Algorytmy Metaheurystyczne

Laboratorium1 - Problem komiwojażera

Autorzy: Zofia Stypułkowska, Bartosz Grelewski

Repozytorium GitHub: https://github.com/Sophie1227/Metaheurystyka1

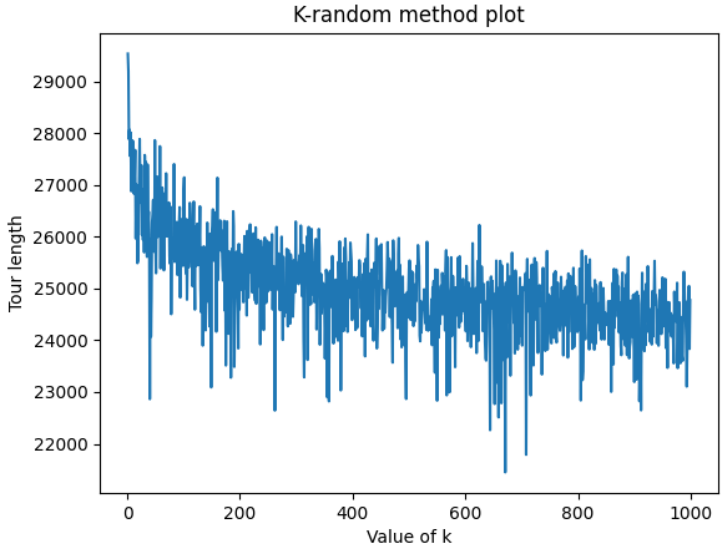
Rozważany Problem

Problem komiwojażera to jedno z najlepiej znanych zagadnień optymalizacyjnych polegające na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Innymi słowy mając zadany układ miast wraz z odległościami między nimi, sprzedawca musi wykonać kurs, odwiedzając każde z nich dokładnie raz i wracając do miasta wyjściowego. Warunkiem jest jednak pokonanie takiej trasy możliwie najmniejszym kosztem, czyli pokonując możliwie najmniejszą trasę. Największa trudność polega na dużej liczbie danych do analizy, a co za tym idzie czasie potrzebnym na znalezienie rozwiązania. W przypadku symetrycznym dla n miast liczba możliwych permutacji wynosi $\frac{(n-1)!}{2}$. Więc przykładowo dla danych z bays29 mamy $\frac{(29-1)!}{2} = 1,5 * 10^{29}$. Z powodu tak dużej liczby danych do sprawdzenia rozwiązując problem komiwojażera możemy to zrobić na 3 sposoby, skupiając się na najlepszym wyniku, najszybszym czasie pracy programu lub starając się znaleźć balans między nimi.

