POLITECHNIKA WROCŁAWSKA

Projektowanie Efektywnych Algorytmów

Sprawozdanie z Etapu 2

Implementacja algorytmów metaheurystycznych dla problemu komiwojażera.

Wykonał:	**Maksymilian Guździoł, 233534**
Termin:	**WT 18.55-20.35**
Data:	**11.12.2018r**
Prowadzący:	**Dr inż. Jarosław Rudy**
Ocena:	

WSTĘP TEORETYCZNY

DEFINICJA TABU SEARCH

Przeszukiwanie tabu (Tabu search, TS) – **metaheurystyka** (algorytm) stosowana do rozwiązywania problemów optymalizacyjnych. Wykorzystywana do otrzymywania rozwiązań optymalnych lub niewiele różniących się od niego dla problemów z różnych dziedzin (np. planowanie, planowanie zadań). Twórcą algorytmu jest **Fred Glover**.

Podstawową ideą algorytmu jest przeszukiwanie przestrzeni, stworzonej ze wszystkich możliwych rozwiązań, za pomocą sekwencji ruchów. W sekwencji ruchów istnieją ruchy niedozwolone, **ruchy tabu**. Algorytm unika oscylacji wokół optimum lokalnego dzięki przechowywaniu informacji o sprawdzonych już rozwiązaniach w postaci listy tabu (TL).

SCHEMAT OGÓLNY ALGORYTMU

```
S=best=INT_MAX

//Inicjowanie listy tabu

t-[]

//Pętla trwa dopóki nie zostaną spełnione warunki końcowe (zadany czas)

WHILE (czasWykonaniaAlgorytmu<CzasZadanyPrzezUżytkownika)

{

//Wykonanie kroku

S = SELECT (sąsiedztwo(s), t)

//aktualizowanie listy tabu

T = UPDATE_TABU (s,t)

//Najlepsze rozwiązanie zapamiętujemy

If (f(najlepsze)<f(s)

Best = s

}

Return best;
```

STRUKTURY SĄSIEDZTWA

Ze względu na przestrzeń rozwiązań należy zdefiniować relację sąsiedztwa na parach elementów tej przestrzeni, która obejmuje całą dziedzinę przestrzeni przeszukiwań. Definicja relacji zależy oczywiście od przestrzeni i formatu rozwiązań (np. rozwiązaniami mogą być wektory binarne, wektory liczb rzeczywistych, permutacje zbioru liczb naturalnych itp...).

W wykonanym algorytmie uwzględniam trzy różne listy sąsiedztwa:

- insert(i,j) – przeniesienia j-tego elementu na pozycję i-tą

- swap(i,j) – zamiana miejscami i-tego elementu z j-tym

- invert(i,j) – odwrócenie kolejności w podciągu zaczynającym się na i-tej pozycji i kończącym na pozycji j-tej

LISTA TABU

Lista Tabu jest to lista ruchów zakazanych, które mają określoną kadencje bycia zakazanymi (w moim programie jest to czas równy ilości miast np. jeśli mamy 47 miast to tabu będzie trwało 47 iteracji).

DYWERSYFIKACJA

Jest to procedura pozwalająca na przeszukiwanie różnych obszarów przestrzeni stanów. W programie wykorzystana została strategia dywersyfikacji nazywana *metodą zdarzeń krytycznych.* Funkcja **diversification ()** sprawdza przez ile kolejnych iteracji nie zostało znalezione lepsze rozwiązanie, a następnie jeśli ta ilość przekroczyła ilość krytyczną (w tym przypadku jest to dwukrotność ilości miast.) to generuje nowe rozwiązanie początkowe. Algorytm ponownie rozpoczyna działanie od wygenerowanego rozwiązania

