POLITECHNIKA WROCŁAWSKA

PROJEKTOWANIE EFEKTYWNYCH ALGORYTMÓW
PROJEKT, WT 17:05
K00-58d

PROBLEM KOMIWOJAŻERA

SYMULOWANE WYŻARZANIE
TABU SEARCH

Katarzyna Hajduk, 259189
ITE 2020, W4N

Prowadzący:
Dr inż. Jarosław Mierzwa

20 grudnia 2022

1. Wstęp teoretyczny

Zadanym problemem jest zaimplementowanie oraz dokonanie analizy efektywności algorytmu symulowanego wyżarzania oraz przeszukiwania tabu dla asymetrycznego problemu komiwojażera (*ATSP*). Problem komiwojażera jest to zagadnienie optymalizacyjne polegające na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie. Cykl Hamiltona jest to taki cykl w grafie, w którym każdy wierzchołek grafu został odwiedzony dokładnie raz w wyłączeniem pierwszego wierzchołka.

2. Założenia

Podczas realizacji zadania należy przyjąć następujące założenia:

- używane struktury danych powinny być alokowane dynamicznie (w zależności od aktualnego rozmiaru problemu),
- program powinien umożliwić wczytanie danych testowych z pliku - te pliki to: ftv47.atsp (1776), ftv170.atsp (2755) , rgb403.atsp (2465). W nawiasach podano najlepsze znane rozwiązanie dla danych zawartych w pliku - długość drogi.
- program musi umożliwiać wprowadzenia kryterium stopu jako czasu wykonania podawanego w sekundach,
- implementacje algorytmów należy dokonać zgodnie z obiektowym paradygmatem programowania, używanie „okienek" nie jest konieczne i nie wpływa na ocenę (wystarczy wersja konsolowa),
- kod źródłowy powinien być komentowany.

3. Implementacja algorytmu symulowanego wyżarzania

Algorytm symulowanego wyżarzania (ang. *Simulated annealing*) jest algorytmem, który polega na powolnym obniżaniu temperatury. Im wyższą wartość ma ten parametr, tym bardziej chaotyczne mogą być zmiany.

Algorytm SA bazuje na metodzie iteracyjnej. Początkowe rozwiązanie wyznaczamy algorytmem zachłannym, a następnie staramy się ulepszyć istniejący rezultat. Przejście z jednego wyniku do drugiego polega na znalezieniu lepszego rozwiązania sąsiedniego. W algorytmie symulowanego wyżarzania istnieje także możliwość wyboru gorszego rozwiązania z pewnym prawdopodobieństwem. Szansa na wybór gorszego rozwiązania jest, między innymi, zależna od ww. temperatury. Ilość przejść wyznacza długość epoki. Algorytm kończy się, kiedy zostanie spełniony warunek stopu – np. upłynie dany czas.

4. Najważniejsze klasy w algorytmie SA

4.1. Implementacja czasu

Pomiar czasu zmierzyłam za pomocą QueryPerformanceCounter(). Zwracany stan licznika jest dzielony przez częstotliwość. Tak otrzymujemy pomiar w sekundach.

```
long long int SimulatedAnnealing::read_QPC() {
    LARGE_INTEGER count;
    QueryPerformanceCounter(&count);
    return((long long int)count.QuadPart);
}
```

4.2. Ustawienie początkowej temperatury

Początkową temperaturę wyznaczam mnożąc współczynnik alfa przez koszt drogi początkowej.

```
double SimulatedAnnealing::setInitTemperature(int cost, double alpha){
    return cost * alpha;
}
```

4.3. Aktualizowanie temperatury

Aktualizuję temperaturę na trzy sposoby – pierwszy, wymagany w projekcie. Drugi – schemat geometryczny i trzeci, schemat liniowy (Cauchy'ego). Alpha i beta to współczynniki zmiany temperatury, era to numer epoki.

```
double SimulatedAnnealing::updateTemperature(double oldTemperature, double alpha){
    return oldTemperature * alpha;
}

double SimulatedAnnealing::updateTemperature2(double oldTemperature, double alpha, int era){
    return ((pow(alpha, (double)era)) * oldTemperature);
}

double SimulatedAnnealing::updateTemperature3(double oldTemperature, double alpha, int era, double beta){
    return (oldTemperature / (alpha + beta * (double)era));
}
```

4.4. Definicja sąsiedztwa

Funkcja polega na zamianie dwóch, losowo wybranych, miast ze sobą.

```
vector<int> SimulatedAnnealing::changeNeighbors(vector<int> path) {
    int city1 = rand() % (size - 1);
    int city2 = rand() % (size - 1);

    while (city1 == city2)
        city2 = rand() % (size - 1);

    swap((path)[city1], (path)[city2]);

    return path;
}
```

4.5. Generowanie początkowej drogi

Funkcja początkowo losuje pierwszy wierzchołek od którego zaczyna się droga. Następnie (w pętli) szukamy, po kolei, najbliższego sąsiada.

Używam tutaj dodatkowej tablicy *visited*, która przechowuje informację o tym, jakie wierzchołki już odwiedziliśmy.

```
vector<int> SimulatedAnnealing::generateInitPath() {
    initX = rand() % (size - 1);

    bool* visited = new bool[size];
    for (int i = 0; i < size; i++)
        visited[i] = false;
    visited[initX] = true;

    vector<int> path;
    path.push_back(initX);
    int minNode = initX;

    for (int i = 0; i < size - 1; i++){
        int minCost = INT_MAX;
        for (int j = 0; j < size; j++){
            if (!visited[j] && matrix[path[path.size()-1]][j] < minCost){
                minCost = matrix[path[path.size() - 1]][j];
                minNode = j;
            }
        }
        path.push_back(minNode);
        visited[minNode] = true;
    }

    path.erase(path.begin());
    delete[] visited;
    return path;
}
```

4.6. Główna pętla programu

Najpierw sprawdzamy warunek stopu. Następnie, przechodząc przez epoki (w tym wypadku przyjmuję N), szukamy najlepszego rozwiązania. Jeżeli jest lepszy od dotychczasowego, to je przyjmujemy. W przeciwnym wypadku liczymy prawdopodobieństwo, które decyduje, czy należy zaakceptować rezultat. Na koniec aktualizujemy czas oraz temperaturę.

```
while (endTime < timeToStop) {
        vector<int> localBest = path;
        int localBestCost = cost;
        vector<int> newPath;

        for (int i = 0; i < eraLength; i++) {
            newPath = localBest;
            newPath = changeNeighbors(newPath);
            int newCost = getPathCost(newPath);

            if (newCost < localBestCost) {
                localBest = newPath;
                localBestCost = newCost;
            }

            else  {
                double probability = getProbability(localBestCost, newCost, temperature);
                double randomNumber = (double)(rand() % 999) / 1000.0;

                if (randomNumber < probability) {
                    localBest = newPath;
                    localBestCost = newCost;
                }

            }
        }
        if (localBestCost < cost) {
            path = localBest;
            cost = localBestCost;
        }

        temperature = updateTemperature(temperature, alpha);
        endTime = ((read_QPC() - startTime) / frequency);
    }
```

