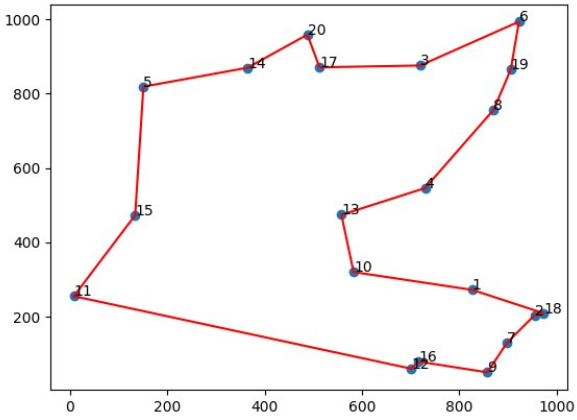
ALGORYTMY METAHEURYSTYCZNE- CZEŚĆ I: HEURYSTYKI

Mateusz Tofil, Mateusz Chęciński

Problem komiwojażera: znalezienie najkrótszego cyklu Hamiltona w zadanym grafie, tj. najktórszej ścieżka przechodzącej dokładnie raz przez każdy wierzchołek (rozwiązanie to permutacja)
Język programowania: Python

1. Wczytywanie macierzy: full (pełna), lower_diag (dolna przekątniowa), euc_2d (punkty na płaszczyźnie) z biblioteki tsplib oraz generowanie ich losowych instancji.
2. Wypisywanie instancji problemu oraz rozwiązania (cyklu).
Przykład (dla 20 punktów na płaszczyźnie)

```
mateusz@thinkpad:~/Studia/Rok 3/Semestr 2/Algorytmy metaheurystyczne/Metaheuristic/l1$ python3 1.py
[0]
[144, 0]
[613, 711, 0]
[290, 409, 330, 0]
[869, 1012, 571, 640, 0]
[728, 790, 234, 487, 791, 0]
[159, 94, 767, 449, 1016, 865, 0]
[486, 559, 192, 253, 722, 242, 628, 0]
[224, 182, 837, 512, 1044, 946, 88, 707, 0]
[248, 389, 572, 269, 660, 754, 368, 523, 385, 0]
[819, 948, 944, 778, 581, 1176, 898, 998, 874, 578, 0]
[246, 291, 816, 486, 937, 960, 208, 717, 157, 285, 719, 0]
[336, 480, 433, 186, 533, 635, 484, 421, 519, 156, 591, 438, 0]
[755, 889, 355, 488, 219, 571, 912, 518, 956, 591, 711, 876, 440, 0]
[721, 863, 711, 600, 346, 945, 837, 789, 838, 474, 251, 701, 424, 459, 0]
[222, 268, 797, 467, 931, 937, 187, 695, 143, 275, 730, 24, 425, 865, 703, 0]
[676, 800, 207, 390, 365, 428, 835, 375, 890, 555, 796, 832, 399, 148, 549, 817, 0]
[159, 19, 713, 416, 1024, 786, 110, 557, 196, 406, 967, 311, 493, 898, 880, 288, 806, 0]
[598, 662, 186, 364, 756, 130, 736, 113, 816, 633, 1085, 830, 523, 541, 865, 808, 393, 659, 0]
[766, 887, 246, 478, 364, 436, 925, 433, 981, 646, 852, 923, 498, 151, 601, 909, 91, 893, 428, 0]
12 -> 16 -> 9 -> 7 -> 2 -> 18 -> 1 -> 10 -> 13 -> 4 -> 8 -> 19 -> 6 -> 3 -> 17 -> 20 -> 14 -> 5 -> 15 -> 11 -> 12
```



3. Liczenie wartości funkcji celu (dlugość cyklu) oraz PRD (błąd względny, nie zawsze znamy optymalne rozwiązanie, więc czasem za optymalne służy najlepsze ze znalezionych).
4. Algorytmy heurystyczne generujące rozwiązania:

K-RANDOM

Generujemy k (parametr) losowych rozwiązań i wybieramy to, którego wartość funkcji celu jest najmniejsza.
Złożoność czasowa: O(kn), zależna od wyboru k, gdy k stałe w stosunku do rozmiaru instacji:
O(n)- generowanie wektora permutacji, liczenie funkcji celu
Złożoność pamięciowa: O(n)- jedna permutacja

NEAREST NEIGHBOUR

Metoda zachłanna, zaczynając od wierzchołka startowego (parametr) idziemy do najbliższego sąsiada. Dodatkowo nie możemy korzystać z odwiedzonych już wierzchołków. Kończymy algorytm, gdy odwiedzimy wszystkie wierzchołki.
Złożoność czasowa: $O(n^2)+O(n)=O(n^2)$- przechodzimy po każdym wierzchołku i jego wszystkich sąsiadach + operacja usuwająca odwiedzone już wierzchołki (ustawiamy je na MAX_INT)
Złożoność pamięciowa: $O(n^2)+O(n)=O(n^2)$- macierz sąsiedztwa + permutacja

2-OPT

Zaczynamy od losowego rozwiązania początkowego. W każdej iteracji algorytmu generujemy otoczenie- odwrócenie losowego fragmentu permutacji, tworzymy ich m (parametr). Następnie jako rozwiązanie wybieramy najlepszy z nich i wykonujemy kolejną iterację. Jeśli nie znajdziemy lepszego, kończymy algorytm z aktualnym rozwiązaniem.
Złożoność czasowa: Ω(mn) - eksperymentalnie
Złożoność pamięciowa: O(mn), zależna od wyboru m, gdy m stałe w stosunku rozmiaru instancji: O(n)- otoczenie

Testy

Testy wykonywane są dla asymetrycznej instacji grafu. Pomiary wykonujemy dla różnych rozmiarów, dla każdego generujemy 10 powtórzeń i uśredniamy wynik. W niektórych statystykach bierzemy też pod uwagę wpływ parametru na algorytm.

