Uniwersytet Gdański
Wydział Matematyki, Fizyki i Informatyki

Informatyka Ogólnoakademicka II stopień

Mateusz Motyl

Nr albumu: 269381
Specjalność: Ogólna
Rodzaj studiów: Stacjonarne

Rozwiązywanie problemu komiwojażera

Gdańsk, 2023

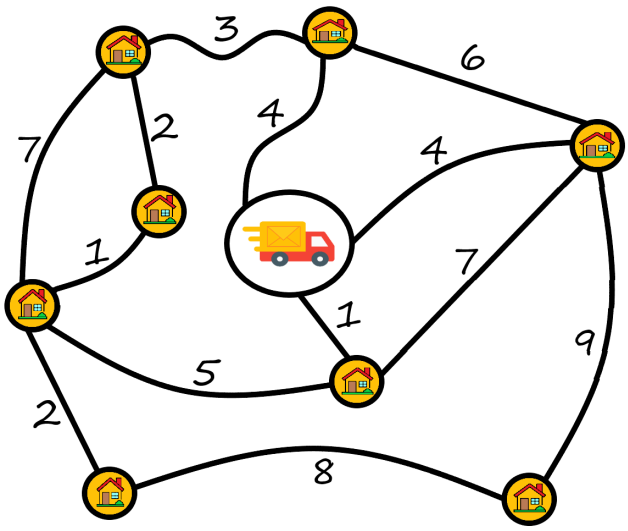
Streszczenie

Zakres prac obejmuje tematykę rozwiązywania problemu komiwojażera w podejściu analitycznym - przy użyciu współrzędnych punktów na płaszczyźnie dwuwymiarowej, zamiast grafu ważonego. Projekt zawiera implementacje problemu za pomocą algorytmu genetycznego, mrówkowego (ang. ant colony optimization) oraz przy użyciu podejścia metaheurystycznego (symulacja wyżarzania), a także porównania czasów oraz dokładności obliczeń programów.

Spis treści

1 Problem komiwojażera

Problem komiwojażera (ang. travelling salesman problem, TSP) to problem znalezienia cyklu w grafie ważonym (ścieżki zaczynającej i kończącej się w tym samym wierzchołku), który będzie zawierał wszystkie wierzchołki, a jego suma wag krawędzi będzie najmniejsza.
Problem jest powszechnie spotykany w życiu codziennym, a jednym z jego przykładów jest praca kuriera, który wyjeżdza z magazynu i musi rozwieźć przesyłki do domów odbiorców, a ostatecznie wrócić do oddziału. Trasa musi być zoptymalizowana, aby stracić możliwie jak najmniej czasu/ paliwa. Waga krawędzi pomiędzy wierzchołkami odpowiada ilości czasu/ paliwa potrzebnej do przedostania sie z jednego do drugiego punktu.



Rysunek 1: Przykładowy problem TSP - graf ważony [Opracowanie własne].

Zadanie przedstawiane jest również w formie bardziej analitycznej - zamiast grafu ważonego dostajemy zbiór punktów na płaszczyźnie, a odległości (euklidesowe) między nimi odpowiadają wagom w grafie. W ten sposób na wejściu podajemy same węzły (współrzędne) bez krawędzi między nimi. Oznacza to, że z każdego punktu możemy dostać się do dowolnego innego. Podejście to zostało przyjęte w opisywanym projekcie.

2 Przygotowanie danych

Dane wykorzystane do przedstawianych programów to zbiór punktów (x, y) z przedziałów:
$x \in [49.3,\ 54.8]$, $y \in [14.2,\ 23.9]$. Są to przybliżone współrzędne geograficzne Polski (szerokość i długość geograficzna). Poniżej przedstawiono funkcję służącą generowaniu danych.

```
from os import system
import numpy as np

def generateCoordinates(n):
    coordinates = [[x, y] for x, y in zip(np.random.uniform(49.3, 54.8, n),
                                          np.random.uniform(14.2, 23.9, n))]
    with open("\INF-D-2023-Mateusz-Motyl-269381\\utils\coords.txt", "w") as f:
        for item in coordinates:
            f.write(str(item) + "\n")
    f.close()
```

Jak można zauważyć pojawił się zapis punktów do pliku tekstowego - pozwala to na korzystanie z tego samego zbioru danych przez wszystkie algorytmy i umożliwia porównanie dokładności oraz ich czasu wykonywania. Ponadto przy ponownym uruchomieniu programu stan pliku nie zmieni się (bez uprzedniego wygenerowania nowych punktów).

3 Algorytmy

Problem komiwojażera jako problem szukania ścieżek warto rozwiązać na kilka sposobów i porównać czasy oraz dokładność obranych podejść (przez dokładność mamy na myśli oczywiście najkrótszą ścieżkę opiewającą we wszystkie węzły).

3.1 Algorytm genetyczny

[1] Algorytm genetyczny (ang. genetic algorithm, AG) to strategia przeszukiwania przestrzeni alternatywnych rozwiązań w celu znalezienia najlepszych rozwiązań. Podejście definiuje środowisko, w którym istnieje pewna populacja osobników, a każdy z nich posiada pewien zbiór informacji stanowiących jego chromosom - zestaw genów podlagający ocenie funkcji przystosowania modelującej środowisko.
Innymi słowy rozwiązanie jest organizmem z zestawem genów, które selekcjonowane są według najlepszego przystosowania do warunków środowiska. Do każdej następnej generacji przedostają się lepsze jednostki z generacji poprzedniej, wymieniając przy tym informacje w nich zawarte, próbując utworzyć przy tym nowe, potencjalnie optymalniejsze rozwiązanie. Warunkiem zakończenia generowania kolejnych pokoleń może być osiągnięta maksymalna liczba iteracji lub znalezienie najlepszego rozwiązania. Poniżej przedstawiono typowy przebieg algorytmu genetycznego.

Rysunek 2: Uproszczony przebieg algorytmu genetycznego [Opracowanie własne].