OPIS NAJWAŻNIEJSZYCH FUNKCJI W PROJEKCIE

//Główna funkcja iterująca

```
void TabS::compute()
{
        random_shuffle(begin(path), end(path));                    //Wymieszanie wektora z miastami
        vector<int> temporaryNewSol;                               //Wektor przechowujący tymczasowe wymieszane miasta
        theBestSolutionCost = pathCostList(path);                  //Przypisanie najlepszego kosztu
        vector<int> BestSolution;          //Wektor z drogą o najniższym obliczonym koszcie
        high_resolution_clock::time_point t1 = high_resolution_clock::now();
                                                                   //Rozpoczęcie liczenia wykonywania algorytmu
        duration<double> time_span;
        while (time_span.count() <= sec) {
                int city_1 = 0;                                    //2 miasta których droga będzie wpisywana do listy tabu
                int city_2 = 0;
                int bestCost = INT32_MAX;                          //startowy koszt drogi
                for (size_t j = 1; j < m_size; j++){//Pierwsze miasto jest miastem startowym dlatego zaczynamy od 1
                        for (size_t k = 2; k < m_size; k++){//rozpoczęte od 2 a nie od 1 bo od nic można iść do siebie samego
                                if (j != k){//warunek pozwalający nie chodzić do tych siebie samego w przyszłości
                                        temporaryNewSol.clear();//Czyszczenie wektora w celu przypisania mu nowej
                                        wymieszanej listy miast
                                        for (size_t m = 0; m < path.size(); m++){
                                                temporaryNewSol.push_back(path.at(m));
                                        }//W zależności od wybranej definicji sąsiedztwa wykonuje się inny warunek
                                        if (choice == 1){ //SWAP
                                                swap(temporaryNewSol, j, k); }
                                        if (choice==2){     //INSERT
                                                insert(temporaryNewSol, j, k); }
                                        if (choice==3){     //INVERT
                                                invert(temporaryNewSol, j, k); }
                                        int newCost = pathCostList(temporaryNewSol); //Nowy koszt dla
                                        obliczonej drogi
                                        //Jeśli nowy koszt mniejszy niż poprzedni to przypisanie go jako najlepszy
```

```
                                    if (newCost < bestCost && tabuList[j][k] == 0){
                                            BestSolution.clear();
                                            for (size_t m = 0; m < path.size(); m++){
            //Przypisanie jednocześnie nowej najlepszej drogi
                                                    BestSolution.push_back(temporaryNewSol.at(m));
                                            }
                                            city_1 = j;  //Miasta z nowego kosztu wpisane będą do Listy Tabu
                                            city_2 = k;
                                            bestCost = newCost;}
                                    }
                            }
                    }

                    if (city_1 != 0){
                            decrease();                 //Funkcja dekrementująca kadencje miast w liście tabu
                            addTabu(city_1, city_2); } //Dodanie miast do listy tabu
                    if (theBestSolutionCost > bestCost){ //Ustalenie najlepszego kosztu z najlepszych
                            theBestSolutionCost = bestCost;
```

//Funkcja obliczająca koszt drogi w danej liscie miast

```
int TabS::pathCostList(vector<int> xpath) {
        cost = 0;                               //Startowy koszt = 0;
        for (int i = 0; i < xpath.size() - 1; i++) {
                cost = cost + cities[xpath.at(i)][xpath.at(i + 1)];
                //dodawanie kosztu drogi danego miasta
        }

        cost = cost + cities[xpath.at(xpath.size() - 1)][xpath.at(0)];
        return cost;
}
```

