Optymalizacja efektywności lokalnego przeszukiwania

Autorzy sprawozdania

Filip Marciniak 148148, Szymon Pasternak 148146

Wstęp

W ramach zadania należało przeprowadzić badania dotyczące wykorzystania ocen ruchów z poprzednich iteracji i ruchów kandydackich w lokalnym przeszukiwaniu. Rozpatrywanym problemem jest zmodyfikowany problem komiwojażera. W problemie należy dla danego zbioru wierzchołków oraz symetrycznej macierzy odległości ułożyć dwa rozłączne cykle, z których każdy zawiera 50% wierzchołków. Celem jest minimalizacja łącznej długości obu cykli. W pracy rozważane są dwie instancje - `kroA200` oraz `kroB200` pochodzące z biblioteki `TSPLib`. Dla każdej z instancji zbadano:

- algorytm lokalnego przeszukiwania w wersji stromej,
- różne rodzaje algorytmów aptymalizacji lokalnego przeszukiwania: algorytm lokalnego przeszukiwania z listą ruchów przynoszących poprawę oraz ruchy kandydackie,
- heurystykę opartą o 2-żal z wagą żalu równą 0,4.

Łącznie uzyskano więc 4 różne wyniki dla każdej z instancji. Dodatkowo przeprowadzono również badania algorytmu z wykorzystaniem ruchów kandydackich, którego celem było porównanie wyników oraz czasów wykonywania algorytmu dla różnej liczby rozpatrywanych sąsiadów.

Kod programu

Kod programu dostępny jest w repozytorium: https://github.com/Johnybonny/IMO

Opis algorytmów

Każdy z algorytmów akceptuje na wejściu macierz odległości pomiędzy danymi wierzchołkami. Jako punkt odniesienia, użyty został algorytm 2-żal zaimplementowany w ramach pierwszego sprawozdania. Drugim punktem odniesienia był algorytm lokalnego przeszukiwania w wersji stromej zaimplementowany w ramach drugiego sprawozdania. Algorytm 2-żal został uruchomiony dla każdego wierzchołka startowego i najdalszego od niego wierzchołka jako startu drugiego cyklu. Każdy z pozostałych algorytmów uruchomiony został 100 razy z losowymi początkowymi rozwiązaniami. W algorytmach, w ruchach wewnątrztrasowych, rozpatrywany był rodzaj sąsiedztwa polegający na wymianie krawędzi w cyklu, a w ruchach międzytrasowych, ruch polegał na wymianie wierzchołków pomiędzy dwoma cyklami.

Algorytm wykorzystania ocen ruchów z poprzednich iteracji

W algorytmie korzystamy z wektora LM, który jest posortowaną od listą ruchów. Ruchy posortowane są według wartości delty. Im jest ona mniejsza, tym wcześniej ruch znajduje się w LM. W LM oprócz ruchu przechowywana jest jego delta, aby nie było potrzeby jej liczyć w każdej iteracji. Podczas dodawania ruchów związanych z wymianą krawędzi w cyklu, dodawane były również ruchy dla odwróconego względnego kierunku krawędzi.

```
wygeneruj losowe rozwiązanie startowe.
Zainicjuj listę LM i dodaj do niej możliwe ruchy przynoszące poprawę zachowując
odpowiednią kolejność.
powtarzaj
{
  dla każdego ruchu z LM
  {
    jeśli ruch jest międzytrasowy
    {
      jeśli ruch nie jest aplikowalny, czyli wierzchołki znajdują się w tym samym
cyklu, usuń go z LM.
      jeśli ruch jest aplikowalny, wykonaj go i usuń z LM.
    }

    jeśli ruch jest wewnątrztrasowy
    {
      jeśli ruch nie jest aplikowalny, czyli krawędzie nie występują w cyklu, usuń
go z LM.
      jeśli ruch jest aplikowalny, czyli krawędzie występują w cyklu w tym samym
kierunku, wykonaj go i usuń z LM.
      jeśli krawędzie występują w cyklu, ale w przeciwnych kierunkach, nie wykonuj
go i przejdź do kolejnej iteracji.
    }

    jeśli ruch został wykonany, to dodaj do LM nowe możliwe ruchy przynoszące
poprawę oraz ich deltę
    {
      jeśli wykonany ruch był międzytrasowy, to dodaj ruchy związane z wymianą
nowych wierzchołków oraz wymianą nowo powstałych krawędzi z wszystkimi innymi
krawędziami w cyklu.
      jeśli wykonany ruch był wewnątrztrasowy, to dodaj ruchy związane z wymianą
nowo powstałych krawędzi z wszystkimi innymi krawędziami w cyklu, w którym
wykonany był ruch.
    }
  }

  jeśli nie znaleziono żadnego aplikowalnego ruchu w LM, to zakończ pętlę.
}
zwróć rozwiązanie.
```