Wykorzystane Metody

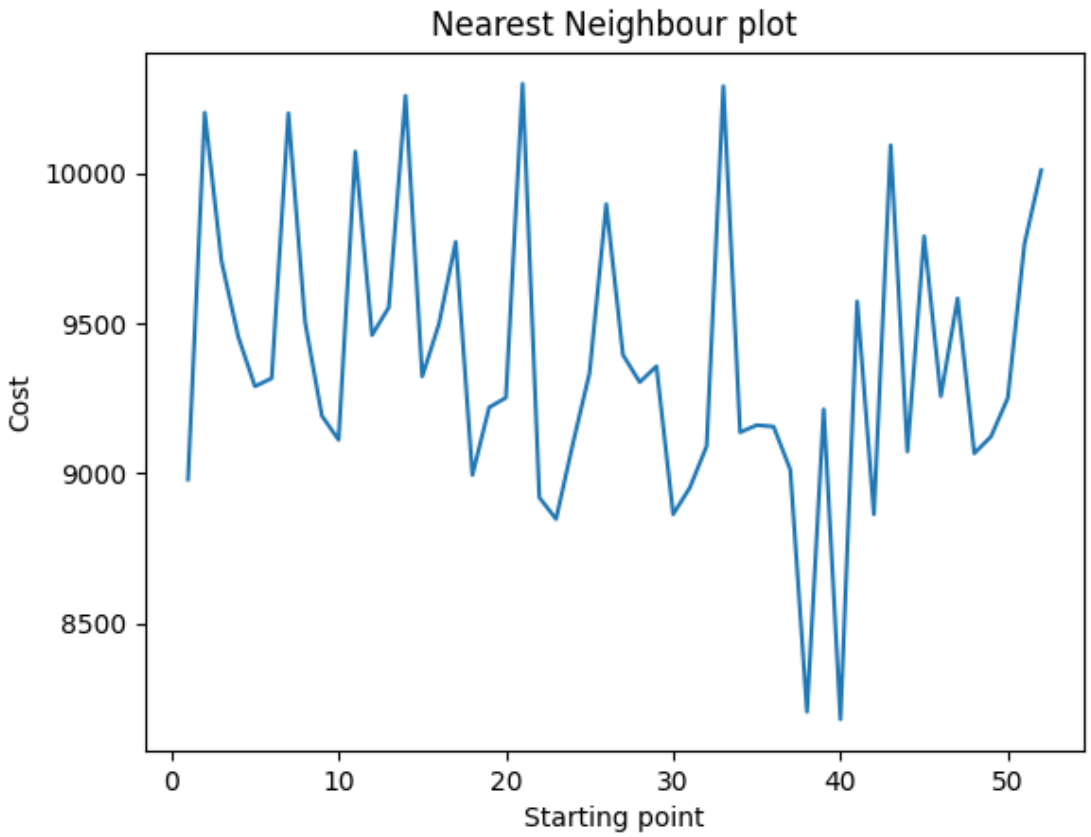
Metoda k-random

Generujemy k losowych dopuszczalnych rozwiązań i szukamy najlepszego z nich. Taka metoda choć prosta do implementacji ma jednak swoje wady. W podstawowej wersji takiego podejścia generujemy jedynie jedną permutacje co sprawia, że rozwiązanie, które otrzymujemy może być bardzo słabe, a nawet najgorsze z możliwych. Aby zredukować szanse na taką sytuację generujemy k możliwych rozwiązań zamiast jednego. Losowo próbkujemy zbiór rozwiązań i zwracamy najlepsze z nich. Teoretycznie, gdy k->∞ prawdopodobieństwo znalezienia optymalnego rozwiązania jest równe 1. Jest to jednak bardzo nieopłacalne pod względem czasu potrzebnego na otrzymanie wyniku. Metoda ta pozwala jednak na proste balansowanie pomiędzy czasem działania i jakością otrzymanego wyniku.



Metoda najbliższego sąsiada

Jest to algorytm zachłanny bazujący na założeniu, że lokalnie optymalne decyzje są również globalnie optymalne. Szybkość tego podejścia polega na małej ilości danych w każdym kroku decyzyjnym. Zamiast przeszukiwać wszystkie możliwe rozwiązania dla każdego punktu sprawdzamy odległości do pozostałych punktów w grafie. Więc dla każdego punktu wyjściowego mamy łącznie $\sum_{i=1}^{n-1} i$ rozwiązań do sprawdzenia. Zaczynamy od wybrania w dowolny sposób wierzchołka wyjściowego i przeszukujemy pozostałe, nieodwiedzone jeszcze wierzchołki w poszukiwaniu najbliższego do niego sąsiada. Robimy to w pętli, aż odwiedzimy wszystkie miasta. Wadą takiego podejścia jest zależność jakości rozwiązania od wyboru punktu wyjściowego, ponieważ po wyborze danego miasta „usuwamy" część krawędzi, które mogły być w kolejnych krokach lokalnie najlepsze.