Do implementacji algorytmu wykorzystana została paczka [2] *pygad* zawierająca zaimplementowany algorytm genetyczny. Trudnością w tym zadaniu jest jego konfiguracja.
Po uwczesnym wczytaniu współrzędnych (x, y) z pliku, przystępujemy do zakodowania chromosomów.

```
# list(0, 1, ..., len(x)), list of all possible values of the gene
gene_space = list(range(0, len(x)))

# number of chromosome genes, number of parameters in the function
num_genes = len(x)
```

- *gene_space* - lista możliwych wartości genów używana w kroku mutacji. Ustawiona jako lista kolejnych numerów $[0, n]$, gdzie n to liczba węzłów do odwiedzenia.
- *num_genes* - liczba genów chromosomu, liczba parametrów w funkcji. Ustawiona jako ilość węzłów do odwiedzenia.

Następnym krokiem jest utworzenie populacji, gdzie do ustawienia mamy parametry wielkości populacji oraz liczby generacji. Im więcej osobników w rodzie, tym większa szansa na znalezienie rozwiązania, jednak zwiększenie tego parametru skutkuje spowolnieniem działania algorytmu.

```
# number of chromosomes/ solutions, number of solutions in the population
if num_genes > 10:
    sol_per_pop = 50
elif num_genes > 20:
    sol_per_pop = 80
elif num_genes > 30:
    sol_per_pop = 100
elif num_genes > 40:
    sol_per_pop = 150
else:
    sol_per_pop = 200

#number of generations
num_generations = 200
```

- *sol_per_pop* - liczba chromosomów/ rozwiązań w populacji. Parametr uzależniony jest od długości chromosomu:
- $len(chromosome) \in [1, 10) \Rightarrow 50$ osobników, $len(chromosome) \in [10, 20) \Rightarrow 80$ osobników, itd.
- *num_generations* - liczba generacji, liczba iteracji algorytmu

Po ustawieniu powyższych parametrów czas przejść do wyboru typu selekcji kolejnych generacji oraz właściwości ich poprzedników (rodziców). Najlepsze osobniki z generacji powinny przekazywać swoje geny następnym pokoleniom.

```
# selection type => steady state selection
parent_selection_type = "sss"

# number of parents to cross, number of solutions to be selected as parents in the
 mating pool
num_parents_mating = int(sol_per_pop/2) + 1

# if 0 => no parent in the current population will be used in the next population
# if -1 => all parents in the current population will be used in the next
population
# if > 0 => the specified value refers to the number of parents in the current
population
# to be used in the next population
keep_parents = int(sol_per_pop * 0.01)
```

- *parent_selection_type* - [3] typ selekcji rodziców kolejnego pokolenia. "sss" - (ang. steady state selection) selekcja stabilnego stanu, wybiera pewien procent najlepszych rodziców z których produkuje potomków i zastępuje najgorsze chromosomy potomkami. Pozostałe typy to: "rws" (- ruletka, szanse wylosowania względem wartości dopasowania, w najgorszym przypadku może wyeliminować najlepiej przystosowanych), "sus" (- podobnie jak w "rws" ale zamiast ciągłego kręcenia kołem, aż zostaną wybrane wszystkie osobniki (signle selection point), wybiera wszystkich rodziców na raz), "rank" (- przydatny, gdy populacja ma bardzo zbliżone wartości dopasowania. Opiera prawdopodobieństwo wylosowania nie na wartości dopasowania, ale na wartości samego osobnika), "random" (- rodzice wybierani losowo, bez względu na ich kondycję), "tournament" (- wygrywa osobnik z najlepszą wartością przystosowania. Im większy rozmiar turnieju, tym większą szanse na przegraną będą miały słabsze jednostki, walcząc tym samym z większą ilością przeciwników).
- *num_parents_mating* - ilość rodziców do krzyżowania, wartość to około 50% populacji
- *keep_parents* - ilość rodziców do zachowania (kilka procent), dodatkowo przyjmuje wartości:
  0 - żaden rodzic z bieżącej populacji nie zostanie wykorzystany w następnej populacji
  -1 - wszyscy rodzice z bieżącej populacji zostaną wykorzystani w następnej populacji

Nasuwa się pytanie: jakim sposobem oceniane są osobniki? Odpowiada za to funkcja przystosowania (ang. fitness function), która określa jak blisko jest dane rozwiązanie do optymalnego rozwiązania problemu. Algorytmy genetyczne przedstawiają zwkle rozwiązanie za pomocą ciągu liczb (chromosomu). Rezultaty muszą zostać przetestowane i ocenione względem tego jak blisko były spełnienia ogólnej specyfikacji pożądanego rozwiązania. Wyniki te generowane są przez zastosowanie funkcji dopasowania. Im wyższa wartość zwrócona przez funkcję, tym rozwiązanie jest lepsze.

```
    def fitness_func(solution, solution_idx):
        return countDistances(x, y, solution)
```

Parametrami funkcji $fitness\_func$ są:

- *solution* - lista numerów wierzchołków, kolejność w której będziemy odwiedzać wierzchołki
- *solution_idx* - numer iteracji w danym pokoleniu

Powyższa metoda wywołuje kolejną funkcję $countDistances(x, y, order)$, która oblicza odległość między dwoma kolejnymi punktami i sumuje wyniki. Jest to tak zwana odległość euklidesowa określona wzorem: $d = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$

```
def countDistances(x, y, order):
    dist = 0.0

    order = numpy.append(order, order[0])
    for item in zip(order, order[1:]): #list(1,2,3,4) => (1,2)(2,3)(3,4) + (4,1)
       dist += ((x[int(item[1:][0])] - x[int(item[:-1][0])])**2 + (y[int(item[1:][0])]
        - y[int(item[:-1][0])])**2)**0.5

    return -dist
```

- x - lista współrzędnych x (szerokości geograficznych) wierzchołków
- y - lista współrzędnych y (długości geograficznych) wierzchołków
- *order* - kolejność odwiedzania wierzchołków

Przed rozpoczęciem obiegu pętli ostatni punkt dodawany jest na początek, żeby utworzyć cykl. Zwracany wynik poprzedzony jest minusem, tak aby najkrótsza trasa dawała najwyższy wynik. Kolejnym krokiem po selekcji najlepiej przystosowanych osobników, według diagramu [2] są mutacja oraz krzyżowanie. [3] Krzyżowanie określa sposób generowania dzieci z wybranych rodziców; innymi słowy, jak działa reprodukcja. Obsługiwane są 4 algorytmy:

- *signel_point* - krzyżowanie jednopunktowe, losowy punkt przecięcia. Wszystkie geny na prawo od tego punktu są następnie wymieniane między dwoma chromosomami rodzicielskimi tworząc dwoje potomstwa,
- *two_points* - krzyżowanie dwupunktowe, podobnie jak w powyższym przypadku, ale zamiast jednego przecięcia mamy dwa,
- $uniform$ - krzyżowanie jednolite, każdy gen wybierany jest od jednego z rodziców z rónym prawdopodobieństwem,
- *scattered* - krzyżowanie rozproszone, generowany jest losowy wektor binarny długości chromosomów. Geny dziecka pobierane są od jednego rodzica, gdy jego indeks w wektorze binarnym wynosi 0 - od drugiego, gdy wynosi 1.

[3] Mutacja określa jakie transformacje jakie transformacje są wykonywane na dzieciach. Pomaga to utrzymać różnorodność genetyczną. Obsługiwanych jest 5 algorytmów:

- *random* - (domyślna) do losowego zbioru dopisywana jest losowa wartość z dopuszczalnych wartośći,
- *swap* - losowo wybierane dwa geny są zamieniane,
- *inversion* - odwrócenie części sekwencji genów,

- *scramble* - losowo tasowane sekwencje genów,
- *adaptive* - dzieci z wysoką sprawnością przechodzą mniej mutacji,
- *None* - nie jest to algorytm, a jedynie wyłączenie mutacji.

```
# crossover type => single point
# if None => step bypassed
crossover_type = "single_point"

# mutation type => random
# if None => step bypassed
mutation_type = "random"

# mutation percent, default=10%
# no action if mutation_probability or mutation_num_genes exist
mutation_percent_genes = int((1 / len(x)) * 100) + 1
```

- *crossover_type* - typ krzyżowania. W tym przypadku jednopunktowy. Gdyby parametr nie został sprecyzowany, krok krzyżowania zostałby pominięty,
- *mutation_type* - typ mutacji. W tym przypadku losowy. Gdyby parametr nie został sprecyzowany, krok mutacji zostałby pominięty,
- *mutation_percent_genes* - procent, na którym będzie działać mutacja. Nie działa z parametrami *mutation_probability*, *mutation_num_genes*.

Po ówczesnym przygotowaniu parametrów, możemy zainicjować algorytm.

```
ga_instance = pygad.GA(
                        gene_space=gene_space,
                        num_generations=num_generations,
                        num_parents_mating=num_parents_mating,
                        fitness_func=fitness_function,
                        sol_per_pop=sol_per_pop,
                        num_genes=num_genes,
                        allow_duplicate_genes=allow_duplicate_genes,
                        parent_selection_type=parent_selection_type,
                        keep_parents=keep_parents,
                        crossover_type=crossover_type,
                        mutation_type=mutation_type,
                        mutation_percent_genes=mutation_percent_genes)
```

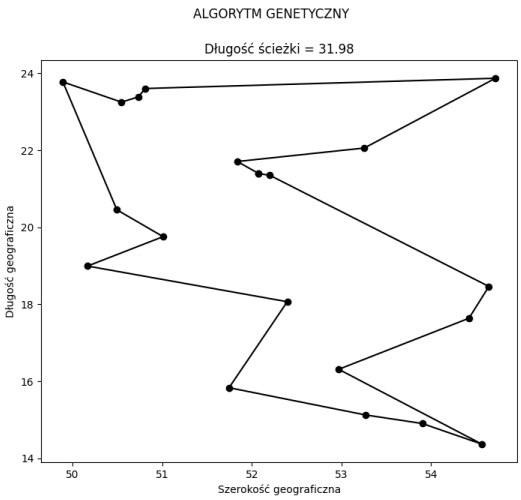
Pojawił się nieopisywany dotąd parametr *allow_duplicate_genes* z wartością $false$. Jako, że problem rozwiązywany jest w formie punktów na płaszczyźnie a nie grafu ważonego z wyznaczonymi już ścieżkami, nie ma potrzeby przechodzenia przez jeden punkt więcej niż jeden raz (ścieżki wyznaczane są z punktu do punktu). Parametr ten odpowiada za unikalność genów w chromosomie.

Algorytm prezentuje następujące wyniki dla 100 generacji:

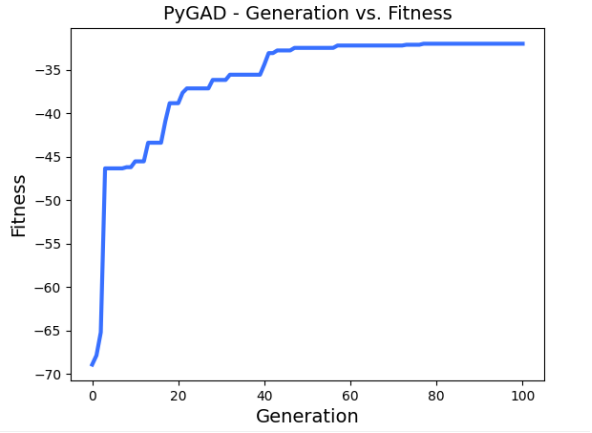
Ilość węzłów	10	20	30	50	100	200
Dystans [° - stopnie]	21.29	31.98	46.98	90.8	328.6	488.17
Czas wykonywania [sek.]	2.4	0.75	0.84	1.42	4.92	6.22

Algorytm dla ilości węzłów mniejszej niż 10 szuka rozwiązania dosyć długo - dla większych wartości sytuacja się normuje; czas wydłuża się proporcjonalnie względem ilości węzłów.

Po zakończeniu pracy dla najlepszego rozwiązania generowane są wykresy.