//Funkcja zamieniająca i-te miasto z j-tym

```
void TabS::swap(vector<int>& xpath, int i, int j) {
        int tmp = xpath.at(i);          //Zmienna pomocnicza
        xpath.at(i) = xpath.at(j);      //Zamiana
        xpath.at(j) = tmp;
}
```

```
//Funkcja wstawiajaca j-ty element na miejsce i-tego
void TabS::insert(vector<int>& xpath, int i, int j)
{
	int tmp = xpath.at(j);				//pomocnicza zmienna
	xpath.erase(xpath.begin() + j);			//usuniecie j-tego elementu
	//wstawienie j-tego elementu na miejsce i-tego
	xpath.insert(xpath.begin() + i, tmp);
}
```

```
//Funkcja odwracająca kolejność w podciągu zaczynającym się na i-tej pozycji i kończącym na pozycji j-tej
void TabS::invert(std::vector<int>& xpath, int i, int j)
{
	vector<int> temporary_1;		//Lista pomocnicza
	vector<int> temporaryReverse;		//Lista pomocnicza do odwrócenia
	int pos1 = i;
	int pos2 = j;
	if (i < j) {
		for (int h = 0; h < i; h++) {		//dopóki nie dojdzie do i dodawaj do listy
			temporary_1.push_back(xpath.at(h));}
		for (int h = i; h < j; h++) {	//odwrócenie wartości w przedziale od i do j
			temporaryReverse.push_back(xpath.at(h));
		}reverse(temporaryReverse.begin(), temporaryReverse.end());
		for (int h = 0; h < temporaryReverse.size(); h++) {
			temporary_1.push_back(temporaryReverse.at(h));}
		for (int h = j; h < xpath.size(); h++) {	//dopisanie wartości za J-tym elementem
			temporary_1.push_back(xpath.at(h));}
	}
	if (i > j) {
		for (int h = 0; h < j; h++) {
			temporary_1.push_back(xpath.at(h));}
		for (int h = j; h < i; h++) {
			temporaryReverse.push_back(xpath.at(h));
		}reverse(temporaryReverse.begin(), temporaryReverse.end());
		for (int h = 0; h < temporaryReverse.size(); h++) {
			temporary_1.push_back(temporaryReverse.at(h));}
		for (int h = i; h < xpath.size(); h++) {
			temporary_1.push_back(xpath.at(h));}
	}

	xpath.clear();		//Czyszczenie głownej listy dróg
    for (int h = 0; h < temporary_1.size(); h++){	//Przypisanie nowej kolejności miast do listy
		xpath.push_back(temporary_1.at(h));}
}
```

//Funkcja odpowiedzialna za dywersyfikacje

```
void TabS::diversification(int bestCost, vector<int> bestBestSol) {
	if (theBestSolutionCost <= bestCost){	//Jesli kolejna iteracja nie dala lepszego kosztu
		divLvl++;}
	else{					//Jeśli znaleziono lepszy koszt
		divLvl = 0;}
	if (divLvl > maxDivLvl){		//Jeśli wartość krytyczna przekroczyła max
		vector<int> tmp;		//Tworzenie wektora tymczasowego
		for (int n = 0; n < path.size(); n++){
			tmp.push_back(path.at(n));}
		for (int m = 0; m < m_size; m++){
			random_shuffle(begin(path), end(path));  //mieszanie wektora tymczasowego
			if (pathCostList(path) > pathCostList(tmp)){ //jeśli w tymczasowym wektorze znaleziono lepszą droge
				path.clear();		//to zapisuje tą drogę do wektora ze ścieżką
				for (int n = 0; n < tmp.size(); n++)
				{
					path.push_back(tmp.at(n));}
				if (theBestSolutionCost > pathCostList(path)){
					bestBestSol.clear();		//to samo z lepszym kosztem
					for (int n = 0; n < path.size(); n++){
						bestBestSol.push_back(path.at(n));}
				}
			}
		}
		for (int m = 0; m < m_size; m++){
			for (int n = 0; n < m_size; n++){
				tabuList[m][n] = 0;}		//Czyszczenie tabu
		}
		divLvl = 0;}
}
```

//Funkcja dodająca do listy tabu nowe miasta

```
void TabS::addTabu(int city1, int city2) {
	tabuList[city1][city2] += time;	//przypisanie kadencji do miast
	tabuList[city2][city1] += time;	//kadencje dostaje jednocześnie droga np. 4-3 i 3-4
}
```

```
//funkcja dekrementująca liste tabu
void TabS::decrease() {
	for (int i = 0; i < m_size; i++) {		//pętla przeszukująca liste tabu
		for (int j = 0; j < m_size; j++) {
			if (tabuList[i][j] > 0) {	//Jeśli kadencja > 0
				tabuList[i][j]--;}	//Zmniejszenie kadencji
		}
	}
}
```