- ZŁOŻONOŚĆ CZASOWA

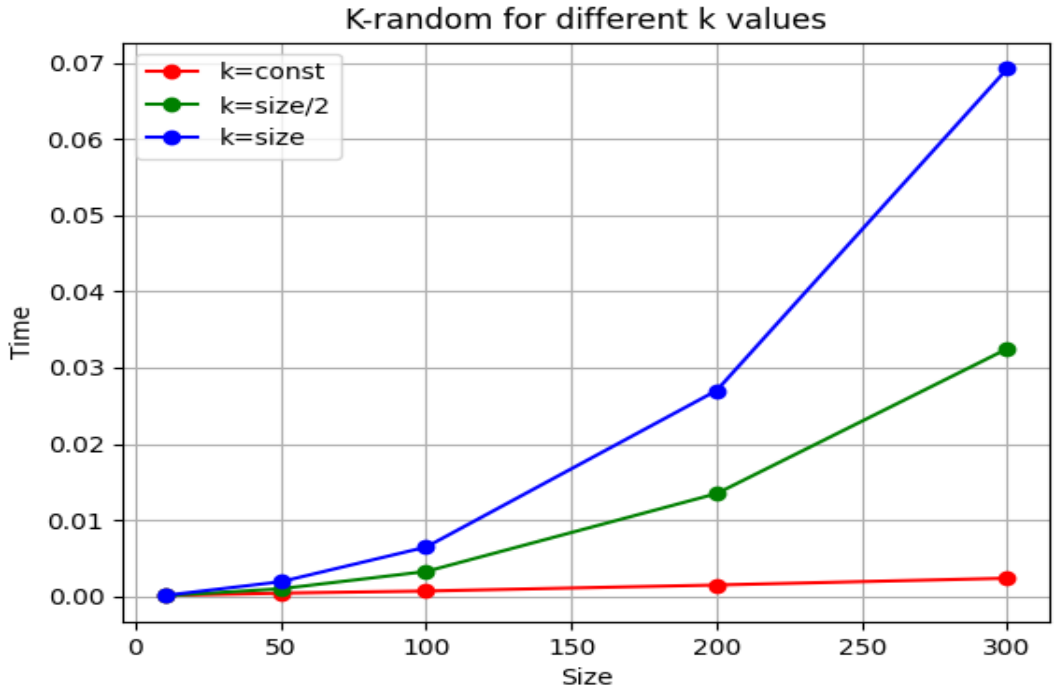


Figure 1: Złożoność czasowa dla róznych wartości k

Można zauważyć, że dobór parametru k ma wpływ na złożoność czasową. Normalnie jest ona liniowa, lecz przy k rzędu rozmiaru grafu przechodzi już w kwadratową. Powyższa sytuacja wzrostu złożoności zachodzi też dla 2-OPT. Z uwagi na ten wpływ na złożoność będziemy analizować czas dla ustalonych wartości parametrów.