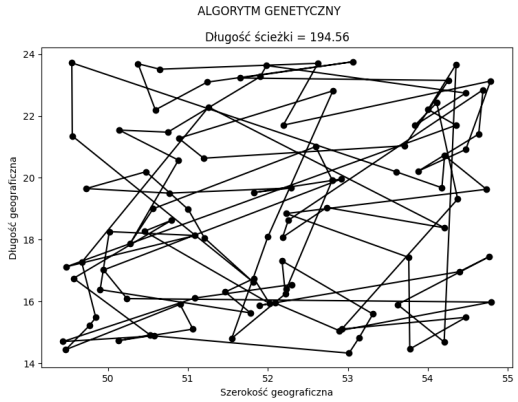
(a) Wizualizacja ścieżki



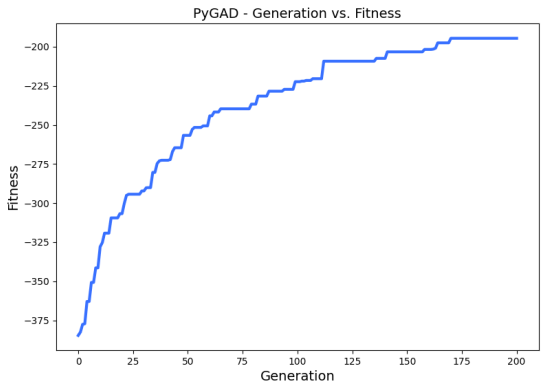
(b) Wykres ocen na przestrzeni pokoleń

Rysunek 3: AG: wykresy zwrócone dla 20 węzłów

Jak można zauważyć wyznaczona ścieżka jest (prawdopodobnie) najoptymalniejszą ścieżką, a wykres ocen na przestrzeni pokoleń nie ma dłuższych odcinków w których wartości są stałe (cały czas udoskonala następne pokolenia). Sprawa komplikuje się przy większej ilości węzłów.



(a) Wizualizacja ścieżki

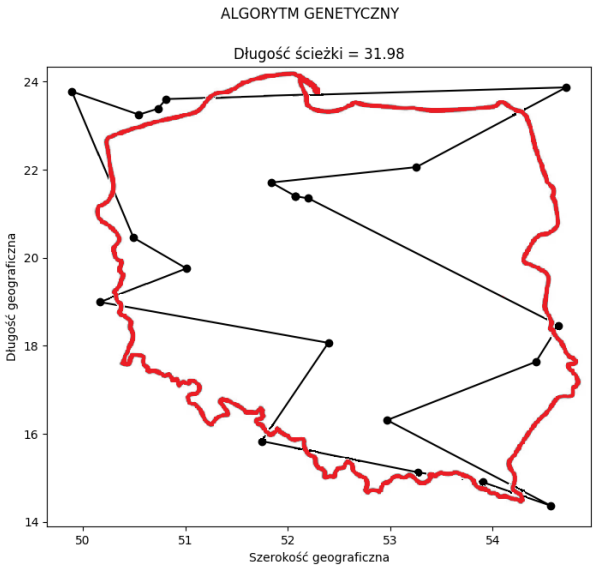


(b) Wykres ocen na przestrzeni pokoleń

Rysunek 4: AG: wykresy zwrócone dla 100 węzłów

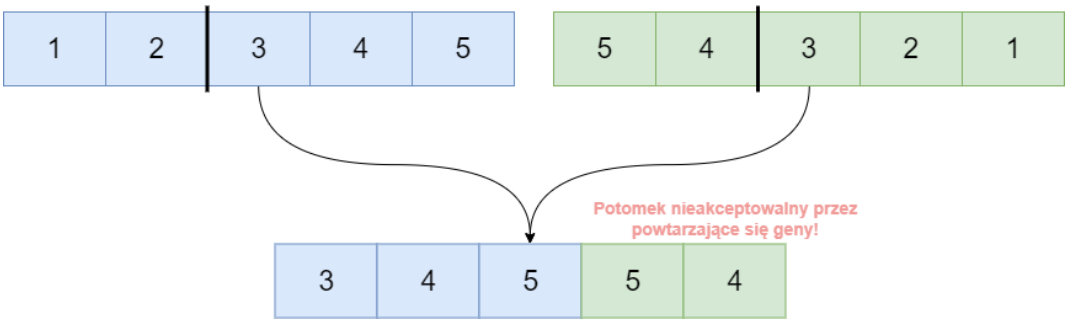
Zwrócona ścieżka z pewnością nie jest najoptymalniejszą. Wykres ocen na przestrzeni pokoleń wskazuje na ciągłe ulepszanie kolejnych pokoleń, jednak mimo znacznego zwiększenia liczby generacji (z 200 do 500) nie zmienił znacząco finalnego porządku odwiedzania wierzchołków. Co warto zauważyć zmiana liczby generacji ze 100 do 200 skróciła ścieżkę o prawie 60% - 328.6 $\rightarrow$ 194.56 (czas trwania algorytmu: 100 generacji - 4.92; 200 generacji - 7.4).

Zwracane przez algorytm długości ścieżki podawane są w stopniach geograficznych (1 $^\circ \approx 111$ km). Współrzędne punktów generowane są z zakresów, jakie obejmują maksymalnie oddalone punkty w Polsce. W podanym przykładzie punkty mniej więcej pokrywają granice, więc dla przetestowania poprawności algorytmu możemy porównać te dystansy.
Długość ścieżki $= 31.98 \cdot 111km = 3549.78km$
Długość granic Polski $= 3572.69km$



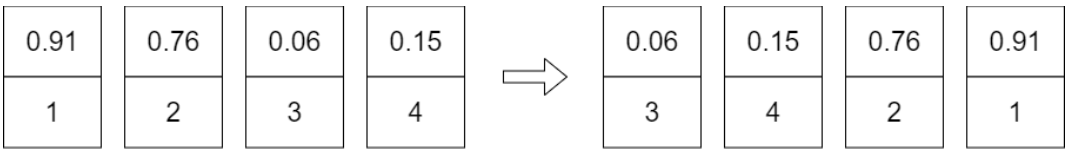
Rysunek 5: Kontur Polski na wykresie ścieżki [Opracowanie własne].

Przy obranym parametrze *gene_space* i wymaganej unikalności wartości genów w chromosomie niewielu osobników przechodzi krzyżowanie oraz mutację.