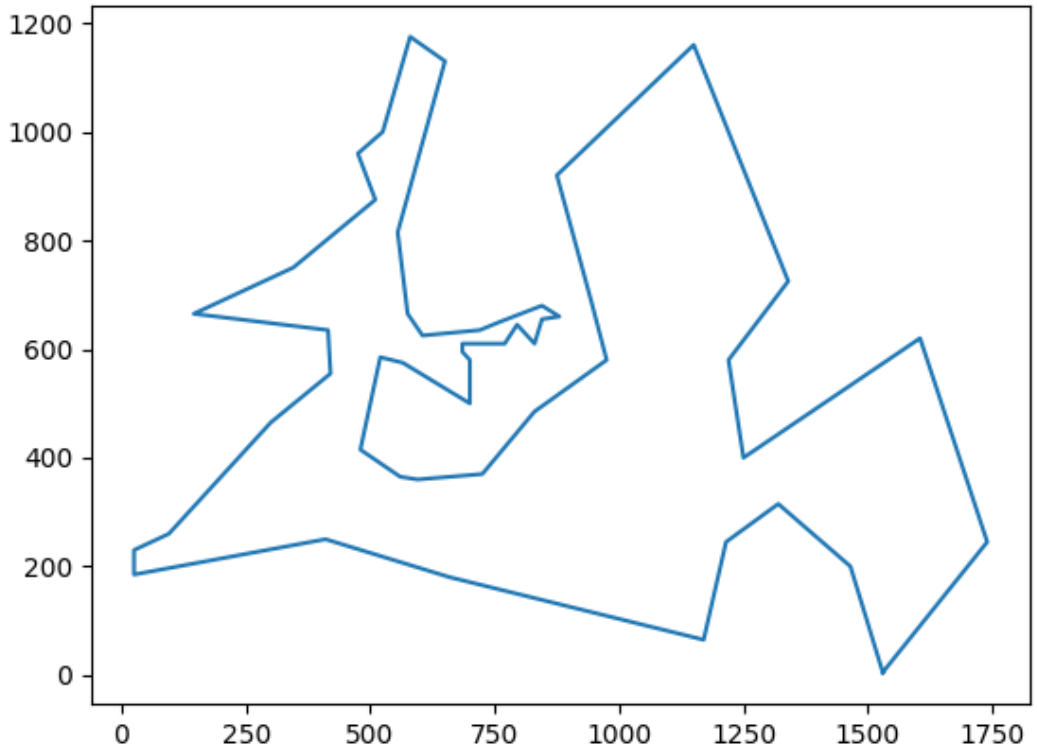
Rysunek 1- Zależność długości trasy od punktu wyjściowego dla berlin52

Rozszerzona metoda najbliższego sąsiada

Podejście to jest bardzo podobne do podstawowego algorytmu najbliższego sąsiada, usuwa jednak zależność od punktu wyjściowego i znajduje globalnie najlepsze rozwiązanie dla swojej podstawowej wersji. Nie usuwa jednak problemu związanego z brakiem dostępu do wykorzystanych już krawędzi, które mogły okazać się lokalnie lepsze.

Metoda 2-OPT

Jest to heurystyka dedykowana dla problemu komiwojażera, jednak przy drobnych zmianach może znaleźć zastosowanie dla innych zagadnień. Jest to algorytm należący do algorytmów popraw. Losujemy pierwsze rozwiązanie, a algorytm próbuje je poprawić. Poprzez lokalne przeszukiwanie tworzymy „sąsiadów" aktualnego rozwiązania i wybieramy najlepszego z nich. Sprawdzamy, czy jest ono lepsze od naszego aktualnego rozwiązania i jeśli jest to aktualizujemy najlepsze rozwiązanie i powtarzamy dla niego proces. Jeśli nie jest ono lepsze to kończymy pracę algorytmu. W naszym rozwiązaniu korzystamy z metody invert, czyli zamiany miejscami 2 wierzchołków. Pozwala to na „rozplątanie" nieoptymalnych pętli i poprawienie wylosowanego rozwiązania, często do poziomu otrzymania globalnie najlepszej trasy. Jest to jednak metoda należąca do NP-trudnych, czyli niedająca gwarancji znalezienia optymalnego rozwiązania w rozsądnym czasie wykonywania pętli algorytmu.