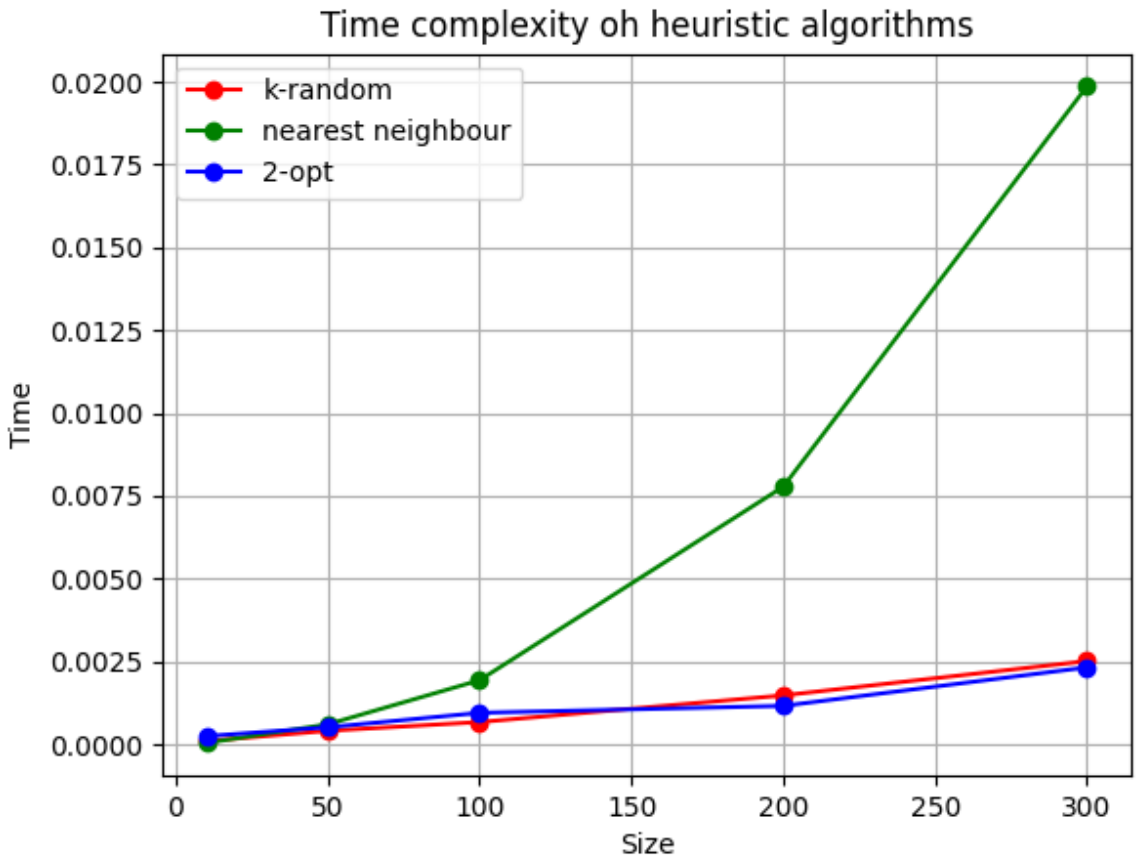


Figure 2: Złożoność obliczeniowa algorytmów dla stałego parametru

Widzimy, że zgodnie z przypuszczeniami algorytm nearest neighbour ma złożoność $O(n^2)$, pozostałe algorytmu opierają działanie na parametrze, który w tym przypadku jest stały, co daje ich złożoność równą O(n).

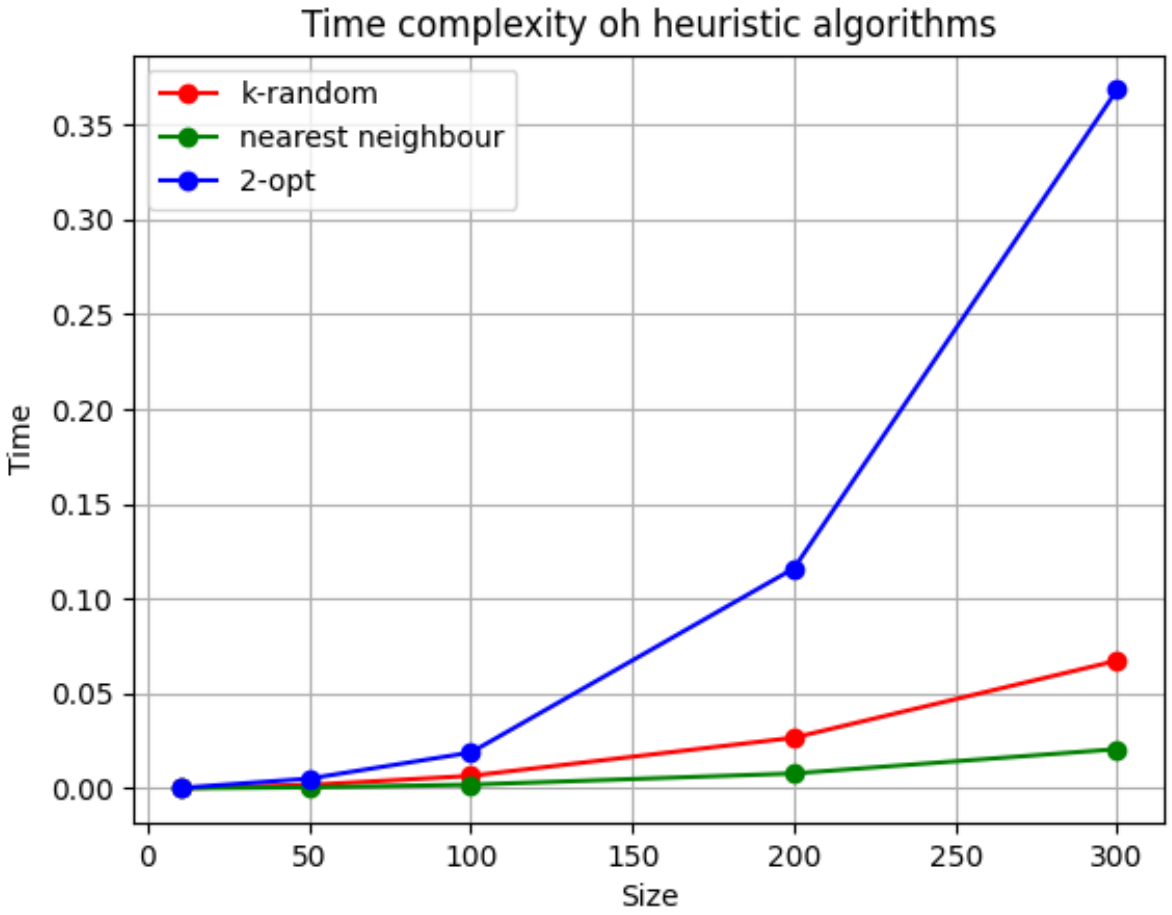


Figure 3: Złożoność algorytmów dla parametru rzędu rozmiaru grafu

Widzimy, że parametr ma wpływ: wzrosła złożoność k-random oraz 2-opt. Złożoność nearest neighbour pozostała niezmienna. K-random także ma teraz złożoność kwadratową. Ciekawą obserwacją jest, że mimo, że dla stałego parametru k-random i 2-opt miały złożoności liniowe, a teraz czas wykonania nearest neigbour znacząco wzrósł w stosunku do k-random. Pozwala to przypuszczać, że złożoność algorytmu 2-opt jest ograniczona jedynie z dołu przez $n \cdot m$ i tak naprawdę dobór parametru ma większy wpływ na czas algorytmu.

