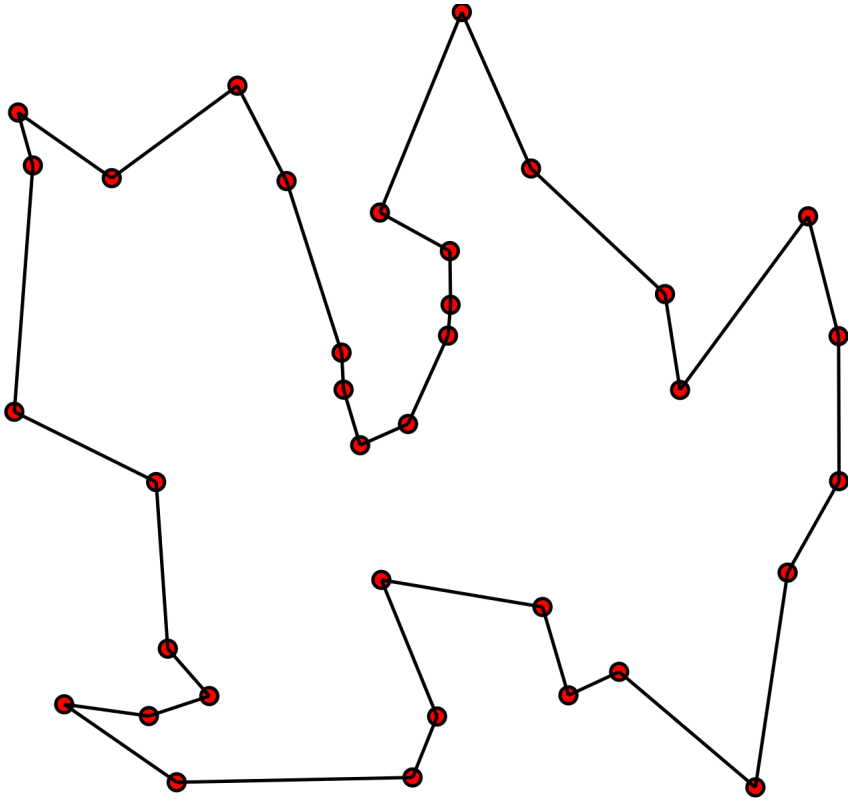
Problem komiwojażera

Karol Janik, Michał Kazanecki

January 2021

1 Wprowadzenie

Aby zrozumieć, na czym polega problem komiwojażera i dlaczego jest tak problematyczny, omówimy pokrótce klasyczny przykład problemu.



Źródło: Wikipedia

Wyobraź sobie, że jesteś sprzedawcą i otrzymałeś mapę podobną do tej wyżej. Widzisz, że mapa zawiera 36 lokalizacji, a Twoim zadaniem jest odwiedzić każdą z nich, aby dokonać sprzedaży.

Zanim rozpoczniesz podróż, prawdopodobnie będziesz chciał zaplanować trasę, aby zminimalizować czas podróży. Na szczęście ludzie są w tym całkiem dobrzy, możemy z łatwością wytyczyć w miarę dobrą trasę bez konieczności robienia czegoś więcej poza zerknięciem na mapę. Kiedy znaleźliśmy trasę, która naszym zdaniem jest najlepsza, jak mżemy sprawdzić, czy trasa, którą wybraliśmy jest naprawdę optymalna?

Krótko mówiąc, nie możemy - przynajmniej nie praktycznie.

Aby zrozumieć, dlaczego tak trudno jest udowodnić optymalną trasę, rozważmy podobną mapę zaweirająca 3 lokalizacje, zamiast oryginalnej - zawierającej 36 lokalizacji. Aby znaleźć trasę, najpierw musimy wybrać lokaliza-

cję startową z trzech możliwych miejscowości na mapie. Następnie do wyboru mamy dwa miasta dla drugiej lokalizacji, a na koniec zostało jedno miasto, aby ukończyć trasę. Oznaczałoby to, że do wyboru jest w sumie $3 \times 2 \times 1$ różnych tras.

