Algorytmy ewolucyjne i metaheurystyczne

Marzec 2020

Testy globalnej wypukłości

1.1 Opis zadania

Zadanie polegało na przeprowadzeniu testów globalnej wypukłości. Należało wygenerować dla każdej instancji 1000 losowych optimów lokalnych, tj. rozwiązań uzyskanych z losowych rozwiązań startowych po zastosowaniu lokalnego przeszukiwania w wersji zachłannej. Następnie dla każdego z tych rozwiązań należało policzyć podobieństwo do najlepszego rozwiązania i średnie podobieństwo dla wszystkich pozostałych rozwiązań. Na osi x nanieśliśmy wartość funkcji celu, na y (średnie) podobieństwo. Policzyliśmy też wartość współczynnika korelacji.

Stosujemy dwie miary podobieństwa (oddzielnie):
- Liczba wspólnych wierzchołków wybranych do rozwiązania.
- Liczba wspólnych krawędzi.

1.2 Pseudokod

1.2.1 Porównanie wspólnych wierzchołków

Algorithm 1 Wspólne wierzchołki

1: Dla pary rozwiązań zlicz wszystkie wierzchołki, które występują w obu rozwiązaniach

1.2.2 Wspólne krawędzie

Algorithm 2 Wspólne krawędzie

1: Dla każdego rozwiązania z pary wygeneruj listę wszystkich krawędzi występujących w rozwiązaniu
2: Zlicz krawędzie występujące w obu rozwiązaniach, pamiętając o uwzględnieniu krawędzi z przeciwnym zwrotem.

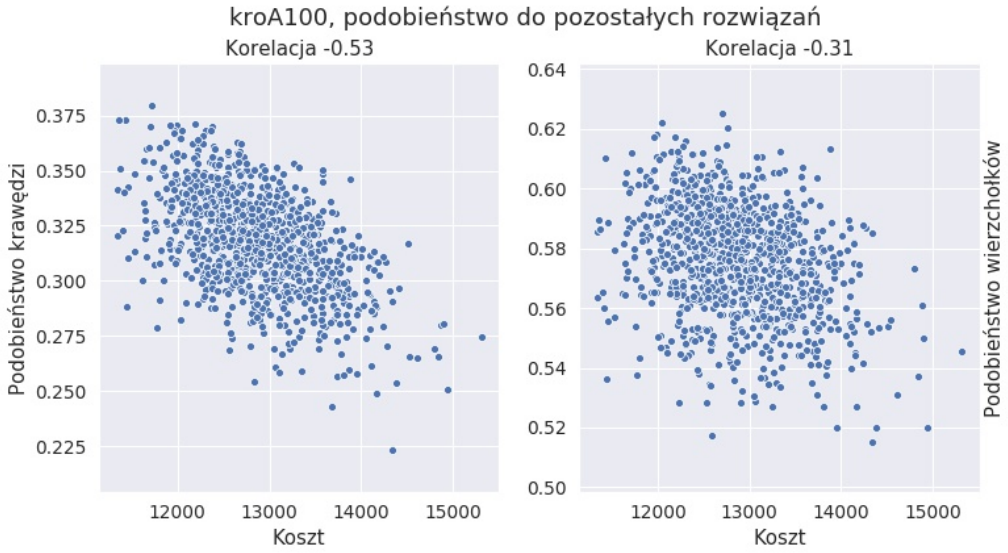
Porównianie do najlepszego rozwiązania i do wszystkich rozwiązań

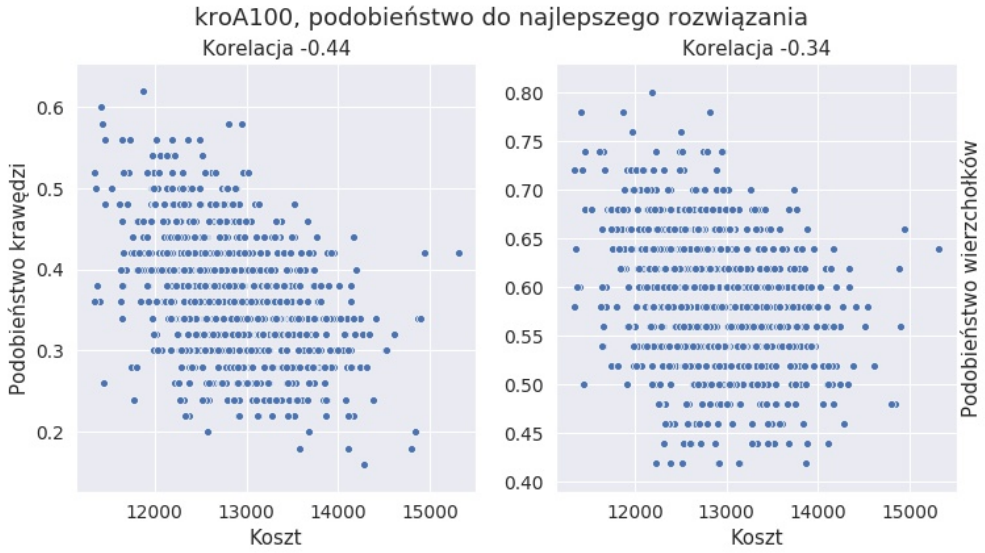
Wyżej przedstawione metody zastosowaliśmy do porównania wszystkich rozwiązań z najlepszym rozwiązaniem, tj. rozwiązaniem o najmniejszym koszcie, jak i do porównania rozwiązań każdy z każdym. W tym celu iterowaliśmy po wszystkich parach rozwiązań, sumując wyniki częściowe.

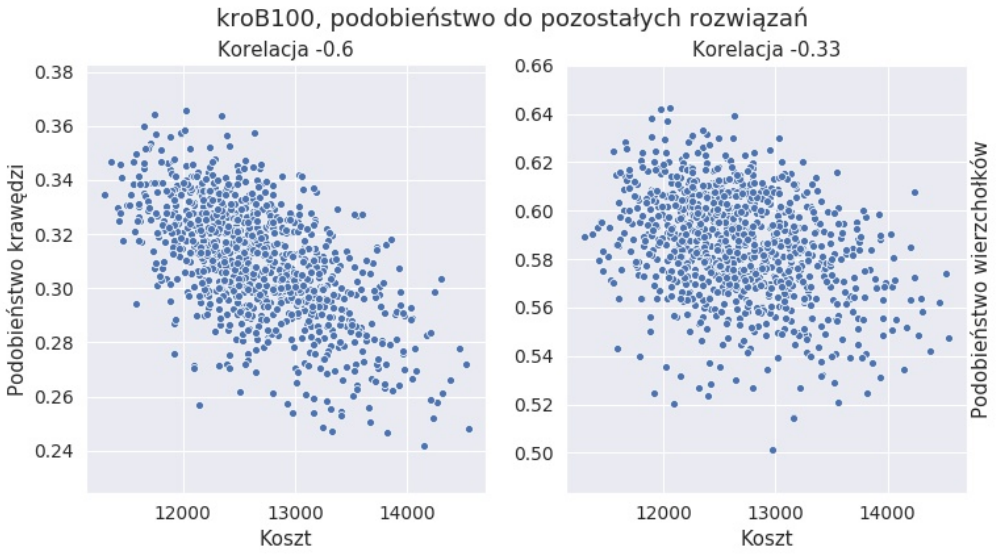
1.3 Wyniki eksperymentu obliczeniowego

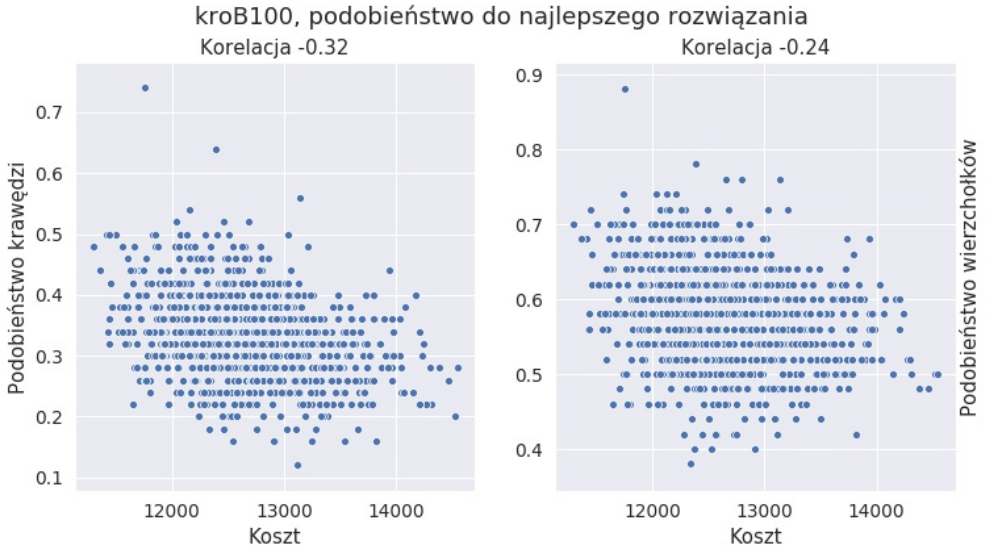
Testy przeprowadzone były na wszystkich instancjach, których używaliśmy do poprzednich zadań - kroA100, kroB100, kroA200, kroB200. Dla każdej instancji wygenerowaliśmy 1000 rozwiązań za pomocą algorytmu Local Search w wersji greedy.