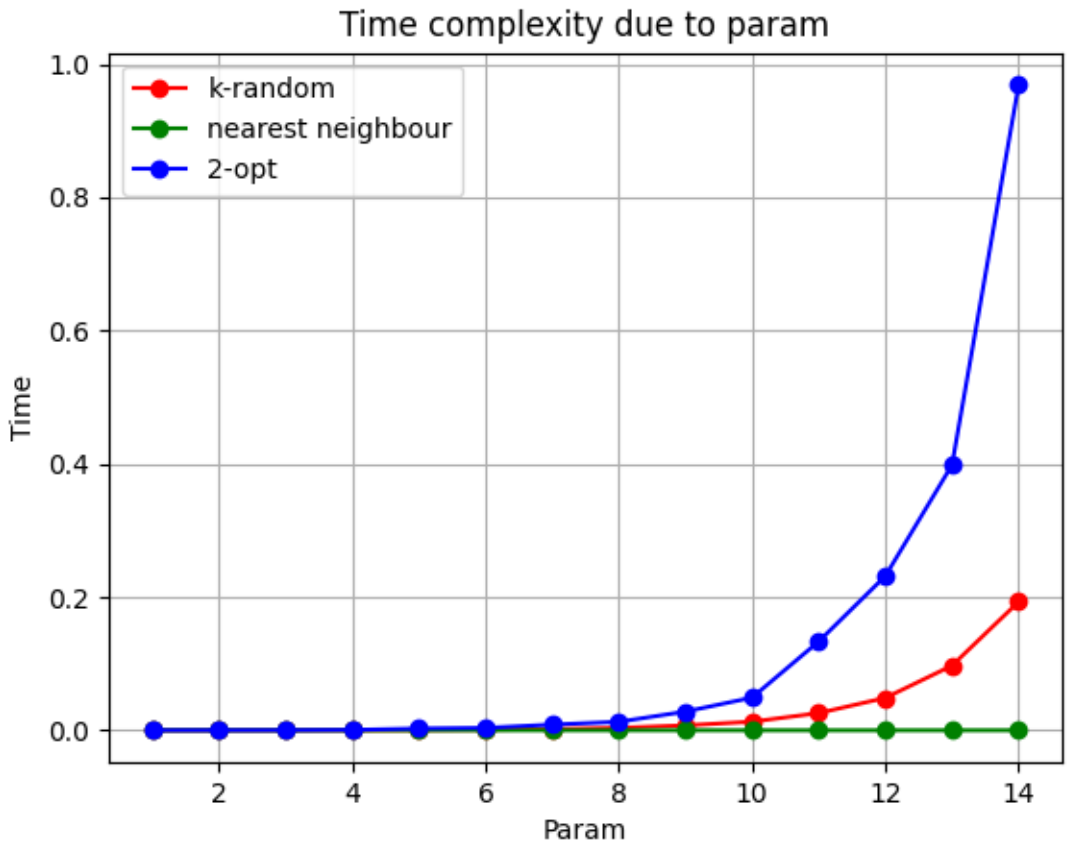


Figure 4: Złożoność czasowa algorytmów w zależności od parametru

Graf bierzemy z przykładu “br17.atsp”, param to w tym przypadku 2^{param}. Widzimy, że parametr nie ma żadnego wpływu na nearest neighbour, jednak dla k-random daje spory wzrost złożoności przy tych samych danych, a dla 2-opt jeszcze większy.

