Politechnika Poznańska

Wydział Informatyki i Telekomunikacji

Instytut Informatyki

Optymalizacja metod lokalnego przeszukiwania

Inteligentne metody optymalizacji

Paweł Chumski, 144392
pawel.chumski@student.put.poznan.pl

Michał Ciesielski, 145325
michal.p.ciesielski@student.put.poznan.pl

Prowadzący:
prof. dr hab. inż. Andrzej Jaszkiewicz
andrzej.jaszkiewicz@cs.put.poznan.pl

16-04-2023

Spis treści

Wstęp

Celem zadania była poprawa efektywności lokalnego przeszukiwania, dla zmodyfikowanego problemu komiwojażera. Pierwszy wykorzystany algorytm opierał się na wykorzystaniu ocen ruchów z poprzedniej iteracji lokalnego przeszukiwania. Drugim z kolei mechanizmem były ruchy kandydackie znajdowane dla k najbliższych wierzchołków.

Wszystkie badane algorytmy wykorzystują zarówno ruchy między-, jak i wewnątrztrasowe. Na podstawie poprzednich zajęć ruchy wewnątrztrasowe obejmują wymiany dwóch krawędzi, gdyż ten wariant okazał się lepszy niż wymiana wierzchołków. Pozostałe dwa wykorzystane algorytmy do porównania to heurystyka konstrukcyjna oparta na największym ważonym 2-żalu oraz algorytm lokalnego przeszukiwania w wersji stromej. Każdy z czterech algorytmów na każdej instancji wejściowej uruchamiany był co najmniej 100 razy, a w przypadku algorytmów lokalnego przeszukiwania rozwiązaniami startowymi były rozwiązania losowe.

Eksperymenty zostały przeprowadzone na instancjach kroA200.tsp oraz kroB200.tsp z biblioteki TSPLib.

Algorytmy lokalnego przeszukiwania poprawiające efektywność

1.1 Funkcja pomocnicze - liczenie delty oraz metoda next_moves

Do implementacji wskazanych algorytmów lokalnego przeszukiwania konieczne było wykorzystanie funkcji obliczających deltę funkcji celu. Poza tymi funkcjami dość istotna była również implementacja metody next_moves, wykorzystywanej w algorytmie wykorzystującym oceny ruchów z poprzednich iteracji. Poniżej prezentujemy jej pseudokod:

```
1  new_moves := [];
2  if move_type == SWAP_EDGES then
3  |  for (a, b) in (swap_candidates in cycle 1st or 2nd) do
4  |  |  move := swap edges (a, succ(a)) and (b, succ(b));
5  |  |  if delta(move) < 0 then
6  |  |  |  add move to new_moves list;
7  |  |  end
8  |  end
9  end
10 if (move_type == SWAP_NODES) and (a in 1st cycle) and (b in 2nd cycle) then
11 |  for x in cycle 2nd do
12 |  |  move := swap nodes a and x;
13 |  |  if delta(move) < 0 then
14 |  |  |  add move to new_moves list;
15 |  |  end
16 |  end
17 |  for x in cycle 1st do
18 |  |  move := swap nodes b and x;
19 |  |  if delta(move) < 0 then
20 |  |  |  add move to new_moves list;
21 |  |  end
22 |  end
23 end
24 return new_moves;
```

1.2 Lokalne przeszukiwanie wykorzystujące oceny ruchów z poprzednich iteracji

1 Opis algorytmu

Implementacja algorytmu w postaci pseudokodu znajduje się poniżej. Ten algorytm lokalnego przeszukiwania od wersji stromej wyróżnia aplikacja pierwszego możliwego ruchu z listy posortowanych według wartości delta ruchów. Nieaplikowalne ruchy usuwane są z listy możliwych ruchów, a do listy ruchów następnie w uporządkowany sposób dodawane są nowe ruchy za pomocą metody next_moves. Warunkiem stopu jest brak znalezienia aplikowalnego ruchu.

