IMO - Zadanie 2

Autorzy: Sławomir Gilewski, Dariusz Max Adamski

Wprowadzenie

W sprawozdaniu opisane są algorytmy lokalnego przeszukiwania, rozwiązujące zmodyfikowany problem komiwojażera. Algorytmy te startują z wybranego rozwiązania początkowego, a następnie przeglądają przestrzeń rozwiązań (sąsiedztwo) dostępnych w danym kroku, akceptując jedynie rozwiązania przynoszące poprawę, tak długo jak to jest możliwe.

Rozważano algorytmy lokalnego przeszukiwania w wersjach stromej (steepest) i zachłannej (greedy). W obydwu wersjach zbiór możliwych ruchów stosowanych do bieżącego rozwiązania obejmuje ruchy dwóch rodzajów - międzytrasowe oraz wewnątrztrasowe. Dla ruchów wewnątrztrasowych możliwe są dwa rodzaje ruchów - wymiana dwóch wierzchołków lub wymiana dwóch krawędzi. Istnieją zatem 4 możliwe kombinacje - wersje lokalnego przeszukiwania: greedy-wymiana wierzchołków, greedy-wymiana krawędzi, steepest-wymiana wierzchołków oraz steepest-wymiana krawędzi.

Wyniki zaimplementowanych algorytmów badano startując z rozwiązań losowych oraz rozwiązań uzyskanych za pomocą heurystyki “Największy 2-żal” (algorytm rozbudowy cyklu), na podstawie instancji “kroA100” i “kroB100” z biblioteki TSPLib.

Dodatkowo, jako punkt odniesienia wykorzystano algorytm losowego błądzenia, który w każdej iteracji wykonuje losowo wybrany ruch (niezależnie od jego oceny) i zwraca najlepsze znalezione w ten sposób rozwiązanie. Algorytm ten działa w takim samym czasie jak średnio najwolniejsza z wersji lokalnego przeszukiwania.

Kod źródłowy dostępny on-line: https://github.com/Uxell/LocalSearch-TSP

Algorytmy delty funkcji celu

Do badania jakości proponowanego rozwiązania wykorzystano algorytmy obliczające deltę funkcji celu w trzech wariantach: wymiana wierzchołków między cyklami, wymiana wierzchołków wewnątrz cyklu, wymiana krawędzi wewnątrz cyklu. Każdy z nich, w zależności od badanego sąsiedztwa, liczy różnicę wyników po rozwiązaniu i przed rozwiązaniem, zatem delta ta będzie minimalizowana. Algorytmy te przejawiają się minimalnym poziomem skomplikowania, zatem ich pseudokod w większości pokrywa się z kodem programu.

W zmiennej **distance** przechowywana jest macierz odległości euklidesowej pomiędzy wierzchołkami z danej instancji
Zmienna **cycle** lub **cycles** to cykl lub cykle (w przypadku delty wymiany między cyklami), przechowujące indeksy wierzchołków w macierzy **distance**
Zmienne **i** oraz **j** to indeksy wierzchołków, które chcemy wymienić. W przypadku wymiany krawędzi **i** to wierzchołek końcowy pierwszej krawędzi, a **j** to pierwszy wierzchołek drugiej krawędzi.

Delta wstawienia wierzchołka **city** na miejscu **i** cyklu **cycle**:

```
function delta_replace_vertex(cities, cycle, i, city):
	a, b, c = cycle[(i - 1) mod |cycle|], cycle[i], cycle[(i+1) mod |cycle|]
	return distance[a, city] + distance[city, c] - distance[a, b] - distance[b, c]
```

Delta wymiany wierzchołków o indeksach **i** oraz **j** pomiędzy dwoma cyklami **cycle[0]** oraz **cycle[1]**:

```
function delta_replace_vertices_outside(cities, cycles, i, j):
	return delta_replace_vertex(cities, cycles[0], i, cycles[1][j]) +
		delta_replace_vertex(cities, cycles[1], j, cycles[0][i])
```