5. Implementacja algorytmu przeszukiwania z zakazami

Algorytm przeszukiwania z zakazami (ang. *Tabu Search*) jest algorytmem, który polega przeszukiwaniu sąsiedztwa w celu wybrania najlepszego rozwiązania. Ostatnio wykonany ruch jest wprowadzany na listę zabronionych ruchów (*tabu list*), które nie mogą być wykonane przez daną ilość iteracji. Lista tabu ma za zadanie wyeliminować prawdopodobieństwo zapętleń przy przeszukiwaniu. Czasami jednak lista tabu zabrania wykonywania korzystnych kroków, mimo iż nie powoduje to powstaniu cykli. Istnieją więc sytuacje, w których opłacalne byłoby złamanie jej zakazu – nazywamy je kryterium aspiracji. Dozwolone jest złamanie zakazu tabu w przypadku znalezienia rozwiązania lepszego od najlepszego znalezionego rozwiązania. Dodaję także strategię dywersyfikacji, która pozwala na przeglądanie różnych obszarów rozwiązań. Jest wywoływana gdy przez daną liczbę iteracji nie wydarzyła się żadna zmiana. Algorytm kończy się, podobnie jak Simulated Annealing, kiedy zostanie spełniony warunek stopu – np. upłynie dany czas.

6. Najważniejsze klasy w algorytmie TS

Implementacja czasu oraz wyznaczanie rozwiązania początkowego wyglądają identyczne, dlatego nie będę ich przytaczać ponownie.

6.1. Sprawdzanie listy tabu

Funkcja sprawdza czy takie przejście przez miasta jest w liście ruchów zakazanych.

```
bool TabuSearch::isInTabu(vector<int> tabuList, int city1, int city2){
    bool inTabu = false;
    for(int i = 0; i < tabuList.size() - 1; i++){
        if(tabuList.at(i) == city1){
            if(tabuList.at(i + 1) == city2)
                inTabu = true;
        }
    }
    return inTabu;
}
```

6.2. Generowanie losowej drogi

Funkcja przetasowuje dotychczasową drogę, tworząc nową ścieżkę. Funkcja jest używana do dywersyfikacji.

```
vector<int> TabuSearch::generateRandomPath() {
    vector<int> path;

    for (int i = 0; i < size; i++) {
        if(i != initX)
            path.push_back(i);
    }

    random_shuffle(path.begin(), path.end());

    return path;
}
```

6.3. Definicja sąsiedztwa

Pierwsza funkcja polega na zamianie dwóch, losowo wybranych, miast ze sobą. Sprawdza jednocześnie, czy ruchy są w tabu list. Jeżeli tak, to dodatkowo kontroluje, czy zamiana nie byłaby bardziej opłacalna (kryterium aspiracji).

Druga funkcja przenosi element z takePositon na insertPosition i analogicznie sprawdza kryterium aspiracji.

Trzecia funkcja odwraca kolejność w podciągu zaczynającym się na pozycji takePositon i kończącym na pozycji insertPosition.

```
vector<int> TabuSearch::changeNeighbors2(vector<int> path, int cost, int aspiration) {
    vector<int> localPath = path;
    int takePosition, insertPosition;
    bool inTabu = false;
    for (int i = 0; i < size; i++) {
        do {
        takePosition = rand() % (size - 1);
        insertPosition = rand() % (size - 1);
        while (insertPosition == takePosition)
            insertPosition = rand() % (size - 1);
        if(tabuList.size() != 0) {
            inTabu = isInTabu(tabuList, takePosition, insertPosition);
            if(inTabu){
                vector <int> tempPath = path;
                tempPath.insert(tempPath.begin() + insertPosition, tempPath.at(takePosition));
                if(insertPosition > takePosition)
                    tempPath.erase(tempPath.begin() + takePosition);
                else
                    tempPath.erase(tempPath.begin() + takePosition - 1);

                int tempCost = getPathCost(tempPath);

                if (tempCost < cost && aspiration > 10) {
                    inTabu = false;
                }
            }
        }
    } while (inTabu);
    path.insert(path.begin() + insertPosition, path.at(takePosition));
    if(insertPosition > takePosition)
        path.erase(path.begin() + takePosition);
    else
        path.erase(path.begin() + takePosition - 1);

        if (getPathCost(path) < cost) {
            localPath = path;
            move1 = takePosition;
            move2 = insertPosition;
        }
    }
    return localPath;
}
```

```
vector<int> TabuSearch::changeNeighbors(vector<int> path, int cost, int aspiration) {
    vector<int> localPath = path;
    int city1, city2;
    bool inTabu = false;

    for (int i = 0; i < size; i++) {
        do {
            city1 = rand() % (size - 1);
            city2 = rand() % (size - 1);

            while (city1 == city2)
                city2 = rand() % (size - 1);

            if (tabuList.size() != 0) {
                inTabu = isInTabu(tabuList, city1, city2);

                if (inTabu) {
                    aspiration++;
                    vector<int> tempPath = path;
                    swap((tempPath)[city1], (tempPath)[city2]);
                    int tempCost = getPathCost(tempPath);

                    if (tempCost < cost && aspiration > 10) {
                        inTabu = false;
                    }
                }
            }
        } while (inTabu);

        swap((path)[city1], (path)[city2]);

        if (getPathCost(path) < cost) {
            localPath = path;
            move1 = city1;
            move2 = city2;
        }
    }
    return localPath;
}
```

```
vector<int> TabuSearch::changeNeighbors3(vector<int> path, int cost, int aspiration) {
    vector<int> localPath = path;
    int takePosition, insertPosition;
    bool inTabu = false;
    for (int i = 0; i < size; i++) {
        do {
            takePosition = rand() % (size - 1);
            insertPosition = rand() % (size - 1);
            while (insertPosition == takePosition)
                insertPosition = rand() % (size - 1);
            if(tabuList.size() != 0) {
                inTabu = isInTabu(tabuList, takePosition, insertPosition);
                if(inTabu){
                    vector <int> tempPath = path;
                    if(takePosition < insertPosition)
                        reverse(tempPath.begin() + takePosition, tempPath.begin() + insertPosition + 1);
                    else
                        reverse(tempPath.begin() + insertPosition, tempPath.begin() + takePosition + 1);
                    int tempCost = getPathCost(tempPath);
                    if (tempCost < cost && aspiration > 10) {
                        inTabu = false;
                    }
                }
            }
        } while (inTabu);
    if(takePosition < insertPosition)
        reverse(path.begin() + takePosition, path.begin() + insertPosition + 1);
    else
        reverse(path.begin() + insertPosition, path.begin() + takePosition + 1);
        if (getPathCost(path) < cost) {
            localPath = path;
            move1 = takePosition;
            move2 = insertPosition;
        }
    }

    return localPath;
}
```