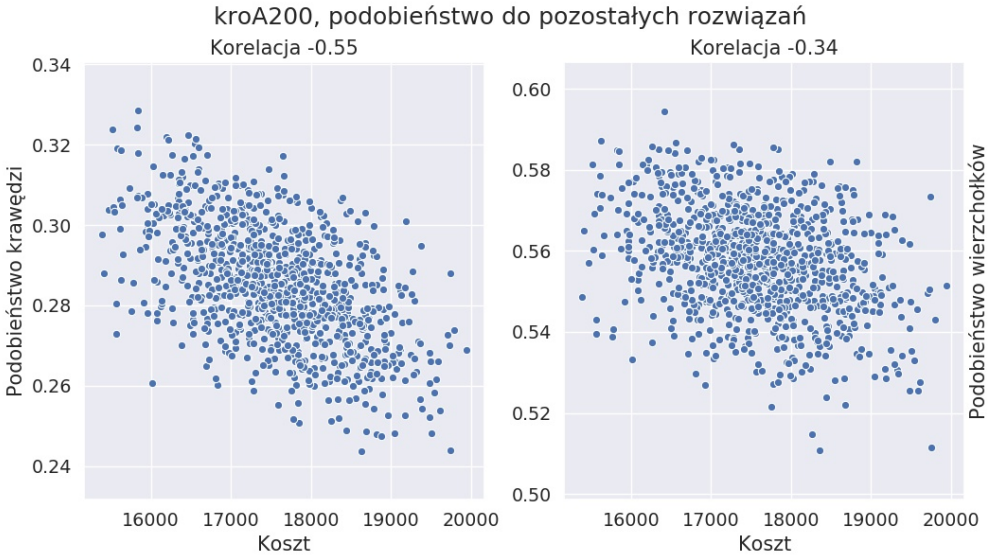
1.4 Wizualizacje

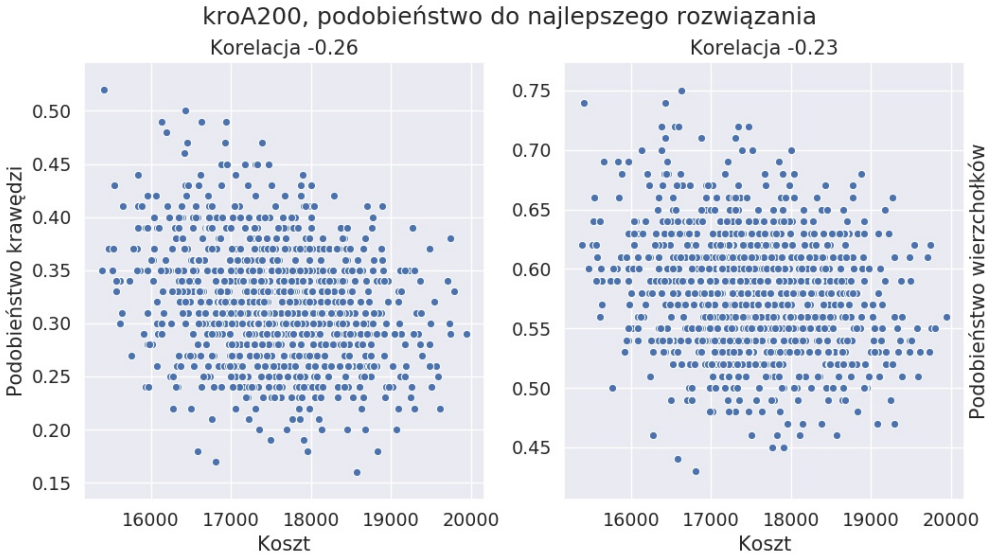


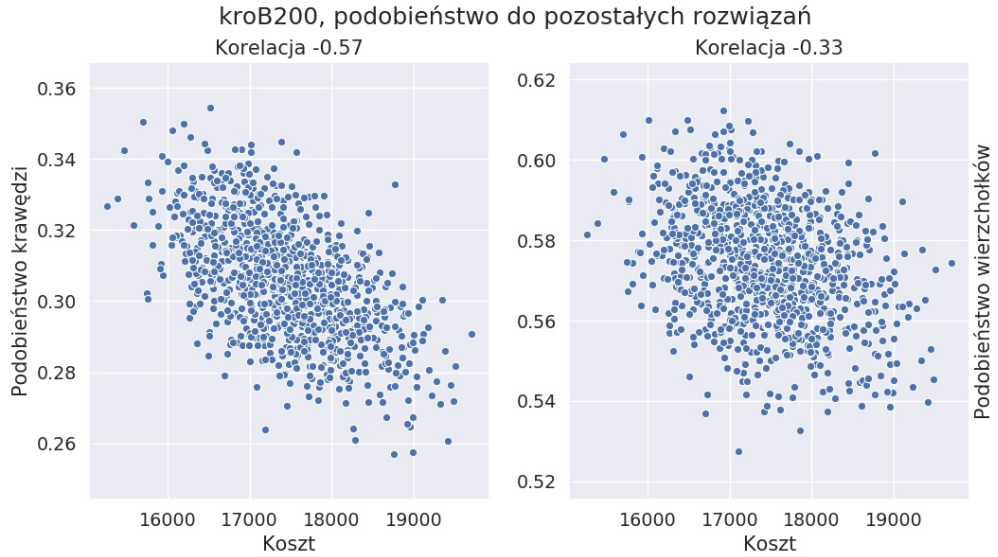


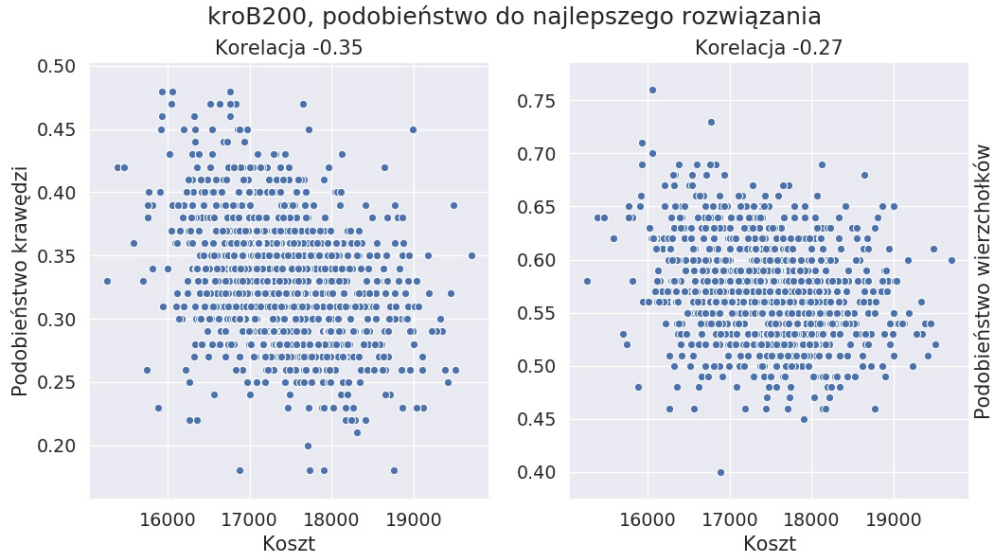

kroB100, podobieństwo do pozostałych rozwiązań
Korelacja -0.6
Korelacja -0.33
Podobieństwo krawędzi
Podobieństwo wierzchołków
Koszt
Koszt


kroB100, podobieństwo do najlepszego rozwiązania
Korelacja -0.32
Korelacja -0.24
Podobieństwo krawędzi
Podobieństwo wierzchołków
Koszt
Koszt


kroA200, podobieństwo do pozostałych rozwiązań
Korelacja -0.55
Korelacja -0.34
Podobieństwo krawędzi
Podobieństwo wierzchołków
Koszt
Koszt


kroA200, podobieństwo do najlepszego rozwiązania
Korelacja -0.26
Korelacja -0.23
Podobieństwo krawędzi
Podobieństwo wierzchołków
Koszt
Koszt





1.5 Wnioski

Jak widać na wizualizacjach wyników doświadczenia, istnieje korelacja pomiędzy podobieństwem krawędzi i wierzchołków a kosztem rozwiązania. Na wszystkich zbiorach danych otrzymaliśmy korelację zarówno w przypadku wierzchołków jak i krawędzi. Dzięki temu, możemy wnioskować że rozwiązania znajdujące się blisko optimum są podobne, co stanowi cechę funkcji wypukłych. Oznacza to, że mimo tego, że funkcja ma wiele minimum lokalnych, nie są one całkowicie różne od globalnego optimum.

1.6 Kod programu

Repozytorium z kodem programu dostępne jest pod adresem: `https://github.com/hancia/AEM`

Steady State

2.1 Opis zadania

Celem zadania była implementacja hybrydowego algorytmu ewolucyjnego i porównanie go z metodami MSLS i ILSx.

2.2 Pseudokod

Algorithm 3 Steady State

```
Wygeneruj losowe populację o rozmiarze 20
Popraw wszystkie losowe rozwiązania w populacji za pomocą Local Search
while Czas wykonania mniejszy niż 240 sekund do
    Wybierz dwa rozwiązania z populacji
    Przeprowadź krzyżowanie dwojga rodziców, generując jednego potomka
    Popraw rozwiązanie potomne za pomocą Local Search
    if W populacji nie ma takiego rozwiązania then
        if Koszt dziecka jest mniejszy od najgorszego rozwiązania w populacji then
            Zamień najgorsze rozwiązanie w populacji na rozwiązanie potomne
        end if
    end if
end while
Zwróć najlepsze rozwiązanie w populacji
```

Krzyżowanie

Krzyżowanie w naszym algorytmie wykonujemy następująco: z populacji losujemy dwa rozwiązania. Wybieramy losowy punkt z rozwiązania pierwszego, od którego przepisujemy 50 % rozwiązania do potomka. Następnie, z drugiego rozwiązania przepisujemy wierzchołki zaczynając od losowego punktu, jeżeli nie znajdują się jeszcze w rozwiązaniu, aż do uzyskania odpowiedniej długości ścieżki.

