IMO - Zadanie 3

Autorzy: Dariusz Max Adamski, Sławomir Gilewski

Wprowadzenie

W sprawozdaniu opisane są implementacje dwóch mechanizmów poprawy efektywności lokalnego przeszukiwania, dla zmodyfikowanego problemu komiwojażera. Pierwszym wykorzystanym mechanizmem było wykorzystanie ocen ruchów z poprzednich iteracji lokalnego przeszukiwania. Drugim mechanizmem były ruchy kandydackie znajdowane dla k najbliższych wierzchołków.

Długość cykli i efektywność czasową mechanizmów porównano z heurystyką konstrukcyjną opartą na największym 2-żalu, oraz algorytmem lokalnego przeszukiwania w wersji stromej. Wyniki zaimplementowanych algorytmów badano startując ze wspólnego rozwiązania losowego dla instancji “kroA200” i “kroB200” z biblioteki TSPLib.

Kod źródłowy jest dostępny on-line[1].

Funkcje pomocnicze

Dla zwięzłości, opisane w poprzednim zadaniu funkcje obliczające deltę oceny rozwiązania po wykonaniu danego ruchu oraz generujące i wykonujące początkowe ruchy zostały pominięte w sprawozdaniu. Ich szczegółowe opisy i pseudokod znajdują się w sprawozdaniu dotyczącym lokalnego przeszukiwania.

Funkcją pomocniczą wprowadzoną w tym sprawozdaniu jest funkcja *next_moves*, która zwraca nowe możliwe ruchy (zamiana wierzchołków lub krawędzi) po wykonaniu danego ruchu.

```
function next_moves(cities, cycles, move)
      new_moves := []
      if move swapped edges in cycle A
            for a, b in (edge swap candidates in cycle A)
```

[1] https://github.com/maxadamski/wiwiwi/tree/master/imo/3-local-optimization

```
                move := swap edges (a, succ(a)) and (b, succ(b))
                if delta(move) < 0 add move to new_moves

        if move swapped nodes a in A and b in B
                for x in cycle B
                        move := swap nodes b in cycle A and x in cycle B
                        if delta(move) < 0 add move to new_moves
                for x in cycle A
                        move := swap nodes x in cycle A and a in cycle B
                        if delta(move) < 0 add move to new_moves
        return new_moves
```

Lokalne przeszukiwanie wykorzystujące oceny ruchów z poprzednich iteracji

Pseudokod algorytmu lokalnego przeszukiwania wykorzystującego mechanizm przechowywania ocen ruchów z poprzednich iteracji jest opisany poniżej. Na początku procedura inicjalizuje listę moves identycznie jak algorytm lokalnego przeszukiwania w wersji stromej. Następnie z posortowanej rosnąco według delt, listy ruchów wybierany jest pierwszy aplikowalny ruch. Do listy ruchów w uporządkowany sposób dodawane są ruchy prowadzące do sąsiednich rozwiązań. Nowe ruchy zwraca procedura *next_moves*. Nieaplikowalne ruchy są usuwane z listy ruchów. Gdy nie znaleziono aplikowalnego ruchu w liście, procedura jest przerywana i zwracane jest rozwiązanie.

```
function search_memory(cities, cycles)
        moves := init_moves(cities, cycles) ordered by delta
        while true
                for move in moves
                        if move swaps edges u and v
                                if u and v do not exist in any cycle
                                        remove move from moves
                                if u and v exist and have the same direction
                                        remove move from moves
                                        best_move := move
                                        break
                        if move swaps nodes a in cycle c1 and b in cycle c2,
                                        given edges x1-a-y1 and x2-b-y2
                                if c1 == c2 or edges x1-a-y1 or x2-b-y2 don't exist
                                        remove move from moves
                                else
                                        remove move from moves
                                        best_move := move
                                        break
                if did not find best_move break
                moves := moves + next_moves(cities, cycles, best_move) ordered by delta
                cycles := best_move(cycles)
        return cycles
```

Lokalne przeszukiwanie wykorzystujące ruchy kandydackie

Poniższy algorytm opisuje algorytm lokalnego przeszukiwania wykorzystujący mechanizm ruchów kandydackich. Jego działanie można interpretować następująco. Dla każdego wierzchołka znajdowane jest **k** najbliższych wierzchołków. Następnie oceniane są ruchy polegające na dodaniu krawędzi pomiędzy daną parą wierzchołków i usunięciem krawędzi pomiędzy ich następnikami. Oraz ruchy polegające na wymianie wierzchołków pomiędzy cyklami. Ostatecznie wybierany jest ruch przynoszący najlepszą poprawę rozwiązania. Algorytm wykonuje się do czasu kiedy nie znaleziono żadnego ruchu przynoszącego poprawę rozwiązania.

```
function search_candidates(cities, cycles, k = 10)
      # for extra speed, precompute k closest cities here
      while true
            best_move, best_delta := null, 0
            for a in cities
                  for b in (k closest cities to a)
                        find cycles c1, c2 and indices i, j of nodes a and b
                        if c1 == c2
                              move := swap edges (a, succ(a)) and (b, succ(b))
                        else
                              move := swap nodes a and b
                        if delta(move) < best_delta
                              best_delta, best_move := delta(move), move
            if did not find best_move break
            cycles := best_move(cycles)
      return cycles
```