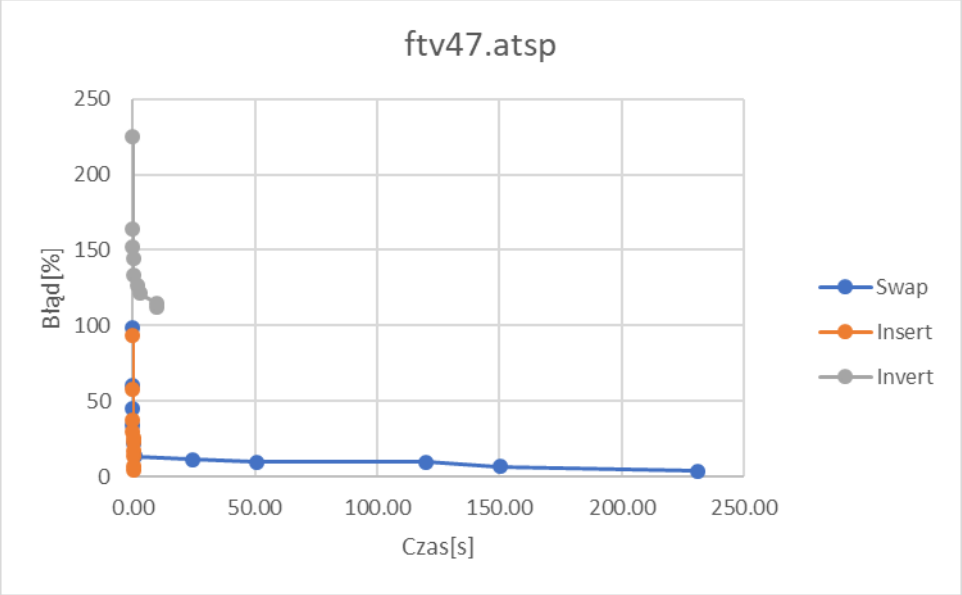
WYNIKI POMIARÓW

Algorytm w każdym przypadku został uruchomiony na 4 minuty. Program został przetestowany z dywersyfikacją. Pliki na których testowany był algorytm: ftv47, ftv170, rbg403.

ftv47-swap		
Wynik	**Czas[s]**	**Błąd względny[%]**
3525	0.02	98.4797
2852	0.04	60.5856
2575	0.06	44.9887
2385	0.10	34.2905
2168	0.16	22.0721
2018	0.95	13.6261
1978	24.35	11.3739
1948	50.65	9.68468
1946	120.07	9.57207
1895	150.12	6.70045
1838	230.97	3.49099

ftv47-insert		
Wynik	**Czas[s]**	**Błąd względny[%]**
3437	0.03	93.5248
2802	0.04	57.7703
2439	0.06	37.3311
2305	0.11	29.786
2223	0.17	25.1689
2180	0.22	22.7477
2075	0.36	16.8356
2017	0.47	13.5698
1895	0.48	6.70045
1860	0.50	4.72973

ftv47-invert		
Wynik	**Czas[s]**	**Błąd względny[%]**
5780	0.02	225.45
4691	0.09	164.133
4482	0.12	152.365
4343	0.16	144.538
4139	0.20	133.052
4032	2.15	127.027
3932	2.98	121.396
3819	9.75	115.034
3762	9.76	111.824