Oznacza to, że w tym przykładzie do wyboru jest tylko 6 różnych tras. Tak więc, w przypadku, gdy mamy tylko 3 lokalizację obliczenie każdej z 6 tras i znalezienie najkrótszej ścieżki jest trywialne. Bardziej spostrzegawcze osoby mogły zdać sobię sprawe na czym polega problem już na pierwszy rzut oka. Liczba możliwych tras jest silnią od liczby miejsc do odwiedzenia, a problem z silniami polega na tym, że rosną one niezwykle szybko!

Wracając do naszego pierwotnego problemu, jeżli chcemy znaleźć najkrótszą trasę dla naszej mapy 20 lokalizacji, musielibyśmy oszacowac **371993326789901217467999448150835200000000** różnych tras! Nawet przy nowoczesnej mocy obliczeniowej jest to strasznie niepraktycze, a w przypadku jeszcze większych problemów jest prawie niemożliwe.

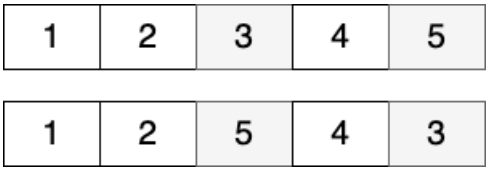
2 Szukanie Rozwiązania

Znalezienie rozwiązania problemu komiwojażera wymaga wyspecjalizowanego algorytmu genetycznego. Na przykład prawidłowe rozwiązanie musiałoby reprezentować trasę, w której każda lokalizacja jest uwzględniona tylko raz. Jeśli trasa zawiera jedną lokalizację więcej niż raz lub całkowicie pominęła lokalizację, nie byłoby to poprawne rozwiązanie.

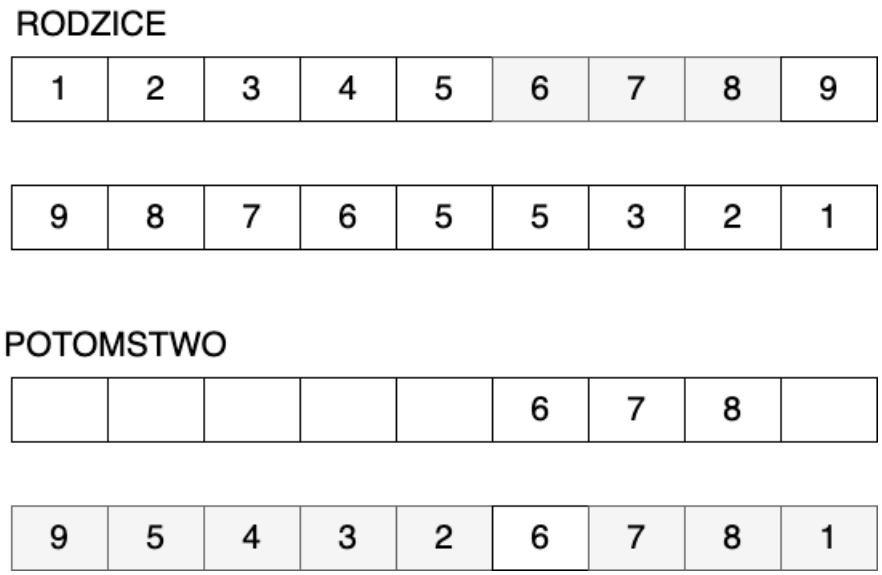
Aby algorytm genetyczny rzeczywiście spełniał ten wymóg, potrzebne są specjalne typy mutacji i metod krzyżowania.

Metoda mutacji powinna umożliwiać tylko tasowanie trasy, nie powinna nigdy dodawać ani usuwać lokalizacji z trasy, w przeciwnym razie groziłoby to stworzeniem nieprawidłowego rozwiązania. Jednym z typów metod mutacji, których możemy użyć jest mutacja zmiany.