2 Pseudokod

```
1  moves := find initial moves ordered by delta;
2  while True do
3      for move in moves do
4          if move_type == SWAP_EDGES then
5              a, b - edges given from move;
6              if a and b edges don't exist in any cycle then
7                  remove move from moves list;
8              end
9              if a and b exist and have the same direction or opposite then
10                 remove move from moves list;
11                 best_move := move;
12                 break;
13             end
14         end
15         if move_type == SWAP_NODES then
16             a, b - nodes given in cycle1 and cycle2 from move;
17             x1-a-y1, x2-b-y2 - given edges in cycle1 and cycle2 from move;
18             if cycle1 == cycle2 or edges (x1-a-y1 or x2-b-y2) don't exist then
19                 remove move from moves;
20             end
21             else
22                 remove move from moves;
23                 best_move := move;
24                 break;
25             end
26         end
27     end
28     if best_move doesn't exist then
29         break;
30     end
31     moves := moves + next_moves(length_matrix, cycles, best_move) ordered by delta;
32     apply_best_move(cycles, best_move);
33 end
34 return cycles;
```

1.3 Lokalne przeszukiwanie wykorzystujące ruchy kandydackie

1 Opis algorytmu

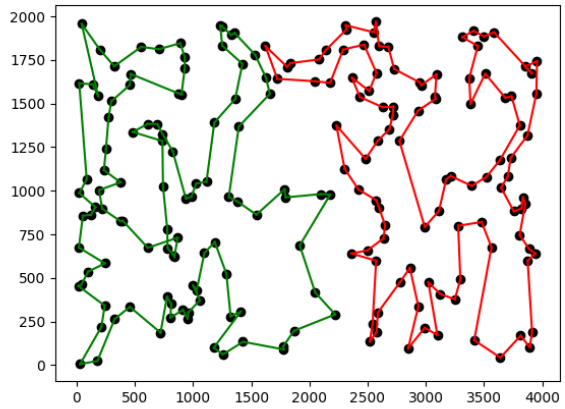
Prezentowany algorytm wyróżnia wykorzystanie mechanizmu ruchów kandydackich. Jako kandydackie oznaczamy ruchy wprowadzające do rozwiązania co najmniej jedną krawędź kandydacką. Krawędzie kandydackie definiowane są w naszym przypadku wyznaczając dla każdego wierzchołka k=10 innych najbliższych wierzchołków. Następnie oceniane są ruchy polegające na dodaniu krawędzi pomiędzy daną parą wierzchołków i usunięciem krawędzi pomiędzy ich następnikami oraz ruchy polegające na wymianie wierzchołków między cyklami. Ostatecznie wybierany jest ruch przynoszący najlepszą poprawę rozwiązania.

2 Pseudokod

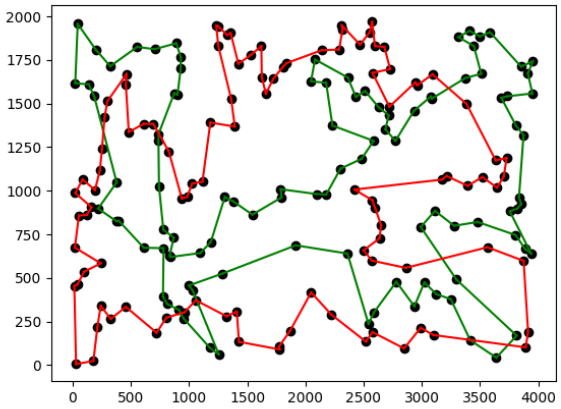
```
1  find k nearest nodes for all available nodes;
2  while True do
3  |  best_move := null;
4  |  best_delta := 0;
5  |  for a in nodes do
6  |  |  for b in (k closest nodes to a) do
7  |  |  |  find cycles c1_index, c2_index and indices i, j of nodes a and b, respectively;
8  |  |  |  if c1_index == c2_index then
9  |  |  |  |  move := swap edges (a, succ(a)) and (b, succ(b));
10 |  |  |  end
11 |  |  |  else
12 |  |  |  |  move := swap nodes a and b;
13 |  |  |  end
14 |  |  |  if delta(move) < best_delta then
15 |  |  |  |  best_delta := delta(move);
16 |  |  |  |  best_move := move;
17 |  |  |  end
18 |  |  end
19 |  end
20 |  if best_move doesn't exist then
21 |  |  break;
22 |  end
23 |  apply_best_move(cycles, best_move);
24 end
25 return cycles;
```

