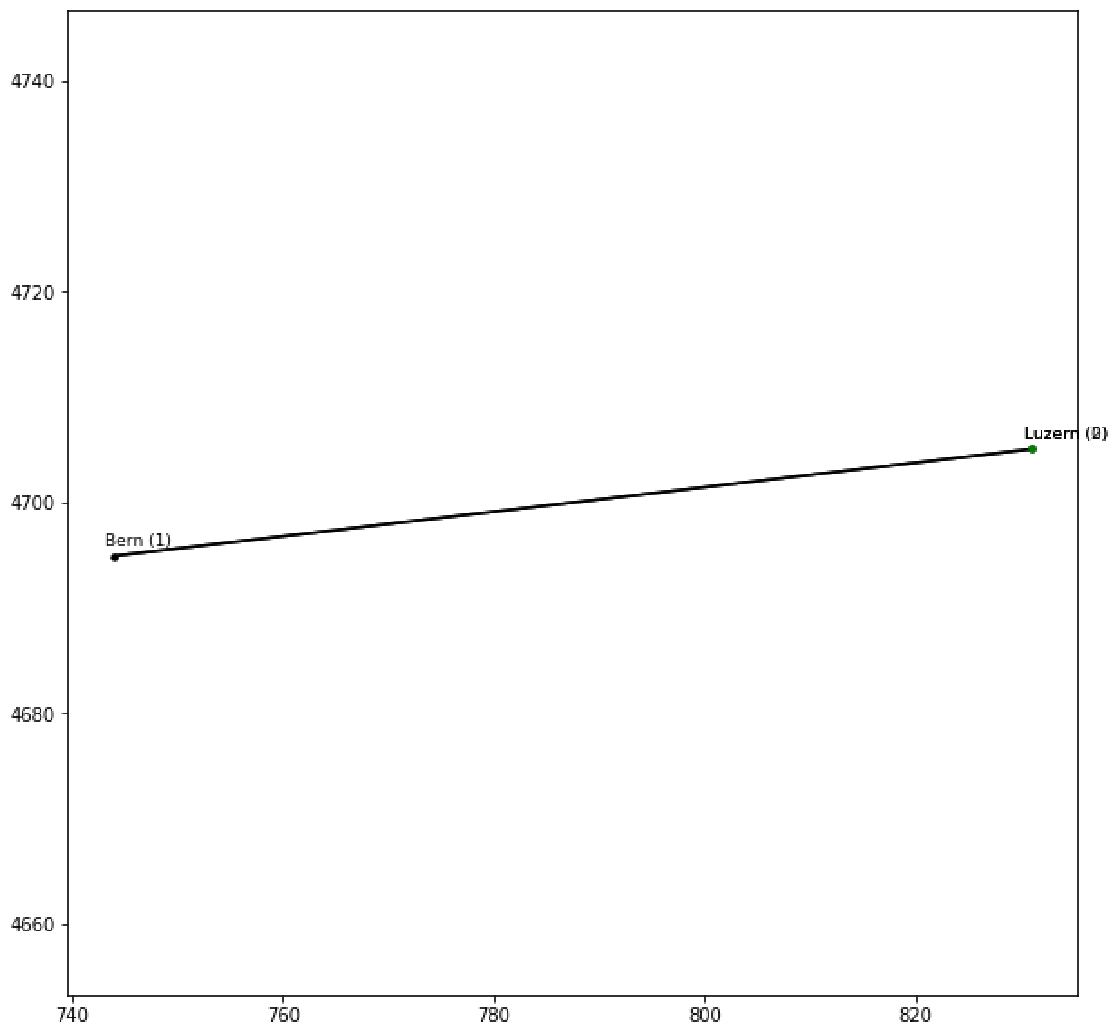
```
In [204]: import matplotlib.pyplot as plt

          def plot_path(path, sbb):
              fig = plt.figure(figsize=(10,10))
              last_city = ""
              hub_coordinates = sbb.get_hub_locations()
              count = 0
              for city in path:
                  if last_city == "":
                      first_city = city;
                  if last_city != "":
                      if count == 0:
                          plt.plot([hub_coordinates[city][0], hub_coordinates[last_city][0]], [h
                      else:
```

```
                    plt.plot([hub_coordinates[city][0], hub_coordinates[last_city][0]], [h
                plt.text(hub_coordinates[city][0]-0.75, hub_coordinates[city][1]+1.0, city+" (
                last_city = city;

                count = count + 1

            plt.plot([hub_coordinates[first_city][0], hub_coordinates[last_city][0]], [hub_coo
            plt.axis('equal')

        def evaluate_path(path, sbb):
            l = list(path.copy())
            length = 0
            for i in range(len(l)-1):
                length = length + sbb.get_distance_between(l[i], l[i+1])

            return length
```