6.4. Główna pętla programu

Analogicznie jak w SA najpierw sprawdzamy warunek stopu. Następnie zmieniamy sąsiedztwa, szukając najlepszego rozwiązania. Jeżeli jest lepsze od dotychczasowego, to je przyjmujemy. Ruchy wpisujemy na listę tabu. Sprawdzamy, czy minęła już długość kadencji (N/2) – jeżeli tak, to usuwamy początkowe ruchy. Ostatni „if" sprawdza czy nie należy przeprowadzić dywersyfikacji.

```
while (endTime < timeToStop) {
        bestPath = path;
        bestPathCost = cost;
        vector<int> newPath;

            if (neighbor == 1)
                newPath = changeNeighbors(path, cost); //Zamieniamy 2 węzły
            else if (neighbor == 2)
                newPath = changeNeighbors2(path, cost); //Insert
            else
                newPath = changeNeighbors3(path, cost); //Invert

            iterCounter++;
            int newCost = getPathCost(newPath);

            if (newCost < bestPathCost) {
                bestPath = newPath;
                bestPathCost = newCost;
                iterCounter = 0;

                tabuList.push_back(move1);
                tabuList.push_back(move2);
            }

            if (tabuList.size() == size / 2) { //Długość kadencji
                tabuList.erase(tabuList.begin());
                tabuList.erase(tabuList.begin());
            }

            if (iterCounter == size) {
                path = generateRandomPath();
                iterCounter = 0;

                tabuList.clear();
            }
        endTime = ((read_QPC() - startTime) / frequency);
    }
```

7. Wyniki algorytmu SA

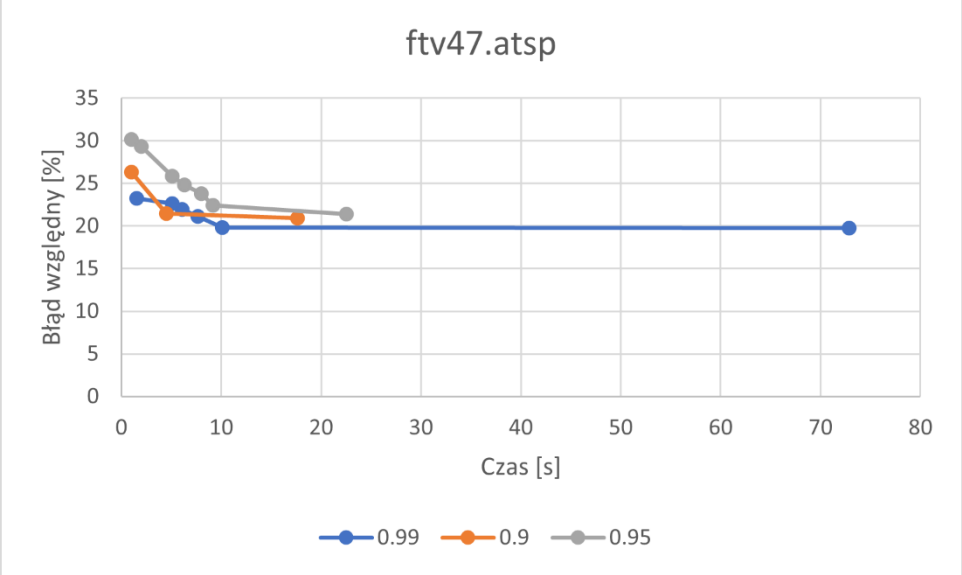
Dla trzech plików przyjmuję wartości współczynnika a = {0.99, 0.95, 0.90} oraz limit czasowy = 120s, a także porównuję trzy schematy schładzania. Algorytm powtarzam 10 razy. Porównanie - wykres błędu względnego w funkcji czasu (a = 0.99)

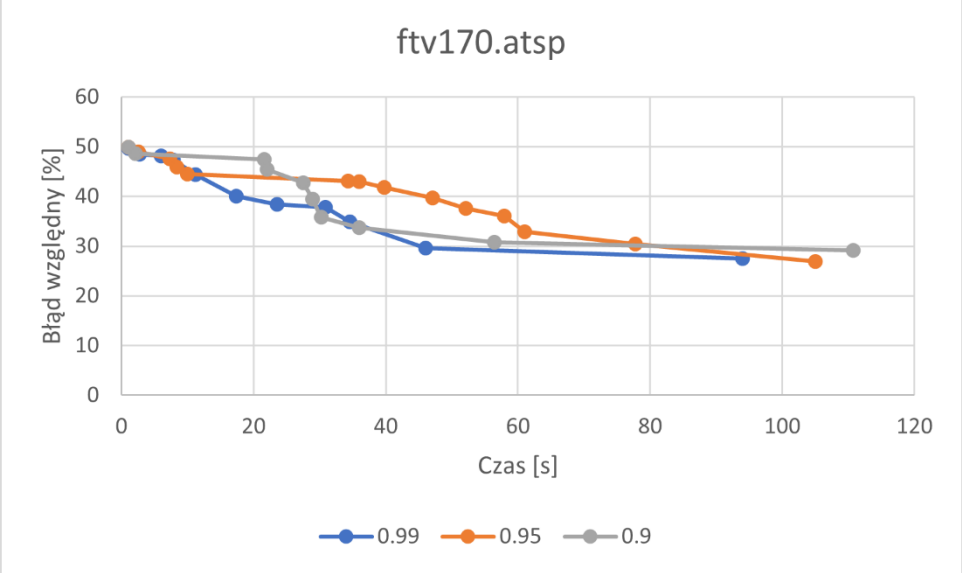
N	Optymalny koszt	Koszt	Średni błąd [%]	Temperatura końcowa [T]	Parametry
48	1776	2218.67	19.93067	2.42E-322	a=0.99, t=120s
48	1776	2419	26.31113	4.44659E-323	a=0.95, t=120s
48	1776	2225	20.142967	2.47033E-323	a=0.90, t=120s