Wizualizacje najlepszych rozwiązań

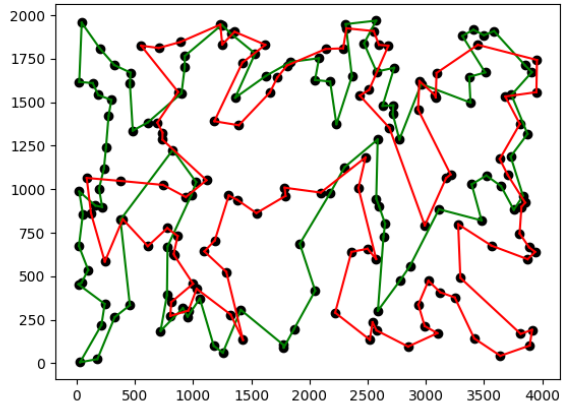
2.1 kroA200



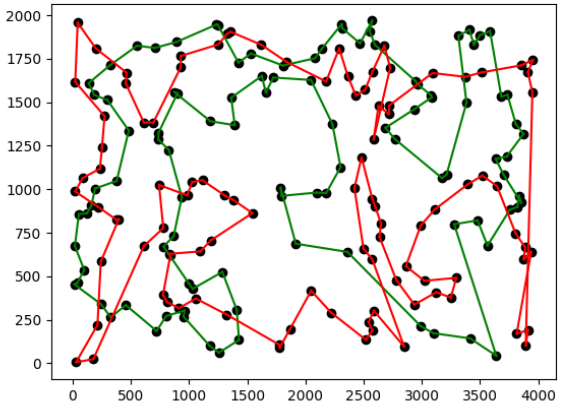
Rys. 1. Weighted 2-Regret



Rys. 2. Steepest Search

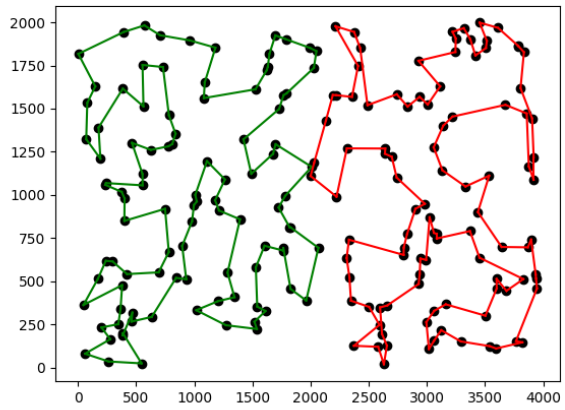


Rys. 3. Memory Search

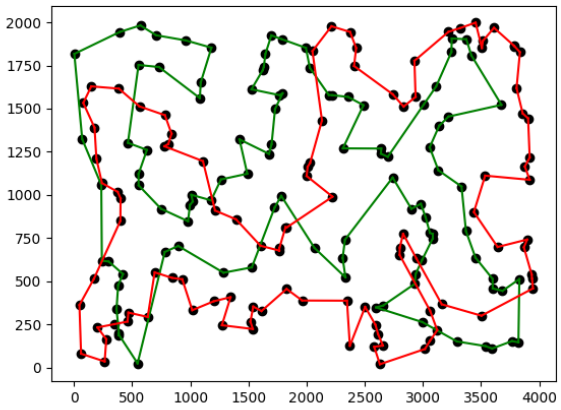


Rys. 4. Candidates Search

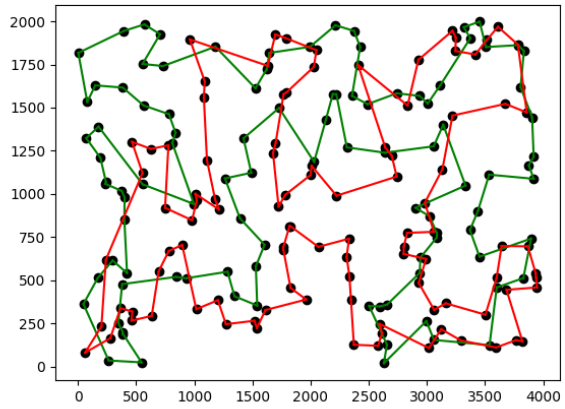
2.2 kroB200



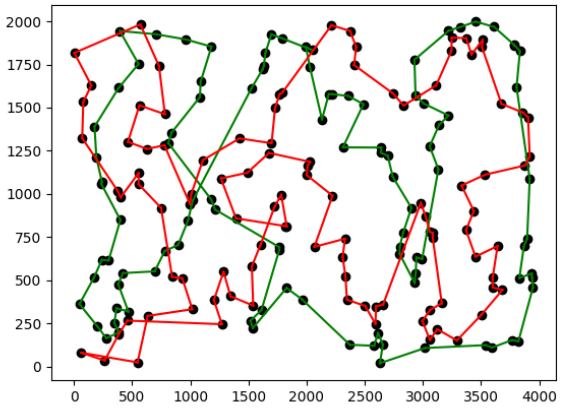
Rys. 5. Weighted 2-Regret



Rys. 6. Steepest Search



Rys. 7. Memory Search



Rys. 8. Candidates Search

Porównanie wyników eksperymentu obliczeniowego

Każdy algorytm był testowany 100 razy dla każdej instancji. Dla rozwiązań początkowych uzyskiwanych za pomocą ważonego 2-Żalu wyjątkowo zawsze był wybierany po kolei inny wierzchołek startowy (czyli ta heurystyka była uruchamiana 200 razy, za każdym razem dla innego wierzchołka startowego). Natomiast dla algorytmów lokalnego przeszukiwania rozwiązanie startowe było generowane za każdym razem losowo.

Tab. 1. Tabela zawierająca długości ścieżek dla poszczególnych algorytmów

Instancja	kroA200.tsp			kroB200.tsp		
Algorytm	Minimum	Średnia	Maksimum	Minimum	Średnia	Maksimum
Weighted 2-Regret	33457	37078	40314	33082	37179	39092
Steepest	36344	38716	42183	36580	38688	41463