W przypadku mutacji zmiany, dwie lokalizacje na trasię są wybierane losowo, a następnie ich pozycję są po prostu zamieniane. Na przykład, jeśli zastosujemy mutację swap do listy [1,2,3,4,5] możemy otrzymać [1,2,5,4,3]. Tutaj pozycje 3 i 5 zostały zamienione, tworząc nową listę z dokładnie tymi samymi wartościami, tylko w innej kolejności. Ponieważ mutacja zmiany polega tylko na zamianie wcześniej istniejących wartości, nigdy nie utworzymy listy, która ma brakujące lub zduplikowane wartości w porównaniu z oryginałem. Tego właśnie chcemy w przypadku problemu komiwojażera.



Jedną z metod skrzyżowania, która jest w stanie wygenerować prawidłową trasę, jest uporządkowane skrzyżowanie. W tej metodzie wybieramy podzbiór z pierwszego rodzica, a nastepnie dodajemy ten podzbiór do potomstwa. Wszelkie brakujące wartości są następnie dodawane do potomstwa od drugiego rodzica w celu ich znalezienia. Aby lepiej to zrozumieć spójrzmy na przykład.



Podzbiór trasy jest pobierany od pierwszego rodzica (6,7,8) i dodawany do trasy potomstwa. Następnie brakujące lokalizację tras są dodawane w kolejności od drugiego rodzica. Pierwsza lokalizacja na trasie drugiego rodzica to 9, która nie znajduje się na trasie potomstwa, więc jest dodawana na pierwszej dostępnej pozycji. Następna pozycja na trasie rodziców to 8, znajduje się na trasie potomstwa, więc jest pomijana. Proces ten trwa do momentu, gdy potomstwo nie będzie miało pustych wartości. Jeśli zaimplementowano poprawnie, wynikiem końcowym powinnabyć trasa zawierająca wszystkie pozycje, które wykonali rodzice, bez brakujących lub zduplikowanych pozycji.

3 Implementacja algorytmu

Najpierw zdefiniujemy klase, która będzie reprezentowała populację. Konstruktor klasy tworzy następujące pola **bag** do reprezentowania całej populacji, **parents** do reprezentacji kilku wybranych, **score** do przechowywania wyniku najlepszego chromosomu w populacji, **best** do przechowywania najlepszego chromosomu oraz **adjacency_mat** macierz sąsiedztwa, której będziemy używać do obliczenia odległości w kontekście TSP.

```
class Population:
  def __init__(self, bag, adjacenty_mat):
    self.bag = bag
    self.parents = []
    self.score = 0
    self.best = None
    self.adjacency_mat = adjacenty_mat
```

Również stworzymy funkcję **init_population** do losowego generowania pierwszej populacji.

```
def init_population(cities, adjacency_mat, n_population):
  return Population(
    np.asarray([np.random.permutation(cities) for _ in range(
                                    n_population)]),
    adjacency_mat,
  )
```

Teraz potrzebujemy metody, która określi sprawność chromosomu. W kontekście problemu komiwojażera sprawność definiuje się w bardzo prosty sposób: im krótszy dystans całkowity, tym bardziej sprawny i lepszy jest chromosom. Przypomnij sobie, że wszystkie potrzebne informacje o odległości są przechowywanie w macierzy sąsiedztwa. Możemy obliczyć sumę wszystkich odległości między dwoma sąsiednimi miastami w sekwencji chromosomów.

```
  def fitness(self, chromosome):
    return sum(
      [
        self.adjacency_mat[chromosome[i], chromosome[i+1]]
        for i in range(len(chromosome) - 1)
      ]
    )
```

