Algorytm genetyczny dla problemu komiwojażera

Piotr Popis

Czerwiec 2020

1 Wprowadzenie

1.1 Problem komiwojażera(Traveling Salesman Problem)

Jest jednym z najszerzej przebadanych problemów optymalizacji kombinatorycznej. Polega na zminimalizowaniu dystansu trasy Salesmana, gdzie trasa musi przechodzić przez zadane n (załóżmy, że) *miast* z jego listy, które powinien przejść dokładanie raz oraz znane są odległości pomiędzy miastami. Inaczej mówiąc szukamy minimalnego cyklu Hamiltona w pełnym grafie ważonym.
Problem możemy przedstawiać w różnych wariantach na przykład:

1. symetryczny sTSP
2. asymetryczny aTSP
3. wielokrotny mTSP
4. zgeneralizowany gTSP

W symetrycznym przypadku odległość z miasta A do B jest taka sama jak odległość z B do A. W asymetrycznym odległości te mogą być różne ($c_{ij} \neq c_{ji}$). W 3 przypadku wielokrotnym miasto może być odwiedzone więcej niż raz, a w przypadku zgeneralizowanym(uogólnionym) nie wszystkie miasta muszą być odwiedzone. Tego typu problemy powstają w różnych zastosowaniach ciekawymi przykładami są między innymi vehicle routing problem, printed circuit board punching sequence problems, wing nozzle design problem in air craft design. Należy do problemów NP- trudnych.

1.2 Algorytm Genetyczny(Genetic Algorithm)

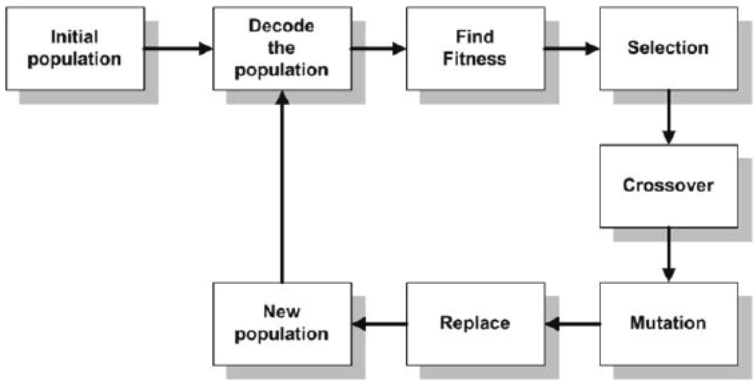
1.2.1 Opis

Jest heurystyką, a mianowicie członkiem grupy algorytmów ewolucyjnych. Jest jedną z metod inteligencji obliczeniowej. Metodologia jest inspirowana efektywnością naturalnej selekcji w ewolucji biologicznej. W przeciwieństwie do innych metaheurystyk takich jak tabu serach czy simulated annealing nie generuje jednego rozwiązania, a grupę rozwiązań (generację). Bieżąca generacja nazywana jest populacją. Każdego członka bieżącej grupy nazywamy nieraz chromosomem. Do stworzenia nowej generacji używamy genotypów z poprzedniej generacji. Utworzenie nowego chromosomu, w zasadzie grupy chromosomów - generacji następuje zgodnie z różnymi operacjami.
Trzy najczęściej używane to:

1. selekcja(selection),
2. krzyżowanie(crossover),
3. mutacja(mutation).

Operacja selekcji pozwala wybrać odpowiednich osobników do nowych generacji. Wybiera zazwyczaj tych najlepszych, ale warto zostawić dla tych gorszych pewne prawdopodobieństwo wybrania w celu ucieczki z lokalnych optimum. Istnieje też możliwość rekombinacji czyli bezpośrednim przekazywaniu najlepszym osobników z populacji do nowej generacji w celu zachowania rozwiązań o bardzo wysokiej jakości. Krzyżowanie pozwala na wymieszanie genów wybranych w selekcji osobników. Mutacja natomiast polega na zazwyczaj losowym zmienieniu niektórych genów. Każda z powyższych operacji ma wiele efektywnych implementacji, które są zależne od problemu. Znaczy to tyle, że die da się jednoznacznie określić najlepszej implementacji, tylko jedna jest bardziej dokładna obliczeniowo, inna znacznie szybsza.

1.2.2 Schemat



1.3 Kilka wybranych implementacji operacji(pseudokod)

1.3.1 Krzyżowanie

One-point crossover -Zamiana suffixów dwóch osobników.

```
v,w - vectors ( paths)
c = randomint(1,l)
if c!=1:
  for i in (c,l):
    swap(vi,wi)
return v,w
```

Two-point crossover Wymiana genów na zadanym przedziale c,d.

```
v,w - vectors ( paths)
c,d = randomint(1,l),randomint(1,l)
if c>d:
  c,d=d,c
if c!=d:
  for i in (c,d-1):
    swap(vi,wi)
return v,w
```

Uniform crossover Zamiana każdego genu z prawdopodobieństwem p.

```
p - probab of swapping an index
v,w - vectors ( paths)
c,d = randomint(1,l),randomint(1,l)
for i in (1,l):
  if p>random.uniform(0,1):
    swap(vi,wi)
return v,w
```

1.3.2 Mutacja

Standard Swap Zamiana genów na pozycjach (losowych i,j).

```
i,j - random cities range(0,n)
return swap(path,i,j)
```