Rysunek 2- Wizualizacja przykładowej trasy dla rozwiązania 2-OPT z wykorzystaniem danych berlin52

Wnioski

Każdy z powyższych algorytmów skupia się na innych celach. Mimo, że usiłują one rozwiązać ten sam problem to każdy z nich ma inne priorytety dotyczące rozwiązania. Mimo, że teoretycznie algorytmy rozszerzonego najbliższego sąsiada i 2-OPT dają najlepsze rozwiązania, to są one niewydajne czasowo. Z kolei algorytm k-random jest szybki dla niskiej wartości k, jednak nie daje on dobrych wyników. W naszym programie policzyliśmy jednak również rozwiązania dla wyników metody k-ranom i najbliższego sąsiada po rozplątaniu przez 2-OPT. Dla losowej permutacji długość trasy wyniosła 27076, a po rozplataniu za pomocą algorytmu 2-OPT zmniejszyła się do 8111. Z kolei dla rozwiązania znalezionego metodą najbliższego sąsiada, wstępna trasa wyniosła 8980, a po rozplątaniu 7842.

Widzimy więc, że zastosowanie algorytmy 2-OPT nałożonego na algorytm najbliższego sąsiada daje nam najlepsze wyniki, ale jest on również dość wydajny czasowo. Jeśli jednak zależy nam szczególnie na czasie, to warto rozważyć zastosowanie czystej metody najbliższego sąsiada lub rozszerzonej metody najbliższego sąsiada, które dają stosunkowo dobre wyniki i mają mniejszą złożoność obliczeniową od 2-OTP.

LABORATORIUM 2 – ALGORYTMY LOKALNEGO POSZUKIWANIA

Wykorzystana metoda:

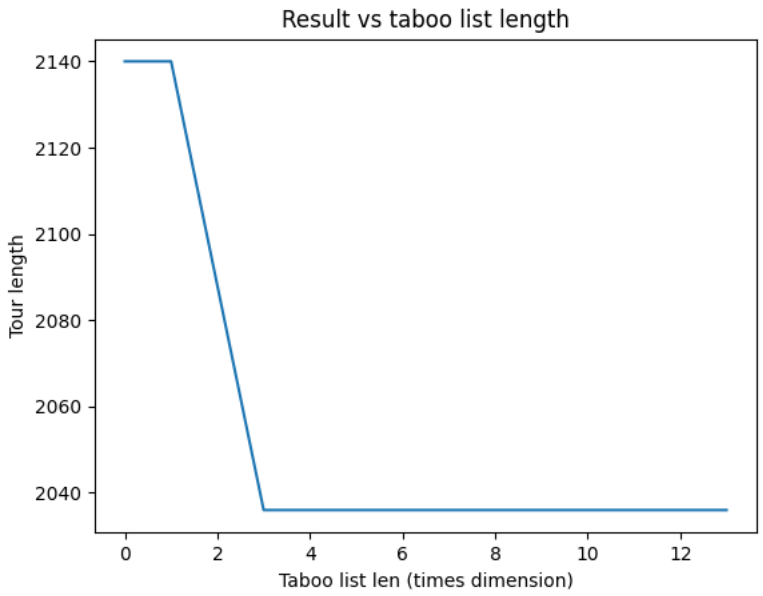
Algorytm tabu-search:

Opiera się on na przeszukiwaniu najbliższego otoczenia utworzonego za pomocą sekwencji ruchów. Istnieją jednak ruchy zakazane, zawarte w tak zwanej tabu liście. Ruchy, które pogarszają wynik są zakazane, jednak w drodze wyjątku mogą być dopuszczalne, jeśli nie znaleziono, żadnego lepszego rozwiązania. Podobnie jest z ruchami zakazanymi, ich lista zapobiega wpadaniu algorytmu w pętlę między na przykład 2 rozwiązaniami, ale za pomocą konkretnych warunków możemy pozwolić na ponowne wejście do tabu listy. Największą trudnością w implementacji takiego algorytmu jest znalezienie optymalnej długości tabu listy. W naszym wypadku stworzyliśmy test zwracający nam optymalną długość listy dla danej instancji i wynik ten wykorzystujemy w tworzeniu porównania wyniku w zależności od długości listy, gdzie pozostałymi próbkami jest długość równa 7 (wartość przedstawiona na wykładzie) oraz 3n (wartość znaleziona w artykule naukowym). Dodatkowo ograniczamy działanie naszego przeszukiwania maksymalnym czasem wykonywania. Aby dodatkowo zoptymalizować otrzymywane wyniki, jako instancję wejściową bierzemy wynik algorytmu 2-OPT oraz Rozszerzonego najbliższego sąsiada, które pisaliśmy w poprzedniej części laboratoriów.