Delta wymiany wierzchołków o indeksach **i** oraz **j** cyklu **cycle:**

```
function delta_replace_vertices_inside(cities, cycle, i, j):
	a, b, c = cycle[(i - 1) mod |cycle|], cycle[i], cycle[(i + 1) mod |cycle|]
	d, e, f = cycle[(j - 1) mod |cycle|], cycle[j], cycle[(j + 1) mod |cycle|]
	if j-i == 1:
		return distance[a,e]+distance[b,f]-distance[a,b]-distance[e,f]
	elif (i, j) == (0, |cycle|-1):
		return distance[e,c]+distance[d,b]-distance[b,c]-distance[d,e]
	else:
		return distance[a,e] + distance[e,c] + distance[d,b] + distance[b,f]
			-distance[a,b] - distance[b,c] - distance[d,e] - distance[e,f]

function delta_replace_edges_inside(cities, cycle, i, j):
	if (i,j) == (0, |cycle|-1):
		a, b, c, d = cycle[i], cycle[(i+1) mod |cycle|], cycle[(j-1) mod |cycle|], cycle[j]
	else:
		a, b, c, d = cycle[(i - 1) mod |cycle|], cycle[i], cycle[j], cycle[(j+1) mod |cycle|]
	return distance[a, c] + distance[b, d] - distance[a, b] - distance[c, d]
```

Generatory par wierzchołków

Nasze algorytmy przeszukiwania wykorzystują generatory par wierzchołków, które dla wybranego przeszukiwania zwracają odpowiednią listę par indeksów kandydatów, które można sprawdzić.

Kandydaci na wymianę wierzchołków między cyklami:

```
function outside_candidates(cycles):
	cycle[0], cycle[1] = cycles
	return every possible permutation of indices between cycle[0] and cycle[1]
```

Kandydaci na wymianę wierzchołków/krawędzi w ramach jednego cyklu:

```
function inside_candidates(cycle):
	return every possible combination of indices in cycle
```

Algorytmy lokalnego przeszukiwania

W obydwóch wersjach przeszukiwania, dostępne ruchy przeglądane są systematycznie tak, aby ostatecznie przejrzeć całe sąsiedztwo. W wersji zachłannej kolejność przeglądania powinna być losowa, a akceptowane jest pierwsze modyfikacja wprowadzająca poprawę rozwiązania. W wersji stromej natomiast, kolejność nie ma znaczenia, ponieważ akceptowana jest modyfikacja wprowadzająca najbardziej znaczącą poprawę.

Niezależnie od wersji, na początku wybierany jest wariant ruchów wewnątrztrasowych, tak aby później jedynie wykorzystać odpowiednią funkcję dotyczącą przeszukiwania wewnątrztrasowego. Funkcje dotyczące delty oraz wymiany wybranego wariantu przechowywane są w **delta_replace_inside** oraz **replace_inside**

Algorytm w wersji zachłannej

Ogólny algorytm:

```
repeat for every cantidate solution:
	if a new, better solution is found:
		replace the previous solution with a new one
until no better solution is found after browsing every candidate
```

Wymiana wierzchołków między cyklami w wersji zachłannej (pierwsza poprawa):

```
function outside_vertices_trade_first(cities, cycles):
	candidates = outside_candidates(cycles)
	shuffle the candidates
	for i, j in candidates:
		delta = delta_replace_vertices_outside(cities, cycles, i, j)
		if delta < 0:
			replace_vertices_outside(cycles, i, j)
			return delta
	return delta
```

Wymiana wewnątrz cyklu w wersji zachłannej (pierwsza poprawa):

```
function inside_trade_first(cities, cycles):
	shuffle the two cycles order
	for cycle in cycles:
		candidates = inside_candidates(cycle)
		shuffle the candidates
		for i, j in candidates:
			delta = delta_replace_inside(cities, cycle, i, j)
			if delta < 0:
				replace_inside(cycle, i, j)
				return delta
	return delta
```