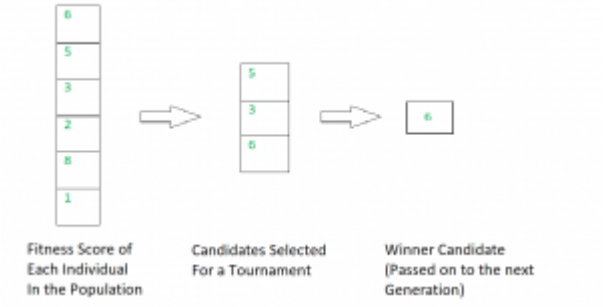
Teraz znając przykładowe implementacje operacji w algorytmie genetycznym głównie opierające się na permutacjach, możemy zacząć rozważania. Możemy postawić, że naszym celem jest znalezienie jak najbardziej optymalnego rozwiązania w jak najkrótszym czasie. Problemem stają się lokalne optima, których musimy pokonać odpowiednio dobierając operacje mutacji, krzyżowania oraz selekcji. Rozważmy kilka solucji problemu komiwojażera wykorzystując optymalny algorytm genetyczny.

2 Różne strategie selekcji

Jedną z operacji w algorytmie genetycznym jest selekcja. Służy do wybrania osobników, które następnie chcemy skrzyżować. Przy doborze rodziców musimy uważać, żeby nasze wybory nie doprowadziły do zamknięcia się na jeden genotyp

2.1 Tournament Selection

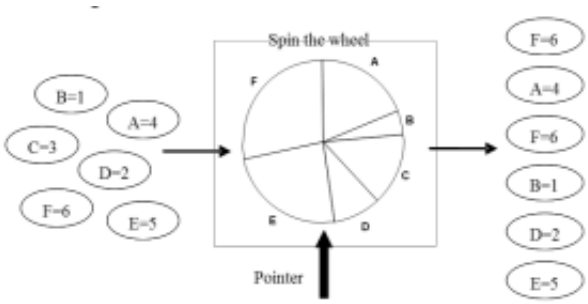
Bardzo popularną metodą jest selekcja turniejowa. Z większej populacji wybieramy x przedstawicieli losowo, wielkość x nazywamy rozmiarem turnieju. Binary Tournament zatem, to selekcja, w której do konkurencji wybieramy tylko 2 chromosomy. Następnie z wyodrębnionego zbioru wybieramy najlepszego. Takie podejście pozwala zachować różnorodność.




```
P - population
t- tounament size >0
best = random.choice(P)
for i from 2 to t:
  next = random.choice(P)
  if fitness(next)>fitness(Best)
    best=next
return best
```

2.2 Proportional Roulette Wheel Selection

Osobniki są wybierane z prawdopodobieństwem bezpośrednio proporcjonalnym do ich jakości (f -fitness value). Odpowiada to nieco kole ruletki.Koło możemy podzielić na segmenty odpowiadające proporcjom rodziców. Prawdopodobieństwo takie możemy wyznaczyć korzystając z wzoru $p_i = \frac{f_i}{\sum_{j=1}^{n} f_i}$. Główną zaletą jest to, że żaden z elementów populacji nie zostaje przekreślony. Ma też niestety swoje wady to znaczy, że jeśli populacja początkowa zawiera załóżmy, 2-3 silne genotypy, ale nie idealne, a pozostali członkowie populacji są bardzo słabi - zamknie się na tych dwóch najlepszych (prawdopodobnie). Z drugiej strony słabe rozwiązania są szybko eliminowane. Jednak jeśli rozważamy problem minimalizacji to jest nieco uciążliwe rozwiązanie, bo musimy zaimplementować funkcje konwertującą funkcje minimalizującą do maksymalizującej. Wprowadza zatem lekkie zamieszanie.



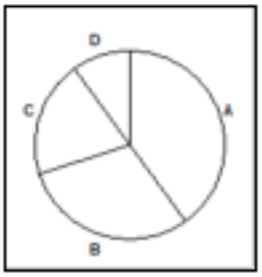

```
do once per generation
  global p = [ind_1,...,ind_ps]
  global f = [fitness(pi) for pi in p]
  if f is all 0.0s:
    convert f to all 1.0s
  for i from 2 to l:
    fi = fi+ f_{i-1}
perform each time
  n = random(0,fl)
  for i from 2 to l:
    if  f_{i-1}<n<=fi:
      return pi
   return p1
```

2.3 Rank-based Roulette Wheel Selection

Prawdopodobieństwo wybrania chromosomu jest tutaj zależna od fitness value oraz jest relatywna do całej populacji. Najpierw osobnicy zostają posortowani zgodnie z ich jakością, a następnie wyliczane jest prawdopodobieństwo na podstawie ich rankingu ,a nie bezpośrednim fitness. Potrzebujemy funkcji mapującej indeksy na posortowanej liście do listy jej prawdopodobieństw selekcji. Funkcja ta może, ale nie musi być liniowa. Bias może być regulowany przy użyciu nacisku wyboru SP dla liniowej np 2>SP>1. Pozycja zatem może być skalowana zgodnie z formułą:

$$Rank(Pos) = 2 - SP + (2.(SP - 1).\frac{Pos - 1}{n - 1})$$

Można uniknąć skalowania, ale to może stać się bardziej kosztowne obliczeniowo z powodu sortowania. Takie podejście pozwala zablokować sytuację powodującą utknięcie w najlepszych chromosomach początkowych. Różnica pomiędzy kołami ruletki dla odpowiednio proporcjonalnej i rank-based selekcji.