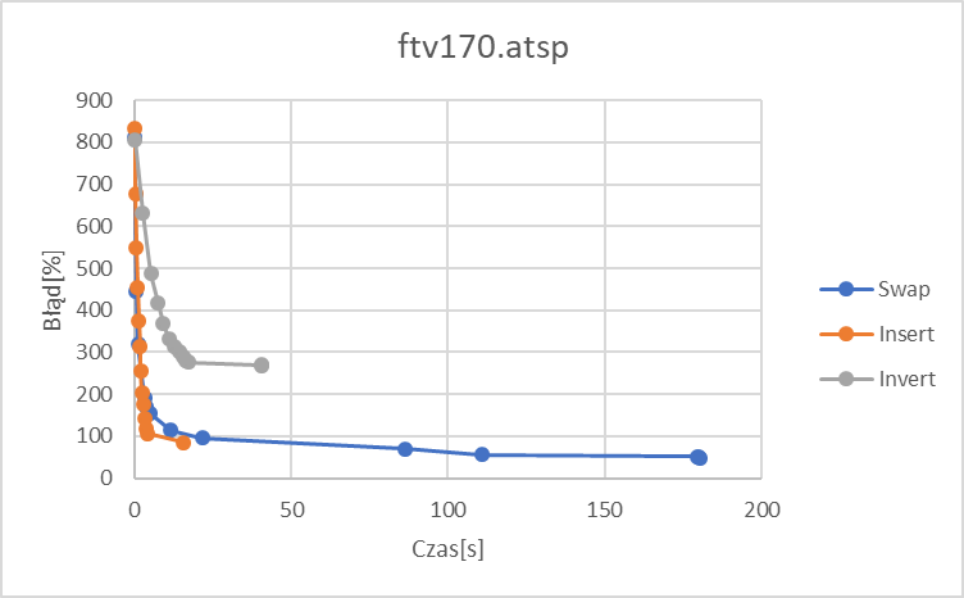


Z przeprowadzonych wyżej pomiarów wynika, że przy zastosowaniu definicji sąsiedztwa SWAP, w której to zamieniamy i-ty element z j-tym, można uzyskać najczęściej koszt zbliżony do tego optymalnego. Na potwierdzenie tego stwierdzenia został obliczony błąd względny, który przy pierwszej metodzie jest najmniejszy. Przy definicji sąsiedztwa INSERT wyniki również zbytnio nie odbiegają od optymalnego rozwiązania. Sytuacja zmienia się podczas użycia sąsiedztwa INVERT, której to wyniki odbiegają znacząco i koszty przejścia 47 miast są czasem nawet dwukrotnie wyższe od optymalnych. Warto również zauważyć, że przy definicji sąsiedztwa INSERT, droga o wartości 1860 została znaleziona po 0.5 sekundy, a w definicji sąsiedztwa SWAP, droga o wartości na przykład 1895 została znaleziona dopiero po 150 sekundach.

ftv170-swap		
Wynik	**Czas[s]**	**Błąd względny[%]**
25102	0.031183	811.143
15012	0.60	444.9
11533	1.12608	318.621
8058	3.11647	192.486
7406	3.7379	168.82
7001	4.99533	154.12
5898	11.5303	114.083
5392	21.6915	95.7169
4667	86.3104	69.4011
4303	111.039	56.1888
4178	179.64	51.6515
4135	180.119	50.0907
4038	180.454	46.5699

ftv170-insert		
Wynik	**Czas[s]**	**Błąd względny[%]**
25713	0.0291042	833.321
21440	0.304124	678.221
17916	0.643493	550.309
15271	0.994127	454.301
13112	1.35188	375.935
11360	1.71538	312.341
9829	2.078	256.77
8335	2.47538	202.541
7609	2.86051	176.189
6693	3.22795	142.94
6010	3.62101	118.149
5690	4.00906	106.534
5094	15.7521	84.9002

ftv170-invert		
Wynik	**Czas[s]**	**Błąd względny[%]**
24933	0.189969	805.009
20151	2.59452	631.434
16172	5.24949	487.005
14237	7.57366	416.77
12897	9.16151	368.131
11932	10.936	333.103
11405	12.5246	313.975
11031	14.1673	300.399
10750	15.4207	290.2
10502	16.4838	281.198
10368	17.1899	276.334
10198	40.3466	270.163
10161	40.6942	268.82

