Projektowanie Efektywnych Algorytmów

Projekt
24.01.2023

254307 Paul Paczyński

(7) Algorytm Mrówkowy

1. Sformułowanie zadania

Zadanie polega naopracowaniu, implementacji i zbadaniu efektywności (metody) algorytmu mrówkowego rozwiązującego problem komiwojażera dla instancji o różnych wielkościach. Dodatkowo należy zaimplementować jedną z metod algorytmu mrówkowego i sprawdzić jego pracę dla podanych przez Marco Dorigo parametrów wejściowych.

2. Metoda

Metoda użyta do rozwiązania problemu to „Ant colony optimalization”. Jest to probabilistyczna metoda rozwiązywania problemów, która próbuje naśladować zachowanie mrówek szukających pożywienia. Mrówki poruszają się w sposób losowy zostawiając za sobą ślad składający się z feromonów. W momencie napotkania takiego śladu, mrówka podąża za nim, Substancja ta po jakimś czasie wyparowuje, więc jeśli mrówki nie podążają wyznaczoną przez feromon ścieżką, to znaczy, że najprawdopodobniej znalazły inną, lepsza ścieżkę której używa większa ilość mrówek.

W metodzie „ACO” wykorzystywane są mrówki sztuczne które różnią się od swoich oryginałów. Sztuczne mrówki potrafią bowiem samodzielnie podejmować decyzję co do wyboru wierzchołka, potrafią określić odległość między wierzchołkiem początkowym i końcowym, posiadają listę tabu w której zapamiętują przebytą ścieżkę, oraz mogą aktualizować pozostawiony feromon w dowolnym momencie.

Dodatkowo w systemie mrówkowym wyróżnia się trzy rodzaje algorytmów (wg Marco Dorigo), które różnią się głównie sposobem obliczania pozostawionego feromonu, oraz momentem jego zmiany. Algorytmy te to:

- Algorytm Gęstościowy (DAS)
- Algorytm Ilościowy (QAS)
- Algorytm Cykliczny (CAS)

W projekcie zaimplementowano algorytm cykliczny (CAS)

3. Algorytm

3.1 Klasa Ant

```
class Ant():
        def __init__(self, starting_vertex,id):
            self.tabu = [starting_vertex]  # List of visited vertexes
            self.possible_moves = None  # List of possible moves.
            self.first_move = True
            self.last_visited = starting_vertex
            self.id = id # for testing purposes

        def reset(self):
            self.tabu = self.tabu[:1]
            self.first_move = True
            self.last_visited = self.tabu[0]
```

Klasa Ant posiadała w sobie listę tabu, gdzie zapamiętywała ścieżkę którą się poruszała, listę z dostępnymi ruchami, oraz atrybuty first_move i last_visited, gdzie pierwszy odpowiadał za kod programu inicjujący listę possible_moves, a drugi przetrzymywał indeks wierzchołka ostatnio odwiedzonego. Dodatkowo metoda reset używana była dla każdej mrówki po zostawieniu feromonu.

3.2 Obliczanie początkowego rozwiązania

```
def init_distance(self):
        vertexes = []
        vertexes.extend(range(0, self.N))
        random.shuffle(vertexes)
        cost = self.calculate_path_cost(vertexes)
        return cost
```

Początkowy koszt ścieżki obliczany był poprzez wygenerowanie pseudolosowej ścieżki i obliczenie jej długości. Wartość ta jest używana do zainicjalizowana początkowej macierzy przechowującej startową ilość feromonu.

3.3 Obliczenie startowej ilości feromonu

```
def init_pheromone(self):
        tau_zero = self.ants_number / self.init_distance()

        for i in range(self.N):
            self.pheromone_matrix.append([])
        for i in range(self.N):
            for j in range(self.N):
                self.pheromone_matrix[i].append(tau_zero)
```

Początkowa ilość feromonu powstaje poprzez podzielenie ilości mrówek przez oszacowany koszt pseudolosowego rozwiązania.

3.4 Obliczanie atrakcyjności miast

```
 def calculate_denominator(self, ant: Ant):
        denominator = 0.0
        atractivness = dict()
        for i in ant.possible_moves:
            pheromone_ammount = float(
                self.pheromone_matrix[ant.last_visited][i])
            distance = float(matrix[ant.last_visited][i])
            if distance == 0:
                distance = 0.00001
            atractivness[i] = pow(
                pheromone_ammount, self.alpha) * pow(1.0/distance,
self.beta)

            denominator += atractivness[i]

        return denominator, atractivness
```

Metodę calculate_denominator mrówka wywołuje za każdym razem kiedy musi podjąć decyzję. Metoda zwraca słownik zawierający atrakcyjność każdego dostępnego miasta (licznik), oraz sumę wszystkich tych wartości zawartych w słoniku (mianownik). Korzystano w tej metodzie z poniższego wzoru.