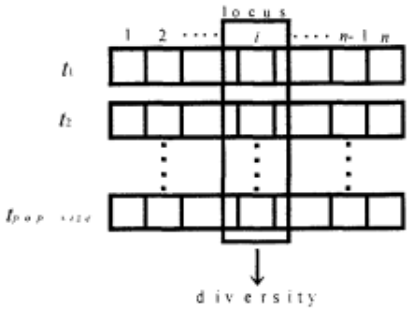
Porównanie selekcji:

Instances	Known optimal solution	Tournament	Proportional	Rank-based
10-city	-	2.8567	2.8567	2.8567
20-city	-	4.0772	4.0772	4.0772
30-city	-	4.8352	4.9075	4.6683
40-city	-	6.1992	6.5127	5.7311
burma14	30.8785	30.8785	30.8785	30.8785
bay29	9074	9077	9079	9074
dantzig42	679	725	760	679
eil51	425	470	513	430

3 Algorytm genetyczny bazujący na entropii

3.1 Opis

Aby poradzić sobie z problemem lokalnego optimum Yasuhiro TSUJIMURA and Mitsuo GEN zaprezentowali rozwiązanie bazujące na entropii. Wykorzystali operację cycle crossover, swap mutation oraz selekcję - roulette wheel selection. Jak wiemy zwiększenie się ilości podobnych chromosomów w populacji może doprowadzić do utknięcia w lokalnym optimum. W EBGA mierzymy różnorodność chromosomów w każdej nowej generacji i ulepszamy populacje o małej różnorodności. Jak wyznaczyć zatem różnorodność?



w tak ustawionych chromosomach przechodzimy po każdej kolumnie i sprawdzamy w niej różnorodność. Locus diversity H_i i-tego locus możemy wyznaczyć ze wzoru:

$$H_i = \sum_{c \in C} pr_{ic} ln(pr_{ic})$$

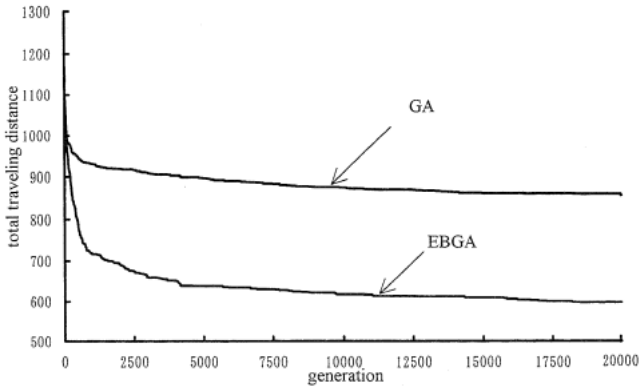
, gdzie $pr_{ic} = \frac{na_{ic}}{population_{size}}$
C - zbiór miast
na_{ic} -ilość wystąpień miasta c na(index kolumny) locus i H_i osiąga maksymalną wartość $ln(population_{size})$, gdy każde miasto z C występuje jednolicie, a 0 gdy jedno z miast występuje znacznie częściej niż pozostałe. H_i musimy oceniać na podstawie jakiejś stałej (próg(ang. threshold)) w tym przypadku $ln2$.Jeśli próg jest większy lub równy H_i to mamy niską różnorodność. Każde umiejscowienie porównujemy z progiem i jeśli jest większe od $\frac{n}{a}$ to zwiększamy podzielność. Parametr a to po prostu integer z przedziału [2,5]. Jak zwiększyć różnorodność? Korzystamy z procedury Med - Pop to znaczy, wybieramy $m = random(\frac{pop_{size}}{a}, pop_{size} - 1)$ chromosomów z populacji. W każdym wybranym chromosomie wymieniamy geny wzdłuż loca z innym mającym mniejszą różnorodność na locus niż próg.

3.2 Podsumowanie

1. Wygeneruj pop_{size} chromosomów losowo.
2. Ewaluacja wyznacz jakość każdego chromosomu (fitness) i oceń, który jest najlepszy $eval(t_k) = \frac{1}{\sum_{i=1}^{n} d(c_i,c_{i+1}) + d(c_n,c_1)}$
3. Krzyżowanie CX cycle crossover na wybranych chromosomach z prawdopodobieństwem pr
4. Mutacja zamiana miast z prawdopodobieństwem pm

5. selekcja wybierz pop_{size} chromosomów korzystając z roulette wheel selection.

6. Ulepsz różnorodność populacji korzystając z procedury med-pop.



Jak widać EBGA znajduje dużo lepsze rozwiązania niż GA.

4 Algorytm genetyczny metodą losowych kluczy dla GTSP

Tak jak już wspomnieliśmy GTSP to zgeneralizowany wariant powszechnie znanego TSP. Różnica polega na tym, że nie każde z miast musi zostać odwiedzone. Mianowicie zbiór miast C jest dzielony na m zbiorów rozłącznych tak, by ich suma była równa C, tzn każde miasto musi należeć do jakiegoś zbioru C_i. Tsp jest zatem specjalnym przypadkiem GTSP takim, że C jest podzielone na |C| podzbiorów. W tym przypadku skorzystamy z operacji reprodukcji, czyli skopiowaniu najlepszych elementów z populacji do nowej generacji. W tym problemie skorzystamy również z random keys w celu kodowania rozwiązań. Użycie ich jest możliwe, gdy nasz problem może opierać się na permutacjach integerów i , w których one- lub two- point crossover stwarza problemy. Przeanalizujmy metodę random- key na przykładzie. Załóżmy, że mamy ścieżkę 4 2 1 5 3 , nadajemy kolejnym elementom losową wartość (random key) na przedziale (0,1) i teraz do naszych kluczy przyporządkujemy wartości ze zbioru 1,...,n rosnąco. to znaczy, że jeśli mamy 0.1 0.42 0.3 0.9 0.7 to nasz ciąg zostanie zakodowany na 1 3 2 5 4. Wróćmy do naszego zbioru C podzielonego na załóżmy 3 podzbiory C_i, C_j, C_k. Najpierw wykorzystując random keye