Rysunek 6: AG: Błąd krzyżowania [Opracowanie własne].

Mocno ograniczony zbiór wartości genów nie pozwalał na większe modyfikacje wśród chromosomów. Podobnie sytuacja ma się przy mutacji. Do tej pory parametr *mutation_type* przyjmował wartość *random*, co znaczy, że w kroku mutacyjnym do losowego zbioru dopisywana była losowa wartość z dopuszczalnych (lepiej sprawdziło by się *inversion* - odwrócenie częściowej sekwencji genów) - krok był pomijany/ gen wymieniany na ten sam.
Rozwiązaniem problemu jest zmiana *gen_space* oraz modyfikacja funkcji dopasowania. Po ustawieniu zakresu dostępnych wartości chromosomu na liczby zmiennoprzecinkowe z zakresu od 0 do 1, prawdopodobieństwo duplikacji - tym samym wykluczenia potomka z dalszej reprodukcji - znacznie maleje oraz znika problem modyfikacji (zbiór do wylosowania jest większy od ilości genów w chromosomie). Otrzymaną w ten sposób listę wartości sortujemy rosnąco, a kolejność sorotowania to kolejność odwiedzania następnych węzłów w poszukiwaniu najkrótszej ścieżki.



Rysunek 7: AG: Sortowanie genów [Opracowanie własne].

Z powyższego przykładu wynika, że kolejność odwiedzania wierzchołków to $[3, 4, 2, 1]$. Czas zaimplementować powyższe funkcje.

```
gene_space = {
  'low': 0,
  'high': 1
}
```

Przy pomocy paremtrów *low*, *hight* możemy zdefiniować przedziały, z któych losowane będą wartości genów.

```
def floatOrder(floats):
    nodes = list(range(0, len(x)))
    order = list(zip(nodes, floats))  # ((0, float), (1, float), ... (n, float))
    sort = sorted(order, key=lambda a: float(a[1]))  # sorted by float values

    return list(zip(*sort))[0] # unzip: (list(order), list(float))[0]
```

Sortowaniu posłuży funkcja $floatOrder()$, której parametrem jest lista wartości zmiennoprzecinkowych przekazywana z funkcji dopasowania ($fitness\_func(\underline{\textbf{solution}}, solution\_idx)$). Metoda tworzy listę krotek (tuples) i sortuje ją według indeksu występowania, a ostatecznie zwraca indeksy jako kolejność odwiedzania węzłów, którą możemy przekazać do *countDistances*.
Po modyfikacji algorytmu wyniki prezentują się następująco:

Ilość węzłów	10	20	30	50	100	200
Dystans [° - stopnie]	21.29	31.23	46.38	75.35	195.07	394.9
Czas wykonywania [sek.]	4.44	1.6	1.92	2.41	4.21	7.05

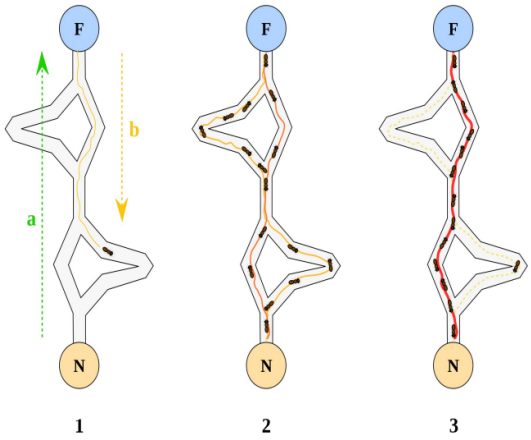
Dla przypomnienia tabela AG z $gene\_space \in [0, 1, ..., len(nodes)]$:

Ilość węzłów	10	20	30	50	100	200
Dystans [° - stopnie]	21.29	31.98	46.98	90.8	328.6	488.17
Czas wykonywania [sek.]	2.4	0.75	0.84	1.42	4.92	6.22

Jak widać wyniki dla większych wartości prezentują się znacznie lepiej, jednak co z tym idzie - zwiększony jest czas wykonywania algorytmu. Wartości *gene_space* w przedziale całkowitoliczbowym działają bardzo dobrze przy niezbyt dużej ilości obliczeń. Co można również zauważyć, niezależnie od parametrów zbioru wartości genów czas wykonywania dla małych ilości węzłów jest nieproporcjonalnie wysoki.

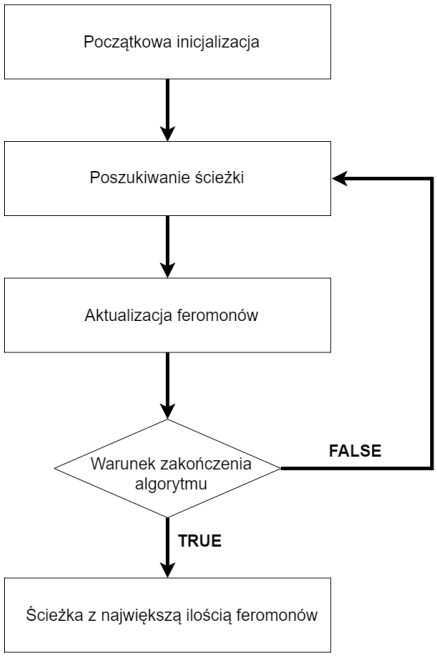
3.2 Algorytm mrówkowy

[4] Algorytm mrówkowy (ang. Ant Colony Optimization, ACO) to algorytm zainsirowany zachowaniem mrówek szukających pożywienia dla swojej kolonii. W prawdziwym świecie poruszają się one w sposób losowy w poszukiwaniu pożywienia. Gdy je znajdą, wracają do swojej kolonii, pozostawiając ślad składający się z feromonów, który naprowadza kolejne mrówki. Algorytm przeznaczony jest do znajdowania najbardziej optymalnych ścieżek. Trasy lepsze z czasem stają się mocniej oznaczone feromonami, przez co wybiera je więcej mrówek. Ostatecznie zwracana jest ścieżka najczęściej wybierana - najoptymalniejsza. Feromony po czasie wyparowują - ich siła maleje. Im dłuższa trasa, tym feromony mają więcej czasu na ulotnienie, więc ich koncentracja na drogach krótszych będzie silniejsza i zwabi więcej mrówek.