- ZŁOŻONOŚĆ PAMIĘCIOWA

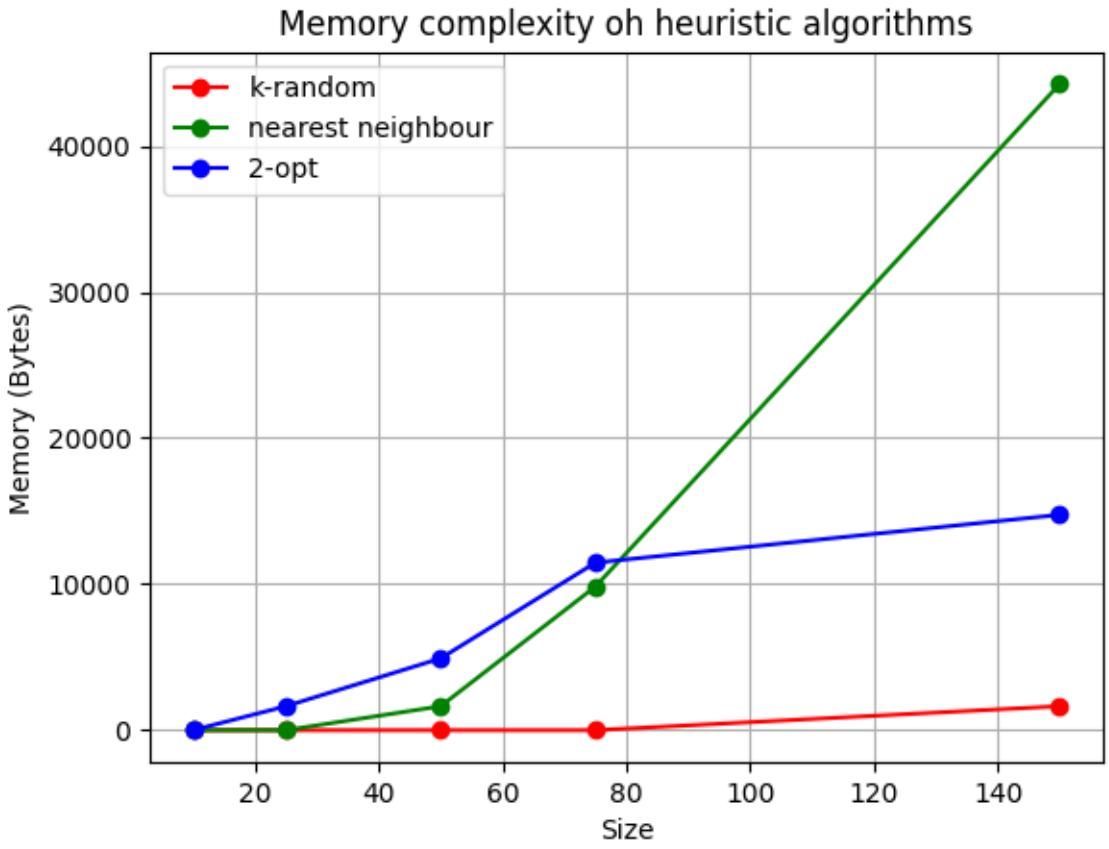


Figure 5: Złożoność pamięciowa algorytmów

Widzimy, że największą złożoność pamięciową ma nearest neighbour- $O(n^2)$. Wynika to z faktu, że potrzebujemy do jego obliczeń cała macierz sąsiedztwa (kopiujemy docelową i usuwamy z niej odwiedzone już wierzchołki, oszczędza to złożoność czasową, ale zwięszka pamięciową). Dla algorytmu k-random pamiętamy jedynie wektor permutacji. Dla 2-opt pamiętamy cała listę wektorów permutacji (jej rozmiar jest zależny od parametru, ale w tym przypadku jest stały). Powoduje to, że te algorytmu mają złożoność O(n).

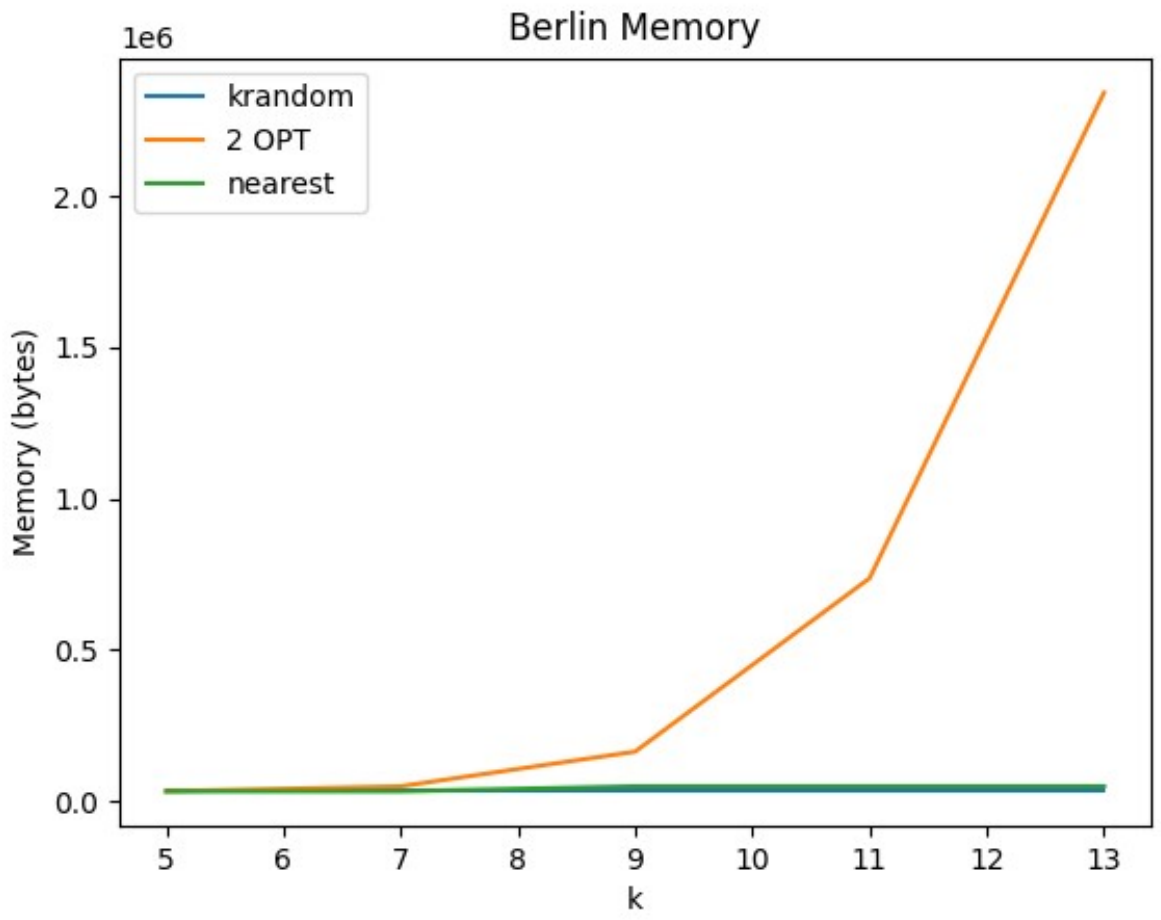


Figure 6: Złożoność pamięciowa algorytmów w zależności od parametrów

Graf bierzemy z przykładu “br17.atsp”, k to w tym przypadku 2^k. Widzimy, że parametr nie ma żadnego wpływu na nearest neighbour i k-random. Ich zużycie pamięci jest od niego niezależne. Dla 2-opt następuje spory wzrost złożoności pamięciowej przy wzroście k. Wynika to z faktu pamiętania całej listy wektorów permutacji.