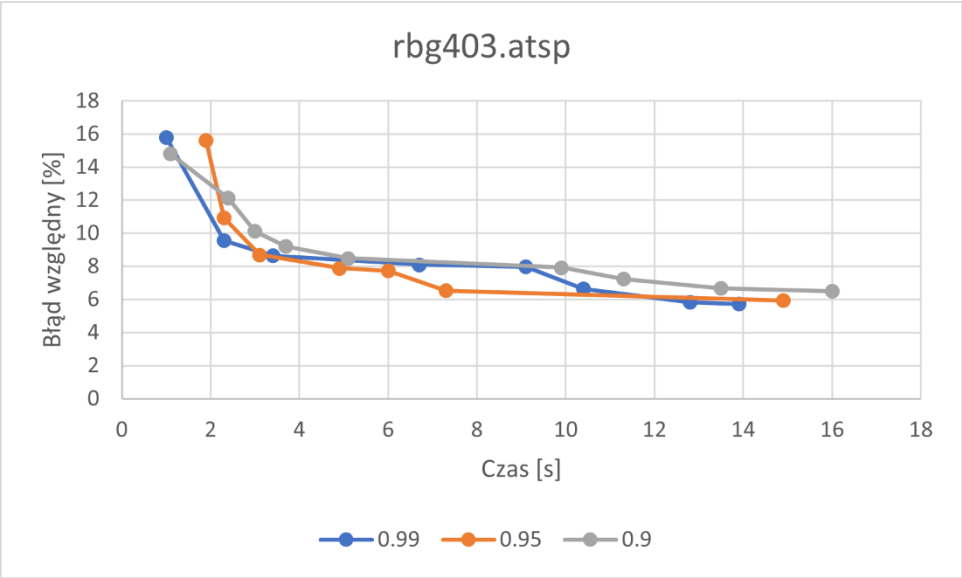
N	Optymalny koszt	Koszt	Średni błąd [%]	Temperatura końcowa [T]	Parametry
171	2755	3802.667	27.5313	6.31E-56	a=0.99, t=120s
171	2755	3730.667	26.12463	6.30E-306	a=0.95, t=120s
171	2755	3791.333	27.30733	2.47E-323	a=0.90, t=120s

N	Optymalny koszt	Koszt	Średni błąd [%]	Temperatura końcowa [T]	Parametry
403	2465	2613	5.661993	5.56E-18	a=0.99, t=120s
403	2465	2623.667	6.046967	2.89E-103	a=0.95, t=120s
403	2465	2636.667	6.50253	1.30E-23	a=0.90, t=120s

7.1. Porównanie - wykres kosztu w funkcji czasu dla zmiennego a (t = 120s)





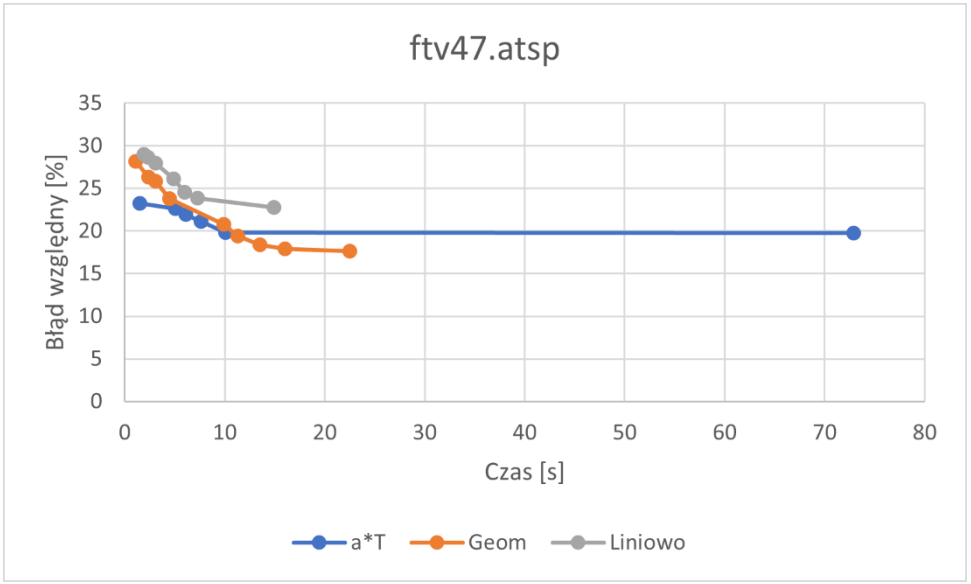


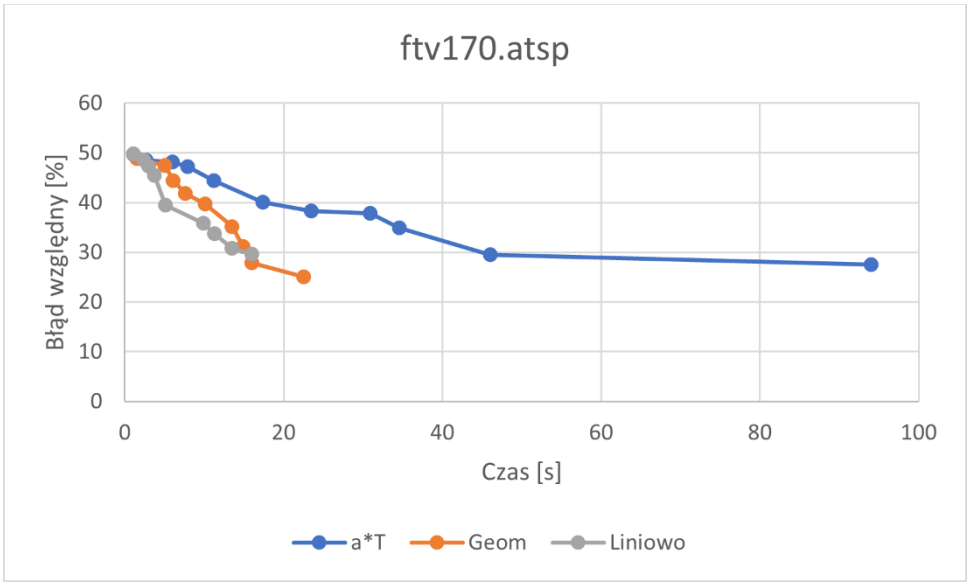
N	Optymalny koszt	Koszt	Średni błąd [%]	Parametry
48	1776	2218.67	19.93067	T(i+1)=a*T(i)
48	1776	2159.667	17.6536	Geometrycznie
48	1776	2303.667	22.7886	Liniowo

