Najważniejszy fragment kodu źródłowego

```
1.  def findNearest(lst, current):
2.      smallest = 100000
3.      for city in lst:
4.          if city == current:
5.              continue
6.          else:
7.              distance = calculateDistance(current, city)
8.              if distance < smallest:
9.              smallest = distance
10.             bestNeigh = city
11.     return bestNeigh
```

Uzasadnienie

Wybrałem powyższy fragment kodu, ponieważ jest on podstawą użytego przeze mnie algorytmu tzn. powtarzalny algorytm najbliższego sąsiada (ang. *repetitive nearest neighbour algorithm*). Ponadto w znacznym stopniu pomógł mi on zrozumieć ideę algorytmów zachłannych. W każdym kroku rozważamy tylko aktualnie najlepiej rokujące posunięcie, nie zastanawiając się nad tym jak wpływa ono na kolejne etapy wykonywania algorytmu. Przekładając to na problem komiwojażera- za każdym razem sprawdzamy, które miasto jest aktualnie najbliżej (co właśnie robi powyższy kod) i tam się udajemy. W rezultacie otrzymujemy zazwyczaj wynik niewiele gorszy od optymalnego.

Podsumowanie

W pierwszej części ćwiczenia sprawdziłem jaka jest całkowita długość trasy w przypadku kiedy przechodzimy po wszystkich miastach po kolei tak jak są zapisane w pliku, zaczynając od miasta pierwszego. Uzyskany rezultat to w przybliżeniu 5084. To rozwiązanie rzeczywiście jest pewnym sposobem na znalezienie ścieżki, w której komiwojażer zagląda do każdego miasta dokładnie raz, a następnie wraca do miasta, z którego wyruszył. Niemniej jednak losowy wybór kolejności odwiedzanych punktów w wielu przypadkach skutkuje otrzymaniem całościowego kosztu trasy znacząco przewyższającego koszt w rozwiązaniu optymalnym.

Najważniejszą cechą problemu komiwojażera jest to, że w celu znalezienia rozwiązania optymalnego należy obliczyć wszystkie możliwe rozwiązania. Jest to zatem problem NP zupełny. Nie jesteśmy więc w stanie zaimplementować takiego algorytmu, który zwróciłby optymalne rozwiązania w stosunkowo niedługim czasie. Oprócz tego pojawia się także problem ograniczenia pamięci, którą dysponuje program. Ze względu na to poszukujemy rozwiązań, które dają rezultaty możliwie zbliżone do optymalnych przy jak najmniejszej złożoności czasowej i pamięciowej. Ja zdecydowałem się na algorytm zachłanny, mianowicie powtarzalny algorytm najbliższego sąsiada . Jest to tak naprawdę rozszerzona wersja algorytmu, który do znalezienia najlepszej trasy za każdym razem przechodzi do najbliższego wierzchołka. W rezultacie otrzymałem ścieżkę, której długość wynosi 894. Jest to rozwiązanie znacznie lepsze niż w przypadku losowego wyboru miast.

Warto tutaj omówić wydajność prezentowanego algorytmu. Ponieważ w każdej chwili możemy wybrać dowolny nieodwiedzony jeszcze wierzchołek mamy tutaj do czynienia z grafem pełnym. Zatem złożoność czasowa algorytm najbliższego sąsiada jest rzędu kwadratowego. Ale nie poprzestajemy na tym. W celu znalezienia najlepszego rozwiązania sprawdzamy długość trasy zaczynając za każdym razem w innym wierzchołku. To powoduje, że ostatecznie otrzymujemy złożoność czasową O(n^3). Jeśli chodzi o złożoność pamięciową to jest ona bardzo niewielka, ponieważ de facto zapamiętujemy jedynie wierzchołki, które zostały już odwiedzone.

Myślę, że najważniejszą rzeczą, której nauczyłem się w tym laboratorium jest to, że rozwiązanie optymalne nie zawsze jest rozwiązaniem najlepszym. Czasami z pewnych względów implementacja i stosowanie takiego rozwiązania może okazać się niezwykle kosztowne chociażby z powodu złożoności naszego problemu. Warto jest wtedy pomyśleć o innym rozwiązaniu, które niekoniecznie jest optymalne, ale jego użycie jest stosunkowo proste i przynosi znaczną oszczędność czasową.