Algorytm ruchów kandydackich

```
wygeneruj losowe rozwiązanie startowe.
wygeneruj listę k najbliższych sąsiadów dla każdego punktu.
powtarzaj
{
  dla każdego punktu w zbiorze wszystkich punktów
  {
    dla każdego sąsiada z listy najbliższych sąsiadów
    {
```

```
      jeżeli punkt i sąsiad są w różnych cyklach
      {
        oblicz deltę wymiany wierzchołka przed punktem i sąsiada pomiędzy cyklami.
        jeżeli delta jest mniejsza od 0 oraz najmniejsza z dotychczasowych to
zapamiętaj ją oraz ruch.

        oblicz deltę wymiany wierzchołka za punktem i sąsiada pomiędzy cyklami.
        jeżeli delta jest mniejsza od 0 oraz najmniejsza z dotychczasowych to
zapamiętaj ją oraz ruch.

        oblicz deltę wymiany punktu i wierzchołka przed sąsiadem pomiędzy cyklami.
        jeżeli delta jest mniejsza od 0 oraz najmniejsza z dotychczasowych to
zapamiętaj ją oraz ruch.

        oblicz deltę wymiany punktu i wierzchołka za sąsiadem pomiędzy cyklami.
        jeżeli delta jest mniejsza od 0 oraz najmniejsza z dotychczasowych to
zapamiętaj ją oraz ruch.
      }

      jeżeli punkt i sąsiad są w tym samym cyklu
      {
        oblicz deltę wymiany krawędzi rozpoczynającej się od punktu oraz krawędzi
rozpoczynającej się od sąsiada.
        jeżeli delta jest mniejsza od 0 oraz najmniejsza z dotychczasowych to
zapamiętaj ją oraz ruch.

        oblicz deltę wymiany krawędzi kończącej się na punkcie oraz krawędzi
kończącej się na sąsiedzie.
        jeżeli delta jest mniejsza od 0 oraz najmniejsza z dotychczasowych to
zapamiętaj ją oraz ruch.
      }
    }
  }

  wykonaj zapamiętany ruch.
  jeżeli nie została znaleziona żadna ujemna delta, to zakończ pętlę.
}
zwróć rozwiązanie.
```

Wyniki

W tabeli przedstawione zostały najlepsze, średnie i najgorsze wyniki dla zbadanych algorytmów.

	kroA200			kroB200		
	min	mean	max	min	mean	max
2-regret	32518	36024	40288	34477	36917	38341
lokalne przeszukiwanie - strome	36566	38526	41002	35572	38744	40726

	kroA200			kroB200		
oceny poprzednich ruchów	37407	40924	45363	37369	40942	44831
ruchy kandydackie (k=10)	35589	38747	43351	37063	38905	40900

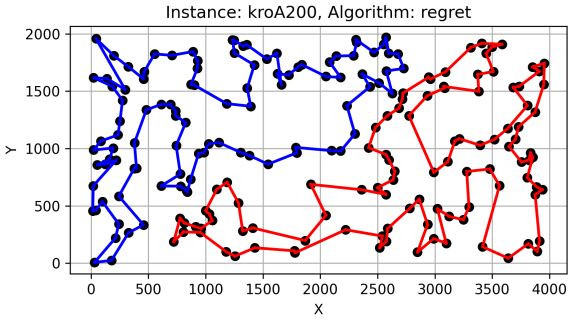
W tabeli poniżej przedstawione zostały czasy wykonania poszczególnych algorytmów w milisekundach.

	kroA200			kroB200		
	min	mean	max	min	mean	max
2-regret	131	145	208	133	158	300
lokalne przeszukiwanie - strome	693	890	1176	712	840	1170
oceny poprzednich ruchów	512	723	988	538	691	975
ruchy kandydackie (k=10)	407	510	731	419	545	722