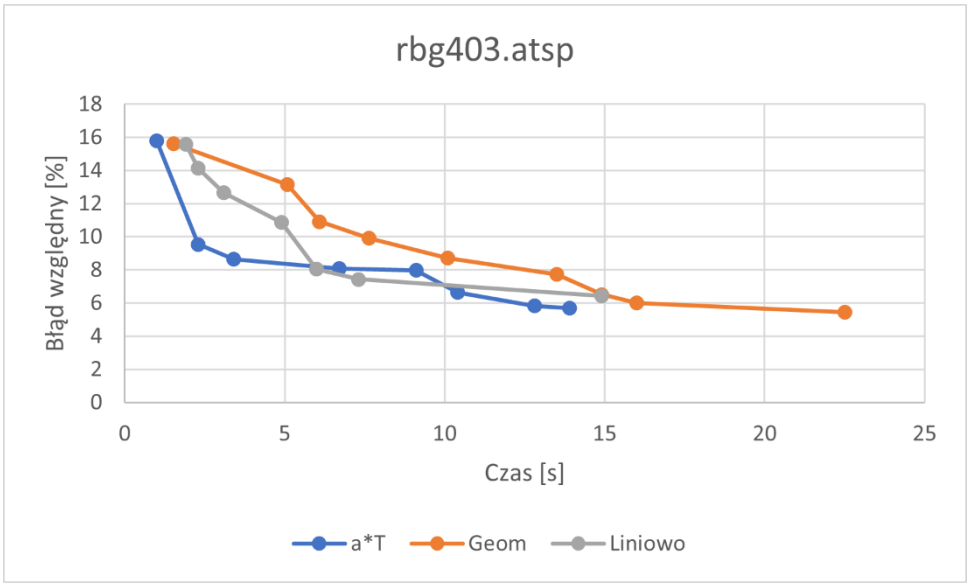
N	Optymalny koszt	Koszt	Średni błąd [%]	Parametry
171	2755	3802.667	27.5313	T(i+1)=a*T(i)
171	2755	3678	25.03475	Geometrycznie
171	2755	3918.5	29.66775	Liniowo

N	Optymalny koszt	Koszt	Średni błąd [%]	Parametry
403	2465	2613	5.661993	T(i+1)=a*T(i)
403	2465	2609	5.455112	Geometrycznie
403	2465	2635	6.44998	Liniowo

7.2. Porównanie sposobów schładzania (alfa = 0.99, t = 120s)







8. Najlepsze uzyskane wyniki dla SA

- ftv47.atsp

44 -> 15 -> 16 -> 45 -> 20 -> 38 -> 18 -> 17 -> 36 -> 14 -> 35 -> 34 -> 13 -> 46 -> 23 -> 12 -> 32 -> 7 -> 31 -> 6 -> 9 -> 33 -> 27 -> 28 -> 2 -> 41 -> 43 -> 22 -> 40 -> 47 -> 26 -> 42 -> 0 -> 25 -> 1 -> 10 -> 3 -> 24 -> 4 -> 29 -> 30 -> 5 -> 8 -> 11 -> 37 -> 39 -> 21 -> 19 -> 44

Koszt: 2049

- ftv170.atsp

152 -> 142 -> 141 -> 134 -> 131 -> 113 -> 164 -> 127 -> 126 -> 125 -> 124 -> 121 -> 120 -> 122 -> 123 -> 162 -> 102 -> 103 -> 117 -> 118 -> 119 -> 129 -> 128 -> 130 -> 135 -> 138 -> 139 -> 140 -> 6 -> 7 -> 8 -> 9 -> 10 -> 76 -> 74 -> 75 -> 11 -> 12 -> 18 -> 19 -> 20 -> 21 -> 22 -> 23 -> 26 -> 27 -> 28 -> 29 -> 30 -> 31 -> 35 -> 34 -> 156 -> 40 -> 39 -> 38 -> 37 -> 49 -> 170 -> 73 -> 77 -> 1 -> 2 -> 0 -> 81 -> 80 -> 79 -> 82 -> 78 -> 72 -> 71 -> 60 -> 50 -> 51 -> 52 -> 53 -> 43 -> 55 -> 54 -> 58 -> 59 -> 61 -> 68 -> 67 -> 167 -> 70 -> 87 -> 85 -> 86 -> 83 -> 84 -> 69 -> 66 -> 63 -> 64 -> 56 -> 57 -> 62 -> 65 -> 88 -> 153 -> 154 -> 89 -> 90 -> 91 -> 94 -> 96 -> 97 -> 99 -> 98 -> 95 -> 92 -> 93 -> 166 -> 108 -> 107 -> 106 -> 105 -> 165 -> 163 -> 100 -> 101 -> 104 -> 114 -> 110 -> 109 -> 115 -> 116 -> 136 -> 137 -> 147 -> 148 -> 149 -> 150 -> 161 -> 160 -> 14 -> 13 -> 32 -> 158 -> 36 -> 157 -> 33 -> 155 -> 41 -> 42 -> 45 -> 44 -> 46 -> 47 -> 48 -> 168 -> 3 -> 4 -> 5 -> 133 -> 169 -> 111 -> 112 -> 132 -> 143 -> 144 -> 151 -> 17 -> 24 -> 15 -> 159 -> 16 -> 25 -> 146 -> 145 -> 152