2.3 Wyniki eksperymentu obliczeniowego

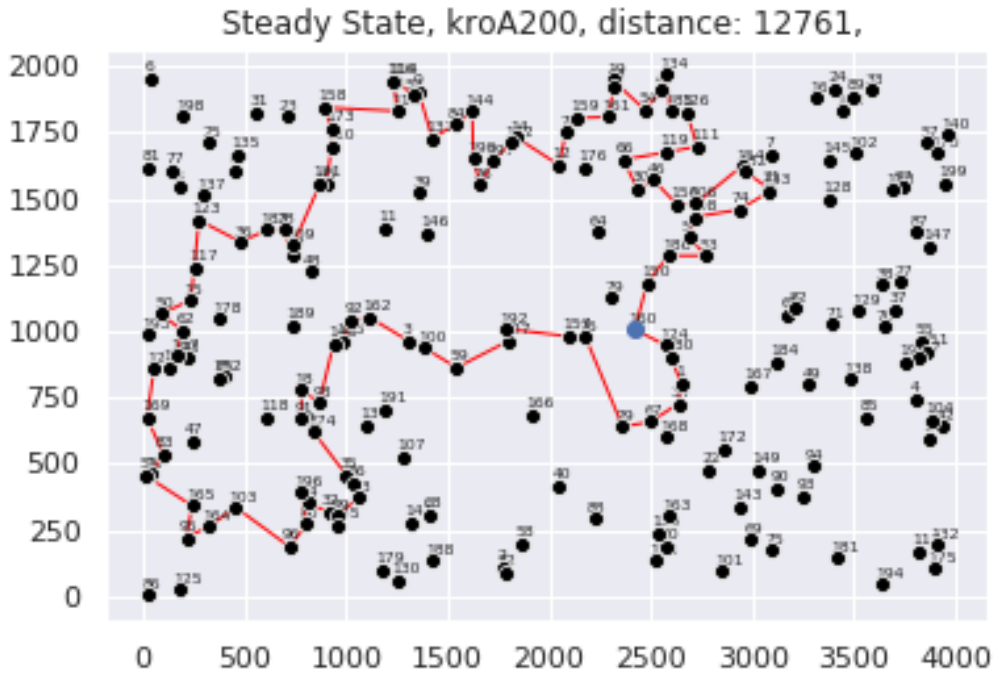
Wszystkie nasze rozwiązania były testowane na rozmiarach instancji 200 oraz z wykorzystaniem algorytmu Local Search w wersji steepest z ruchami wewnątrz trasowymi poprzez zamianę krawędzi. Dokonaliśmy takiego wyboru na podstawie wcześniejszych dobrych wyników dla tych wariantów oraz krótszego czasu wykonywania. Pomijamy tabele zawierającą czasy przetwarzania, ponieważ każde uruchomienie trwa 240 sekund.

instance	strategy	cost min	mean	max
kroA200	Multiple Start Local Search	14640	15032	15395
	Iterated Local Search	13589	13926	14168
	Destroy Repair Local Search	12882	13285	14103
	Steady State	12761	13020	13642
kroB200	Multiple Start Local Search	14760	15329	15715
	Iterated Local Search	14119	14416	14736
	Destroy Repair Local Search	12963	13198	13597
	Steady State	12541	13014	13318

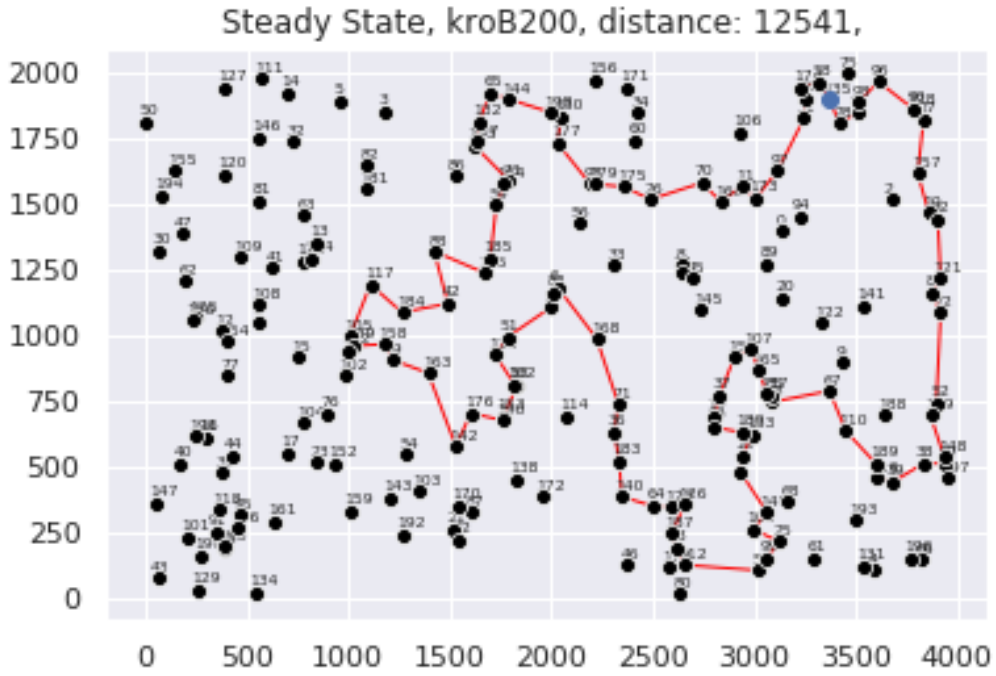
Tabela 2.1: Wyniki dla instancji kroA200 oraz kroB200.

2.4 Wizualizacje

2.4.1 KroA200



2.4.2 KroB200



2.5 Wnioski

Jak widać w tabeli porównującej wyniki poszczególnych rozwiązań, nasz algorytm ewolucyjny radzi sobie lepiej od poprzednich metod. Zarówno średnia, jak i minimalna wartość jest najmniejsza w steady state. Tak dobre działanie algorytmu możemy zawdzięczać stosunkowo szybkim procesem generowania nowego rozwiązania w fazie krzyżowania, co pozwala na wiele iteracji algorytmu w zadanym czasie. Krzyżowanie w sposób wybrany przez nas pozwala na zachowanie pewnej losowości, a zatem różnorodności rozwiązań wynikowych, co zwiększyć przestrzeń poszukiwanych rozwiązań. Taka wersja algorytmu genetycznego nadal jest mocno zależna od populacji początkowej, którą generujemy losowo - widzieliśmy różnice wyników w trakcie kolejnych uruchomień algorytmu.
Testowaliśmy również podejście z mutacją pomiędzy fazą krzyżowania a Local Search. Mutacja polegała na dodaniu kilku losowych ruchów wewnątrz i zewnątrz trasowych. Taka modyfikacja działała jednak średnio gorzej od wersji bez perturbacji - możliwe, że powstający w ten sposób potomkowie za bardzo różnili się od rodziców, co wprowadzało zbyt duży element losowości. Z tego powodu zdecydowaliśmy się na wersję bez mutacji.

2.6 Kod programu

Repozytorium z kodem programu dostępne jest pod adresem: `https://github.com/hancia/AEM/tree/master/strategies`

Metody bazujące na lokalnym przeszukiwaniu

3.1 Opis zadania

Zadanie polegało na implementacji oraz przetestowaniu trzech algorytmów, które wykorzystują wielokrotne uruchamianie algorytmu Local Search.

3.2 Pseudokod

3.2.1 Multiple Start Local Search

Algorithm 4 Multiple Start Local Search

1: Wygeneruj 100 początkowych rozwiązań z których każde zaczynamy w innym punkcie.
2: Każde rozwiązanie początkowe popraw używając LS
3: Zwróć najkrótsze rozwiązanie

3.2.2 Iterated Local Search

Algorithm 5 Iterated Local Search

1: Wygeneruj losowe rozwiązanie
2: **while** Czas wykonania mniejszy niż 240 sekund **do**
3: Wykonaj 12 losowych ruchów zewnątrz trasowych
4: Wykonaj 12 losowych ruchów wewnątrz trasowych
5: Popraw obecne rozwiązanie używając Local Search
6: **end while**
7: Zwróć najlepsze rozwiązanie pośród wszystkich iteracji

Po wygenerowaniu losowego rozwiązania wykonujemy po 12 ruchów wewnątrz trasowych oraz zewnątrz trasowych. Dokonaliśmy testów na wartościach od 6-14 i błąd spadał do wartości 10, potem już spadek nie był znaczący bądź nie było go wcale. Natomiast wraz ze zwiększaniem perturbacji rozwiązania cały czas rósł czas znajdowania rozwiązania, a za tym malała ilość możliwych iteracji do wykonania. Biorąc pod uwagę przetarg między czasem a jakością rozwiązania wybraliśmy wartość 12.

3.2.3 Iterated local search Destry-Repair

Algorithm 6 Iterated Local Search Destry-Repair

1: Wygeneruj losowe rozwiązanie
2: **while** Czas wykonania mniejszy niż 240 sekund **do**
3: Wybierz wierzchołki do usunięcia
4: Usuń wybrane wierzchołki z rozwiązania zostawiając miejsce do wypełnienia
5: Uzupełnij miejsca wykorzystując algorytm Greedy Cycle bez żalu
6: Popraw obecne rozwiązanie używając Local Search
7: **end while**
8: Zwróć najlepsze rozwiązanie pośród wszystkich iteracji

Postaramy się dokładniej przybliżyć w jaki sposób wykonujemy etap destroy repair.

- Wyznaczamy liczbę wierzchołków, które za każdym razem mają być usuwane. Przykładowo dla ścieżki o długości 6 i 50% wierzchołków będzie to 3. Następnie tworzymy binarną maskę reprezentującą który wierzchołek jest usuwany, a który nie. Utworzymy wektor sześciu zer i wylosujemy 3 pozycje jedynek. W rezultacie otrzymamy $[0, 1, 1, 0, 1, 0]$ - co znaczy że, drugi, trzeci oraz piąty wierzchołek zostanie usunięty z trasy.

- Tak utworzoną maskę aplikujemy na naszym obecnym rozwiązaniu usuwając te wierzchołki których indeksy odpowiadają indeksom jedynek w masce. W tym przykładzie po aplikacji maski na $[79, 1, 20, 48, 5, 23]$ uzyskamy $[79, \_, \_, 48, \_, 23]$.

- W tym momencie mamy poprzerywaną ścieżkę. Teraz grupujemy nasze "dziury" przechowując pierwszy i ostatni indeks dziury - w naszym przykładzie uzyskamy $[(0, 3), (3, 5)]$, co będzie odpowiadało $([79, \_, \_, 48], [48, \_, 23])$.

- Teraz dla każdej takiej "dziury" tworzymy ścieżkę z pierwszej i ostatniej pozycji np. $[79, 48]$ i rozszerzamy ją używając algorytmu Greedy Cycle dopóki nie osiągnie on rozmiaru naszej dziury, w tym konkretnym przypadku będą to dwie iteracje.