$$p_{ij} = \begin{cases} \dfrac{(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}{\sum_{c_{i,l} \in \Omega}(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}} & \bigvee c_{i,l} \in \Omega \\ 0 & \bigvee c_{i,l} \notin \Omega \end{cases}$$

gdzie:

c - kolejne możliwe (nie znajdujące się na liście $tabu_k$ miasto),

Ω - dopuszczalne rozwiązanie (nieodwiedzone miasta, nienależące do $tabu_k$),

η_{ij} -wartość lokalnej funkcji kryterium; np. $\eta = \frac{1}{d_{ij}}$ (*visibility*), czyli odwrotność odległości pomiędzy miastami,

α - parametr regulujący wpływ τ_{ij},

β - parametr regulujący wpływ η_{ij}.

3.5 Wybór wierzchołka przez mrówkę

```
    def pick_vertex(self, ant: Ant, denominator: float, atractivness:
dict):
        sum = 0
        chance = random.random() # interval (0,1]
        sorted_list = sorted(atractivness.items(), key=lambda x: x[1])
# sort by value from lowest to highest
        #for i in atractivness.keys():
        for struct in sorted_list:
            i = struct[0]
            if denominator == 0:
                denominator = SMALL_FLOAT
            sum += atractivness[i] / denominator
            if sum>chance:
                ant.pick_vertex(i)
                return

        vertex = sorted_list[-1][0]  # (key,value)
        ant.pick_vertex(vertex)
```

Mrówka wybiera ścieżkę na podstawie wcześniej obliczonych atrakcyjności poszczególnych miast. Słownik sortowany jest od najmniejszej do największej wartości, jego poszczególne wartości dzielone są przez mianownik i dodawane do sumy prawdopodobieństwa. Losowana jest jednokrotnie wartość w przedziale (0,1] i porównywana z sumą prawdopodobieństwa. Jeśli po przejściu pętli żaden wierzchołek nie zostanie wybrany, mrówka wybierze ostatni wierzchołek w słowniku, czyli z najwyższym prawdopodobieństwem.

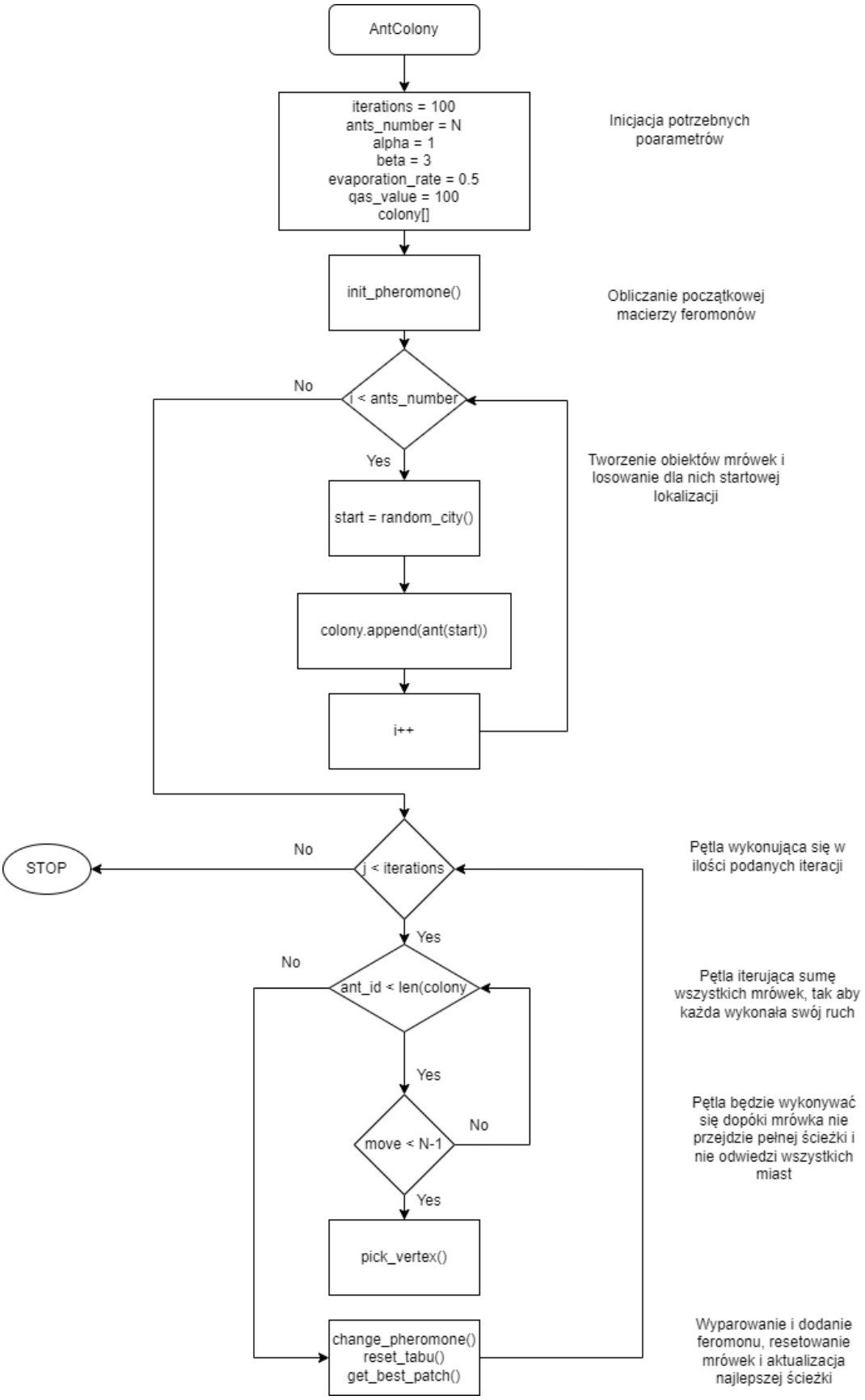
3.6 Zmiana feromonu

```
 def change_pheromone(self):
        # Evaporate pheromone from all paths
        for i in range(self.N):
            for j in range(self.N):
                if i == j:
                    pass
                else:
                    self.pheromone_matrix[i][j] =
self.pheromone_matrix[i][j] * self.evaporation_rate

        # Calculate cost of route and put down some pheromone
        for ant in self.colony:
            cost = self.calculate_path_cost(ant.tabu)
            if cost < self.min_cost:
                self.min_cost = cost
                self.best_path = ant.tabu.copy()

            for i in range(self.N):
                if i == self.N - 1:
                    #self.pheromone_matrix[ant.tabu[i]][ant.tabu[-1]]
+= float(self.qas_value / cost)
                    # TODO: DUMMY
                    a = 0
                else:
                    self.pheromone_matrix[ant.tabu[i]][ant.tabu[i+1]]
+= float(self.qas_value / cost)
            ant.reset() # Clear lists for future iterations
```