- BŁĄD WZGLĘDNY

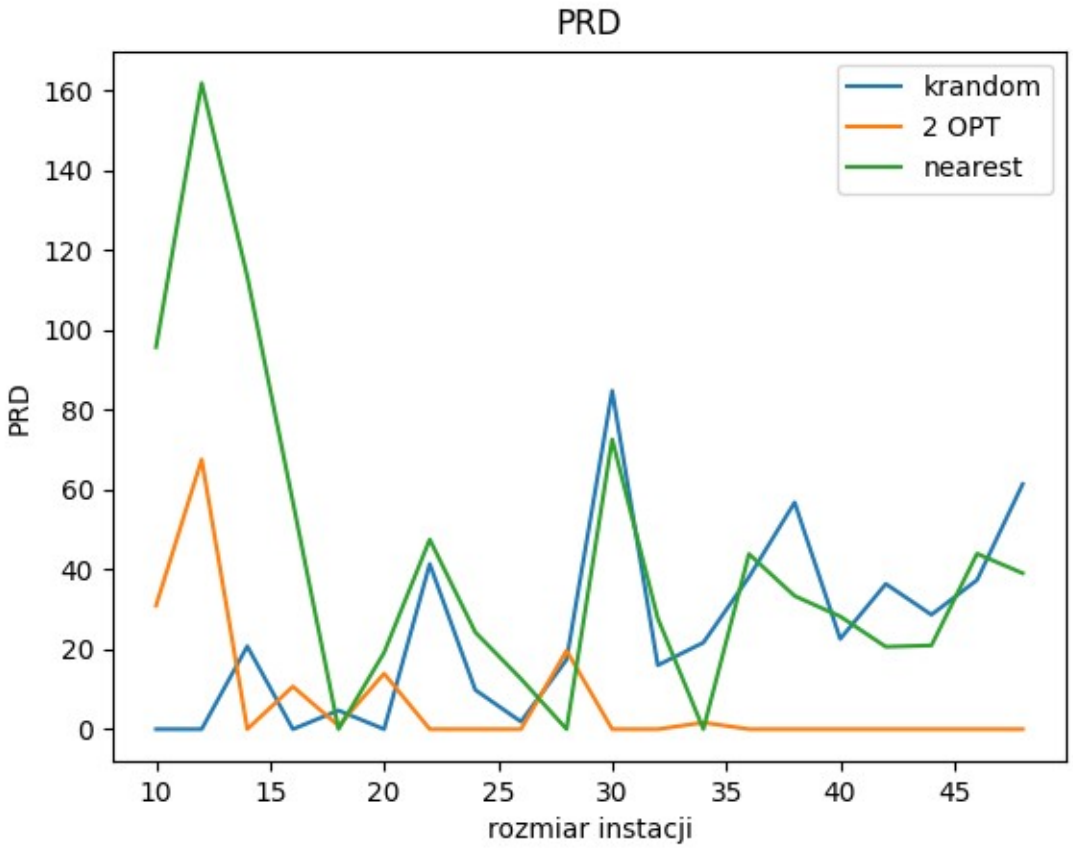


Figure 7: Błąd względny dla małych rozmiarów instacji

W tym przypadku nie znamy rozwiązanie optymalnego. Otrzymujemy je, więc wykonując 3 algorytmy i biorąc najlepszy z wyników. Wynika z tego, że w wiekszości przypadków 2-opt dawał najlepszy rezultat, gdyż jego błędy względne są zdecydowanie najmniejsze.

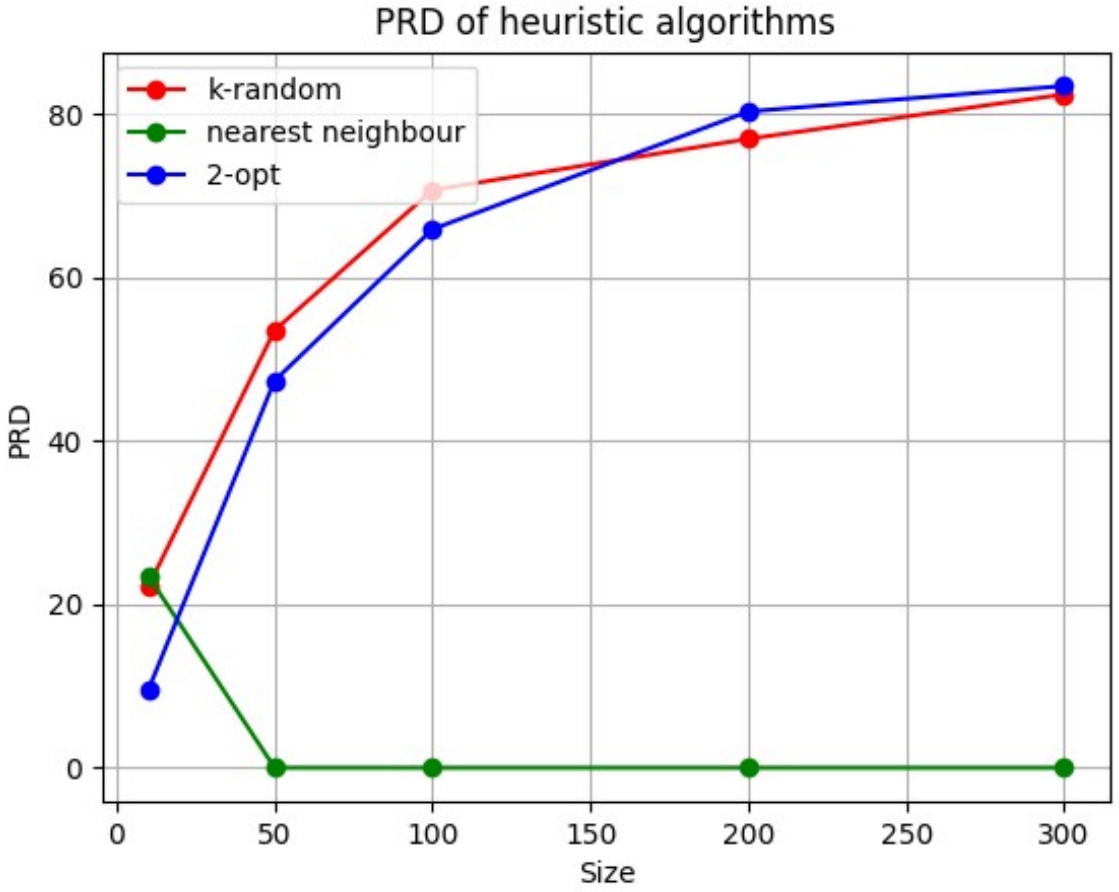


Figure 8: Błędy względne dla małej wartości parametru

W tym przypadku najlepiej działa nearest neighbour. Wynika to z faktu, że parametr podany dla k-random oraz 2-opt jest mały, więc nie algorytmu nie wynajdują wielu alternatywnych rozwiązań.