Koszt: 3554

- rbg403.atsp

271 -> 46 -> 23 -> 14 -> 62 -> 13 -> 205 -> 204 -> 24 -> 36 -> 32 -> 274 -> 83 -> 33 -> 376 -> 68 -> 38 -> 270 -> 19 -> 18 -> 402 -> 287 -> 107 -> 61 -> 257 -> 43 -> 34 -> 295 -> 50 -> 56 -> 229 -> 225 -> 58 -> 8 -> 6 -> 64 -> 3 -> 2 -> 386 -> 47 -> 112 -> 322 -> 272 -> 28 -> 85 -> 129 -> 95 -> 55 -> 307 -> 143 -> 310 -> 29 -> 397 -> 5 -> 299 -> 259 -> 286 -> 273 -> 281 -> 77 -> 31 -> 52 -> 40 -> 39 -> 78 -> 226 -> 147 -> 79 -> 69 -> 245 -> 81 -> 22 -> 21 -> 82 -> 249 -> 360 -> 172 -> 26 -> 217 -> 314 -> 59 -> 97 -> 42 -> 333 -> 267 -> 293 -> 221 -> 67 -> 66 -> 60 -> 44 -> 88 -> 96 -> 94 -> 90 -> 72 -> 57 -> 163 -> 25 -> 15 -> 92 -> 51 -> 49 -> 37 -> 288 -> 247 -> 120 -> 392 -> 351 -> 317 -> 93 -> 145 -> 318 -> 180 -> 369 -> 187 -> 364 -> 303 -> 71 -> 246 -> 387 -> 349 -> 73 -> 166 -> 300 -> 165 -> 359 -> 207 -> 45 -> 373 -> 355 -> 115 -> 114 -> 353 -> 263 -> 102 -> 301 -> 209 -> 356 -> 313 -> 20 -> 304 -> 394 -> 393 -> 65 -> 213 -> 278 -> 122 -> 119 -> 168 -> 104 -> 9 -> 384 -> 383 -> 361 -> 146 -> 144 -> 105 -> 391 -> 154 -> 111 -> 340 -> 308 -> 260 -> 264 -> 113 -> 327 -> 108 -> 255 -> 142 -> 265 -> 275 -> 210 -> 110 -> 326 -> 258 -> 106 -> 374 -> 363 -> 289 -> 123 -> 192 -> 189 -> 124 -> 284 -> 290 -> 125 -> 74 -> 214 -> 401 -> 216 -> 200 -> 198 -> 155 -> 323 -> 305 -> 215 -> 309 -> 162 -> 48 -> 358 -> 350 -> 234 -> 228 -> 224 -> 240 -> 219 -> 328 -> 357 -> 201 -> 89 -> 130 -> 208 -> 285 -> 282 -> 164 -> 132 -> 243 -> 256 -> 291 -> 315 -> 161 -> 148 -> 325 -> 134 -> 306 -> 319 -> 135 -> 280 -> 279 -> 236 -> 227 -> 320 -> 137 -> 54 -> 254 -> 223 -> 344 -> 188 -> 173 -> 354 -> 342 -> 266 -> 141 -> 116 -> 345 -> 276 -> 298 -> 12 -> 149 -> 347 -> 352 -> 150 -> 84 -> 190 -> 100 -> 138 -> 297 -> 193 -> 316 -> 171 -> 63 -> 337 -> 232 -> 176 -> 156 -> 390 -> 153 -> 75 -> 182 -> 331 -> 378 -> 131 -> 233 -> 324 -> 381 -> 170 -> 334 -> 139 -> 366 -> 292 -> 329 -> 294 -> 330 -> 343 -> 151 -> 101 -> 230 -> 382 -> 212 -> 336 -> 321 -> 379 -> 231 -> 222 -> 185 -> 91 -> 283 -> 277 -> 179 -> 368 -> 252 -> 371 -> 370 -> 140 -> 127 -> 399 -> 377 -> 99 -> 389 -> 169 -> 27 -> 195 -> 103 -> 194 -> 380 -> 362 -> 109 -> 178 -> 181 -> 346 -> 338 -> 241 -> 375 -> 191 -> 186 -> 53 -> 400 -> 177 -> 174 -> 250 -> 262 -> 261 -> 202 -> 126 -> 341 -> 159 -> 7 -> 136 -> 152 -> 76 -> 218 -> 203 -> 211 -> 98 -> 372 -> 41 -> 398 -> 396 -> 395 -> 312 -> 35 -> 237 -> 238 -> 385 -> 199 -> 235 -> 121 -> 244 -> 80 -> 248 -> 183 -> 17 -> 242 -> 117 -> 158 -> 269 -> 253 -> 86 -> 11 -> 196 -> 0 -> 302 -> 118 -> 175 -> 157 -> 1 -> 311 -> 10 -> 128 -> 268 -> 16 -> 133 -> 339 -> 160 -> 365 -> 332 -> 296 -> 87 -> 220 -> 197 -> 167 -> 184 -> 4 -> 335 -> 206 -> 30 -> 251 -> 239 -> 70 -> 388 -> 348 -> 367 -> 271

Koszt: 2602

9. Wyniki algorytmu TS

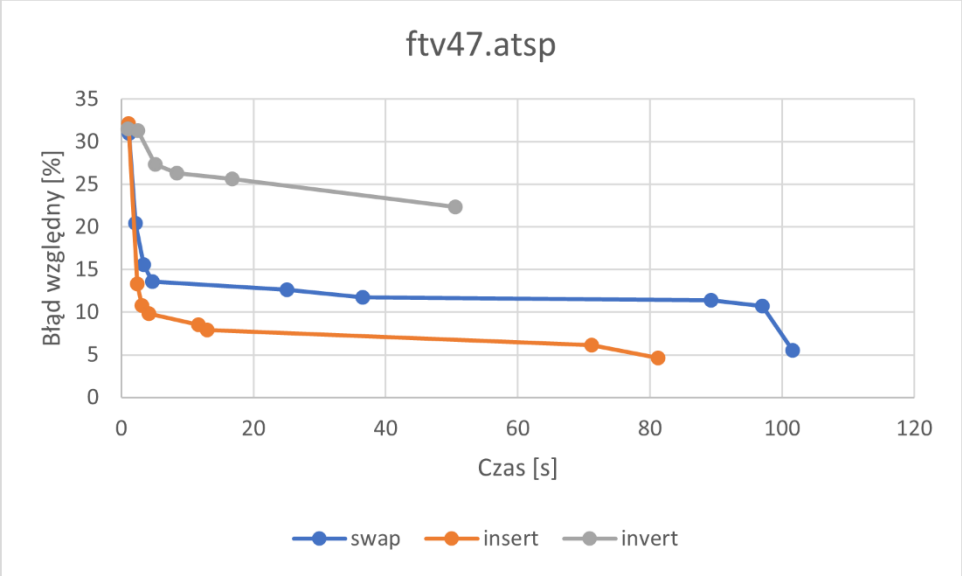
Dla trzech plików przyjmuję wartości limitu czasowego = 120s, a także porównuję trzy definicje sąsiedztwa. Algorytm powtarzam 10 razy.