Główna funkcja przeszukująca:

```
function greedy_search(cities, cycles):
	while True:
		moves = [outside_vertices_trade_first, inside_trade_first]
		shuffle the moves
		delta = outside_vertices_trade_first(cities, cycles)
		if delta >= 0:
			delta = inside_vertices_trade_first(cities, cycles)
			if score >= 0:
				break
	return cycles
```

Algorytm w wersji stromej

Ogólny algorytm:

```
repeat:
	Find the best, new solution
	if the new solution is better than the previous one:
		Replace the previous solution with a new one
until no better solution is found after browsing every candidate
```

Wymiana wierzchołków między cyklami w wersji stromej (najlepsza poprawa):

```
function outside_vertices_trade_best(cities, cycles):
	candidates = outside_candidates(cycles)
	delta_scores = delta_replace_vertices_outside for every indices pair from candidates
	delta = minimum(delta_scores)
	if delta < 0:
		return delta, solution
return delta, _
```

Wymiana wewnątrz cyklu w wersji stromej (najlepsza poprawa):

```
function inside_trade_best(cities, cycles):
	delta_scores = delta_replace_inside for every pair from
			inside_candidates for every cycle in cycles
	delta = minimum(delta_scores)
	if delta < 0:
		return delta, solution
	return delta, _
```

Główna funkcja przeszukująca:

```
function steepest_search(cities, cycles, time_limit):
	while True:
		moves = [outside_vertices_trade_best, inside_trade_best]
		delta_scores, solutions = score for every move from moves
		delta = minimum(delta_scores)
		if delta < 0:
			Apply the solution with the minimal delta
		else:
			break
	return cycles
```

Algorytm losowego błądzenia

Jako punkt odniesienia zaimplementowano również algorytm losowego błądzenia, który w każdej iteracji wykonuje losowo wybrany ruch (niezależnie od jego oceny) i zwraca najlepsze znalezione w ten sposób rozwiązanie. Algorytm ten działa w takim samym czasie jak średnio najwolniejsza z wersji lokalnego przeszukiwania na danej instancji.

```
function random_search(cities, cycles, time_limit):
	best_solution, best_score = cycles, score(cycles)
	while total_time < time_limit:
		moves = [outside_vertices_trade, inside_vertices_trade, inside_edges_trade]
		Make a random move from moves
		new_score = score(cycles)
		if new_score < best_score:
			best_solution = cycles
			best_score = new_score
	return best_solution
```

Wyniki

Każdy algorytm był testowany 100 razy dla każdej instancji. Dla początkowych rozwiązań uzyskanych za pomocą heurystyki “Największy 2-żal”, za każdym razem wybierany był inny wierzchołek startowy (nie w sposób losowy, ale po kolei). Natomiast dla początkowych rozwiązań losowych, za każdym razem losowane było inne rozwiązanie losowe.

Najlepsze, najgorsze i średnie wyniki dotyczące długości ścieżek i czasów wykonania są przedstawione w poniższych tabelach. Przykładowe rozwiązania zostały też zwizualizowane i zamieszczone na końcu raportu. Dla porównania zamieszczono również wyniki przed przeszukiwaniem losowym.

Długości ścieżek:

		kroA100.tsp			kroB100.tsp		
Rozwiązanie początkowe	Losowe przeszukiwanie	Minimum	Średnia	Maksimum	Minimum	Średnia	Maksimum
Losowe	Brak	150156	171194	188586	145224	168350	186964
	Losowe błądzenie	132156	145426	154956	136446	143825	149530
	Zachłanny-wierzchołki	30760	40189	48913	32591	40272	48730
	Stromy-wierzchołki	32873	42522	51302	34312	42636	50953
	Zachłanny-krawędzie	**24984**	**27659**	30274	**25363**	**28215**	**31131**
	Stromy-krawędzie	25218	27714	**30081**	26421	28423	31163
2-Żal	Brak	22914	26840	29510	24172	27767	29378
	Zachłanny-wierzchołki	22733	25352	28088	22968	25967	28569
	Stromy-wierzchołki	22733	25351	27447	23146	25966	28569
	Zachłanny-krawędzie	**22278**	24969	27924	**22741**	25611	28419
	Stromy-krawędzie	**22278**	**24916**	**27380**	**22741**	**25378**	**28237**