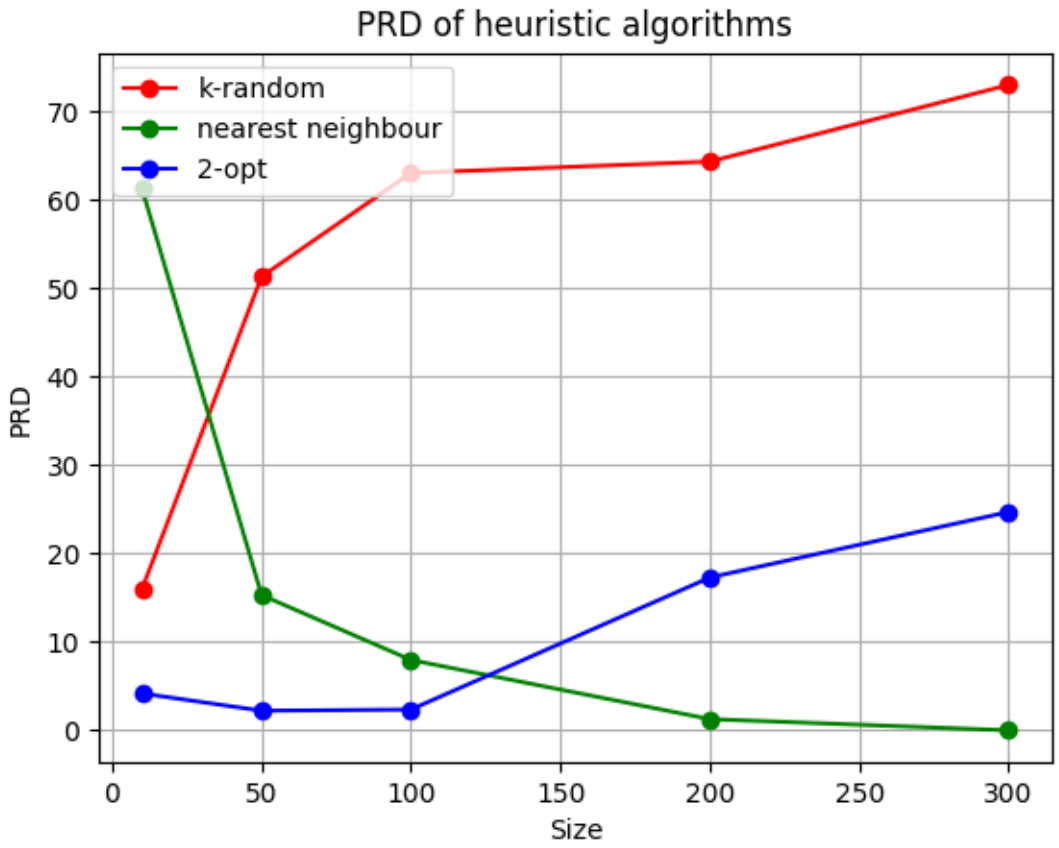
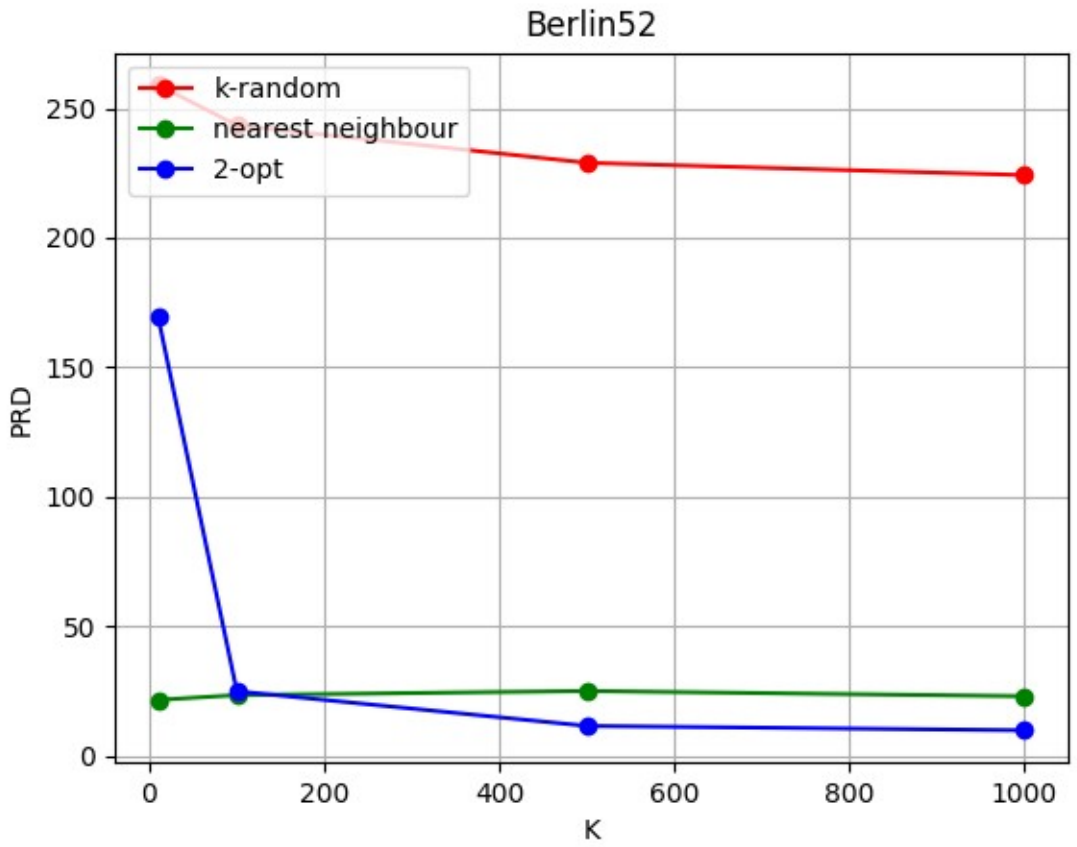
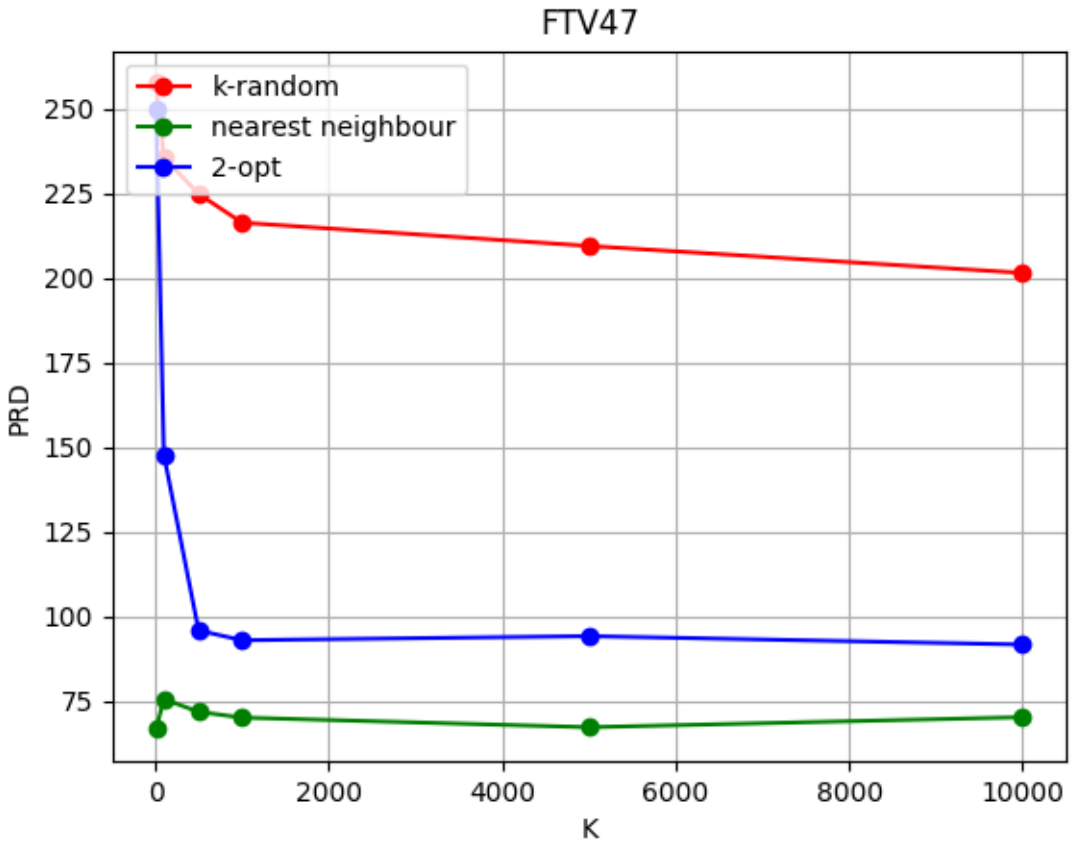