Memory	42355	45848	49629	42799	45673	47659
Candidates	40599	46258	55668	38787	43835	51434

Tab. 2. Tabela zawierająca czasy wykonania dla poszczególnych algorytmów

Instancja	kroA200.tsp			kroB200.tsp		
Algorytm	Minimum [s]	Średnia [s]	Maksimum [s]	Minimum [s]	Średnia [s]	Maksimum [s]
Weighted 2-Regret	0.689521	1.226877	1.857841	0.565515	1.196540	1.477999
Steepest	26.918967	36.251207	43.196131	26.344129	35.443091	43.202985
Memory	16.411431	22.852724	28.066025	17.321161	23.590197	32.437272
Candidates	7.749410	12.953810	15.820175	9.085585	14.022290	18.447464

Podsumowanie

Wnioski

- Dla wariantów algorytmu lokalnego przeszukiwania wykorzystujących listę zapamiętanych ruchów oraz ruchy kandydackie czas wykonania jest krótszy niż w przypadku algorytmu lokalnego przeszukiwania w wersji stromej.
- Najszybszym algorytmem okazała się heurystyka oparta na ważonym 2-żalu, spośród algorytmów lokalnego przeszukiwania najszybszy okazał się wariant wykorzystujący ruchy kandydackie.
- Algorytm zapamiętujący listę ruchów przynoszących poprawę, mimo większego poziomu skomplikowania okazał się wolniejszy niż algorytm wykorzystujący mechanizm ruchów kandydackich, a także przynoszący zazwyczaj gorsze rezultaty.

Kod programu

https://github.com/imo/lab3