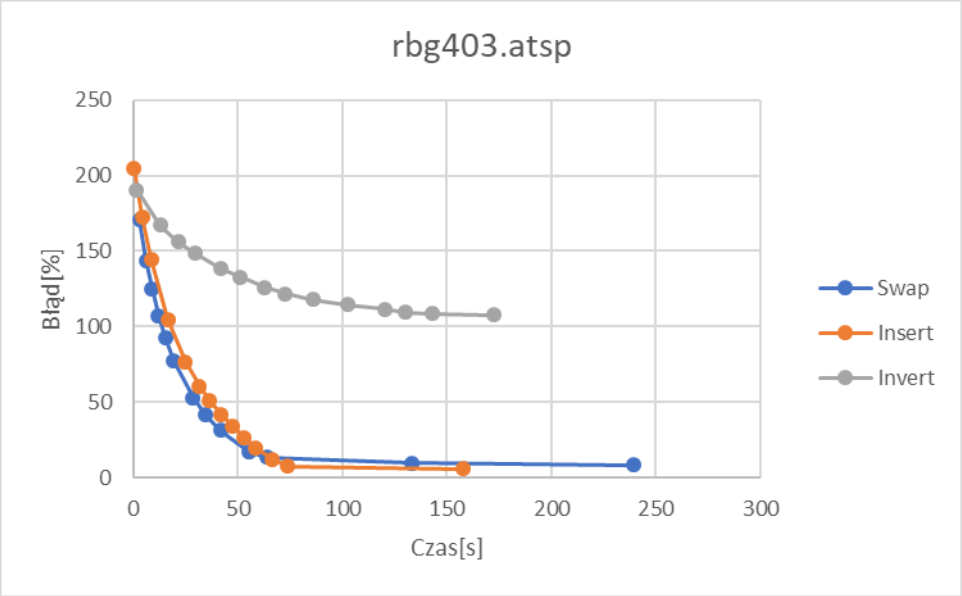


Wyniki pomiarów przy grupie 170 miast są podobne do poprzednich. Używając definicji sąsiedztwa SWAP możemy spodziewać się dokładniejszego wyniku znalezionego pod koniec działania algorytmu (4 minuty). W definicji INSERT wynik bardziej odbiega od optymalnego jednak zostaje znaleziony dużo szybciej. Przy ostatniej metodzie niestety nie można spodziewać się wyniku nawet zbliżonego do tego optymalnego.

rbg403-swap		
Wynik	**Czas[s]**	**Błąd względny[%]**
6665	3.11706	170.385
6013	6.04804	143.935
5547	8.68629	125.03
5105	11.8991	107.099
4752	15.1644	92.7789
4363	19.29	76.998
3767	28.377	52.8195
3494	34.4002	41.7444
3235	41.6101	31.2373
2892	54.9933	17.3225
2798	63.8133	13.5091
2700	133.083	9.53347
2674	239.252	8.4787

rbg403-insert		
Wynik	**Czas[s]**	**Błąd względny[%]**
7505	0.302605	204.462
6718	4.29662	172.536
6036	8.57423	144.868
5043	16.6061	104.584
4356	24.7269	76.714
3955	31.6561	60.4463
3713	36.5112	50.6288
3497	41.9941	41.8661
3305	47.1501	34.0771
3119	52.8347	26.5314
2954	58.314	19.8377
2766	66.5048	12.211
2647	73.8563	7.38337
2604	157.844	5.63895

rbg403-invert		
Wynik	**Czas[s]**	**Błąd względny[%]**
7163	1.44724	190.588
6581	12.7435	166.978
6313	21.5207	156.105
6125	29.6191	148.479
5877	42.0824	138.418
5738	51.2708	132.779
5570	62.7265	125.963
5469	72.7181	121.866
5370	85.6919	117.85
5289	102.511	114.564
5211	120.231	111.4
5158	130.282	109.249
5142	143.141	108.6
5117	172.337	107.586



Przyglądając się wynikom uruchomionego algorytmu tabu search przez 4 minuty dla 407 miast można dojść do wniosku, że definicja sąsiedztwa INSERT jest najlepsza. Znaleziony koszt drogi jest zbliżony do optymalnego (co pokazuje bardzo mały błąd względny) i jednocześnie koszt ten został znaleziony najszybciej z wszystkich trzech definicji. Porównywalna, jednak niewiele mniej dokładna definicja sąsiedztwa SWAP jest również dobrym wyborem w przypadku obliczania drogi. Niestety i tym razem metoda INVERT jest najmniej dokładna i generuje ponad dwukrotnie dłuższą drogę.

WNIOSKI

Wykonanie projektu pozwoliło zapoznać się z tematyką algorytmiki metaheurystycznej. Algorytm został przetestowany ze strategią dywersyfikacji oraz bez niej, lecz nie zostało to przedstawione w sprawozdaniu. W przypadku wykonywania algorytmu bez dywersyfikacji, zwracane wyniki przy większej ilości miast były zauważalnie gorsze.

ŹRÓDŁA

https://cs.pwr.edu.pl/zielinski/lectures/om/localsearch.pdf?fbclid=IwAR2Y_K00u4gSZNwiRLIKqn-anqQPZcPuuUZyuCjXWIZeQO74AIm-icy_R8

http://www.zio.iiar.pwr.wroc.pl/pea/w5_ts.pdf

http://www.cs.put.poznan.pl/mhapke/TO-TS2.pdf

https://pl.wikipedia.org