Najpierw następuje wyparowanie feromonu. Każda wartość w macierzy zostaje pomnożona przez współczynnik wyparowania ustalany na początku programu. Następnie każda mrówka oblicza koszt przebytej przez siebie ścieżki i jeśli jest mniejszy od aktualnego najlepszego wyniku, to wartość i ścieżka są zapamiętywane. Kolejna pęta pozostawia na ścieżce przebytej przez mrówkę ilość feromonu według poniższego wzoru, gdzie L to koszt ścieżki znalezionej przez mrówkę.

$$\Delta\tau_{ij}^{k}(t,t+1) = \begin{cases} \dfrac{Q_{Cycl}}{L^k} & \text{jeżeli } k\text{-ta mrówka przeszła z } i \text{ do } j \text{ na swojej trasie} \\ 0 & \text{w przeciwnym przypadku} \end{cases}$$

3.7 Schemat blokowy



4. Dane testowe

burma14.tsp
gr17.txt
gr21.tsp
gr24.tsp
bays29.tsp
ftv33.txt
ftv44.atsp
ftv70.atsp
pr152.tsp
ftv170.atsp
kroB200.tsp
rbg323.atsp
pcb442.tsp
rbg443.atsp
tsp666.tsp
p1002.tsp

http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

5. Procedura badawcza

Należało zbadać zależność czasu rozwiązania problemu od wielkości instancji. Przy implementacji „AOC" trzeba jeszcze pamiętać o początkowych parametrach potrzebnych do przeprowadzenia badania. Zostały użyte parametry podane w prezentacji (które wywodzą się od zaleceń Dorigo).

- $\alpha = 1$,
- β od 2 do 5,
- $\rho = 0.5$,
- $m = n$ (m - liczba mrówek, n - liczba miast)
- $\tau_0 = \frac{m}{C^{nn}}$, gdzie C^{nn} jest szacowaną długością trasy

Dodatkowym parametrem była ilość iteracji która musi się zakończyć, aby podany był wynik, oraz ilość powtórzeń badań na danej instancji.