decydujemy w jakiej kolejności odwiedzimy grupy miast załóżmy, że i -> j -> k. Następnie losujemy klucze dla grup miast i decydujemy o kolejności w tych podgrupach. W tym przypadku 20% populacji będzie przekazane do nowej generacji, 70% stworzone przez crossover, a pozostała część zostanie wygenerowana przez imigracje. Przy reprodukcji korzystamy z strategii elitizmu. Pozwala to na zachowanie odpowiednich do mieszania genów. Do stworzenia potomków używamy uniform crossover. Operacja imigracji co jakiś czas dodajemy po prostu generujemy nowe losowe rozwiązania, które dołączamy do nowej populacji. Ulepszenie każde nowe rozwiązanie uzyskane w GA staramy się poprawić swapując losowe miasta. Druga operacja to 2-opt próbująca usunąć dwa miejsca i wstawić je w inne miejsca tak, aby uzyskać pojedynczy tour o niższym koście. Swap pozwala zamienić miasta z innych komponentów C_i. Populacja zarządzamy w taki sposób, aby nie dodawać duplikatów, duplikaty to takie chromosomy, których ulepszenia są takie same. Zalety takiej implementacji to prostota implementacji, jest łatwa do przekształcenia dla innych problemów np MTP lub TCP. Schemat byłby niemal identyczy.

5 Algorytm genetyczny z mixed region search dla ATSP

W asymetrycznym TSP mamy do czynienia z różnymi dystansami z miasta A do B, a miasta B do A. Większość algorytmów stworzonych dla STSP mogą zostać zmodyfikowane do ATSP. Mamy n miast do zwiedzenia. Problem ATSP może być sformułowany w następujący sposób:

$$min \sum_{i=1} n \sum_{j=1} n x_{ij} c_{ij}$$

, gdzie x $\in$ =1 gdy, jest jest takie bezpośrednie połączenie pomiędzy miastami i oraz j na naszej trasie lub 0, gdy nie jest. Liczba rozwiązań niewykonalnych dla ATSP z dwiema podtrasami f(n,2) może być znaleziony w taki sposób:

$$f(n,2) = \sum_{i=2}^{floor(0.5n)}$$

, gdzie h to ilość niewykonalnych solucji, gdzie i to ilość nodów w jednej podtrasie, a druga ma n-i nodów. W tym algorytmie bardziej przykuwamy uwagę do operacji krzyżowania. Nasze wyniki mają być później

przepuszczone przez patching algorytm Karpa.PMX i TBX czyli partially matched crossover oraz tie break crossover. TBX, którego będziemy jest lekko zmodyfikowany względem standardowego. Najpierw dzielimy naszą trasę tak jak w 2-point crossover. Następnie wybieramy losową ilość elemntów do wymiany i produkujemy nie prawidłową trasę (z powtórzeniami miast). Kolejno losujemy losowe uniformy na przedziale 0,1 tak jak w random-key values dla miast powtarzających się tzn 1919 generuje 0.1 0.2 0.3 0.4 ciąg (losujemy dwa takie wektory) i dodajemy do naszych ścieżek w powtórzeniami Teraz wartości mniejsze przechodzą do mniejszej z zamienionych, a wartości większe do większych (względem tych, które na początku podmieniliśmy). Dla Uproszczenia przykład: Wybrane jednostki:

(123|4567|89)

(452|1876|93)

Mieszamy, tworzymy szlaki z powtórzeniami:

(123|1867|89)

(452|4576|93)

Losujemy uniformy:

[0.3][0.6][0.7][0.3]

[0.3][0.5][0.6][0.2]

Dodajemy:

([1.3]23|[1.6][8.7]67|[8.3]9)

([4.3][5.5]2|[4.6][5.2]76|93)

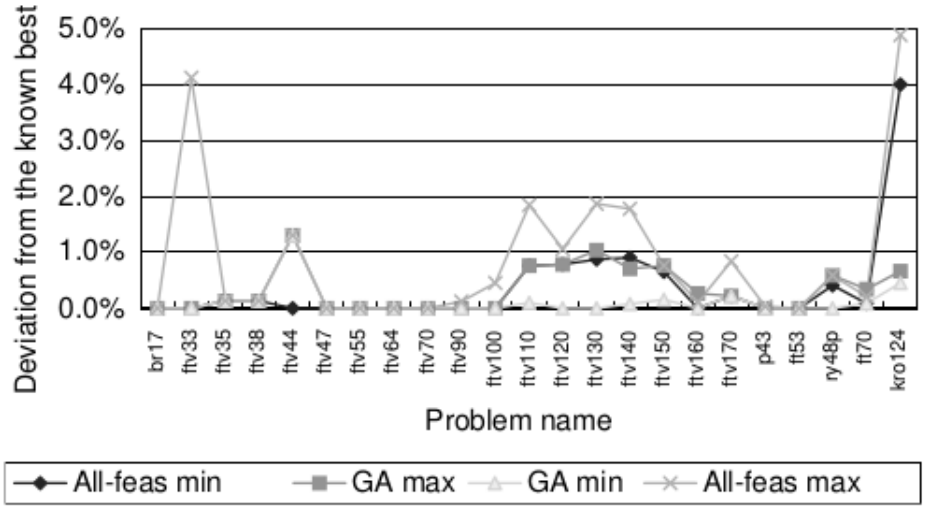
Pozbywamy się powtórzeń:

(123|4876|59)

(182|4567|93)

Do reprodukcji wykorzystujemy stochastyczne próbkowanie z zamianą. (relatywny ranking względem fitness). W celu mutacji korzystamy z three way -swapping. Korzystamy z niej, gdy nie ma żadnych zmian w wartościach funkcji fitness. W każdej iteracji wybieramy grupę elitarną

60% . Nasze rozwiązanie poza grupami elitarnymi utrzymuje dobre rozwiązania - nie wpada w lokalne optima. Z każdej grupy 60/30/10 wybieramy przedstawicieli i mixujemy. Raz na jakiś czas do naprawienia rozwiązań, bo w naszym algorytmie generujemy błędne używamy patching algorytm. Graf porównujący:



Wyniki wskazują na to, że genetyczny algorytm może być wykorzystany w problemie asymetrycznym TSP i osiągnąć bardzo dobre wyniki.

6 Algorytm genetyczny dla MOTSP(multi-objective)[FRAMEWORK PROPOSITION]

Motsp polega na tym, ze skupiamy się na zminimalizowaniu lub zmaksymalizowaniu wielu funkcji np cost, distance. Jakość solucji jest ewaluowana na podstawie optymalności Pareto. Solucje takie mogą być dominujące lub nie. Zbiór dominujący, inaczej optymalność Pareo to zbiór rozwiązań, które przekłamują w przestrzeni wykonalno- decyzyjnej. Model matematyczny MOTSP:

$$\text{minimize } f(x) \begin{cases} f1(x) = c_1^{x(n),x(1)} + \sum_{i=1}^{n-1} c_1^{x(i),x(i+1)} \\ f1(x) = c_1^{x(n),x(1)} + \sum_{i=1}^{n-1} c_1^{x(i),x(i+1)} \\ f1(x) = c_1^{x(n),x(1)} + \sum_{i=1}^{n-1} c_1^{x(i),x(i+1)} \end{cases}$$

,gdzie n to ilość miast, c_j to koszt do j, a to cykliczna permutacja n miast. Euclidean distance jest wykorzystywany do wygenerowania kosztu oraz macierzy dystansów. Dla dwóch wymiarów mamy formułę:

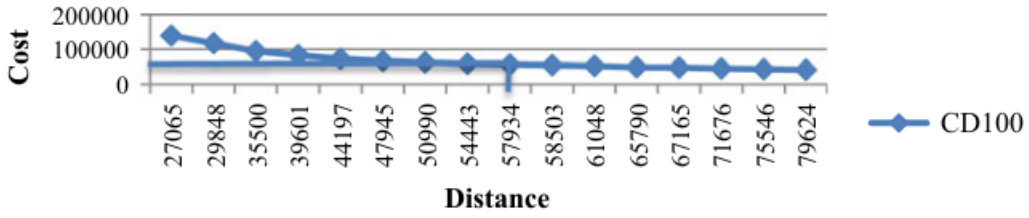
$$ECD = \sqrt[2]{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

. Na podstawie tego wzoru określamy dystans pomiędzy dwoma miastami. A Pareto jest wykorzystywany do wydobycia rozwiązań efektywnych. Algorytm:

1. Wczytaj dane dla multi- objective instaces(Euclid)
2. Znajdź dystans dla instancji korzystając Z ECD
3. Macierz jest formowana zgodnie z danym miastem.
4. GA dla tak wygenerowanej macierzy.
   (a) Stwórz losową populację z macierzy.
   (b) Określ jakość chromosomów.
   (c) Skrzyżuj wybranych rodziców lub zmutuj
   (d) Sprawdź jakość rozwiązań nowych.
5. Zwróć najlepszą trasę z odpowiednim kosztem i dystansem.

S. no.	Instances	Size	Best-so-far		GA optimum	
			Cost	Distance	Cost	Distance
1	euclidAB100	100	55,250	54,172	55,777	53,643
2	euclidCD100	100	56,731	56,626	56,493	57,934
3	euclidEF100	100	53,536	54,258	53,581	54,213
4	euclidAB300	300	120,208	120,149	119,979	120,368
5	euclidCD300	300	119,457	119,453	119,165	119,805
6	euclidEF300	300	119,735	119,726	119,490	120,002
7	euclidAB500	500	172,344	172,376	172,059	172,628
8	euclidCD500	500	174,543	174,563	173,906	175,141
9	euclidEF500	500	175,218	175,144	174,987	176,280

GA zoptymalizowało wszystkie pozycje i znajduje minimalne koszty/dystanse. MOTSP daje przybliżone Pareto optymalne solucje dla każdej pozycji w racjonalnym czasie. Dla EuclidCD100 :



Znaczy to tyle, że GA dla problemu MOTSP to bardzo dobry wybór.

7 Rozwiązanie problemu TSP wykorzystując algorytm genetyczny w rzeczywistej aplikacji

W tej sekcji zderzymy się z brutalną rzeczywistością i ilością danych jakie musimy przeanalizować. Nasze goals to:

1. Zmniejszyć koszt i czas podróży sprzedawcy.
2. Zwiększyć ilość odwiedzanych na dzień.
3. Priorytetować pewnych klientów.
4. Uwzględnić dni pracy i weekendy oraz nadgodziny.

Jednocześnie chcemy jednak zadbać o

1. odpowiednio się przygotować,
2. ulepszać relacje z klientami,
3. zwiększać pewność siebie sprzedawców,
4. poprawić wydajność firmy (ogólną),
5. zmniejszyć koszty,
6. zoptymalizować czas pracy.