Wyniki i wnioski

Każdy algorytm był testowany 100 razy dla każdej instancji. Wyjątkowo, dla rozwiązań uzyskanych za pomocą heurystyki największego 2-żalu, za każdym razem wybierany był inny wierzchołek startowy (200 razy, nie w sposób losowy, ale po kolei). Natomiast dla algorytmów lokalnego przeszukiwania, lokalnego przeszukiwania wykorzystującego oceny ruchów z poprzednich iteracji, oraz lokalnego przeszukiwania wykorzystującego ruchy kandydackie, wspólne startowe rozwiązanie zostało wygenerowane losowo.

W algorytmie lokalnego przeszukiwania z wykorzystaniem ruchów kandydackich, za wartość parametru “k” przyjęto 10.

Najlepsze, najgorsze i średnie wyniki dotyczące długości ścieżek i czasów wykonania są przedstawione w poniższych tabelach. Rozwiązania z najkrótszymi długościami ścieżek zostały też zwizualizowane i zamieszczone na końcu raportu.

Zgodnie z przewidywaniami, mechanizm pamięci ocen ruchów i ruchów kandydackich przyspiesza wykonanie algorytmu lokalnego przeszukiwania. W średnim przypadku, przyspieszenie wynosi 39.3% (ruchy kandydackie) i 27.5% (pamięć). Najszybszym z algorytmem w każdym przypadku, okazało się lokalne przeszukiwanie z ruchami kandydackimi (nie biorąc pod uwagę heurystyki konstrukcyjnej). Dla przyjętej wartości k = 10, ruchy kandydackie pozwalają na osiągnięcie niewiele gorszego wyniku od stromego przeszukiwania. Pamięć ruchów mimo przyspieszenia generuje najgorsze wyniki spośród porównywanych algorytmów. Podejrzewamy, że może to mieć związek z procedurą generującą ruchy do sąsiednich rozwiązań. Gdyby zwracała ona więcej ruchów, z pewnością długość cykli by się skróciła, ale przy tym czas wykonania też by wzrósł.