Następnie oceniamy populację. Mówiąc najprościej, ocena sprowadza się do obliczenia przystosowania każdego chromosomu w całej populacji, określenia, kto jest najlepszy, przechowania informacji o wynikach i zwrócenia wektora prawdopodobieństwa, którego każdy element reprezentuje prawdopodobieństwo, że i-ty element w zbiorze populacji zostaje wybrany jako rodzic. Stosujemy podstawowe przetwarzanie wstępne, aby zapewnić, że najgorszy chromosom nie ma aboslutnie żadnej szansy na wybranie

```
def evaluate(self):
    distances = np.asarray(
      [self.fitness(chromosome) for chromosome in self.bag]
    )
    self.score = np.min(distances)
    self.best = self.bag[distances.tolist().index(self.score)]
    self.parents.append(self.best)
    if False in (distances[0] == distances):
      distances = np.max(distances) - distances
    return distances / np.sum(distances)
```

Teraz stworzymy metodę **select**, której zadaniem będzie wybranie **k** liczby rodziców, którzy będą podstawą dla następnego pokolenia. Używamy prostego modelu ruletki, w którym porównujemy wartość wektora prawdopodobieństwa i losową liczbę pobraną z rozkładu jednorodnego. Jeśli wartość wektora prawdopodobieństwa jest wyższa, odpowiedni chromosom jest dodawany do własnych rodziców. Powtarzamy ten proces, aż bedziemy mieć **k** rodziców.

```
def select(self, k=4):
    fit = self.evaluate()
    while len(self.parents) < k:
      idx = np.random.randint(0, len(fit))
      if fit[idx] > np.random.rand():
        self.parents.append(self.bag[idx])
    self.parents = np.asarray(self.parents)
```

Teraz najważniejsza część, implementacja mutacji. Mutacje opisaliśmy w poprzednim rozdziale.

```
    def swap(chromosome):
  a, b = np.random.choice(len(chromosome), 2)
  chromosome[a], chromosome[b] = (
    chromosome[b],
    chromosome[a],
  )
  return chromosome
```

```
def crossover(self, p_cross=0.1):
        children = []
        count, size = self.parents.shape
        for _ in range(len(self.bag)):
            if np.random.rand() > p_cross:
                children.append(list(self.parents[np.random.randint
                                                  (count, size=1)[0
                                                  ]]))
            else:
                parent1, parent2 = self.parents[np.random.randint(
                                                  count, size=2), :
                                                  ]
                idx = np.random.choice(range(size), size=2, replace
                                                  =False)
                start, end = min(idx), max(idx)
                child = [None] * size
                for i in range(start, end + 1, 1):
                    child[i] = parent1[i]
                pointer = 0
                for i in range(size):
                    if child[i] is None:
                        while parent2[pointer] in child:
                            pointer += 1
                        child[i] = parent2[pointer]
                children.append(child)
        return children
```

```
def mutate(self, p_cross=0.1, p_mut=0.1):
        next_bag = []
        children = self.crossover(p_cross)
        for child in children:
            if np.random.rand() < p_mut:
                next_bag.append(swap(child))
            else:
                next_bag.append(child)
        return next_bag
```

3.1 Dodatkowe metody

W tym podrozdziale zajmiemy się dodatkowymi metodami, które pozwolą generować współrzędne miasta i odpowiadające im macierze sąsiedztwa.

```
def generate_cities(n_cities, factor=0.2):
  x = np.asarray(range(int(-n_cities / 2), int(n_cities / 2) + 1, 1
                                  ))
  y = np.sqrt(n_cities ** 2 / 4 - x ** 2)
  return np.asarray(list(zip(x,y)))
```

generate_cities generuje n losowych współrzędnych miast w postaci tablicy numpy. Teraz potrzebna metoda, która utworzy macierz sąsiedztwa na podstawie współrzędnych miasta.