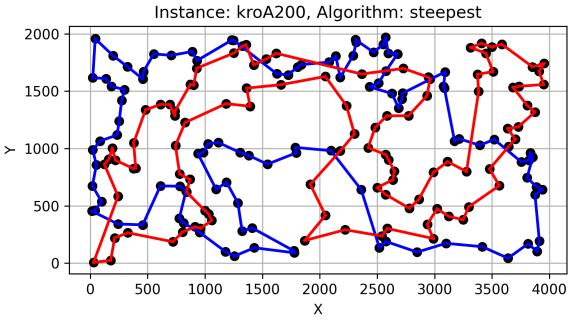
Poniżej umieszczone zostały wizualizacje najlepszych z uzyskanych wyników:

Wizualizacja kroA200.tsp

2-Regret



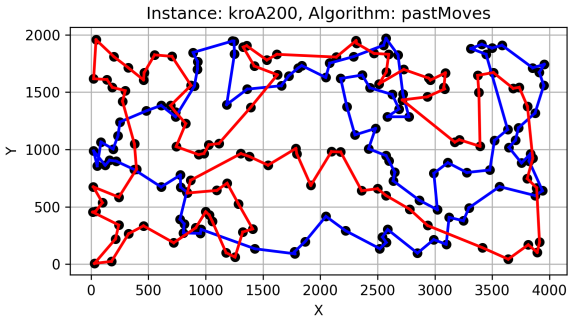
Lokalne przeszukiwanie - strome

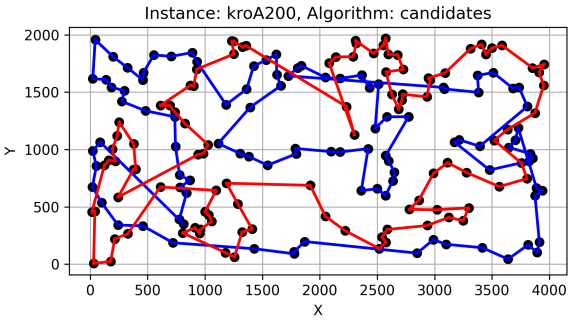


Oceny poprzednich ruchów

Ruchy kandydackie

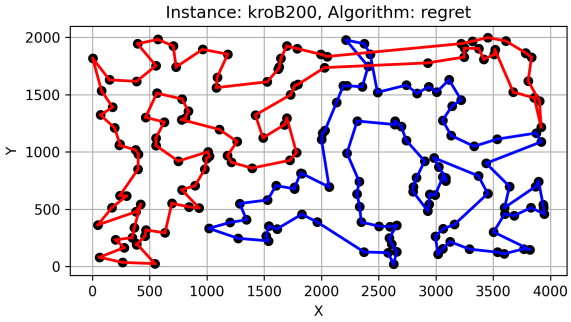
Oceny poprzednich ruchów | **Ruchy kandydackie**

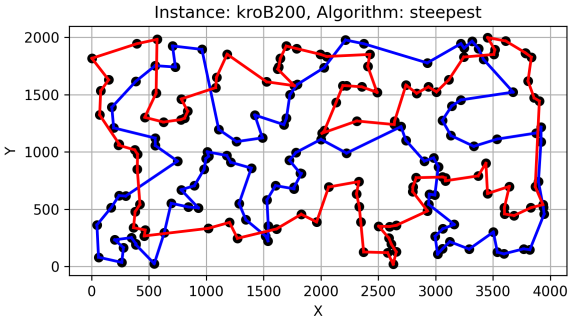




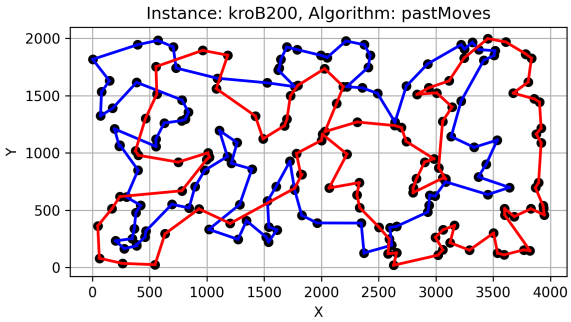
Wizualizacja kroB200.tsp

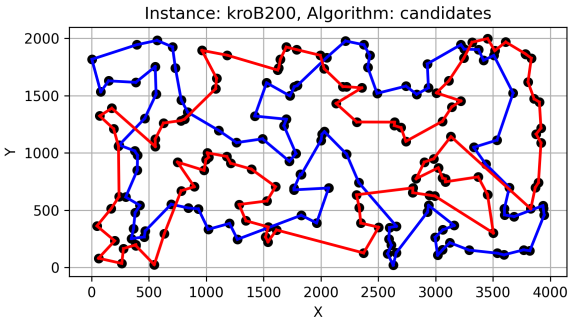
2-Regret | **Lokalne przeszukiwanie - strome**





