IMO - Zadanie 1

Autorzy: Dariusz Max Adamski, Sławomir Gilewski

Wprowadzenie

W sprawozdaniu opisane są trzy algorytmy rozwiązujące zmodyfikowany problem komiwojażera. Algorytmy znajdują dwa cykle w zbiorze wierzchołków, minimalizując łączną długość cykli. Wynikowe cykle mają równą ilość wierzchołków jeśli całkowita liczba wierzchołków jest parzysta, w przeciwnym wypadku pierwszy cykl jest o jeden wierzchołek dłuższy. Efektywność algorytmów była oceniana na podstawie instancji “kroA100” i “kroB100” z biblioteki TSPLib.

Kod źródłowy dostępny on-line: https://github.com/maxadamski/wiwiwi/tree/master/imo/zadanie1

Algorytmy

Wszystkie opisane algorytmy mają podobną zasadę działania. Na początku, w zmiennej **remaining** tworzona jest lista wierzchołków które nie zostały jeszcze dodane do cyklu. Tworzone są też dwa puste cykle w zmiennej **cycles**. Następnie startowy wierzchołek **start** (przekazywany jako argument funkcji obliczającej cykle) jest usuwany z listy **remaining** i dodawany do pierwszego cyklu (**cycles**[0]). Do drugiego cyklu (**cycles**[1]) dodawany jest wierzchołek znajdujący się najdalej od wierzchołka startowego - ten wierzchołek też jest usuwany z listy remaining. Dla trzeciego algorytmu dodatkowo do każdego cyklu jest dodawany jeden wierzchołek najbliższy pierwszemu wierzchołkowi.

Po dodaniu pierwszych wierzchołków do cykli, do każdego cyklu naprzemiennie dodawane są *najlepsze* według danej metody wierzchołki, na *najlepszych* według metody pozycjach w cyklu.

Po usunięciu wszystkich wierzchołków z listy **remaining**, zwracany jest wynik.

W zmiennej **distance** przechowywana jest macierz odległości euklidesowej pomiędzy wierzchołkami z danej instancji.

Najbliższy sąsiad

Pierwszy algorytm jest oparty na prostej heurystyce najbliższego sąsiada - dla danego cyklu wybierana jest para wierzchołków - jeden z cyklu i jeden z listy **remaining** których odległość w macierzy **distance** jest najmniejsza. Po wybranym wierzchołku z cyklu wstawiany jest wybrany wierzchołek z listy **remaining**. Po tym wybrany wierzchołek jest usuwany z listy **remaining**.

```
function solve-nn(start)
	remaining, cycles = list of all cities, two empty cycles
	remove start from remaining and add it to cycles[0]
	remove city farthest from start from remaining and add it to cycles[1]
	while remaining is not empty
		for cycle in cycles
			x, i = argmin for x in remaining for i in 0..|cycle| : distance[cycle[i], x]
			remove x from remaining and add it to cycle at index i
	return cycles
```

Rozbudowa cyklu

Drugi algorytm działa podobnie do algorytmu najbliższego sąsiada. Różnicą jest wybór pary wierzchołków z cyklu i listy **remaining** - zamiast wyboru wierzchołków z najmniejszą odległością wybierany jest wierzchołek, który możliwie najmniej zwiększy całkowitą długość cyklu. Wzrost długości cyklu jest obliczany procedurą *score-delta*. Zauważyliśmy, że aby obliczyć wzrost długości cyklu przez dodanie wierzchołka **c** pomiędzy wierzchołkami **a** i **b** nie trzeba obliczać długości całego cyklu przed i po dodaniu **c**, a jedynie dodać odległości z **a** do **c**, z **c** do **b** i odjąć od nich odległość z **a** do **b**. Ta obserwacja pozwoliła uniknąć zbędnego narzutu czasowego w naszej implementacji.

```
function score-delta(cycle, city, index)
	a = cycle[(index - 1) mod |cycle|]
	b = cycle[index]
	return distance[a, city] + distance[city, b] - distance[a, b]

function solve-cycle(start)
	remaining, cycles = list of all cities, two empty cycles
	remove start from remaining and add it to cycles[0]
	remove city farthest from start from remaining and add it to cycles[1]
	while remaining is not empty
```

```
        for cycle in cycles
            x, i = argmin for x in remaining for i in 0..|cycle| : score-delta(cycle, x, i)
            remove x from remaining and add it to cycle at index i
    return cycles
```

Największy 2-żal

Trzeci algorytm bazuje na algorytmie rozbudowy cyklu. Na początku do każdego cyklu dodawany jest najbliższy wierzchołek do początkowego wierzchołka w tym cyklu. Następnie, dopóki lista **remaining** nie jest pusta, dla każdego cyklu obliczana jest macierz przyrostów długości cyklu **scores**, gdzie wartość **scores**[i][j] to przyrost długości cyklu, gdyby wstawić wierzchołek **remaining**[i] do cyklu na j-tej pozycji. Obliczany jest także wektor **regret** którego i-ta wartość to 2-żal i-tego wierzchołka z listy **remaining**. Warto wspomnieć, że w naszej implementacji rzędy macierzy scores nie są w pełni sortowane do obliczenia 2-żalu, a jedynie wybierane są 2 najmniejsze elementy (zrealizowane macierzowo przy użyciu funkcji *numpy.partition*) - w pseudokodzie sortujemy cały rząd dla prostoty zapisu. Po wybraniu wierzchołka z największym 2-żalem, wybierana jest pozycja w cyklu, która minimalnie zwiększy jego całkowitą długość.

```
function solve-regret(start)
  remaining, cycles = list of all cities, two empty cycles
  remove start from remaining and add it to cycles[0]
  remove city farthest from start from remaining and add it to cycles[1]
  for cycle in cycles
    remove city nearest to cycle[0] from remaining and add it to cycle
  while remaining is not empty
    for cycle in cycles
      # scores is a 2D matrix where rows are cities, and columns are cycle indices
      scores = for x in remaining for i in 0..|cycle| : score-delta(cycle, x, i)
      # regret is a vector where each element is the 2-regret of a remaining city
      regret = differences of second and first columns of (scores with sorted rows)
      x = argmax for i in 0..|remaining| : regret[i]
      i = argmin for i in 0..|cycle| : scores[x, i]
      remove city at index x from remaining and add it to cycle at index i
  return cycles
```