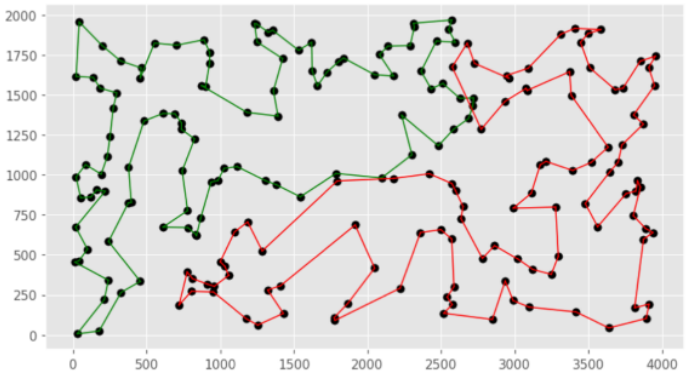
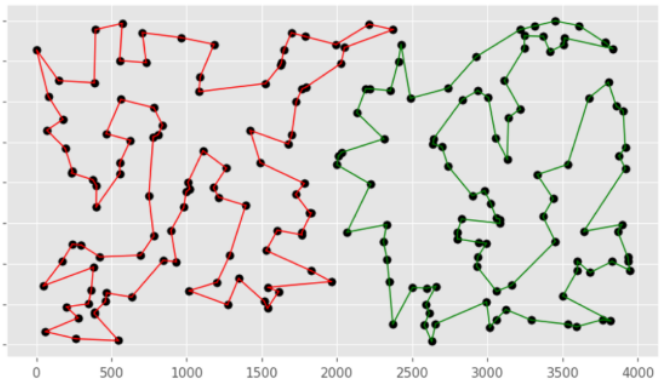
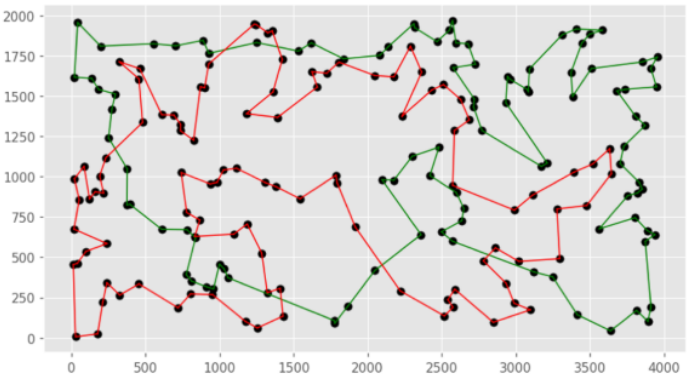
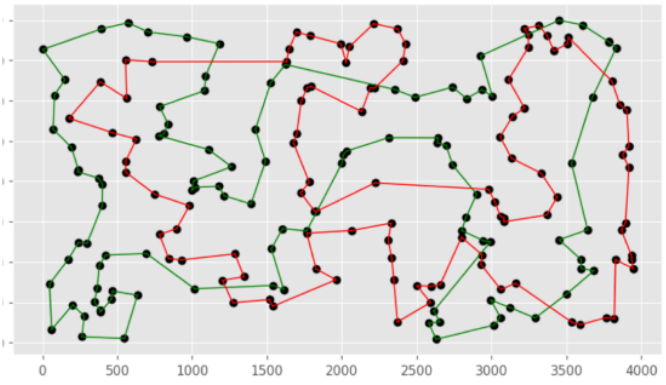
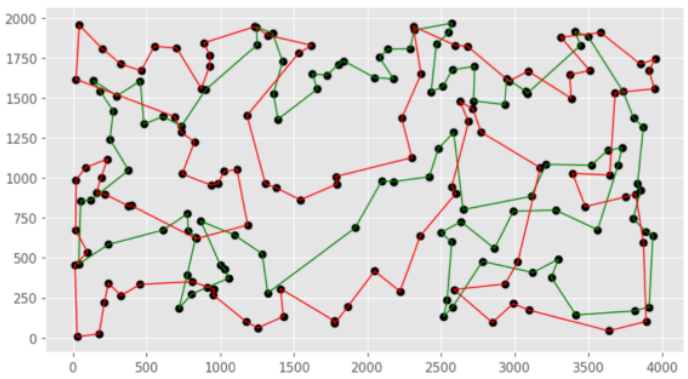
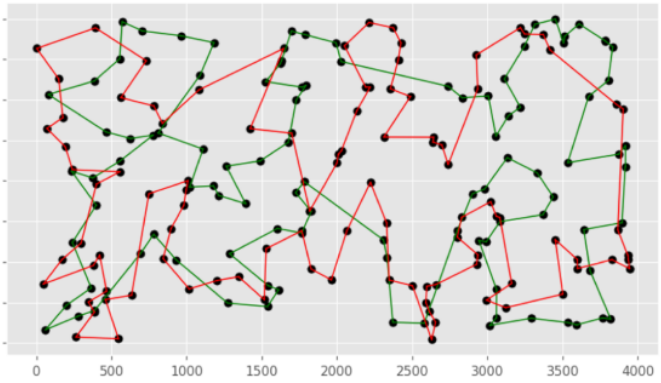
Podsumowując, ruchy kandydackie to prosty mechanizm pozwalający prawie dwukrotnie przyspieszyć lokalne przeszukiwanie, przy niewiele gorszych jakościowo rozwiązaniach.

Długości ścieżek:

	kroA200.tsp			kroB200.tsp		
	Minimum	Średnia	Maksimum	Minimum	Średnia	Maksimum
Heurystyka 2-żalu	**32631**	**34790**	**38907**	**32512**	**36299**	**38640**
Przeszukiwanie strome	**35152**	**38749**	**42396**	**36422**	**38554**	**41428**
Pamięć ocen ruchów	43529	45942	49333	43509	45541	48482
Ruchy kandydackie	36882	39911	42560	37853	39773	42433

Czasy wykonania:

	kroA200.tsp			kroB200.tsp		
	Minimum	Średnia	Maksimum	Minimum	Średnia	Maksimum
Heurystyka 2-żalu	**0.346042**	**0.403103**	**0.591658**	**0.343163**	**0.391453**	**0.634425**
Przeszukiwanie strome	9.583865	10.718405	12.048869	9.699372	10.966508	13.316899
Pamięć ocen ruchów	6.323468	7.761346	10.870667	6.777590	7.999278	9.792198
Ruchy kandydackie	**5.797556**	**6.506545**	**7.569556**	**5.658702**	**6.484034**	**7.879067**

	kroA200.tsp	kroB200.tsp
2-Regret		
Local Steep		
Local + Memory		
Local + Candidate