Czasy wykonania:

		kroA100.tsp			kroB100.tsp		
Rozwiązanie początkowe	Losowe przeszukiwanie	Minimum	Średnia	Maksimum	Minimum	Średnia	Maksimum
Losowe	Zachłanny-wierzchołki	**0.3448**	0.7167	**1.1114**	**0.2908**	0.7355	1.2924
	Stromy-wierzchołki	1.3182	2.3053	2.8507	1.3990	2.3539	3.1089
	Zachłanny-krawędzie	0.3695	**0.6630**	1.1287	0.3106	**0.6617**	**1.0833**
	Stromy-krawędzie	1.0016	1.6783	2.0758	0.8643	1.6560	1.9744
2-Żal	Zachłanny-wierzchołki	**0.0191**	0.1236	0.3626	**0.0328**	0.1387	0.4605
	Stromy-wierzchołki	0.0361	0.2371	0.6547	0.0237	0.2113	0.5635
	Zachłanny-krawędzie	0.0272	**0.0966**	**0.2072**	0.0343	**0.1242**	**0.3463**
	Stromy-krawędzie	0.0521	0.2121	0.4595	0.0376	0.2317	0.5306

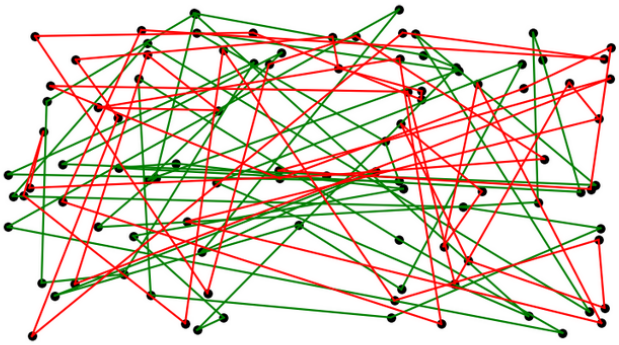
Wnioski

- Zgodnie z oczekiwaniami, algorytm zachłanny osiąga najlepsze czasy w każdym przypadku. Wygląda na to, że czas przeszukiwania nie zależy od wariantu wewnątrztrasowego, ponieważ najlepsze wyniki są rozłożone pomiędzy te warianty.
- Jeżeli chodzi o porównanie wariantów wewnątrztrasowego, wymiana dwóch krawędzi otrzymuje znacząco lepsze wyniki od wymiany wierzchołków. Nie ma żadnej sytuacji, w której wariant wymiany wierzchołków otrzymuje lepszy wynik końcowy.
- Jeżeli chodzi o punkty startowe, wyniki są bardzo interesujące:
    - Okazuje się, że dla losowego rozwiązania początkowego, lepsze minimum prawie zawsze osiąga algorytm zachłanny. Maksymalne i średnie wyniki są zazwyczaj do siebie zbliżone.
    - Natomiast dla dosyć dobrego rozwiązania początkowego osiągniętego algorytmem 2-żal, minimum jest takie samo dla obydwóch instancji, a w przypadku średnich i maksymalnch wyników, to algorytm stromy ma przewagę.
- Potwierdza to naszą tezę, że algorytm zachłanny działa szybciej i osiąga gorsze wyniki przy pojedynczych uruchomieniach, jednak gdy możemy go uruchomić kilka(kilkaset) razy, jest w stanie osiągać wyniki zbliżone lub lepsze niż algorytm stromy. Algorytm zachłanny lepiej sobie również radzi na losowych, “świeżych” rozwiązaniach początkowych.
- W przypadku rozwiązań początkowych, które mają dobre wyniki jeszcze przed uruchomieniem przeszukiwania, lepiej stosować algorytm stromy.