Dodatkowo musimy uwzględnić rzeczy jak dni wolne, przerwy śniadaniowe, przystosować czas pracy do godzin porannych, dni preferowane do odwiedzin przez klientów. W celu rozwiązania problemu musimy skonstruować pewien optymalny model. Zacznijmy od minimalizacji kosztu.

$$ovr_cost = distance(R) * km_cost + total_hours(R) * hour_cost$$

gdzie R jest danym rozwiązaniem, dystans sumą sub- tras. Totalny czas możemy wyliczyć z formuły

$$total_hours(R) = driving_time(R) + waiting_time(R) + service_time(R) + lunch_time(R)$$

Nasz pesymistyczny przypadek kosztu na godzinę to $worst = hour_cost + speed * km_cost$. Znormalizujmy funkcję

$$norm_cost_per_h(R) = \frac{\frac{overall_cost}{total_hours(R)}}{worst}$$

. Przejdźmy do maksymalizacji ogólnej ważności klienta. Ważność klienta będzie najlepiej oceniona jeśli osiągnie max, gdy priorytetowi klienci zostaną odwiedzeni w odpowiednich - porannych godzinach, zatem:

$$importance_{overal}(R) = \sum_{d=1}^{H}(\sum_{i=1}^{N} IMP_i * DAYWEIGHT_d * VISIT_{R,i,d})$$

, visit mówi o tym ile razy powinien być odwiedzony dany klient. Aby tą funkcję znormalizować musimy znowu sprawdzić extrema.W najgorszym przypadku importance =0, kiedy klient nie został odwiedzony. W najlepszym Każdy zostaje odwiedzony w porannej odpowiedniej porze.

$$importance_{best} = \sum_{i=1}^{N} IMP_i * DAYWEIGHT_1 * FREQMAX_i$$

as

$$norm_i importance(R) = importance_{overall}(R)/importance_{best}$$

Kolejny czynnik, który chcemy usprawnić to minimalizacja ilości kierunków, które nie były odwiedzane. Mimo wszystko chcemy zwiedzić jak największa ilość klientów w czasie podróży, ale nie naruszając czasu pracy sprzedawcy.

$$visits_{max} = \sum_{i}^{N} FREQMAX_i$$

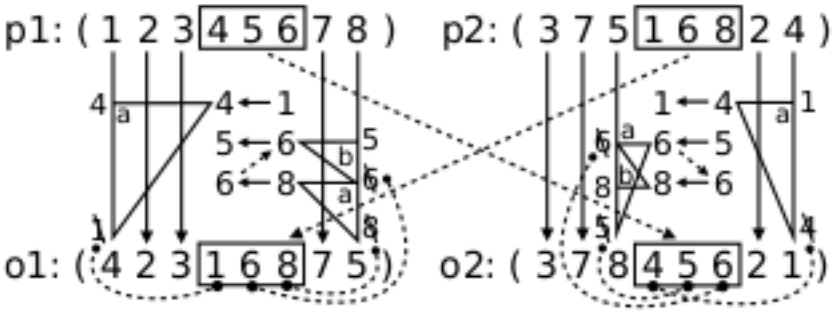
$$visited(R) = visits_{max} = \sum_{d=1}^{H}(\sum_{i=1}^{N} VISIT_{R,i,d})$$

$$wvisited(R) = visits_{max} - visited(R)$$

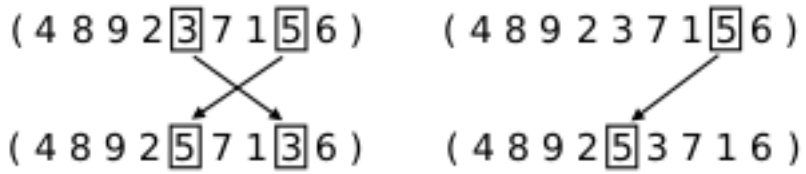
Zatem optymalizując nasz problem zbiega do zminimalizowaniu funkcji fintess(R), gdzie

$$fitness(R) = (a{*}normCostPerHour(R){+}b{*}(1{-}normImportance(R))){*}(c{*}unvisited(R){+}1)$$