Let's define the cities we want to connect and viualize our initial path.

```
In [205]: from sbb import SBB

          sbb = SBB()
          sbb.importData('linie-mit-betriebspunkten.json')

          path = ['Bern', 'Luzern']
          print("Bern - Luzern: path cost = " + str(evaluate_path(path, sbb)))

          path = ['Luzern', 'Bern']
          print("Luzern - Bern: path cost = " + str(evaluate_path(path, sbb)))

          path = ['Luzern', 'Bern', 'Luzern']
          print("Luzern - Bern - Luzern: path cost = " + str(evaluate_path(path, sbb)))

          plot_path(path, sbb)
```

```
successfully imported 2787 hubs
successfully imported 401 train lines
Bern - Luzern: path cost = 87.69281937407082
Luzern - Bern: path cost = 87.69281937407082
Luzern - Bern - Luzern: path cost = 175.38563874814164
```



This is defenitly not the best way how to connect the cities. Let's try our first local search algorithm.

1.2.1 Hill Climbing

If we have 15 cities to connect, we have 15! different possibilities. This is already bigger than 10ˆ12. You can easily see that the problem becomes complex very quickly. We will have problems to systematically explore the search space. Therefore, we will use local search algorithms to tackle this problem.

Local search algorithms start with a solution and try to improve the solution by considering the neighbouring states. The best neighbour will be chosen until no better can be found.

Here, we try to minimize the distance of our path. So instead of hill climbing, we will do the opposite. Instead of trying to find the highest hill (maximum), we're looking at the deepest valley (minimum). But that's not a concern, we can easily change the sign to switch from a maximization to a minimization problem.

Hints: - use the `evaluate_path()` function we have defined earlier - make sure to copy lists or sets properly: `current_path = path.copy()` - you can convert sets to lists by `list(my_set)` - a neighbouring path can be found by switching the position of two cities

```
In [206]: import sys
          import random

          def hill_climbing_TSP(path, sbb):
              search_path = list(path).copy()

              result = search_path.copy()
              new_result = result.copy()

              max_iterations = len(path)*10
              find_result_attempts = len(path)
              remaining_attempts = find_result_attempts

              iteration_count = 0

              print("Max Iterations: {0}".format(max_iterations))
              print("Max Attempts: {0}".format(find_result_attempts))
              print("Remaining Attempts: {0}".format(remaining_attempts))
              print("")

              for iteration in range(max_iterations):
                  iteration_count = iteration_count + 1

                  # https://stackoverflow.com/questions/14971181/hill-climbing-search-algorithm-
                  #
                  # - start with given path, maybe its a good one already
                  # - len(path)-2 because we dont want to change start and end position
                  #
                  for index in range(0, len(path)-2):
                      if index > 0: #check given path normally, without swapping
                          new_result[index], new_result[index+1] = new_result[index+1], new_resu

                      if evaluate_path(new_result, sbb) < evaluate_path(result, sbb):
                          print("INFO: sp_copy: {0} < result: {1}".format(evaluate_path(new_resu
                          result = new_result.copy()
                          print("INFO: new result -> score: {0}, path: {1}".format(evaluate_path
                      else:
                          #no better result
                          remaining_attempts = remaining_attempts - 1

                          #randomize if shifting brings no better result
                          if remaining_attempts <= 0:
                              print("ERROR: {0} times no better result! Randomize!".format(find_
                              r = new_result[1:(len(new_result)-1)]
```