Rysunek 8: Algorytm mrówkowy - poszukiwanie najkrótszej drogi [Wikipedia].

Poniżej przedstawiono typowy przebieg algorytmu genetycznego. W tym przypadku warunkiem zakończenia jest osiągnięcie maksymalnej liczby iteracji.



Rysunek 9: Uproszczony przebieg algorytmu mrówkowego [Opracowanie własne].

Do implementacji algorytmu wykorzystana została paczka [5] *pants* zawierająca zaimplementowany algorytm mrówkowy. W pierwszym kroku należy ustawić parametry algorytmu.

```
    # [float] relative importance of pheromone (default=1)
    alpha = 1

    # [float] relative importance of distance (default=3)
    beta = 3

    # [float] percent evaporation of pheromone (0..1, default=0.8)
    rho = 0.8

    # [float] total pheromone deposited by each ant (>0, default=1)
    q = 1

    # [float] initial pheromone level along each edge of world (>0, default=0.01)
    t0 = .01

    # [int] number of iterations to perform (default=100)
    limit = 200

    # [float] how many ants will be used (default=10)
    ant_count = 10

    # [float] multiplier of the pheromone deposited by the elite ant (default=0.5)
    elite = .5
```

- *alpha* - względne znaczenie feromonu
- *beta* - względne znaczenie odległości
- *rho* - procentowe odparowanie feromonu $rho \in [0, 1]$
- q - całkowita liczba feromonów zdeponowanych przez każdą mrówkę po zakończeniu iteracji $q > 0$
- $t0$ - początkowy poziom feromonów wzdłuż każdej krawędzi
- *limit* - liczba iteracji do wykonania
- *ant_count* - liczba mrówek do użycia
- *elite* - mnożnik feromonu zdeponowanego przez najlepsze mrówki

Do uruchomienia algorytmu niezbędne jest utworzenie *wiata* - przestrzeni rozwiązań. Służy temu funkcja $World$, która przyjmuje dwa parametry:

- *nodes* - listę węzłów w postaci (x, y) przez które przechodzić będą mrówki,
- $fitness\_func$ - funkcję dopoasowania.

Tak jak w przypadku algorytmu genetycznego, funkcja dopasowania będzie wyznaczać odległości euklidesowe między węzłami.

```
def countDistances(x, y):
    return ((x[1] - y[1])**2 + (x[0] - y[0])**2)**0.5
```

Tym razem metoda przyjmuje nie listę a dwa jedynie dwa punkty. Po zdefiniowaniu powyższych parametrów jesteśmy gotowi do uruchomienia algorytmu.

```
solver = pants.Solver(
    alpha=alpha,
    beta=beta,
    rho=rho,
    q=q,
    t0=t0,
    limit=limit,
    ant_count=ant_count,
    elite=elite
)

# return the shortest path found through the given [World]
# param: the [World] to solve
# return: best solution found
# rtype: [Ant]
solution = solver.solve(world)
```

Wyniki prezentują się następująco:

Ilość węzłów	10	20	30	50	100	200
Dystans [° - stopnie]	21.29	31.63	37.27	58.51	93.37	135.51
Czas wykonywania [sek.]	0.14	0.44	0.99	2.58	10.03	44.10

Algorytm testowany był dla tych samych danych co algorytm genetyczny i tej samej liczbie iteracji, więc możemy porównać wygenerowane wartości. Dla przypomnienia poniżej ponownie przedstawiono tabele AG:

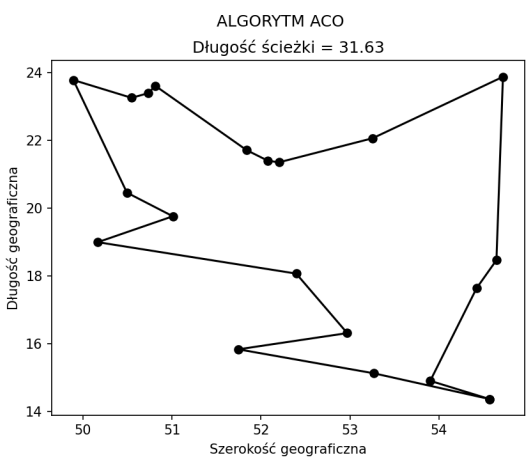
- $float(gene\_space) \in [0, 1]$

Ilość węzłów	10	20	30	50	100	200
Dystans [° - stopnie]	21.29	31.23	46.38	75.35	195.07	394.9
Czas wykonywania [sek.]	4.44	1.6	1.92	2.41	4.21	7.05

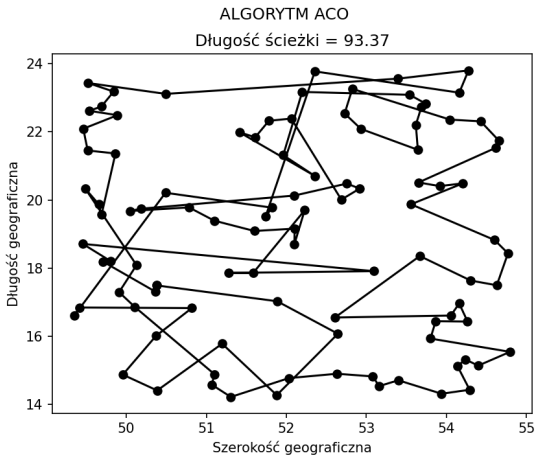
- $int(gene\_space) = [0, 1, ..., len(nodes)$

Ilość węzłów	10	20	30	50	100	200
Dystans [° - stopnie]	21.29	31.98	46.98	90.8	328.6	488.17
Czas wykonywania [sek.]	2.4	0.75	0.84	1.42	4.92	6.22

Algorytm mrówkowy działa lepiej dla większej ilości węzłów podając dokładniejszą kolejność przemierzania wierzchołków i wyliczając krótszy dystans do przemierzenia, jednak znacznie zwiększa czas wykonywania algorytmu.