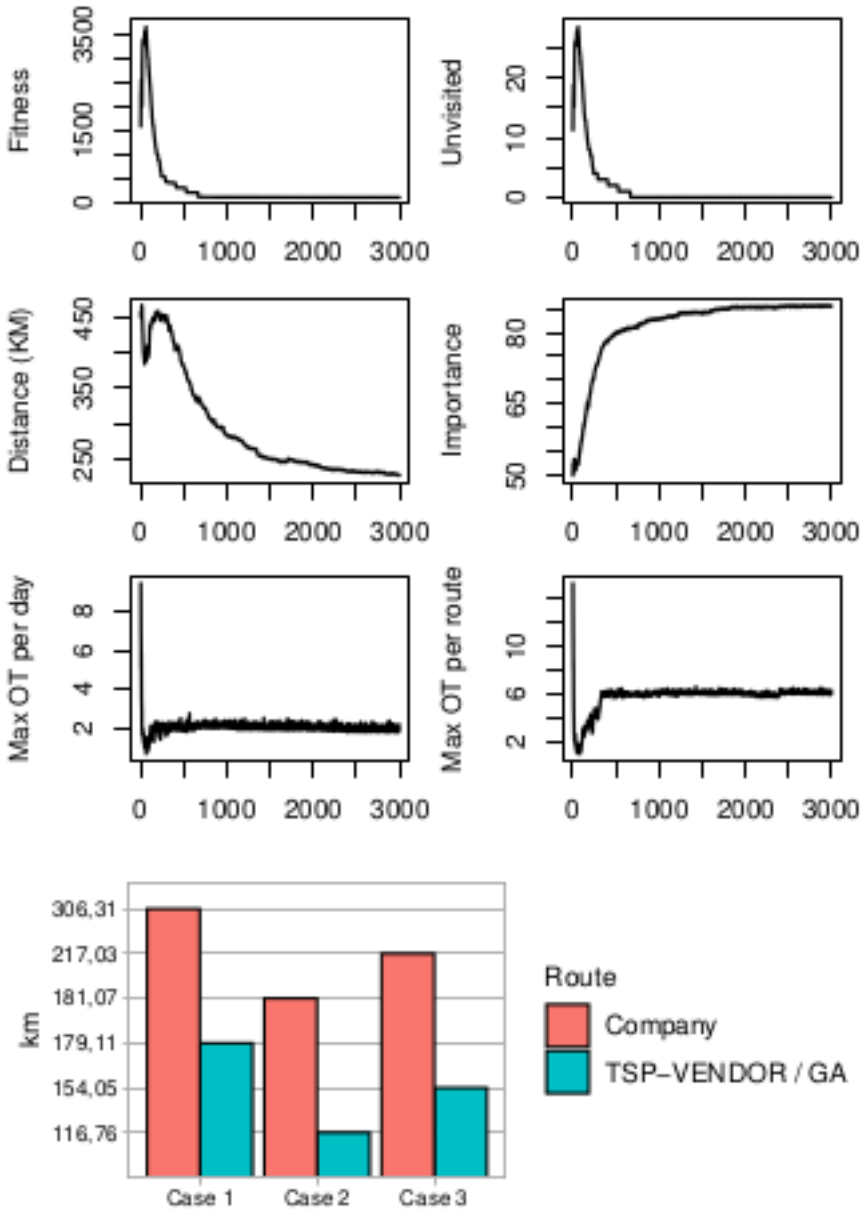
. Nasza reprezentacja wyglądaj ka w poprzednich sekcjach są to ciągi miast 1 2 3 ... Czasem Trasa może mieć kilka dni. Jako operację krzyżowania skorzystamy z omówionego już PMX

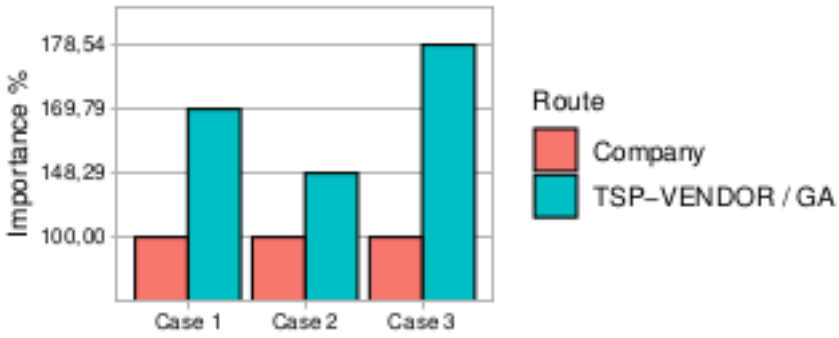


Z takim podejściem nie musimy się martwić o duplikaty miast. Korzystamy z EM (Exchange Mutation)



Dla uproszczenia nasze współczynniki a,b,c stawiamy na 1.





8 Własna implementacja

Implementując GA dla problemu aTSP w python3.

8.1 Reprezentacja

Ścieżki są reprezentowane w listach.Przykład dla 100 miast.

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

8.2 Populacja początkowa

```
def get_random_population(n, pop_size):
    num_list = [i for i in range(n)]
    pop = [num_list.copy() for _ in range(pop_size)]
    for x in pop:
        random.shuffle(x)
    return pop
```

Ale wrzucam też kilka lub jedno rozwiązań " zachłannych", łapie po kolei od src to miasto, do którego odległość jest najmniejsza i zakazuje wchodzić do tych, w których już byłem (mikro tabu list)

```
def get_good_initial(graph, n, src):
    T = [src]
    for i in range(n):
```

```
        minimalCity = get_next_city(graph, T)
        T.append(minimalCity)
    return T[:len(T) - 1]
```

8.3 Selekcja

Korzystam z Tournament selection opisaną w powyższych sekcjach.

```
def tournament_selection(population, distances):
    ts = 3
    best = random.choice(population)
    for i in range(0, ts):
        nex = random.choice(population)
        if compute_distance(nex, distances) < compute_distance(
    best, distances):
            best = nex
    return best
```

8.4 Krzyżowanie

Korzystam z Cycle Crossover opisana w powyższych sekcjach, z pewnym prawdopodobieństwem podmieniam geny z różnych chromosomów.

```
def cycle_crossover(p1, p2, prc):
    for i in range(len(p1)):
        if random.uniform(0, 1) < prc:
            p1[p1.index(p2[i])], p2[p2.index(p1[i])] = p1[i],
    p2[i]
            p1[i], p2[i] = p2[i], p1[i]
    return p1, p2
```

8.5 Mutacja

Korzystam z dwóch rodzajów mutacji jedna to po prostu podmiana częściowa

```
def hard_mutate(path):
    i, j = random.sample([i for i in range(len(path))], 2)
    s, b = i, j
    if j < i:
        s, b = j, i
    return path[s:b] + path[:s] + path[b:]
```

Kolejna to prosty losowy swap

```
def random_swap(path):
    i, j = random.sample([i for i in range(len(path))], 2)
    return swap(i, j, path)
```