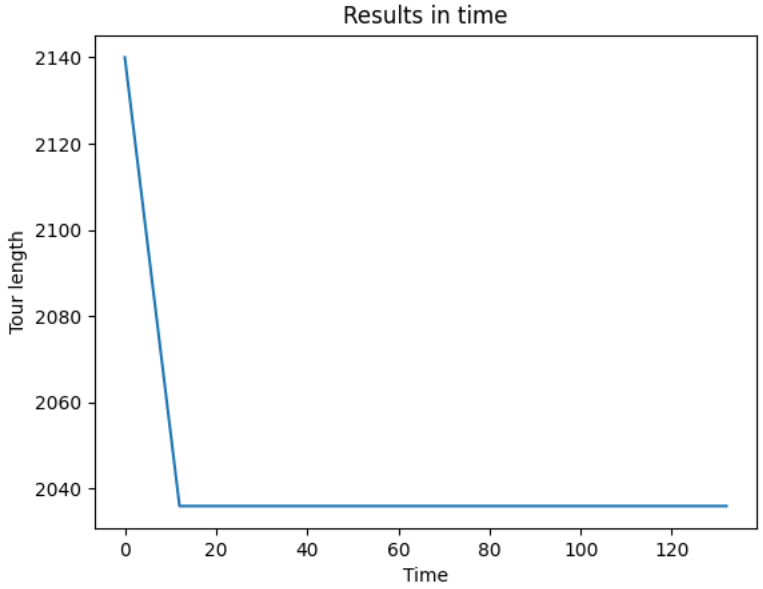
Otoczenie:

W naszym programie wykorzystujemy otoczenie typu INVERT. Jest to stosunkowo jedno z prostszych otoczeń do tego typu problemu. Aby otrzymać mniej efektywne ruchy moglibyśmy brać SWAP lub INSERT. Wykorzystane otoczenie wzięliśmy bezpośrednio z algorytmu TWO-OPT. W naszym kodzie można znaleźć zaimplementowane otoczenia SWAP/INSERT, które są przygotowane do dalszych badań jako osobne metody w tabuSearch.py.

Badania:



Wykres przedstawia trasę, którą nasz Taboo znajduje w zależności od długości listy, którą mu ustawimy. Przy wyższej wartości osi poziomej, czyli długości listy algorytm lepiej sobie radzi ze znajdowaniem lepszych rozwiązań. Badane na instancji FULL_MATRIX – Bays29.



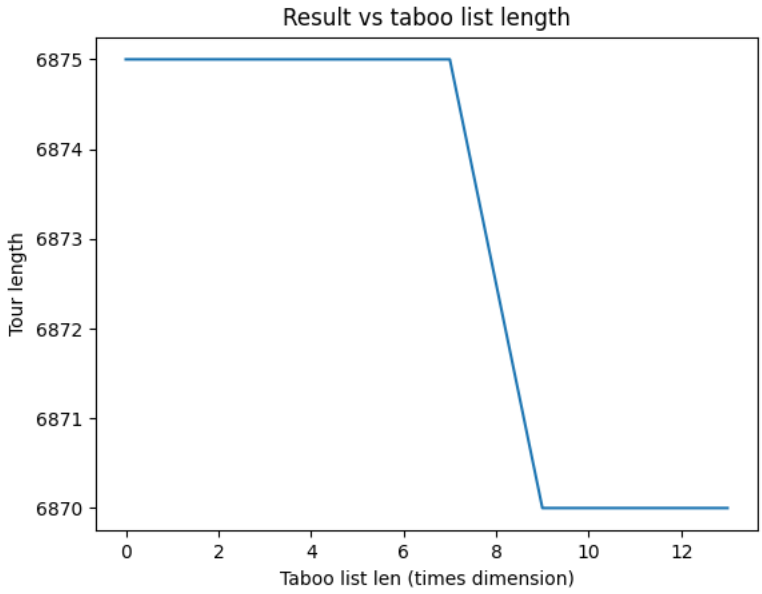
Kolejne przeprowadzone badanie dotyczyło znajdowania dobrego rozwiązania w zależności od podanego czasu. W tym wypadku algorytm bardzo optymistycznie wyznaczył pierwszy punkt startowy znajdujący się na liście rozwiązań. Jest to zaleta użycia algorytmu two_opt zintegrowanego z Nearest Neighbour. Gwarantuje nam to policzenie i wyznaczenie trasy znajdującej się na liście tras danej instancji.