Strojenie parametrów odbyło się analogicznie do wcześniejszej wersji iteracyjnej. Co ciekawe najlepsze wyniki uzyskiwaliśmy między 30%-40% oraz 80%-90%. Ostatecznie wybraliśmy 40%. Algorytm działa bardzo szybko ponieważ po połączeniu ścieżek za pomocą Greedy Cycle prawdopodobnie przestrzeń lepszych rozwiązań jest bardzo mała. Tak duże modyfikowanie ścieżki rozumiemy w ten sposób, że zostawiamy w cyklu kilka przystanków, bądź punktów kluczowych a następnie problem jest zdekomponowany na podproblemy rozwiązywane za pomocą Greedy Cycle. Ze względu na krótki czas przetwarzania przez Local Search możemy sobie pozwolić na dużo iteracji mocno ingerujących w cykl.

3.3 Wyniki eksperymentu obliczeniowego

Wszystkie nasze rozwiązania były testowane na rozmiarach instancji 200 oraz z wykorzystaniem algorytmu Local Search w wersji steepest z ruchami wewnątrz trasowymi poprzez zamianę krawędzi. Dokonaliśmy takiego wyboru na podstawie wcześniejszych dobrych wyników dla tych wariantów oraz krótszego czasu wykonywania.

		cost		
		min	mean	max
instance	strategy			
kroA200	MultipleStartLocalSearch	14640	15032	15395
	IteratedLocalSearch	13589	13926	14168
	DestroyRepairLocalSearch	12882	13285	14103
kroB200	MultipleStartLocalSearch	14760	15329	15715
	IteratedLocalSearch	14119	14416	14736
	DestroyRepairLocalSearch	12963	13198	13597

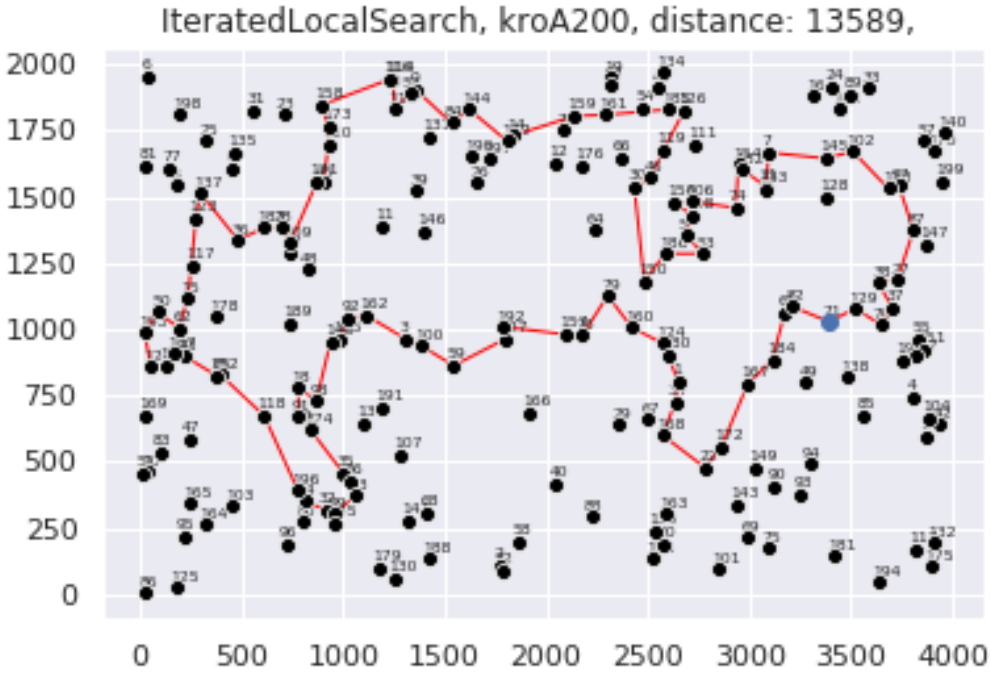
Tabela 3.1: Wyniki dla instancji kroA200 oraz kroB200 dla rozszerzeń LocalSearch.

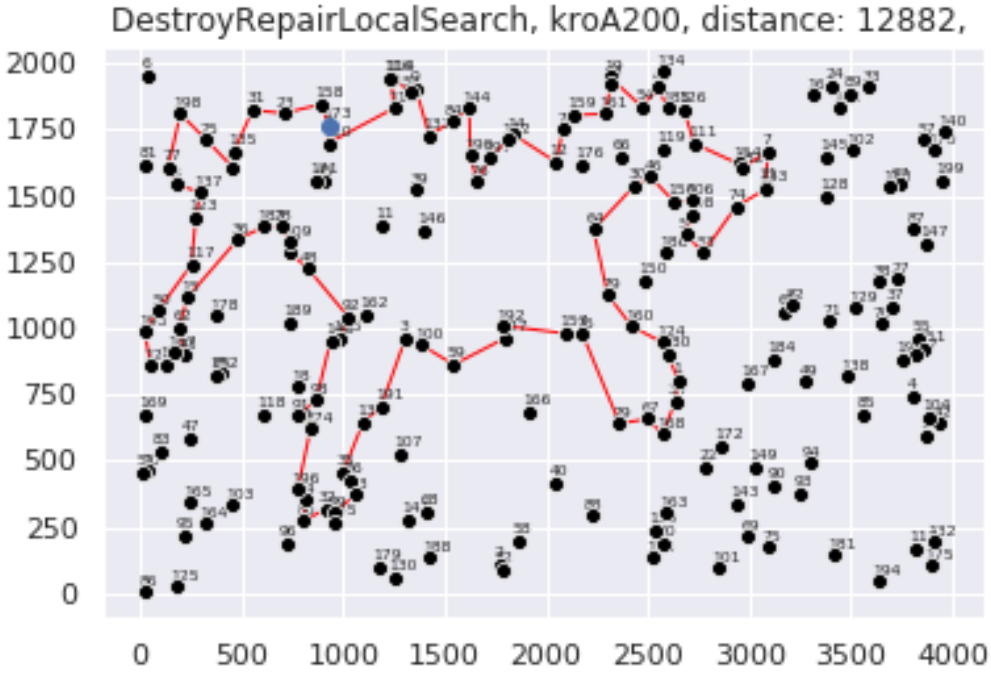
instance	strategy	time min	time mean	time max
kroA200	MultipleStartLocalSearch	229.297	240.476	249.310
	IteratedLocalSearch	240.231	240.624	240.847
	DestroyRepairLocalSearch	240.012	240.131	240.261
kroB200	MultipleStartLocalSearch	231.590	239.021	252.038
	IteratedLocalSearch	240.032	240.416	240.681
	DestroyRepairLocalSearch	240.001	240.123	240.332

Tabela 3.2: Czasy przetwarzania instancji kroA200 oraz kroB200 dla rozszrzeń LocalSearch. Wyniki podane w sekundach

3.4 Wizualizacje

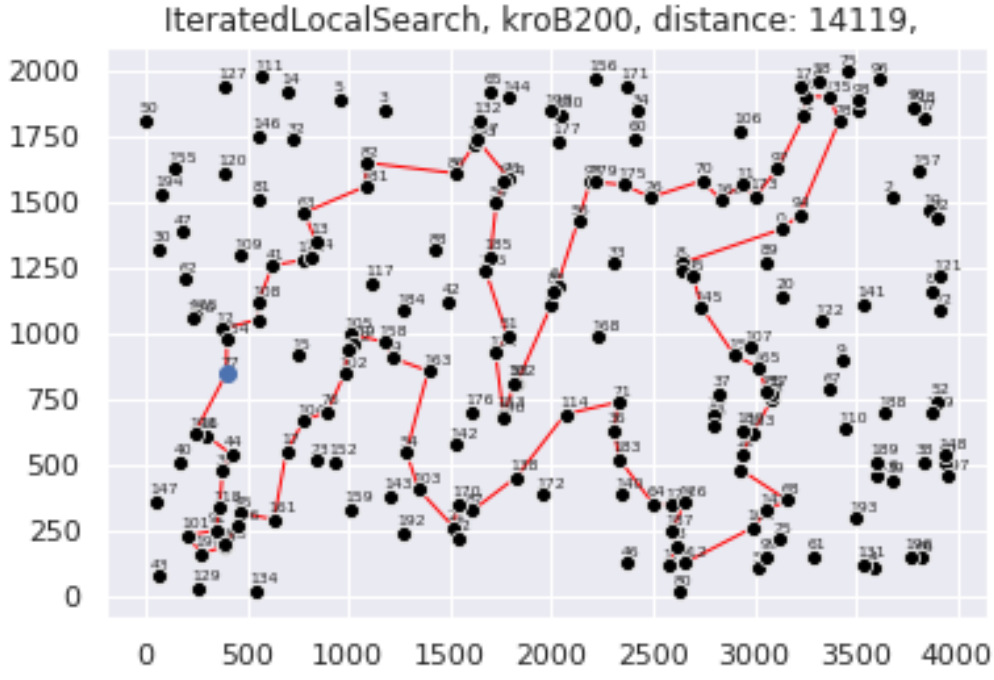
3.4.1 KroA200

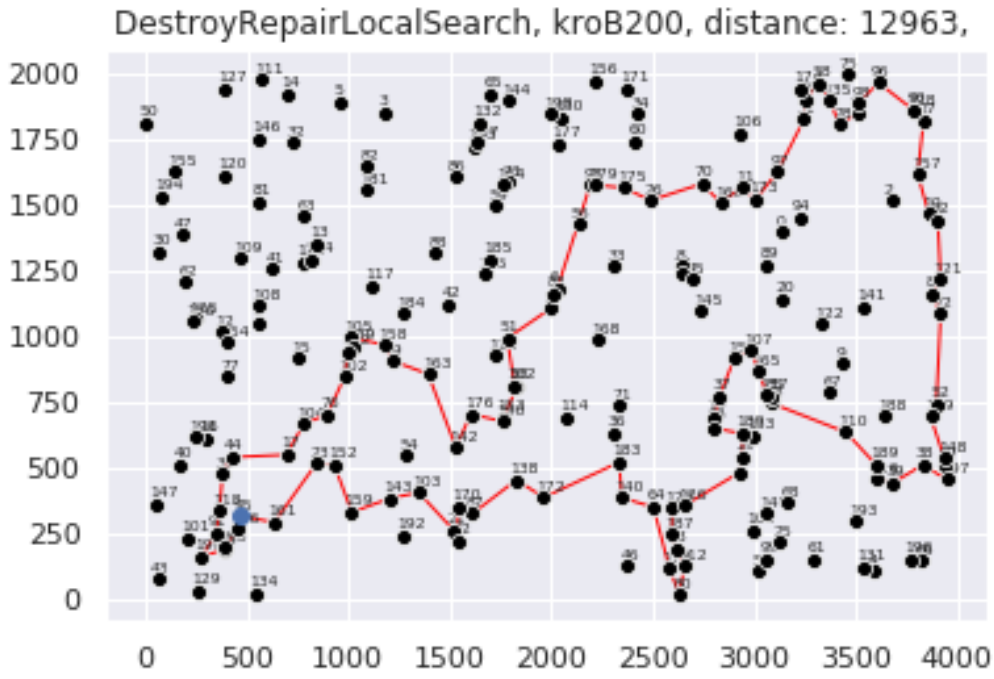

IteratedLocalSearch, kroA200, distance: 13589,
2000
1750
1500
1250
1000
750
500
250
0
0
500
1000
1500
2000
2500
3000
3500
4000