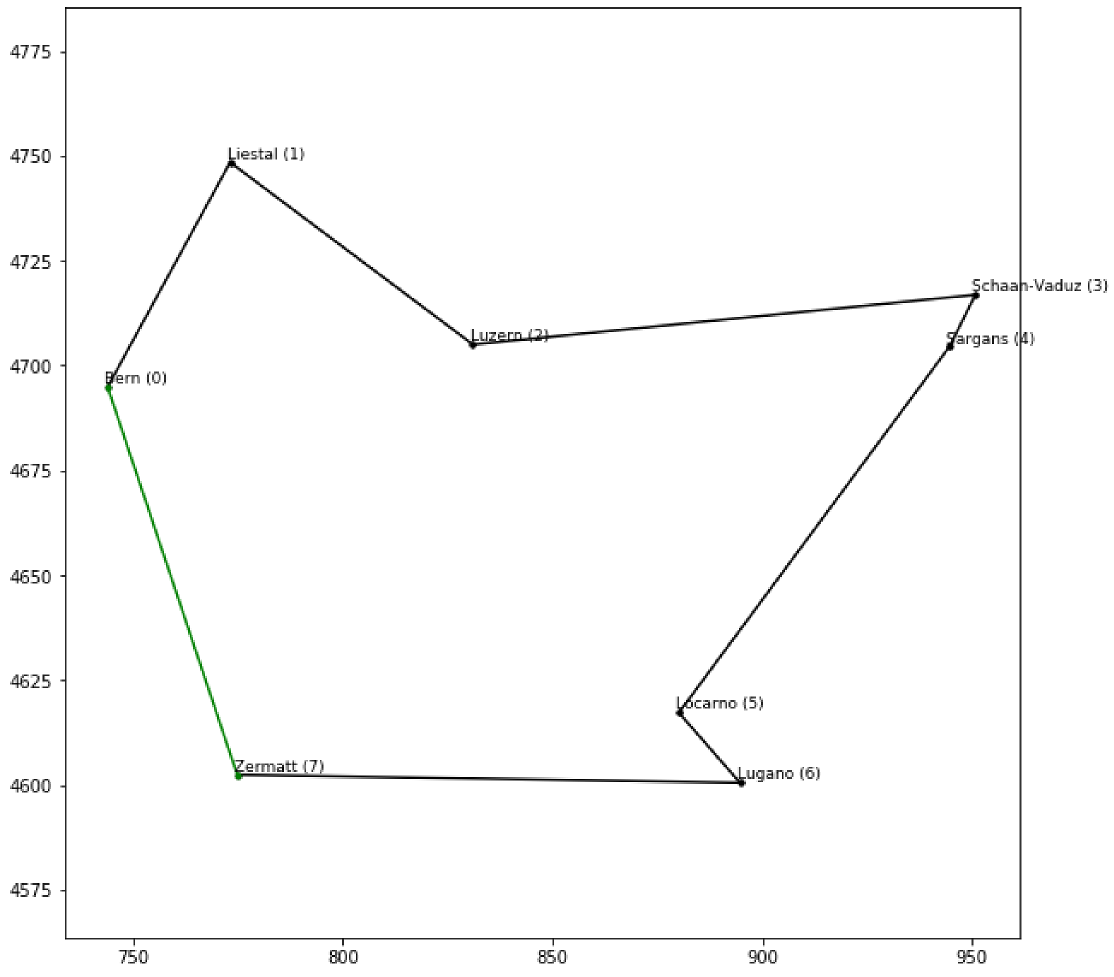
```
                                random.shuffle(r)
                                new_result[1:(len(new_result)-1)] = r

                                #give again same amount of attempts
                                remaining_attempts = find_result_attempts

                return result, iteration_count
```

```
In [207]: path = ['Bern', 'Sargans', 'Liestal', 'Lugano', 'Locarno', 'Luzern', 'Schaan-Vaduz', '

          best_path, iterations = hill_climbing_TSP(path, sbb)

          plot_path(best_path, sbb)
          print("")
          print("Input Path:    " + str(path))
          print("Result Path:   " + str(best_path))
          print("Input Length:  " + str(evaluate_path(path,sbb)))
          print("Result Length: " + str(evaluate_path(best_path,sbb)))
          print("Iterations:    " + str(iterations))
```

```
Max Iterations: 80
Max Attempts: 8
Remaining Attempts: 8

INFO: sp_copy: 806.4269293088845 < result: 806.4269293088845
INFO: new result -> score: 806.4269293088845, path: ['Bern', 'Liestal', 'Sargans', 'Lugano', 'Lo
INFO: sp_copy: 712.5938229521732 < result: 712.5938229521732
INFO: new result -> score: 712.5938229521732, path: ['Bern', 'Liestal', 'Lugano', 'Locarno', 'Lu
INFO: sp_copy: 707.554267062716 < result: 707.554267062716
INFO: new result -> score: 707.554267062716, path: ['Bern', 'Liestal', 'Lugano', 'Locarno', 'Luz
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
INFO: sp_copy: 605.3501579136067 < result: 605.3501579136067
INFO: new result -> score: 605.3501579136067, path: ['Bern', 'Liestal', 'Luzern', 'Locarno', 'Sa
INFO: sp_copy: 605.1713138968423 < result: 605.1713138968423
```

```
INFO: new result -> score: 605.1713138968423, path: ['Bern', 'Liestal', 'Luzern', 'Locarno', 'Sc
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
INFO: sp_copy: 597.2241286025006 < result: 597.2241286025006
INFO: new result -> score: 597.2241286025006, path: ['Bern', 'Liestal', 'Luzern', 'Locarno', 'Lu
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
INFO: sp_copy: 525.0462844915054 < result: 525.0462844915054
INFO: new result -> score: 525.0462844915054, path: ['Bern', 'Liestal', 'Luzern', 'Sargans', 'Sc
INFO: sp_copy: 518.1667903253285 < result: 518.1667903253285
INFO: new result -> score: 518.1667903253285, path: ['Bern', 'Liestal', 'Luzern', 'Schaan-Vaduz'
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
```

```
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!
ERROR: 8 times no better result! Randomize!

Input Path:    ['Bern', 'Sargans', 'Liestal', 'Lugano', 'Locarno', 'Luzern', 'Schaan-Vaduz', 'Ze
Result Path:   ['Bern', 'Liestal', 'Luzern', 'Schaan-Vaduz', 'Sargans', 'Locarno', 'Lugano', 'Ze
Input Length:  1022.4946103114211
Result Length: 518.1667903253285
Iterations:    80
```