8.6 Reprodukcja i zwiększanie różnorodności w populacji

Przekazuję zawsze 30% najlepszych solucji do nowej generacji i dodaje pozostałe 70% losowo wybranych z nowej generacji.

```
def improve_and_reproduce(population, pop_size, distances):
    selected = []
    popu = population.copy()
    while len(selected) < pop_size // 3:
        z = get_min(popu, distances)
        selected.append(z)
        popu.remove(z)
    while len(selected) < pop_size:
        z = random.choice(popu)
        selected.append(z)
        popu.remove(z)
    return selected
```

Co jakiś czas dodaje do generacji nowe rozwiązanie zachłanne.

```
        population.append(get_good_initial(distances, n, random
    .randint(0, n - 1)))
```

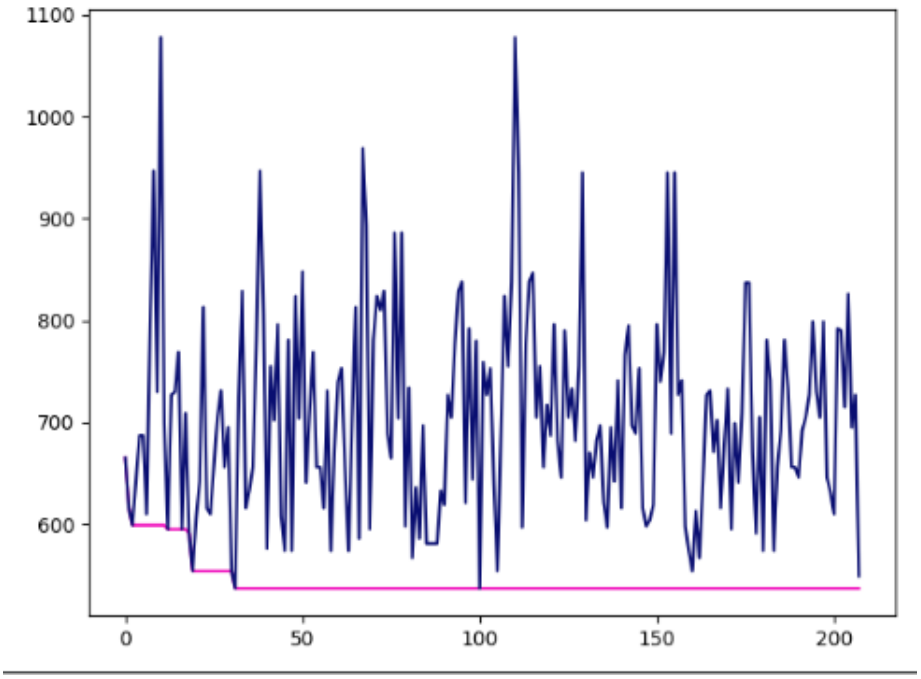
9 Zarys całego algorytmu

```
def genetic_algorithm(t, n, distances, pop_size=24, plot=True,
    prcx=0.5):
    his = []
    end_time = get_current_time() + get_millis(t)
    population = get_random_population(n, pop_size)
    population.append(get_good_initial(distances, n, random.
    randint(0, n - 1)))
    best = get_min(population, distances)
    while get_current_time() <= end_time:
        Q = []
        while len(Q) < pop_size:
            p1 = tournament_selection(population, distances)
            p2 = tournament_selection(population, distances)
            c1, c2 = cycle_crossover(p1, p2, prcx)
            c1 = random_swap(c1)
            c2 = random_swap(c2)
            if c1 not in population:
                Q.append(c1)
            if c2 not in population:
                Q.append(c2)
        population.extend(Q)
        population.append(get_good_initial(distances, n, random
    .randint(0, n - 1)))
```

```
        population = improve_and_reproduce(population, pop_size
    , distances)
        opponent = get_min(population, distances).copy()
        if compute_distance(opponent, distances) <
    compute_distance(best, distances):
            best = opponent
        if plot:
            his.append([compute_distance(best, distances),
                        compute_distance(opponent, distances)])
      # [compute_distance(x, distances) for x in population])
    if plot:
        plot_graph(his)
    return best
```

10 Wyniki

Dla porównania i pokazania, że mój algorytm nie zamyka się w optimach i stale sięga po nowe rozwiązania plot poniżej.



11 Konkluzja

Niestety, aż tak dobrych wyników jak Szanownym Panom Profesorom z artykułów nie udało mi się uzyskać(błąd jest pewnie większy niż 5%).

Jednak jak widać korzystając z operacji w powyższych artykułach udało mi się dobrnąć do dość optymalnego rozwiązania, co było celem algorytmu.

12 Podsumowanie (Semestralne?)

Często duża wiedza daje duże możliwości, w naszym przypadku tą wiedzą staje się znajomość algorytmów metaheurystycznych. W rzeczywistości często napotykamy na różnego rodzaju problemy. Może się zdarzyć, że nasz problem można przybliżyć do pewnego istniejącego już problemu. Tak jak TSP w powyższym przykładzie lub na przykład odszukiwanie odpowiedniego kształtu (ilość neuronów na layer) w sieci neuronowej i dobór odpowiednich aktywacji (tanh,sigmoid,..) wykorzystując np hill climbing, odpowiednie modelowanie i modyfikacje problemu mogą go zgeneralizować lub dopasować się do naszej sytuacji. Algorytm genetyczny jest natomiast jest inteligencją obliczeniową, która pozwala nam z takim problemów (tutaj NP-trudnych) wybrnąć.