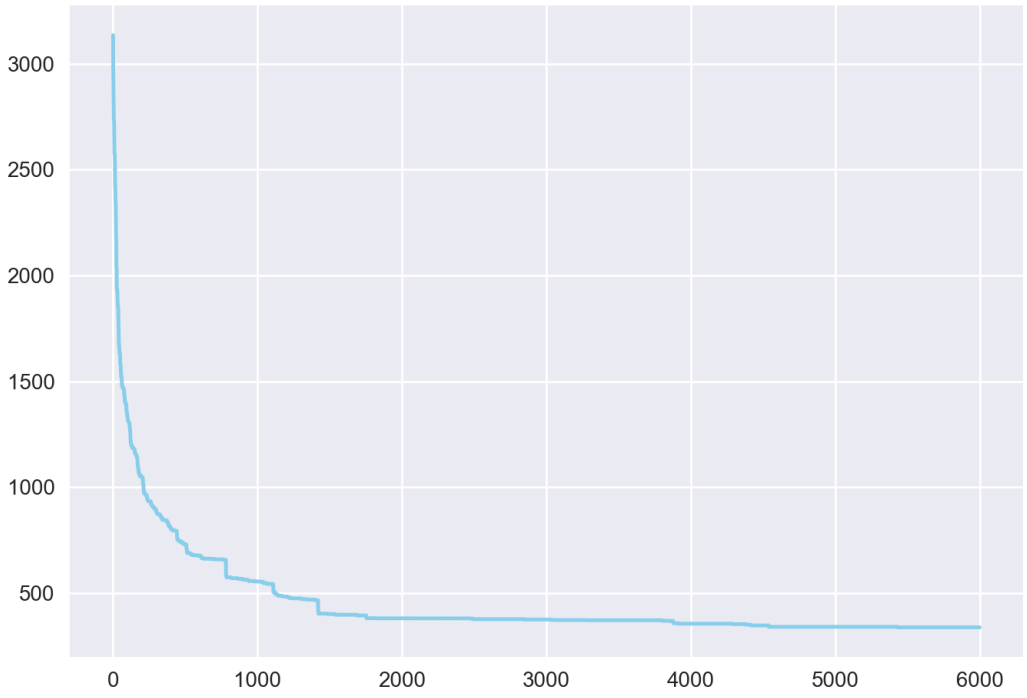
```
def make_mat(coordinates):
    res = [
        [get_distance(city1, city2) for city2 in coordinates] for
                                            city1 in coordinates
    ]
    return np.asarray(res)


def get_distance(city1, city2):
    return np.sqrt((city1[0] - city2[0]) ** 2 + (city1[1] - city2[1
                                        ]) ** 2)
```

Funkcja pomocnicza do rysowania ścieżki wraz z odpowiednimi współrzędnymi miasta.

```
def print_path(best, city_coordinates):
    points = city_coordinates[best]
    x, y = zip(*points)
    plt.plot(x, y, color='skyblue', marker='o')
    plt.show()
```