DestroyRepairLocalSearch, kroA200, distance: 12882,
2000
1750
1500
1250
1000
750
500
250
0
0
500
1000
1500
2000
2500
3000
3500
4000

3.4.2 KroB200


MultipleStartLocalSearch, kroB200, distance: 14760,
0
250
500
750
1000
1250
1500
1750
2000
0
500
1000
1500
2000
2500
3000
3500
4000


IteratedLocalSearch, kroB200, distance: 14119,
0
250
500
750
1000
1250
1500
1750
2000
0
500
1000
1500
2000
2500
3000
3500
4000



3.5 Wnioski

Multiple Start Local Search poprawia wyniki zwykłego Local Search. Jest to jednak spodziewany ponieważ uruchamiamy go 10 razy częściej w naszym porównaniu, finalnie znajdowane jest najlepsze ziarno. Dodatkowe perturbacje wprowadzone w wersji iteracyjnej dają zauważalną i naszym zdaniem satysfakcjonującym poprawę. Ku naszemu zaskoczeniu wersja destroy-repair dała bardzo wysoką poprawę. Nasza intuicja z czego ona wynika została przedstawiona podczas opisu algorytmu.

3.6 Kod programu

Repozytorium z kodem programu dostępne jest pod adresem: `https://github.com/hancia/AEM/tree/master/strategies`

Poprawa efektywności czasowej lokalnego przeszukiwania

4.1 Opis zadania

Celem jest poprawa efektywności czasowej lokalnego przeszukiwania w wersji stromej (steepest) z ruchem wymiany krawędzi. Stosujemy dwa mechanizmy:

- Wykorzystanie ocen ruchów z poprzednich iteracji z uporządkowaną listą ruchów
- Ruchy kandydackie

Jako kandydackie stosujemy ruchy wprowadzające do rozwiązania co najmniej jedną krawędź kandydacką. Krawędzie kandydackie definiujemy wyznaczając dla każdego wierzchołka 5 innych najbliższych wierzchołków.

4.2 Pseudokod

4.2.1 Wykorzystanie ocen ruchów z poprzednich iteracji z uporządkowaną listą ruchów

Algorithm 7 Wykorzystanie ocen ruchów z poprzednich iteracji z uporządkowaną listą ruchów

```
1: Wygeneruj losowe rozwiązanie x
2: Zainicjuj LM - listę ruchów przynoszących poprawę, uporządkowaną od najlepszego
   do najgorszego
3: while Nie znaleziono ruchu m po przejrzeniu całej listy LM do
4:     Przejrzyj wszystkie nowe ruchy i dodaj do LM ruchy przynoszące poprawę,
5:     Przeglądaj ruchy m z LM od najlepszego do znalezienia aplikowalnego ruchu.
6:     Sprawdź czy m jest aplikowalny i jeśli nie, usuń go z LM
7:     if znaleziono ruch m then
8:         x = m(x) (zaakceptuj m(x))
9:     end if
10: end while
```

4.2.2 Ruchy kandydackie

Algorithm 8 Ruchy kandydackie

```
1: Wygeneruj losowe rozwiązanie x
2: while Nie przejrzano wszystkich możliwości x do
3:     Oblicz koszt zamiany wierzchołków
4:     if f(y) > f(x) then                ▷ Dokonaj najlepszej zamiany wierzchołków
5:         x ← y
6:     end if
7: end while
8: while Nie przejrzano 5 najbliższych sąsiadów x do
9:     Oblicz koszt wymiany krawędzi
10:    if f(y) > f(x) then                ▷ Zamień najlepszą parę krawędzi
11:        x ← y
12:    end if
13: end while
14: Jeśli udało Ci się poprawić wróć do linijki 2
```

Ograniczamy przestrzeń poszukiwań krawedzi do takich, ktore sa w takiej relacji, że istnieje miedzy nimi para sasiadow z ktorych przynajmniej jeden jest w gronie najblizszych sasiadow drugiego

4.3 Wyniki eksperymentu obliczeniowego

		cost		
		min	mean	max
instance	strategy			
kroA200	Candidate_moves	17859	21051	26088
	LM	15245	17715	19516
	Local_search	14669	17029	19148
kroB200	Candidate_moves	18466	21644	28988
	LM	15427	17661	19713
	Local_search	15551	17043	18686

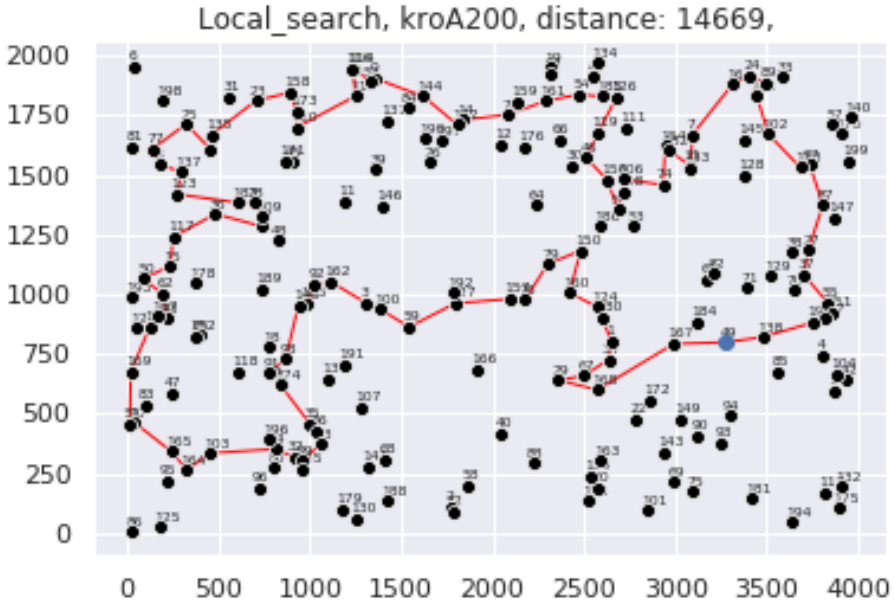
Tabela 4.1: Wyniki algorytmów

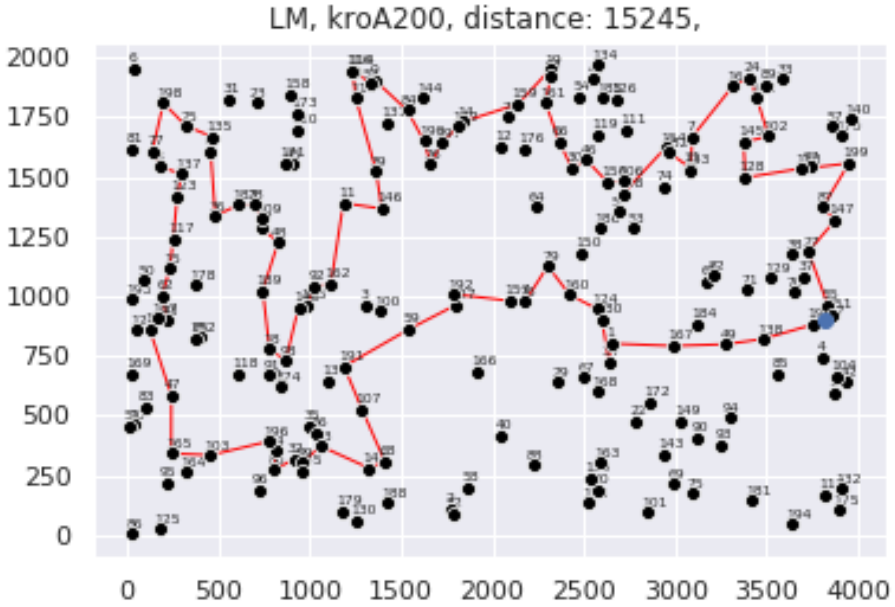
		time		
		min	mean	max
instance	strategy			
kroA200	Candidate_moves	0.567	0.767	1.088
	LM	1.481	1.935	2.591
	Local_search	1.884	2.305	3.046
kroB200	Candidate_moves	0.581	0.744	1.000
	LM	1.557	1.881	2.472
	Local_search	1.944	2.269	2.920

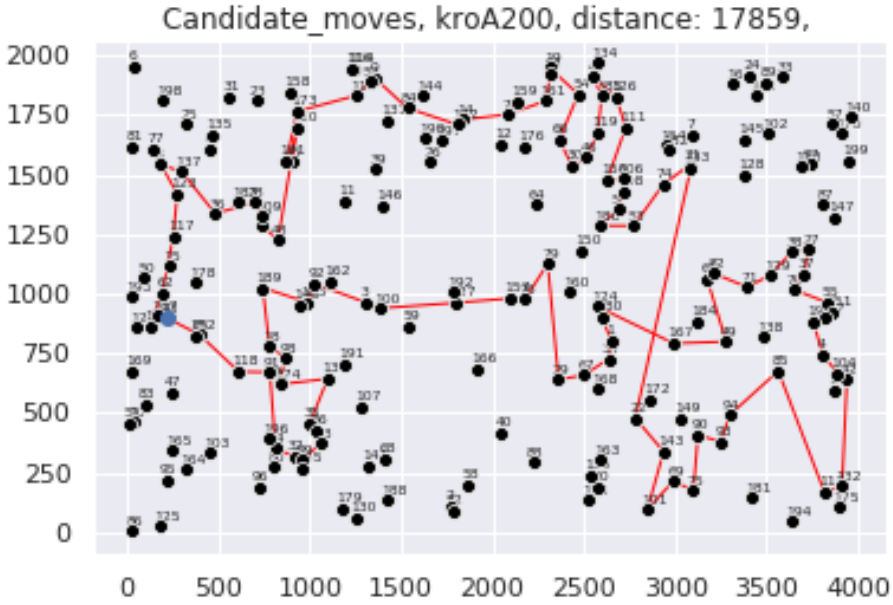
Tabela 4.2: Wyniki czasu przetwarzania jednej instancji dla algorytów Local Search w sekundach.