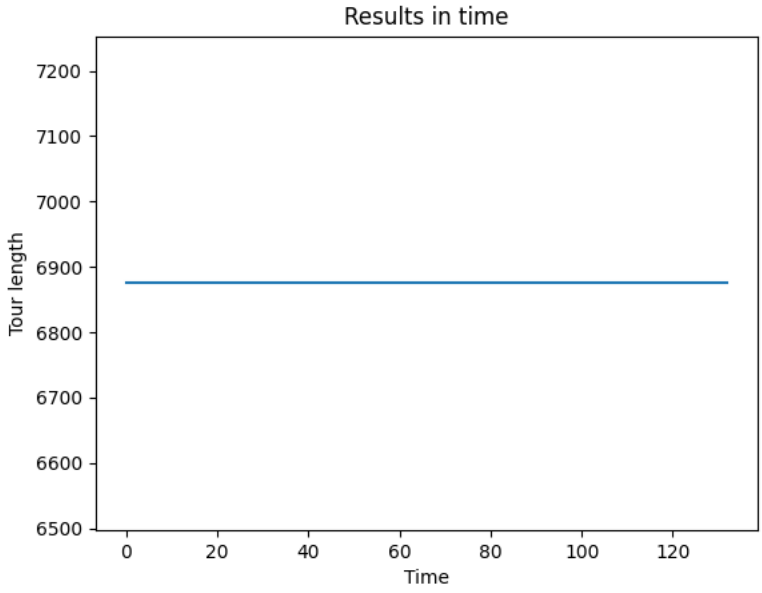
```
Średnia długość listy ze zmianą wyniku:
2
Średnia długość lisy bez zmiany wyniku:
1.0
2
```

Następnym przeprowadzonym eksperymentem jest zagłębienie się w zależność długości listy. Badamy czy wpływ na znajdywanie rozwiązań ma powiększanie listy. Robimy testy polegające na puszczeniu w pętli podwójnej pierwszego wyniku z two_opt sprzężonego z Nearest Neighbour i uzależniamy długość listy od i. Parametr pętli zewnętrznej to 20 iteracji po 30 iteracji, tabuSearch działa po 10 sekund na każde wywołanie. Zliczamy ilość poprawnych rozwiązań tzn. jeśli po wyznaczeniu punktu startowego kolejny będzie pozytywny (mniejszy) od poprzedniego to znaczy, że inkrementujemy zmienną odpowiedzialną za zliczanie dobrych wyników. W przeciwnym wypadku inkrementujemy zmienną podliczającą takie same rozwiązania. Pod koniec wykonujemy operację liczenia średniej uzyskanych pozytywnych wyników przez pozytywne iteracje.