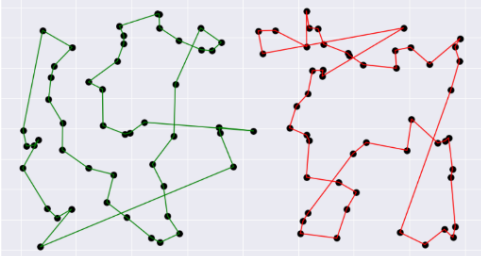
Wyniki

Każdy algorytm był testowany 100 razy dla każdej instancji. Za każdym razem wybierany był inny wierzchołek startowy (nie w sposób losowy, ale po kolei). Najlepsze, najgorsze i średnie wyniki są przedstawione w poniższej tabeli. Najlepsze rozwiązania zostały też zwizualizowane i zamieszczone poniżej.

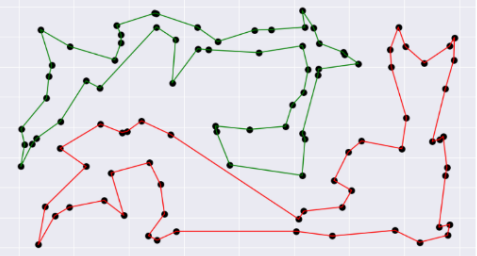
	kroA100.tsp			kroB100.tsp		
Algorytm	Minimum	Średnia	Maksimum	Minimum	Średnia	Maksimum
Najbliższy sąsiad	28761	32609	35175	29852	32574	34424
Rozbudowa cyklu	26304	28708	29980	27180	28538	30197
Największy 2-żal	**22914**	**26840**	**29510**	**24172**	**27767**	**29378**

kroA100.tsp

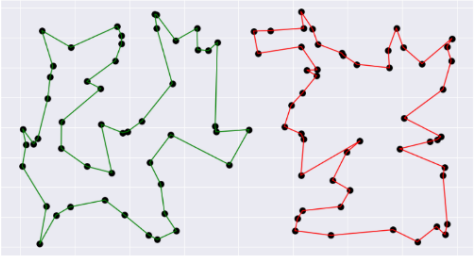
Greedy Nearest



Greedy Cycle

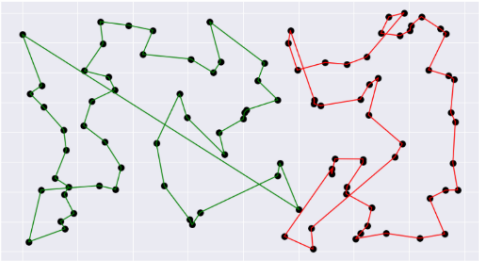


2-Regret

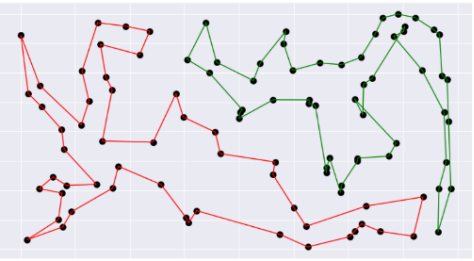


kroB100.tsp

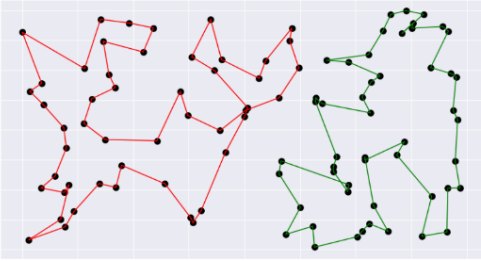
Greedy Nearest



Greedy Cycle



2-Regret



Wnioski

Zgodnie z oczekiwaniami, algorytm najbliższego sąsiada osiąga najgorsze wyniki w każdym przypadku. Na obu instancjach najbardziej efektywnym algorytmem okazał się algorytm korzystający z heurystyki 2-żalu. Użycie tego algorytmu przełożyło się na wynikowe cykle o najmniejszej łącznej długości, zarówno w średnim jak i w najlepszym przypadku. Najgorsze rozwiązanie proponowane przez algorytm największego 2-żalu jest gorsze od najgorszego rozwiązania algorytmu rozbudowy cyklu, ale jest lepsze od najgorszego rozwiązania algorytmu najbliższego sąsiada. Według nas może być to spowodowane przez sposób doboru pierwszych dwóch wierzchołków w cyklu w tym algorytmie lub przez naprzemienne dodawanie do cykli wierzchołków. Alternatywnie moglibyśmy dostosować algorytmy rozwiązujące problem komiwojażera przez podzielenie na początku wszystkich wierzchołków na dwie grupy, a później zbudowanie oddzielnie dwóch cykli z tych grup.