4.4 Wizualizacje

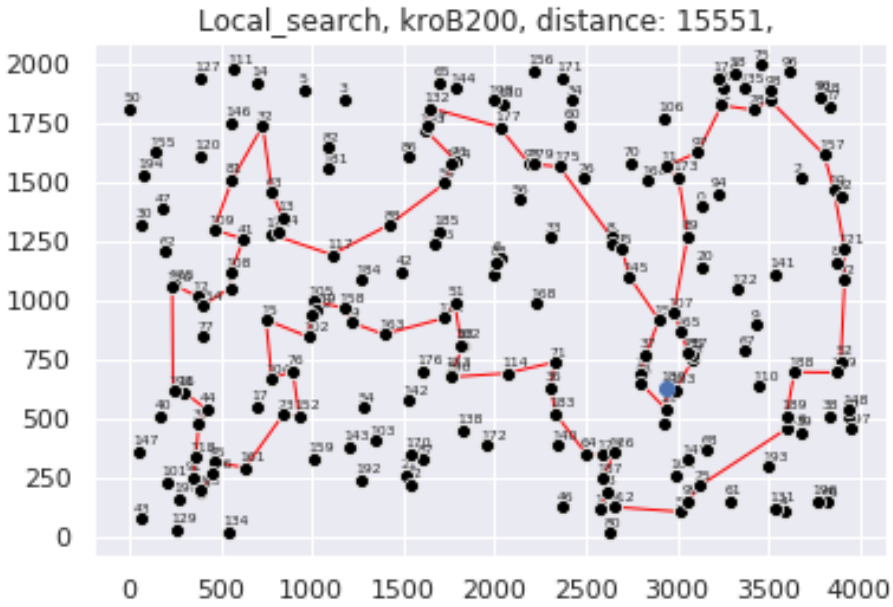
4.4.1 KroA200

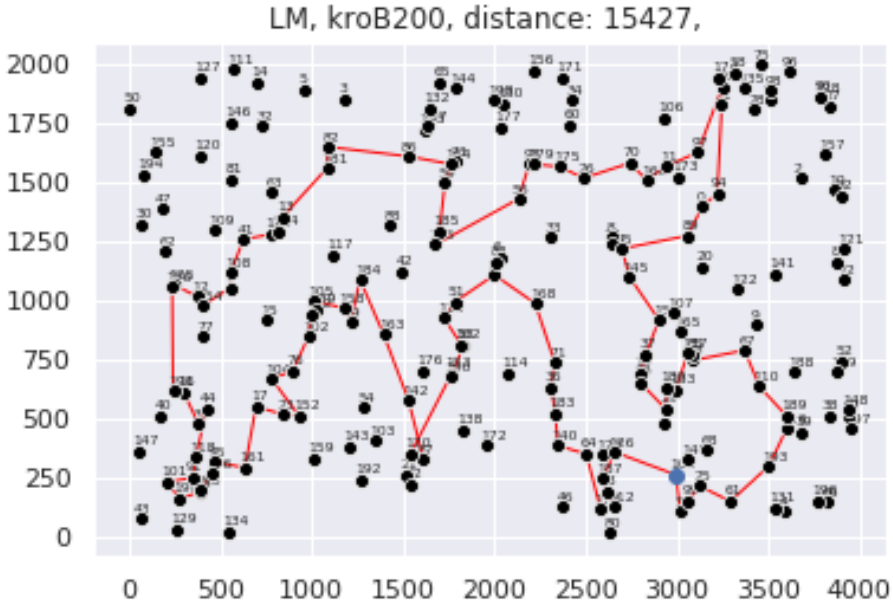


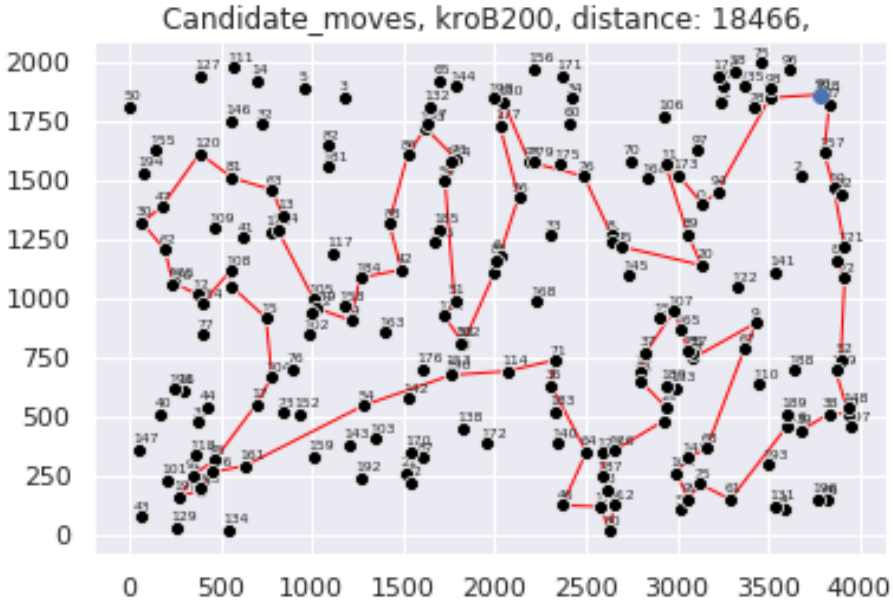



Candidate_moves, kroA200, distance: 17859,
2000
1750
1500
1250
1000
750
500
250
0
0
500
1000
1500
2000
2500
3000
3500
4000

4.4.2 KroB200


Local_search, kroB200, distance: 15551,
2000
1750
1500
1250
1000
750
500
250
0
0
500
1000
1500
2000
2500
3000
3500
4000





4.5 Wnioski

Tak jak oczekiwaliśmy, obie techniki poprawy efektywności czasowej lokalnego przeszukiwania skróciły czas wykonywania naszego programu. W naszej implementacji, najszybciej wykonywała się wersja z ruchami kandydackimi - udało się przyspieszyć klasyczny local search o 70%. Wersja last moves dała nam około 20% poprawy efektywności czasowej. Zyskując na kryterium czasu, tracimy na jakości rozwiązania. W przypadku last moves różnica nie jest aż tak znacząca, a nawet w przypadku najmniejszego kosztu udaje mu się nieznacznie pobić wynik klasycznego local search. Jest to efekt przez nas oczekiwany,

ponieważ przechowujemy najlepsze ruchy, co pozwala nam skrócić czas ich wyznaczania. Natomiast wersja candidate moves radzi sobie trochę gorzej i osiąga o około 20% gorsze wyniki w stosunku do podstawowej wersji local search. Również spodziewaliśmy się takiego wyniku, ponieważ mocno ograniczamy przestrzeń poszukiwań przeglądając tylko najbliższych sąsiadów, przez co możemy nie pomijać niektóre dobre ruchy, za to bardzo szybko otrzymujemy wynik.

4.6 Kod programu

Repozytorium z kodem programu dostępne jest pod adresem: `https://github.com/hancia/AEM/tree/master/strategies`

Local Search

5.1 Opis zadania

Zadanie polega na implementacji lokalnego przeszukiwania w wersjach stromej (steepest) i zachłannej (greedy) z dwoma różnym rodzajami sąsiedztwa - zamianą wierzchołków, bądź krawędzi w trasie.

5.2 Pseudokod

5.2.1 Steepest local search

Algorithm 9 Steepest local search

```
1: Wygeneruj losowe rozwiązanie x
2: while Nie przejrzano całego sąsiedztwa x do
3:     Wygeneruj nowego kandydata y na podstawie sąsiedztwa
4:     if f(y) > f(x) then
5:         x ← y
6:     end if
7: end while
```

5.2.2 Greedy local search

Algorithm 10 Greedy local search

```
1: Wygeneruj losowe rozwiązanie x
2: while Nie znajdziesz lepszego kandydata x do
3:     Wylosuj typ ruchu
4:     Znajdź lepszego sąsiada y bazując na wylosowanym typie
5:     if f(y) > f(x) then
6:         x ← y
7:     end if
8:     Znajdź lepszego sąsiada y bazując na typie przeciwnym do wylosowanego
9:     if f(y) > f(x) then
10:        x ← y
11:    end if
12: end while
```

5.2.3 Sąsiedztwo

W naszym algorytmie rozważamy dwa typy ruchów. Pierwszy z nich to zamiana wierzchołka z obecnego cyklu z takim poza cyklem, która ma na celu szukać odpowiednich miast, które mamy odwiedzić. Drugi ruch wykonujemy zależnie od wybranego sąsiedztwa. Pierwszym z nich jest zamienianie kolejności dwóch wierzchołków które są obecnie w trasie, drugie polega na zamienieniu dwóch krawędzi. Dla algorytmu w wersji steepest szukamy najlepszego sąsiada zawierającego oba typy ruchów. W wersji greedy losujemy kolejność, tzn. albo najpierw używamy sąsiedztwa na podstawie ruchu zewnątrz trasowego, albo wewnątrz trasowego. Przykładowo wylosowaliśmy, że zaczynamy od ruchów zewnątrz trasowych. Rozpoczynamy szukanie lepszego rozwiązania poprzez wymianie wierzchołków.

Jeśli znajdziemy to akceptujemy nowe rozwiązanie i rozpoczynamy szukanie poprzez ruchy wewnątrz trasowe. Jeśli nie znaleźliśmy, również przechodzimy do drugiego typu ruchów. Następnie jeśli znajdziemy rozwiązanie poprzez ruch wewnątrz trasowy to je akceptujemy. Jeśli nie znaleźliśmy lepszego rozwiązania poprzez ruch wewnątrz trasowy to mamy dwie możliwości. Pierwsza to sytuacja w której wcześniej poprawiliśmy ruchem rozwiązanie ruchem zewnątrz trasowym - wtedy powtarzamy procedure zaczynając od losowania. Jeśli nie udało nam się wcześniej poprawić rozwiązania to przerywamy przetwarzanie.