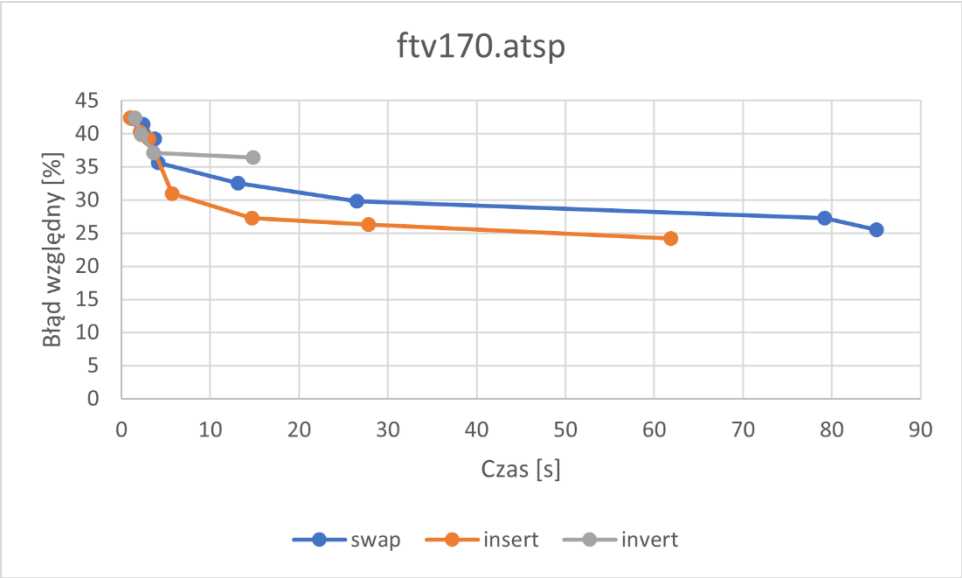
N	Optymalny koszt	Koszt	Średni błąd [%]	Parametry
48	1776	1911	7.60135	Swap
48	1776	1865	5.0671	Insert
48	1776	2173	22.3536	Invert

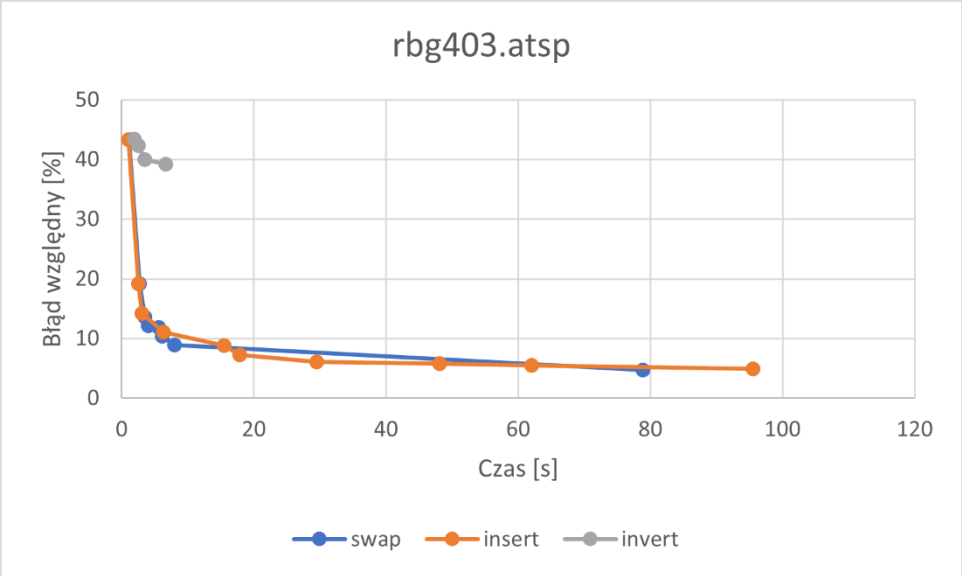
N	Optymalny koszt	Koszt	Średni błąd [%]	Parametry
171	2755	3467	25.84392	Swap
171	2755	3420	24.1561	Insert
171	2755	3759	36.4574	Invert

N	Optymalny koszt	Koszt	Średni błąd [%]	Parametry
403	2465	2566	5.12823	Swap
403	2465	2521	4.94929	Insert
403	2465	3442	39.6349	Invert

9.1. Porównanie definicji sąsiedztwa (t = 120s)







10. Najlepsze uzyskane wyniki dla TS

- ftv47.atsp

0 -> 25 -> 37 -> 20 -> 38 -> 18 -> 17 -> 34 -> 13 -> 46 -> 36 -> 35 -> 14 -> 15 -> 16 -> 45 -> 39 -> 19 -> 44 -> 21 -> 40 -> 47 -> 22 -> 41 -> 43 -> 42 -> 26 -> 1 -> 10 -> 9 -> 33 -> 27 -> 28 -> 2 -> 3 -> 24 -> 4 -> 29 -> 30 -> 5 -> 31 -> 7 -> 23 -> 12 -> 32 -> 6 -> 8 -> 11 -> 0

Koszt: 1874

- ftv170.atsp

24 -> 25 -> 150 -> 160 -> 151 -> 152 -> 142 -> 141 -> 134 -> 131 -> 113 -> 164 -> 127 -> 126 -> 125 -> 119 -> 120 -> 121 -> 122 -> 123 -> 162 -> 101 -> 103 -> 117 -> 118 -> 124 -> 129 -> 128 -> 130 -> 135 -> 138 -> 139 -> 140 -> 6 -> 7 -> 8 -> 9 -> 10 -> 76 -> 74 -> 75 -> 11 -> 12 -> 18 -> 19 -> 20 -> 21 -> 22 -> 23 -> 26 -> 27 -> 28 -> 30 -> 31 -> 33 -> 35 -> 34 -> 156 -> 40 -> 39 -> 38 -> 37 -> 49 -> 170 -> 73 -> 77 -> 2 -> 1 -> 0 -> 81 -> 80 -> 79 -> 82 -> 78 -> 72 -> 71 -> 60 -> 50 -> 51 -> 52 -> 53 -> 43 -> 55 -> 54 -> 58 -> 59 -> 61 -> 68 -> 67 -> 167 -> 70 -> 87 -> 85 -> 86 -> 83 -> 84-> 69 -> 66 -> 63 -> 64 -> 56 -> 57 -> 62 -> 65 -> 88 -> 153 -> 154 -> 89 -> 90 -> 91 -> 94 -> 96 -> 97 -> 99 -> 98 -> 95 -> 92 -> 93 -> 166 -> 108 -> 107 -> 106 -> 105 -> 165 -> 163 -> 100 -> 102 -> 104 -> 110 -> 109 -> 114 -> 115 -> 116 -> 136 -> 137 -> 147 -> 148 -> 149 -> 161 -> 14 -> 13 -> 17 -> 32 -> 158 -> 36 -> 157 -> 41 -> 155 ->

42 -> 45 -> 44 -> 46 -> 47 -> 48 -> 168 -> 3 -> 4 -> 5 -> 133 -> 169 -> 111 -> 112 -> 132 -> 144 -> 143 -> 146 -> 145 -> 15 -> 159 -> 16 -> 29 -> 24