Figure 9: Błąd względny dla dużej wartości parametrów

Tutaj widzmy już, że coraz lepsze rozwiązania generuje już 2-opt. Wciąż najgorszy pozostaje algorytm k-random mimo wzrostu liczby przesukiwanych rozwiązań.



W tym przypadku korzystamy z instacji z TSPLIB. Znamy rozwiązanie optymalne, więc testujemy wpływ parametru na jakość algorytmu. Widzimy, że wraz ze wzrostem parametru algorytm 2-opt jest coraz lepszy. Nearest neighbour także generuje dość mały błąd, lecz w pewnym momencie zaczyna zwracać gorsze wyniki niż 2-opt. Dla k-randoma wzrost parametru ma niewielkie znaczenie, mimo poprawy wciąż generuje spore błędy.



Dla tej instancji problemu otrzymujemy najlepszą skuteczność algorytmu najbliższego sąsiada. Dla wzrostu parametru 2-opt na początku sporo się poprawia, lecz poźniej można zaobserwować praktycznie brak wpływu na działanie.

OGÓLNE WNIOSKI

Teoretycznie największą złożoność obliczeniową i pamięciową ma algorytm nearest neighbour. Warto jednak zauważyć wpływ wielkości parametru na złożoność dla algorytmów k-random i 2-opt. Wybór wielkości parametru jest istotny dla jakości rozwiązania. O ile dla k-random parametr nie ma aż tak dużego, to dla 2-opt znacznie wpływa na błędy względne. Należy więc się zastanowić, na ile możemy zwiększyć złożoność dla zwiększania dokładności rozwiązania. Ważne jednak, aby pamiętać, że otrzymanie rozwiązanie optymalnego jest praktycznie niemożliwe i są to jedynie heurystyki.