5.3 Wyniki eksperymentu obliczeniowego

			cost		
			min	mean	max
instance	version	neigbourhood			
kroA100	greedy	edge	11578	12908	14372
		vertex	13781	17537	21787
	steepest	edge	10840	12542	14096
		vertex	13951	17685	24747
kroB100	greedy	edge	11423	12753	15844
		vertex	14053	17603	22226
	steepest	edge	11297	12477	15167
		vertex	13408	17669	21989

Tabela 5.1: Wyniki algorytmu Local Search

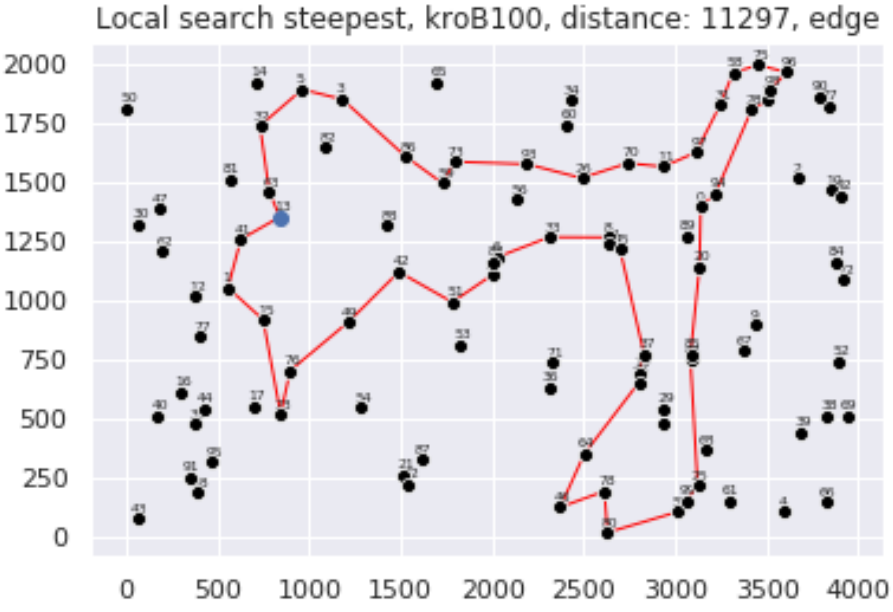
			time		
			min	mean	max
instance	version	neigbourhood			
kroA100	greedy	edge	0.163	0.322	0.607
		vertex	0.334	0.718	1.226
	steepest	edge	0.151	0.200	0.299
		vertex	0.209	0.313	0.475
kroB100	greedy	edge	0.179	0.319	0.507
		vertex	0.410	0.765	1.368
	steepest	edge	0.152	0.209	0.407
		vertex	0.206	0.302	0.476

Tabela 5.2: Wyniki czasu przetwarzania jednej instancji dla algorytmu Local Search w sekundach.

5.4 Wizualizacje

Najlepsze wyniki dla obu instancji uzyskaliśmy używając sąsiedztwa w którym zamieniamy krawędzie. Dla obu instancji lepszy wynik uzyskał algorytm w wersji steepst.


Local search steepest, kroA100, distance: 10840, edge
2000
1750
1500
1250
1000
750
500
250
0
0
500
1000
1500
2000
2500
3000
3500
4000


Local search steepest, kroB100, distance: 11297, edge
2000
1750
1500
1250
1000
750
500
250
0
0
500
1000
1500
2000
2500
3000
3500
4000

5.5 Wnioski

Zgodnie z naszą intuicją sąsiedztwo poprzez zamianę krawędzi zadziałało lepiej niż poprzez zamianę wierzchołków ze względu na mniejszą ingerencje w cykl - sąsiedzi są bardziej podobni. Spodziewaliśmy się, że algorytm w wersji greedy będzie szybszy, jednak zazwyczaj jest on wolniejszy. Ciekawą obserwacją wydaje się być fakt, że sąsiędztwo wewnątrztrasowe poprzez zamiane wierzchołków znacząco wydłuża czas przetwarzania algorytmu greedy, czego nie obserwujemy w wersji steepest. Oczekiwaliśmy, że algorytm w wersji greedy okaże się gorszy od wersji steepest, ku naszemu zaskoczeniu średnie wyniki dla sąsiedztwa poprzez zamiane krawędzi są o około 100 lepsze, a dla sąsiedztwa poprzez zamianę wierzchołków o około 500 gorsze.

5.6 Kod programu

Repozytorium z kodem programu dostępne jest pod adresem: `https://github.com/hancia/AEM/blob/master/strategies/local_search/local_search.py`

Greedy Cycle

6.1 Opis zadania

Rozważany problem to zmodyfikowany problem komiwojażera. Dany jest zbiór wierzchołków i macierz symetrycznych odległości pomiędzy każdą parą wierzchołków. Należy znaleźć najkrótszą ścieżkę zamkniętą przechodzącą przez dokładnie 50% wszystkich wierzchołków (w przypadku nieparzystej liczby wierzchołków zaokrąglamy w górę). Wybór wierzchołków do ścieżki jest elementem rozwiązania. Minimalizowane kryterium to długość zamkniętej ścieżki.

6.2 Pseudokod

6.2.1 Algorytm zachłanny

Algorithm 11 Greedy Cycle

Input
Ważony graf $G(V, E)$
Output
Cykl T

1: Wstaw wierzchołek V do ścieżki T
2: **while** $|T| < \lceil \frac{|V|}{2} \rceil$ **do**
3: $VE_{candidates} \leftarrow \emptyset$
4: **for** $V_i \in G : V \not\subset T$ **do** ▷ Dla każdego wierzchołka, który nie jest w cyklu
5: $VE_{best} \leftarrow \arg\min_{(V_i,E):E \subset T} cost(Vi, E)$ ▷ Znajdź (V_i, E), które min *cost* gdzie $cost(V_i, E) = d(V_1, V_i) + d(V_i, V_2) - d(V_1, V_2)$, E łączy V_1,V_2, a d to odległość
6: $VE_{candidates} \leftarrow VE_{candidates} \cup VE_{best}$
7: **end for**
8: $T \leftarrow T \cup \arg\min_V cost(VE_{candidates})$ ▷ Wstaw do cyklu V kandydata o najmniejszym koszcie
9: **end while**
10: **return** T

6.2.2 Algorytm z żalem

Algorithm 12 Greedy Cycle with k-regret

Input
Ważony graf $G(V, E)$
k - parametr żalu
Output
Cykl T

1: Wstaw wierzchołek V do ścieżki T
2: **while** $|T| < \lceil \frac{|V|}{2} \rceil$ **do**
3: $VCost \leftarrow \emptyset$
4: **for** $V_i \in G : V \not\subset T$ **do** ▷ Dla każdego wierzchołka i, który nie jest w cyklu
5: **for** $E_i \in G : E \subset T$ **do** ▷ Dla krawędzi i, która tworzy cykl
6: $VCost_{V_i} \leftarrow VCost_{V_i} \cup cost(V_i, E_i)$ ▷ gdzie $cost(V_i, E_i) = d(V_1, V_i) + d(V_i, V_2) - d(V_1, V_2)$, E łączy V_1,V_2, a d to odległość
7: **end for**
8: $sort(VCost_{V_i})$ ▷ Posortuj koszty wstawienia dla każdego V
9: **end for**
10: $V_{best} \leftarrow argmax_{V_i} regret(Vcost_{V_i})$ ▷ Znajdz V_i, którego $regret$ jest największy gdzie $regret(Vcost_{V_i}) \leftarrow \sum_l^k Vcost_{V_i}[l] - Vcost_{V_i}[0]$
11: $T \leftarrow T \cup VCost_{V_{best}}[0]$ ▷ Wstaw V_{best} w miejsce E o najmniejszym koszcie
12: **end while**
13: **return** T

6.3 Wyniki eksperymentu obliczeniowego

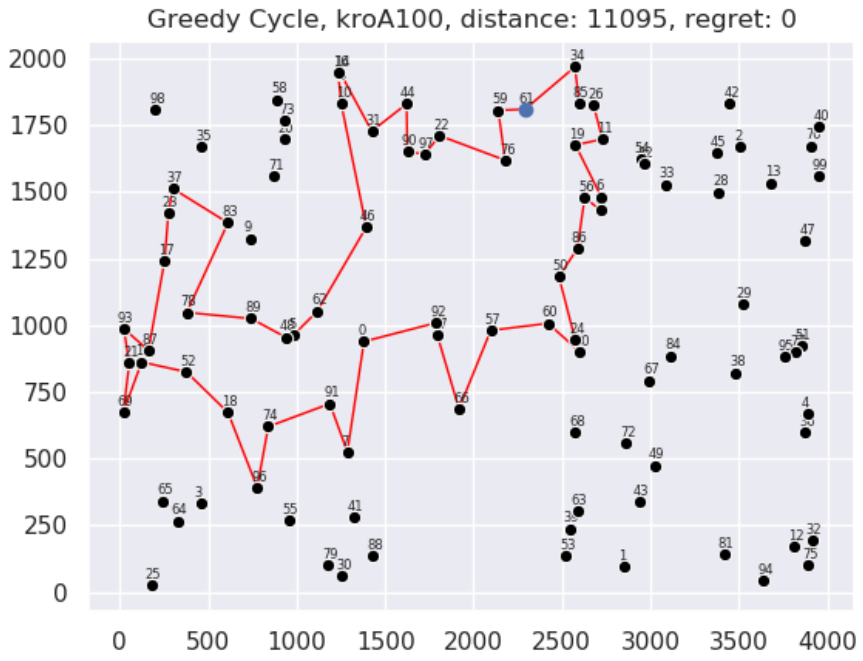
Zbiór danych	Min	Średnia	Max
kroA100	11095	12506	13266
kroB100	10346	11785	13926