Koszt: 3379

- rbg403.atsp

236 -> 220 -> 197 -> 23 -> 14 -> 62 -> 13 -> 205 -> 379 -> 372 -> 28 -> 251 -> 274 -> 293 -> 283 -> 8 -> 168 -> 135 -> 270 -> 19 -> 18 -> 402 -> 287 -> 350 -> 93 -> 203 -> 179 -> 368 -> 397 -> 394 -> 393 -> 335 -> 126 -> 55 -> 164 -> 3 -> 2 -> 61 -> 107 -> 386 -> 47 -> 112 -> 328 -> 257 -> 160 -> 365 -> 142 -> 150 -> 9 -> 273 -> 272 -> 278 -> 86 -> 101 -> 129 -> 177 -> 302 -> 118 -> 310 -> 319 -> 77 -> 31 -> 52 -> 40 -> 39 -> 78 -> 226 -> 154 -> 360 -> 69 -> 245 -> 81 -> 22 -> 21 -> 82 -> 247 -> 249 -> 230 -> 382 -> 144 -> 153 -> 75 -> 206 -> 30 -> 333 -> 267 -> 370 -> 221 -> 67 -> 66 -> 60 -> 98 -> 332 -> 296 -> 208 -> 90 -> 72 -> 337 -> 238 -> 229 -> 225 -> 92 -> 51 -> 49 -> 37 -> 110 -> 108 -> 120 -> 392 -> 351 -> 395 -> 217 -> 145 -> 399 -> 297 -> 155 -> 7 -> 167 -> 36 -> 123 -> 304 -> 5 -> 173 -> 344 -> 99 -> 389 -> 187 -> 152 -> 76 -> 130 -> 373 -> 355 -> 115 -> 114 -> 353 -> 352 -> 33 -> 376 -> 68 -> 264 -> 24 -> 54 -> 196 -> 0 -> 195 -> 213 -> 41 -> 340 -> 122 -> 119 -> 6 -> 87 -> 43 -> 384 -> 383 -> 361 -> 146 -> 320 -> 105 -> 391 -> 258 -> 79 -> 96 -> 338 -> 209 -> 356 -> 322 -> 83 -> 151 -> 11 -> 366 -> 321 -> 289 -> 178 -> 275 -> 147 -> 265 -> 326 -> 390 -> 169 -> 27 -> 363 -> 315 -> 158 -> 1 -> 133 -> 170 -> 284 -> 279 -> 207 -> 45 -> 313 -> 323 -> 216 -> 342 -> 224 -> 254 -> 223 -> 305 -> 215 -> 73 -> 162 -> 48 -> 358 -> 327 -> 234 -> 228 -> 71 -> 308 -> 260 -> 336 -> 292 -> 329 -> 285 -> 282 -> 357 -> 121 -> 100 -> 44 -> 231 -> 306 -> 309 -> 161 -> 148 -> 325 -> 201 -> 89 -> 32 -> 280 -> 12 -> 172 -> 26 -> 298 -> 291 -> 362 -> 387 -> 34 -> 295 -> 218 -> 349 -> 354 -> 200 -> 57 -> 222 -> 185 -> 136 -> 244 -> 80 -> 109 -> 149 -> 347 -> 341 -> 166 -> 199 -> 194 -> 324 -> 374 -> 377 -> 345 -> 299 -> 190 -> 138 -> 266 -> 211 -> 176 -> 156 -> 210 -> 50 -> 56 -> 253 -> 364 -> 303 -> 398 -> 396 -> 227 -> 212 -> 248 -> 348 -> 290 -> 237 -> 241 -> 330 -> 343 -> 233 -> 380 -> 204 -> 188 -> 276 -> 259 -> 103 -> 239 -> 70 -> 240 -> 219 -> 311 -> 334 -> 102 -> 163 -> 25 -> 401 -> 214 -> 312 -> 35 -> 140 -> 65 -> 181 -> 346 -> 94 -> 182 -> 385 -> 300 -> 106 -> 359 -> 137 -> 131 -> 307 -> 143 -> 301 -> 256 -> 184 -> 4 -> 388 -> 294 -> 252 -> 371 -> 317 -> 193 -> 331 -> 378 -> 198 -> 174 -> 250 -> 202 -> 318 -> 262 -> 261 -> 186 -> 53 -> 29 -> 132 -> 243 -> 88 -> 134 -> 288 -> 91 -> 268 -> 16 -> 232 -> 141 -> 116 -> 38 -> 381 -> 159 -> 20 -> 104 -> 84 -> 85 -> 242 -> 117 -> 128 -> 255 -> 235 -> 113 -> 281 -> 97 -> 180 -> 165 -> 127 -> 192 -> 189 -> 139 -> 183 -> 17 -> 111 -> 369 -> 42 -> 367 -> 271 -> 46 -> 58 -> 277 -> 263 -> 95 -> 10 -> 64 -> 15 -> 175 -> 246 -> 316 -> 286 -> 74 -> 400 -> 314 -> 59 -> 125 -> 171 -> 63 -> 375 -> 191 -> 124 -> 339 -> 157 -> 269 -> 236

Koszt: 2349

11. Wnioski

W metodzie „Symulowane wyżarzanie” czynnikami, które mają wpływ na czas wykonywania programu są: temperatura początkowa, długość ery, dobrane współczynniki chłodzenia oraz warunek zakończenia, którym u mnie jest czas. Ze względu na dużą ilość elementów losowych ciężko wyznaczyć najlepsze parametry algorytmu, jednak możemy zobaczyć kilka zależności. Po pierwsze, najlepszym sposobem schładzania okazało się podejście geometryczne, a najgorsze – podejście liniowe. To drugie może jednak wynikać z nieodpowiedniego dopasowania parametru beta (u mnie wynosiło 0.001). W ogólnym rozrachunku najlepsze alfa powinno wynosić 0.99, bo wtedy algorytm najdokładniej sprawdza dane, jednakże u mnie nie sprawdziło się to dla największego pliku. Czas nie ma aż tak dużego wpływu na błąd względny.

W Tabu Search ważnymi czynnikami są kryterium aspiracji, definicja sąsiedztwa, lista tabu i długość kadencji oraz warunek zakończenia – czas, który tutaj również nie okazał kluczowy. Porównując definicje sąsiedztwa widać, że najlepszym sposobem był insert, a najgorszym invert.

Początkowo, porównując oba algorytmy wyszło mi, że algorytm Symulowanego Wyżarzania jest lepszym algorytmem od Tabu Search. Błąd wynikał jednak ze złej implementacji definicji sąsiedztwa. Po naprawie błędu można zauważyć, że TS, z odpowiednią definicją sąsiedztwa, daje lepsze wyniki.