Oh, what happend here? Is this the best we can get? * (Answer Q1) With a simple implementation we do not find the optimal Solution. Therefore a randomization was implemented when we end up in a local minimum. (Start and end position are not randomized). It is far better with this optimization but it is still not a optimal solution.

- Why is this so?
    - (Answer Q2) We end up in a local minimum

- How many steps did we need to get to this solution?
  - When Start and End position should stay then range(0, len(path)-2) -> max swap steps to try (with initial path)
- Can you suggest a method to improve the hill climbing algorithm?
  - (Answer Q3) Randomization (see code example)

1.2.2 Genetic Algorithm

Genetic algorithms (or GA) are inspired by natural evolution and are particularly useful in optimization and search problems with large state spaces.

Given a problem, algorithms in the domain make use of a *population* of solutions (also called *states*), where each solution/state represents a feasible solution. At each iteration (often called *generation*), the population gets updated using methods inspired by biology and evolution, like *crossover*, *mutation* and *natural selection*.

A genetic algorithm works in the following way:

1) Initialize random population.
2) Calculate population fitness.
3) Select individuals for mating.
4) Mate selected individuals to produce new population.
   - Random chance to mutate individuals.
5) Repeat from step 2) until an individual is fit enough or the maximum number of iterations was reached.

Below, you can find some helper functions to implement your genetic algorithm.

First, create a dictionnary that maps a letter to a city name.