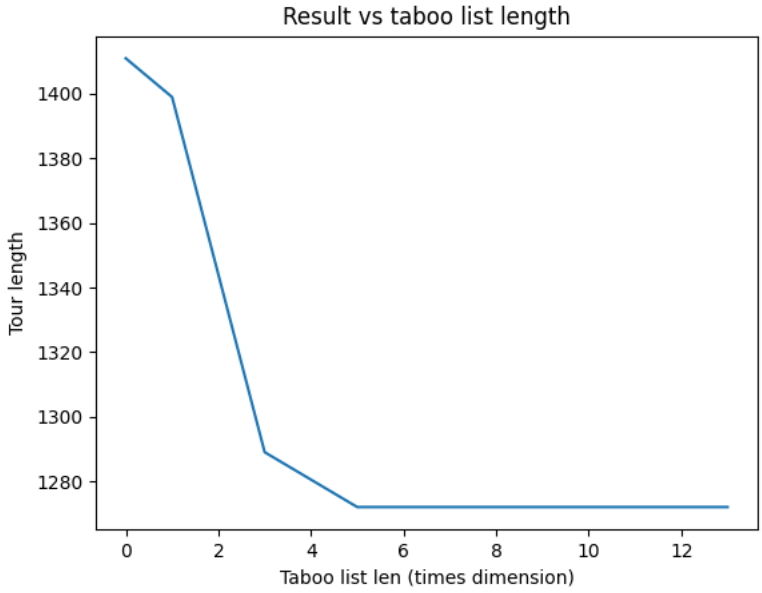
Różne instancje:

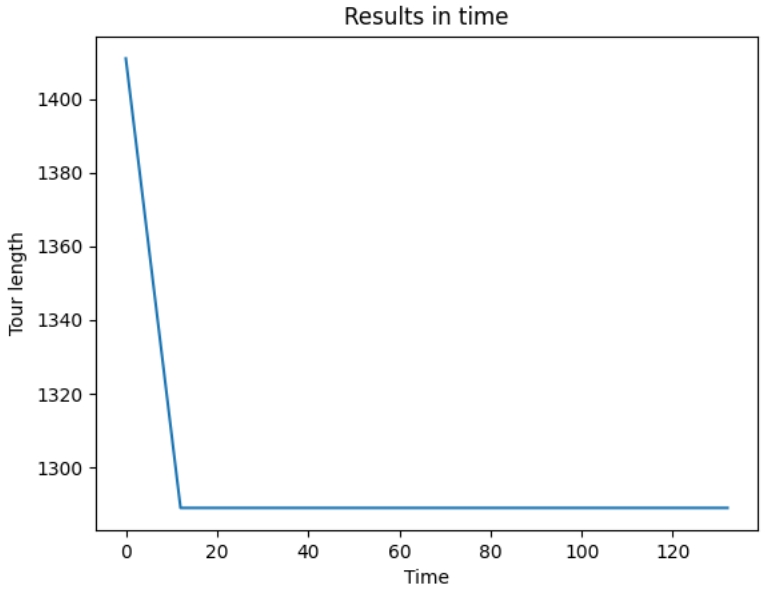
Ulysses16



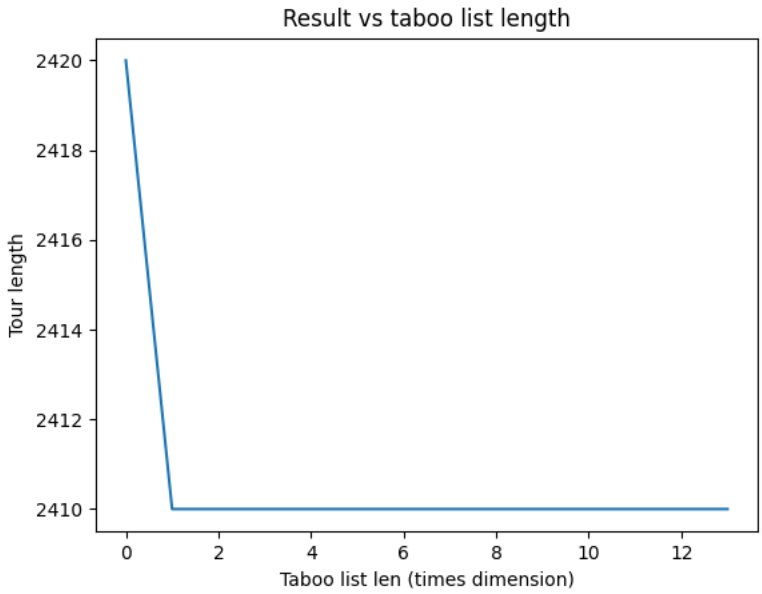

Results in time
Tour length
Time
7200
7100
7000
6900
6800
6700
6600
6500
0
20
40
60
80
100
120