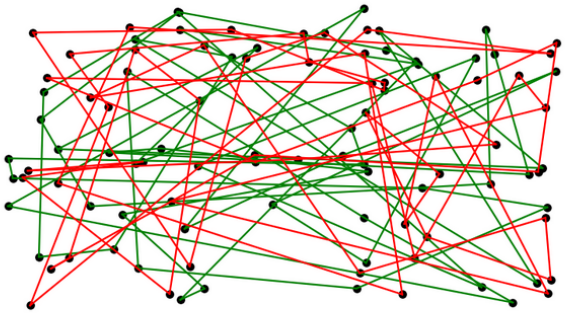
Wizualizacje

kroA100.tsp

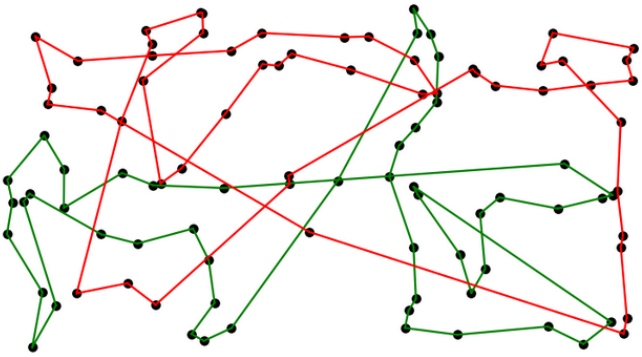
Losowe Początkowe



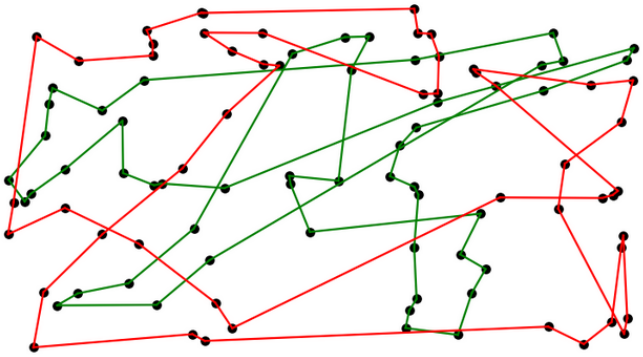
Losowe Błądzenie



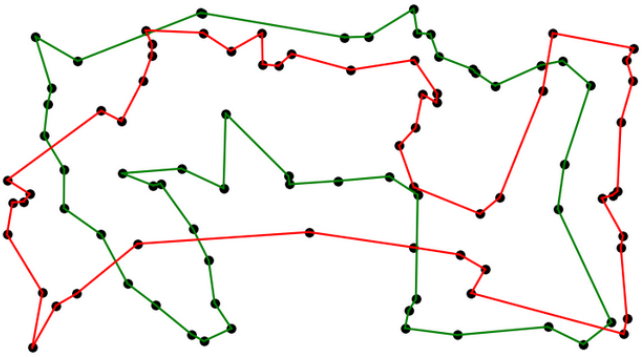
Zachłanny - wierzchołki



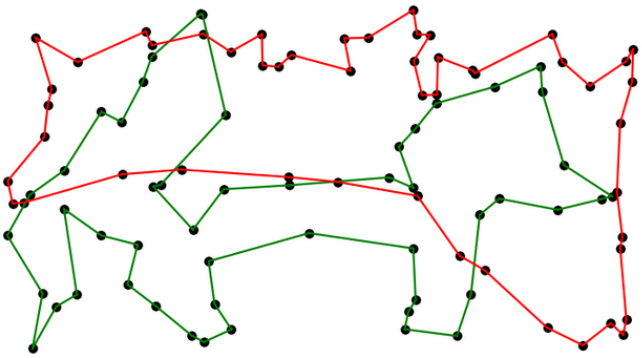
Stromy - wierzchołki



Zachłanny - krawędzie