Tabela 6.1: Wyniki dla algorytmu zachłannego

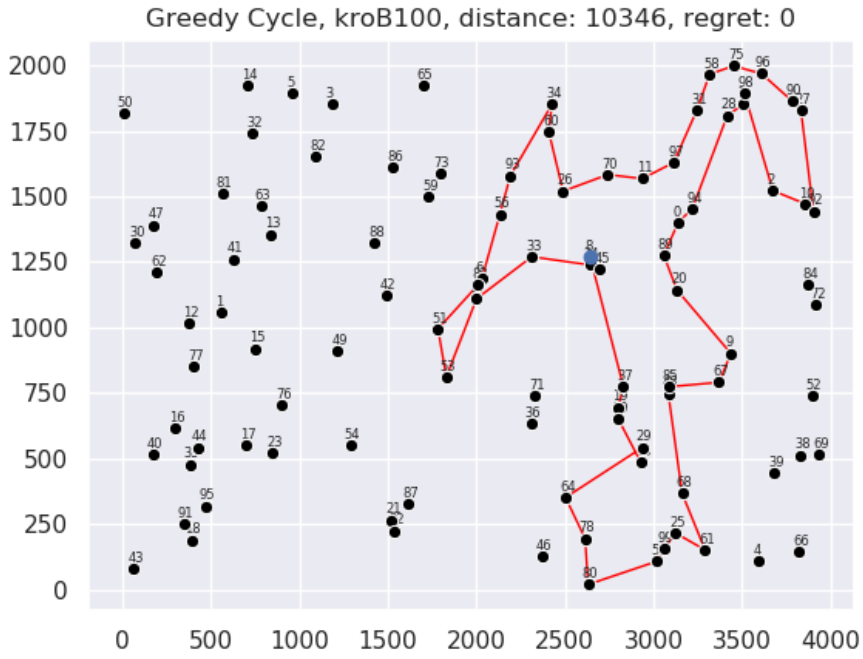
Zbiór danych	Min	Średnia	Max
kroA100	18620	20635	23810
kroB100	17991	21244	23743

Tabela 6.2: Wyniki dla algorytmu z żalem

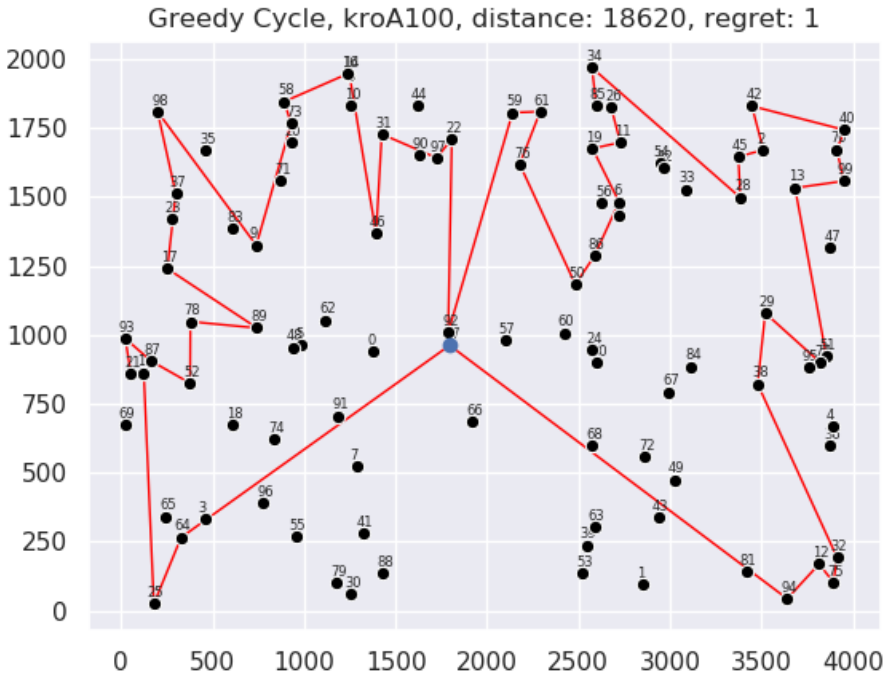
6.4 Wizualizacje



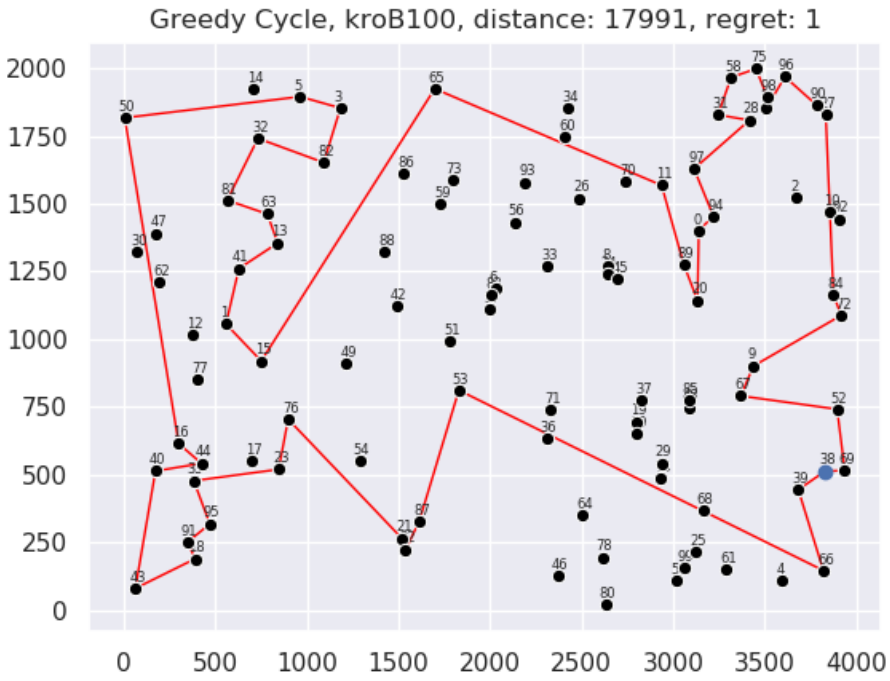
Rysunek 6.1: Algorytm zachłanny, kroA100



Rysunek 6.2: Algorytm zachłanny, kroB100



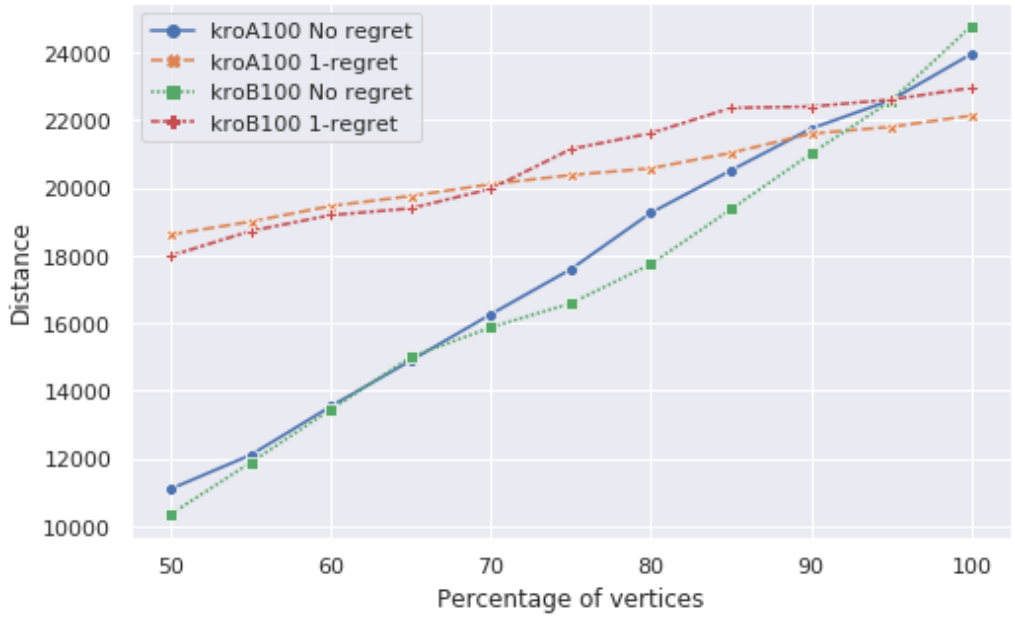
Rysunek 6.3: Algorytm z żalem, kroA100



Rysunek 6.4: Algorytm z żalem, kroB100

6.5 Wnioski

Obserwujemy, że w naszym problemie wersja algorytmu Greedy Cycle bez żalu sprawdziła się lepiej niż ta z żalem. Według nas może to być spowodowane tym, że wersja z żalem wybiega w przyszłość i bierze pod uwagę szerszy kontekst. Powoduje to, że na początku wybierane są gorsze wierzchołki, co zaczyna opłacać się w przyszłości. Jednak w naszym problemie ograniczamy się tylko do 50% miast i jeśli przypadek "problematyczny"[1] powstaje to może zajść sytuacja, w której kolejne miasta wybieranie zachłannie są lepsze a problematyczny przypadek jest pomijany. Idąc tym tokiem myślenia wersja bez żalu powinna stworzyć więcej bardzo kosztownych opcji na koniec, gdzie wersja z żalem by je minimalizowała. Wersja z żalem nie przynosi zysku, ponieważ nie dochodzimy do momentu, w którym te operacje zaczynają się opłacać.
Sprawdziliśmy nasze założenie i szukając ścieżek zawierających wszystkie miasta wersja z żalem wypadła lepiej niż wersja bez. Uzyskaliśmy poprawę dla kroA100 z 23960 na 22137, oraz z 24781 na 22959 dla kroB100.
Sprawdziliśmy zatem od jakiej ilości wierzchołków w cyklu zaczyna być opłacalne stosowanie wersji algorytmu z żalem.



Rysunek 6.5: Wykres przedstawiający różnice odległości między algorytmem w wersji z żalem oraz dla instancji kroA100 i kroB100 względem wzrostu ilości wierzchołków w cyklu.

Obserwujemy, że dla instancji kroA100 przecięcie nastąpiło przy około 88%, a dla kroB przy około 95%. Podsumowując wersja algorytmu z żalem zaczyna być lepsza przy wysokiej liczbie wierzchołków w cyklu.

6.6 Kod programu

Repozytorium z kodem programu dostępne jest pod adresem: `https://github.com/hancia/AEM/blob/master/strategies/greedy_cycle/cheapest_insertion.py`

[1] Przez przypadek problematyczny oznaczamy wierzchołek o dużej wartości żalu