(a) Ścieżka zwrócona dla 20 węzłów



(b) Ścieżka zwrócona dla 100 węzłów

Rysunek 10: ACO: wizualizacja ścieżek

Poprawiła się dokładność wyznaczania ścieżek, co za tym idzie ich wykresy wyglądaja znacznie lepiej, w porównaniu do wykresów algorytmu genetycznego [3] [4]. Warto zauważyć, że ACO zwrócił dokładniejszą ścieżkę dla 100 węzłów przy 100 iteracjach, niż AG przy 200 iteracjach.

3.3 Algorytm metaheurystyczny

[6]Metaheurystyka to ogólny algorytm do rozwiązywania problemów obliczeniowych. Można używać go do rozwiązywania dowolnych problemów, które można opisać za pomocą pewnych definiowanych przez ten algorytm pojęć. Należy pamiętać, że rozwiązanie może różnić się w zależnośći od wykonania (różne perturbacje i punkty początkowe) i nie ma gwarancji optymalności, jednak w większych przypadkach może to być znacznie szybsza alternatywa.
Do implementacji algorytmu wykorzystana została paczka [7] $python - tsp$ zawierającą:

- metody zwracające zawsze optymalne rozwiązanie problemu (skuteczne na małych instacjach problemu):
  - $solve\_tsp\_brute\_force$ - sprawdza permutacje i zwraca najlepszą,
  - $solve\_tsp\_dynamic\_programming$ - wykorzystuje programowanie dynamiczne; szybszy niż poprzedni, ale wymaga więcej pamięci,
- metody nie mające gwarancji znalezienia najlepszego rozwiązania (przeważnie zwracają wystarczająco dobre rozwiązanie w bardziej rozsądnym czasie w przypadku większych problemów):
  - $solve\_tsp\_local\_search$ - heurystyczne wyszukiwanie lokalne; szybkie ale może utknąć w minimum lokalnym,
  - $solve\_tsp\_simulated\_annealing$ - metaheurystyka symulowanego wyżarzania; może być wolniejszy, ale ma większe szanse na uniknięcie uwięzienia w minimum lokalnym,

Wykorzystamy ostatnią z wymienionych funkcji. [8]Symulowane wyżarzanie (ang. simulated annealing) to metoda obliczeniowa czerpiąca inspirację z dziedziny fizyki - symuluje proces wyżarzania ciała stałego. W tym procesie materiał metalurgiczny podgrzewany jest do wysokiej temperatury, a następnie pozostawiany do ostygnięcia, co pozwala lokalnym obszarom uporządkowania rosnąć na zewnątrz, zmniejszając w ten sposób naprężenia i zwiększając ciągliwość materiału (- przeciwieństwo hartowania). [9]Cechą charakterystyczną tej metody jest występowanie parametru sterującego zwanego temperaturą, która maleje w trakcie wykonywania algorytmu. Im wyższa wartość, tym bardziej chaotyczne mogą być zmiany (im wyższa temperatura metalu, tym bardziej jest on plastyczny).

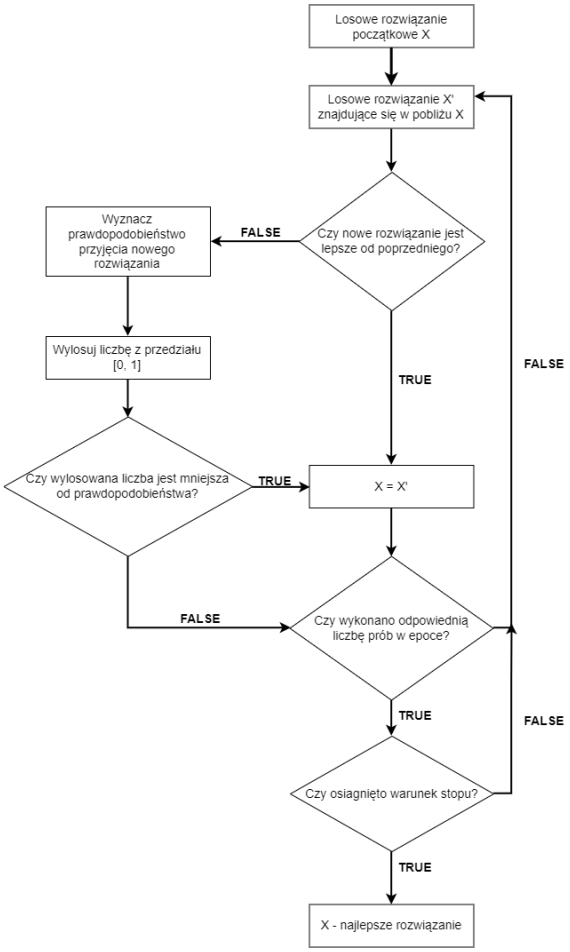
Początkowo losowane jest pewne rozwiązanie, które w kolejnych krokach jest modyfikowane. Jeśli w danym kroku uzyskamy rozwiązanie lepsze - wybieramy je zawsze. Istotną cechą symulowanego wyżarzania jest to, że z pewnym prawdopodobieństwem może być zaakceptowane również rozwiązanie gorsze (ma to na celu wyjście z maksimum lokalnego). Prawdopodobieństwo przyjęcia gorszego rozwiązania wyrażane jest przez rozkład Boltzmanna (równanie określające sposób obsadzania stanów energetycznych przez atomy, cząsteczki lub inne indywidua cząsteczkowe w stanie równowagi termicznej):

$$p = e^{\frac{f(X)-f(X')}{T}}$$

.

- p - prawdopodobieństwo zaakceptowania nowego kandydata na rozwiązanie,
- X - poprzednie rozwiązanie,
- X' - nowe rozwiązanie,
- T - temperatura, która jest parametrem kontrolnym,
- f - funkcja dopasowania (ang. fitness function), im wyższa wartość tym lepsza.

Ze wzoru można zauważyć, że prawdopodobieństwo przyjęcia gorszego rozwiązania spada ze spadkiem temperatury i wzrostem róznicy jakości obu rozwiązań.