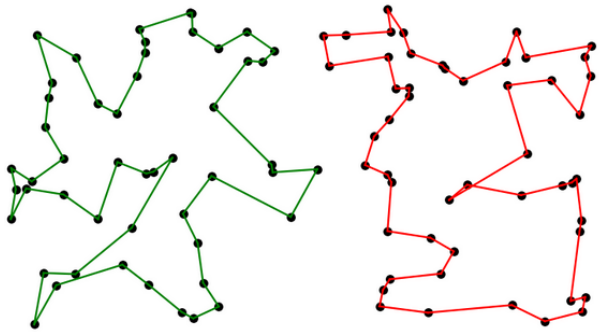


Stromy - krawędzie

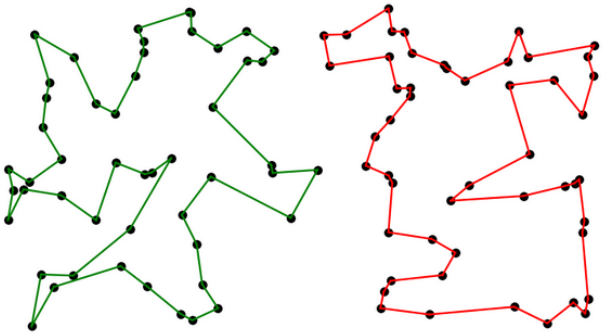


kroA100.tsp

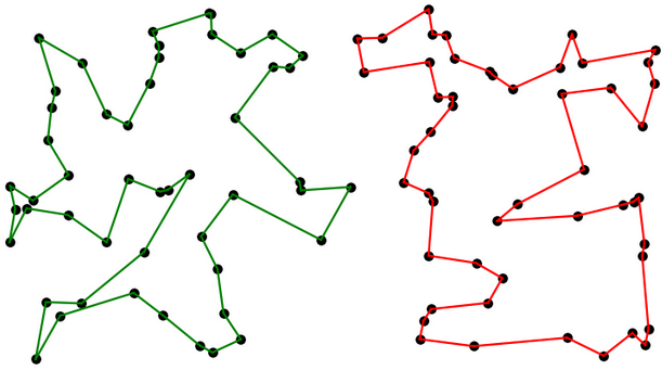
2-Żal początkowe



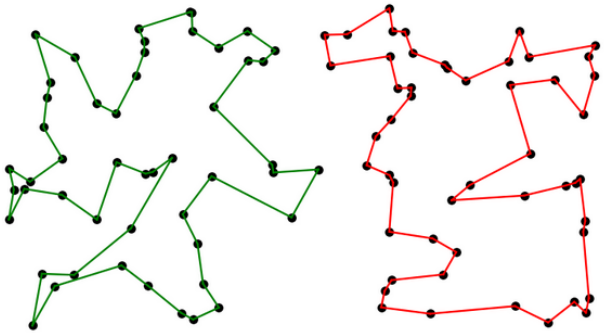
Zachłanny - wierzchołki



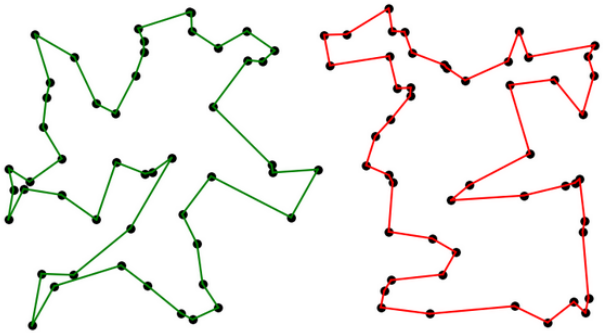
Stromy - wierzchołki



Zachłanny - krawędzie