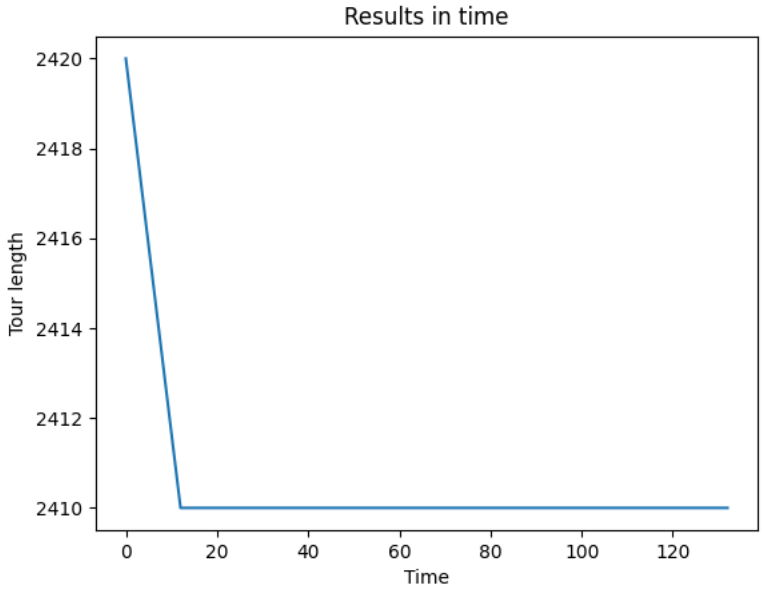
Gr24


Result vs taboo list length
Tour length
Taboo list len (times dimension)
1400
1380
1360
1340
1320
1300
1280
0
2
4
6
8
10
12

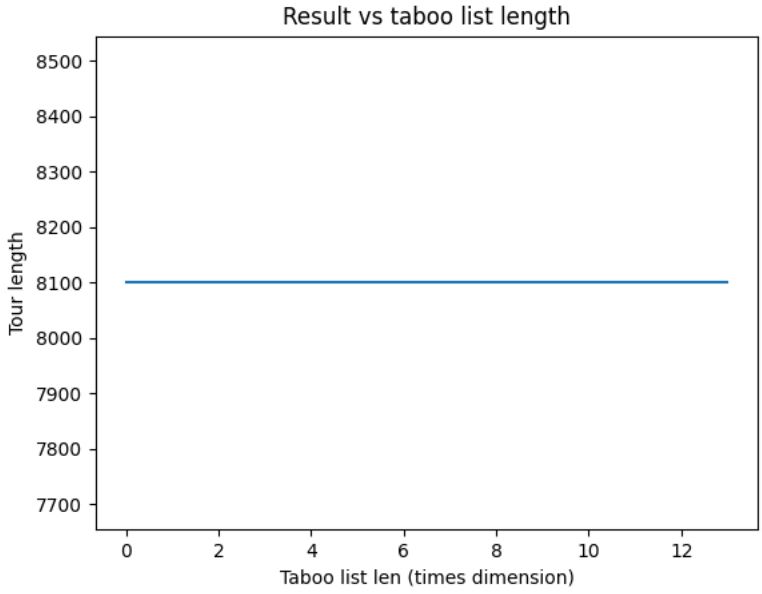

Results in time
Tour length
1400
1380
1360
1340
1320
1300
0
20
40
60
80
100
120
Time

Brg180


Result vs taboo list length
Tour length
2420
2418
2416
2414
2412
2410
0
2
4
6
8
10
12
Taboo list len (times dimension)


Results in time
Tour length
Time
2420
2418
2416
2414
2412
2410
0
20
40
60
80
100
120

Berlin52


Result vs taboo list length
Tour length
Taboo list len (times dimension)
8500
8400
8300
8200
8100
8000
7900
7800
7700
0
2
4
6
8
10
12