Rysunek 11: Koncept algorytmu symulowanego wyżarzania [Opracowane wg. [9]$algorytmy - ency$]. $*epoka$ - próby z tą samą temperaturą

Algorytm jest przykładem metaheurystyki, więc schemat jest jedynie konceptem przebiegu programu. Przykładowo, w rozwiązywaniu problemu komiwojażera pobliskim rozwiązaniem może być zamiana miejscami dwóch węzłów. Czas przejść do kodu źródłowego.

```
permutation, distance = solve_tsp_simulated_annealing(
    distance_matrix,
    x0 = None,
    perturbation_scheme = "two_opt",
    alpha = 0.9,
    max_processing_time = None,
    log_file = None,
    verbose = False
)
```

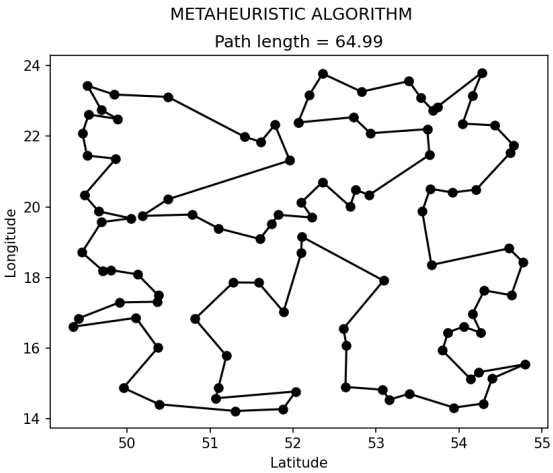
- *distance_matrix* - macierz odległości rozmiaru (n x n) z wpisem (i, j) wskazującym odległość między węzłami i oraz j,
- $x0$ - początkowa permutacja. Jeśli nie zostanie podana, zaczyna się od losowej ścieżki,
- *perturbation_scheme* - mechanizm wykorzystywany do generowania nowych rozwiązań,
- $alfa$ - współczynnik redukcyjny do obniżania temperatury. Z reguły 0.99 trwa dłużej i może przynieść lepsze rozwiązania, podczas gdy 0.9 jest szybsze, ale może nie być tak dobre,
- *max_processing_time* - maksymalny czas przetwarzania w sekundach. Gdy niepodany - metoda zostanie zatrzymana gdy wystąpią 3 cykle temperaturowe bez poprawy,
- log_file - tworzy plik dziennika (ang. log file) z szczegółami algorytmu
- *verbose* - jeśli $True$, wypisuje status algorytmu w każdej iteracji

Funkcja zwraca permutację węzłów od 0 do $n - 1$, która daje najmniejszy dystans (niekoniecznie najbardziej optymalny), oraz sam dystans. Paczka posiada również pomocniczą metodę *great_circle_distance_matrix*(), która tworzy odpowiednią macierz odległości.

Wyniki algorytmu prezentują się następująco:

Ilość węzłów	10	20	30	50	100	200
Dystans [° - stopnie]	21.29	31.71	37.59	49.35	64.99	92.47
Czas wykonywania [sek.]	0.32	0.35	0.65	1.12	5.88	25.75

Algorytm sam wylicza liczbę iteracji potrzebnych do rozwiązania problemu (lub kończy prace przy 3 cyklach bez zmiany wartości), której nie można ustawić, tak jak było to w przypadku powyższych algorytmów. Liczba ta określana jest jako $10 * len(x)$, gdzie $len(x)$ to liczba węzłów. Poniżej wykres ścieżki zwrócony dla 100 węzłów.



Rysunek 12: Algorytm metaheurystyczny: ścieżka zwrócona dla 100 węzłów.

4 Podsumowanie

Podsumujmy jeszcze raz wyniki wszystkich algorytmów. Poniżej przedstawiono tabelę sum dystansów odległości między punktami według kolejności permutacji zwróconej przez właściwe algorytmy.

Liczba węzłów / Algorytm	10	20	30	50	100	200	suma
AG: $gene\_space \in [0, 1]$	21.29	31.23	46.38	75.35	195.07	394.9	764.22
AG: $gene\_space = (0, ..., len(nodes))$	21.29	31.98	46.98	90.8	328.6	488.17	1007.82
Ant Colony Optimization	21.29	31.63	37.27	58.51	93.37	135.51	377.58
Metaheuristic algorithm	**21.29**	**31.71**	**37.59**	**49.35**	**64.99**	**92.47**	**297.4**

Jak można zauważyć wyniki zwracane przez algorytm metaheurystyczny okazują się znacznie lepsze od pozostałych, jednak należy zauważyć, że projekt skupiał się na porównaniu działania algorytmów dla 100 iteracji, podczas gdy liczba iteracji owego algorytmu jest odgórnie ustalana przy jego wykonywaniu. Drugim najlepszym algorytmem jest ACO - algorytm mrówkowy. Porównajmy teraz czasy działania programów.

Liczba węzłów / Algorytm	10	20	30	50	100	200	suma
AG: $gene\_space \in [0, 1]$	4.44	1.6	1.92	2.41	4.21	7.05	21.63
AG: $gene\_space = (0, ..., len(nodes))$	**2.4**	**0.75**	**0.84**	**1.42**	**1.92**	**6.22**	**13.55**
Ant Colony Optimization	0.14	0.44	0.99	2.58	10.03	44.10	58.28
Metaheuristic algorithm	0.32	0.35	0.65	1.12	5.88	25.75	34.07

Algorytm najmniej dokładny okazał się być najszybszym. Dwa programy (ACO i metaheurystyczny) zajmujące czołowe miejsca względem dystansów zajmują ostatnie miejsca w klasyfikacji czasowej. Wniosek: algorytmy o większej dokładności obliczeniowej są tym samym bardziej kosztowne czasowo.

Literatura

[1] Wikipedia [$AlgorytmGenetyczny$]

[2] Pypi [$PyGAD$]

[3] DEVcommunity [$GeneticAlgorithmswithPyGAD$]

[4] Wikipedia [$AlgorytmMrowkowy$]

[5] Pypi [$ACO - Pants$]

[6] Wikipedia [$Metaheurystyka$]

[7] Pypi [$Python - TSP$]

[8] Medium [$SimulatedAnnealing$]

[9] Algorytmy-ency [$SymulowaneWyzarzanie$]