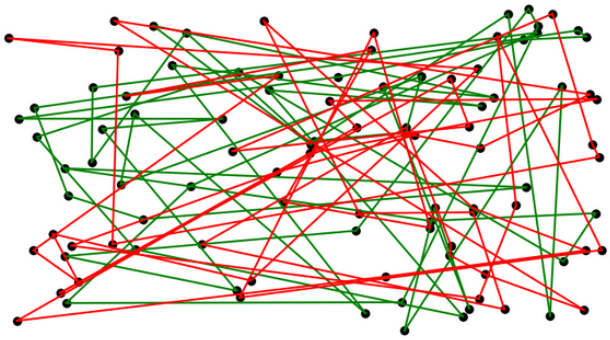


Stromy - krawędzie

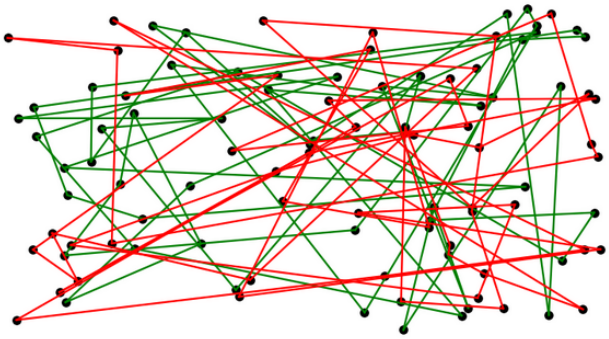


kroB100.tsp

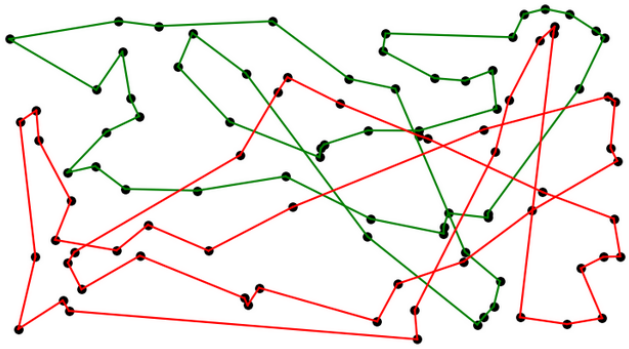
Losowe Początkowe



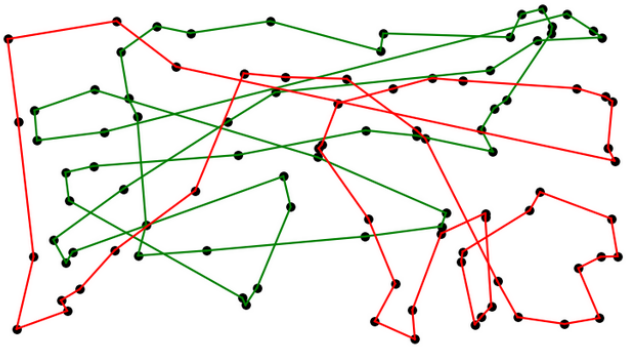
Losowe Błądzenie



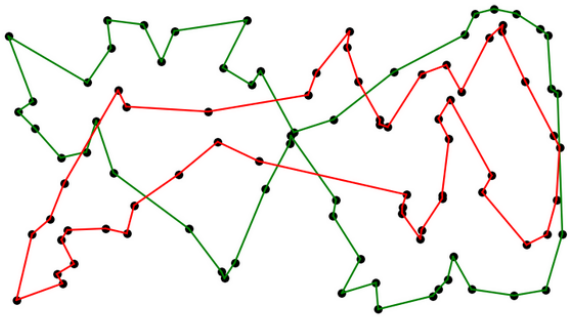
Zachłanny - wierzchołki



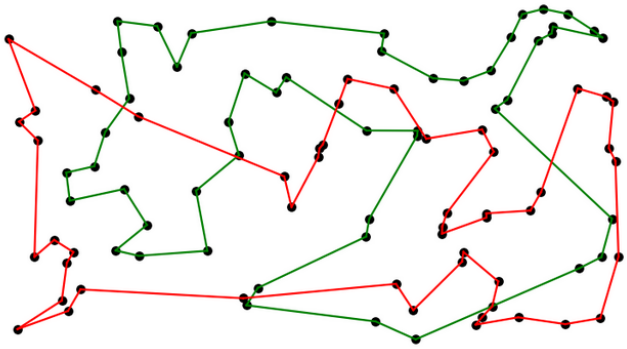
Stromy - wierzchołki