Our solution will be a path through all the cities. To simplify, we will encode each city with a letter from the alphabet. So your first initial path through the cities will have the code "ABCDEFGHIJK..". We can easily convert a letter to a city by `letter2city('A')` or `city2letter('Rotkreuz')`.

```
In [208]: import string

          number_of_cities = len(path)
          letter2city = dict()
          city2letter = dict()
          for i in range(number_of_cities):
              letter2city[string.ascii_uppercase[i]] = list(path)[i]
              city2letter[list(path)[i]] = string.ascii_uppercase[i]

          def path2string(path):
              s = ""
              for city in path:
                  s+=city2letter[city]
              return list(s)

          def path2cities(path):
              s = list()
              for letter in path:
                  s.append(letter2city[letter])
              return s

          path_code = path2string(path)
          print("the path has the following code : ")
          print(path_code)
```

```
the path has the following code :
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
```

Let's inizialize a random population:

```
In [209]: import random

          def init_population(pop_number, cities):
              """Initializes population for genetic algorithm
              pop_number  :  Number of individuals in population
              cities      :  cities in letter code """
```

We can calculate the fitness of a path using the evaluate_path function. Note that shorter paths are considered fitter.

```
In [210]: def fitness(sample):
```

```
          File "<ipython-input-210-3263e2f4dfc5>", line 1
        def fitness(sample):
                            ^
    SyntaxError: unexpected EOF while parsing
```

Create a function to select two individuals for mating. Fitter individuals are more likely to be selected for reproduction than less fit individuals. Therefore, we have to calculate the weights of each indiviudal that corresponds to the likelyhood of being chosen for reproduction.

```
In [211]: import random
          from random import choices


          def calculate_weights(population):
              # calculate the weight of each individual
```

```
    return


def select(population, weights):
    # return two individuals for reproduction
    # fitter individuals should be more likely to be selected
    # hint: use random.choices to chose from the population based on the weights of ea
    return

population = init_population(10, path_code)
weights = calculate_weights(population)
print(select(population, weights))
```

```
None
```

Now that we can select two individuals, we make them reproduce using crossover and mutation. We need to consider that we want to visit every city exactly once. For example, for the crossover, you can take a random lenght of individual 1 and fill up the remaining cities based on the order of the unvisited cities in individual 2.

```
In [212]: def crossover(x, y):
              # create an offspring from the parents x and y
              return

          def mutate(x, p_mutate):
              # switch the location of two cities
              return

          # test your code
          x = path_code
          y = random.sample(path_code, len(path_code))
          xy = crossover(x,y)
          print(x)
          print(y)
          print(xy)
          mutate(xy, 0.5)
          print(xy)
```

```
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
['A', 'G', 'B', 'D', 'C', 'E', 'H', 'F']
None
None
```

We have now all the ingredients to create our genetic algorithm:

1.2.3 Simulated Annealing

The intuition behind Hill Climbing was developed from the metaphor of climbing up the graph of a function to find its peak. There is a fundamental problem in the implementation of the algorithm however. To find the highest hill, we take one step at a time, always uphill, hoping to find the highest point, but if we are unlucky to start from the shoulder of the second-highest hill, there is no way we can find the highest one. The algorithm will always converge to the local optimum. Hill Climbing is also bad at dealing with functions that flatline in certain regions. If all neighboring states have the same value, we cannot find the global optimum using this algorithm. Let's now look at an algorithm that can deal with these situations. Simulated Annealing is quite similar to Hill Climbing, but instead of picking the *best* move every iteration, it picks a *random* move. If this random move brings us closer to the global optimum, it will be accepted, but if it doesn't, the algorithm may accept or reject the move based on a probability dictated by the *temperature*. When the *temperature* is high, the algorithm is more likely to accept a random move even if it is bad. At low temperatures, only good moves are accepted, with the occasional exception. This allows exploration of the state space and prevents the algorithm from getting stuck at the local optimum.

The temperature is gradually decreased over the course of the iteration. This is done by a scheduling routine:

```
In [213]: def exp_schedule(k=20, lam=0.005, limit=100):
              """One possible schedule function for simulated annealing"""
              return lambda t: (k * math.exp(-lam * t) if t < limit else 0)
```