Oceny poprzednich ruchów | **Ruchy kandydackie**

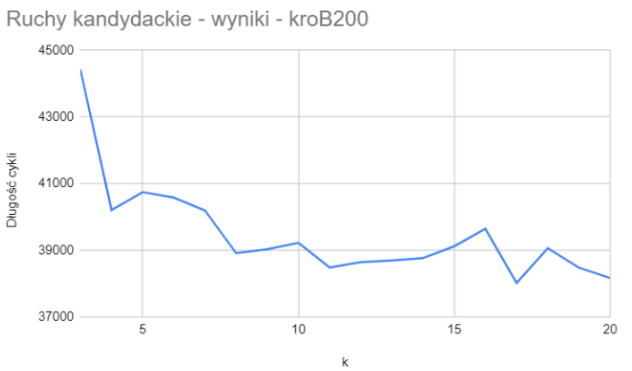




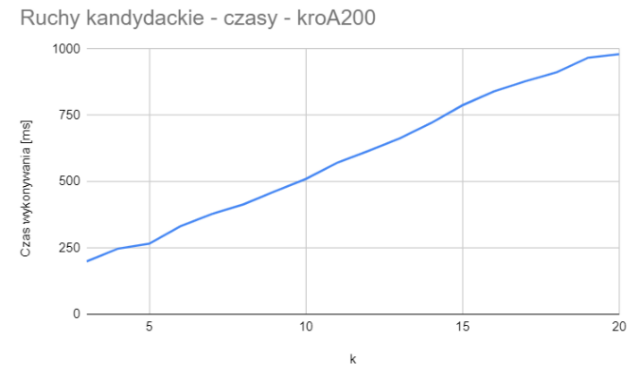
Badanie wpływu liczby sąsiadów na wynik oraz czas działania algorytmu ruchów kandydackich

Wyniki - kroA200

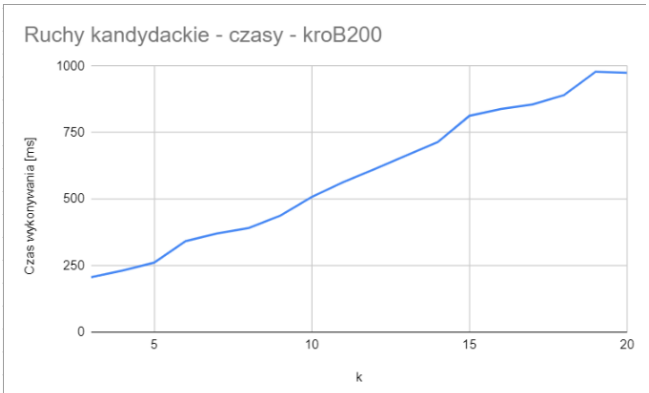
Wyniki - kroB200



Czasy - kroA200



Czasy - kroB200



Wnioski

1. Lokalne przeszukiwanie z zapamiętywaniem poprzednich ruchów jest efektywną metodą optymalizacji lokalnego przeszukiwania. W każdym przypadku udało się poprawić rozwiązanie wygenerowane jako startowe.
2. Algorytm ruchów kandydackich jest efektywną heurystyką poprawiającą rozwiązania startowe. W każdym przypadku udało się je poprawić.
3. Lokalne przeszukiwanie z zapamiętywaniem poprzednich ruchów radzi sobie gorzej niż przeszukiwanie strome, ale za to działa szybciej. Wadą tej metody jest duża trudność w implementacji algorytmu. Algorytm ruchów kandydackich radzi sobie nieco gorzej niż przeszukiwanie strome, ale lepiej niż lokalne przeszukiwanie z zapamiętywaniem poprzednich ruchów. Zaletą algorytmu jest jego prostota i możliwość dobrania wartości k oznaczającej liczbę rozpatrywanych sąsiadów. Najlepsze wyniki oraz czasy osiągała heurystyka konstrukcyjna oparta na 2-żalu.
4. Dobór wartości k w algorytmie wykorzystującym ruchy kandydackie znacząco wpływa na otrzymane wyniki oraz czas działania algorytmu. Dla k>8 można oczekiwać podobnych, dobrych wyników. Dalsze zwiększanie k może poprawić wynik, ale nie w tak znaczący sposób. Widocznie wydłuża się natomiast czas działania algoytmu. Zwiększa się on liniowo przy dobieraniu coraz większych wartości k.