Zachłanny - krawędzie

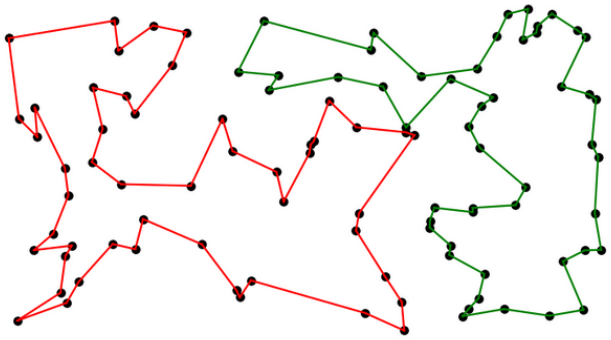


Stromy - krawędzie

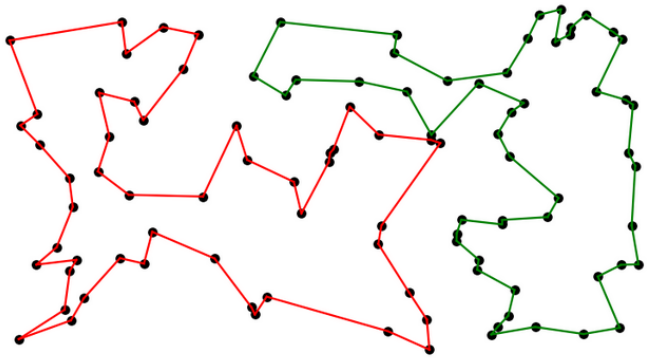


kroB100.tsp

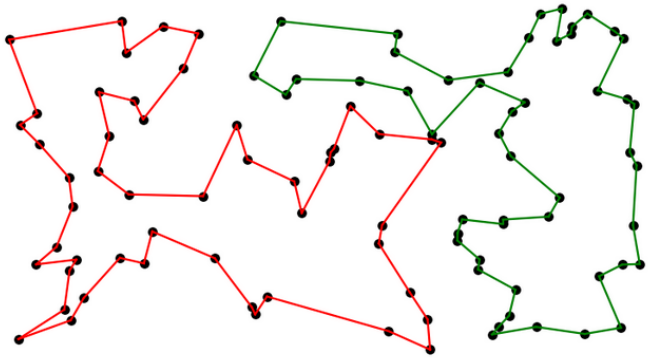
2-Żal początkowe



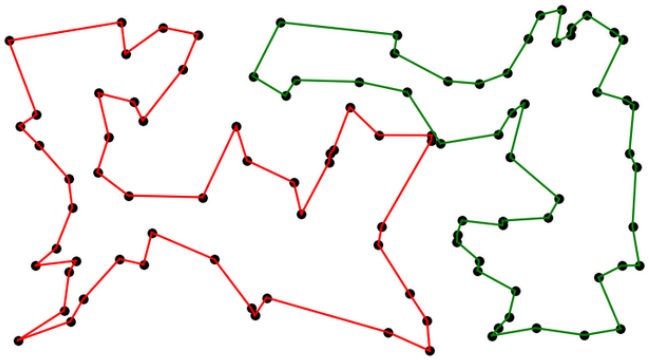
Zachłanny - wierzchołki



Stromy - wierzchołki



Zachłanny - krawędzie



Stromy - krawędzie