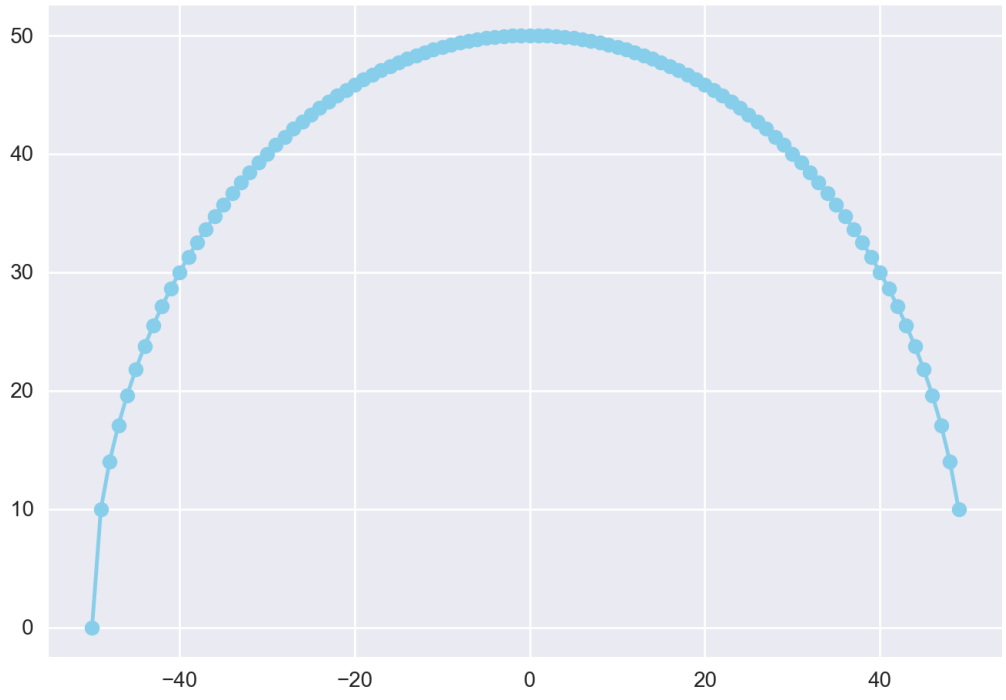
Stworzymy teraz 100 fałszywych miast i uruchomimy algorytm genetyczny, aby zoptymalizować scieżkę. Jeśli algorytm z powodzeniem znajdzie optymalną scieżkę, będzie to pojedyńcza krzywa od jednego końca półkola w pełni połączonego do drugiego końca.

```
cities = range(100)
    city_coordinates = better_generate_cities(len(cities))
    adjacency_mat = make_mat(city_coordinates)
    best, history = genetic_algorithm(
        cities,
        adjacency_mat,
        n_population=500,
        selectivity=0.05,
        p_mut=0.05,
        p_cross=0.7,
        n_iter=6000,
        print_interval=500,
        verbose=False,
        return_history=True,
    )
    plt.show()
    print(best)
```


3000
2500
2000
1500
1000
500
0
1000
2000
3000
4000
5000
6000

```
Generation 0: 3136.0756854145816
Generation 500: 729.1945091938047
Generation 1000: 553.9423036123877
Generation 1500: 400.0095906722208
Generation 2000: 379.43806161584513
Generation 2500: 375.9221319772076
Generation 3000: 373.96566697095267
Generation 3500: 370.69902591673235
Generation 4000: 354.35352986043705
Generation 4500: 346.13028286649615
Generation 5000: 339.70177451397956
Generation 5500: 337.2379099506411


[5, 6, 7, 8, 9, 13, 14, 15, 16, 17, 18, 22, 23, 30, 31, 32, 33, 34,
                                    35, 36, 37, 38, 39, 40, 41, 42,
                                   43, 44, 45, 46, 47, 48, 88, 87,
                                   86, 85, 84, 83, 82, 81, 80, 79,
                                   78, 77, 76, 75, 74, 73, 72, 71,
                                   70, 69, 68, 67, 66, 65, 64, 63,
                                   62, 61, 60, 59, 58, 57, 56, 55,
                                   54, 53, 52, 51, 50, 49, 29, 28,
                                   27, 26, 25, 24, 21, 20, 19, 12,
                                   11, 10, 4, 3, 2, 1, 0, 89, 90, 91
                                   , 92, 93, 94, 95, 96, 97, 98, 99]
```

Algorytm wydaje się być zbieżny, ale najlepsza zwrócona ścieżka nie wydaje się być scieżką optymalną, ponieważ nie jest to posortowana tablica od 0 do 99, jak sie spodziewamy. Warto zauważyć, że algorytm znalazł coś, co można nazwac segmentami optymalnymi: zawuażamy, że istnieją odcinki scieżek, zawierające kolejne liczby, czego oczekiwalibysmy na optymalnej ściezce. Optymalną scieżką bedzie posortowanie tej tablicy i wykreślenie grafu:


50
40
30
20
10
0
−40
−20
0
20
40

Literatura

[1] *https://jaketae.github.io/study/genetic-algorithm/*

[2] *https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35*

[3] *https://www.youtube.com/watch?v=hnxn6DtLYcYab_channel=TheCodingTrain*