Struktura pliku config.ini

<ilość instancji>
<NazwaPliku> <OptRozwiązanie> <Próby> <Iteracje> <alfa> <beta> <evaporation_rate> <Qcycl>
#nazwa_pliku_wyjściowego // nazwa poprzedzona „#"

Plik config użyty do badań

```
16
burma14.tsp 3323 10 100 1 3 0.5 100
gr17.txt 2085 10 100 1 3 0.5 100
gr21.tsp 2707 10 100 1 3 0.5 100
gr24.tsp 1272 10 100 1 3 0.5 100
bays29.tsp 2020 10 100 1 3 0.5 100
ftv33.txt 1286 10 100 1 3 0.5 100
ftv44.atsp 1613 10 100 1 3 0.5 100
ftv70.atsp 1950 3 25 1 3 0.5 100
pr152.tsp 73682 3 10 1 3 0.5 100
ftv170.atsp 2755 3 10 1 3 0.5 100
kroB200.tsp 26130 3 5 1 3 0.5 100
rbg323.atsp 1326 3 5 1 3 0.5 100
pcb442.tsp 50778 3 5 1 3 0.5 100
rbg443.atsp 2720 3 5 1 3 0.5 100
tsp666.tsp 294358 3 5 1 3 0.5 100
pr1002.tsp 259045 1 3 1 3 0.5 100
#wyniki
```

6. Wyniki

6.1 Sprzęt użyty do badania

Procesor: Intel(R) Core(TM) i5-4690K CPU @ 3.50GHz 3.50 GHz
RAM: 8 GB
System: Windows 64bit, procesor x64

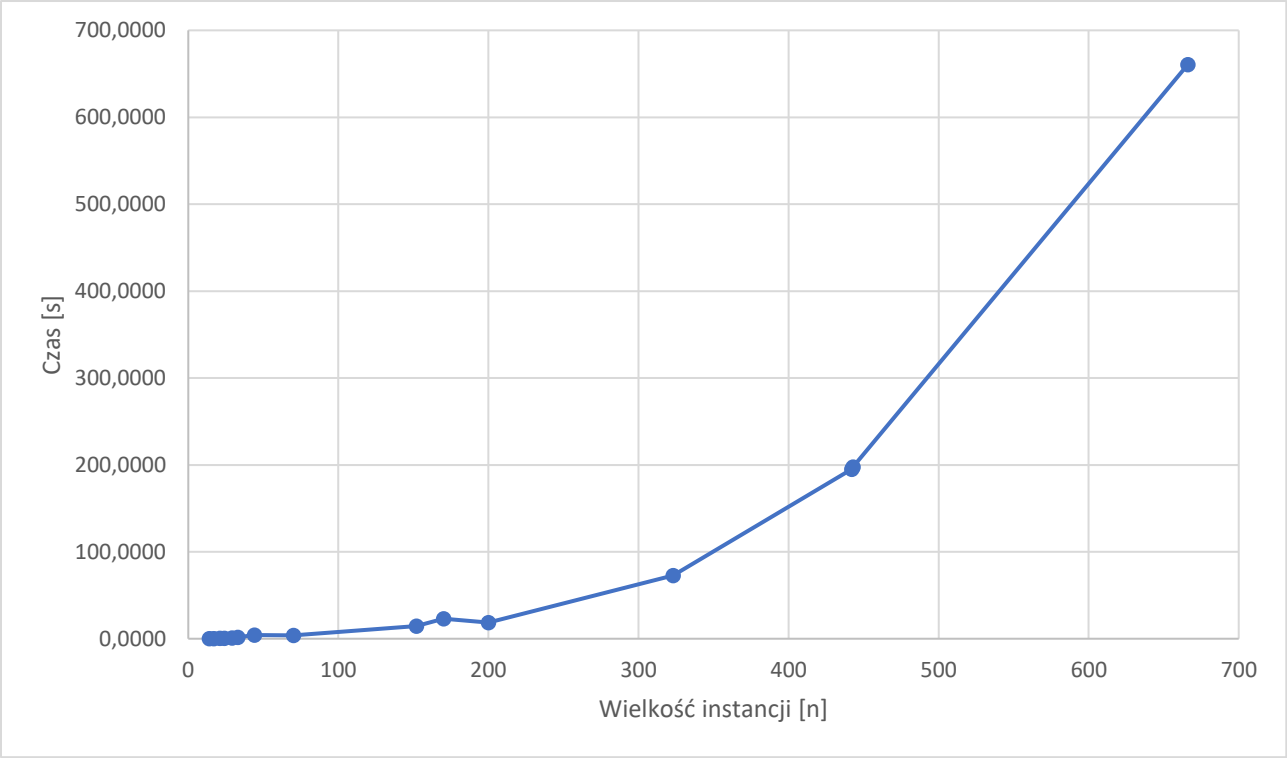
6.2 Sposób pomiaru czasu

Do pomiaru czasu wykorzystuję bibliotekę time dla języka python. Tworzone są dwie zmienne, start_time i end_time, które przyjmują wartość zwróconą z funkcji time.time(), zwracającej stan zegara. Umiejscowienie algorytmu pomiędzy dwoma zmiennymi, a następnie odjęciu start_time od end_time daje nam czas wykonania algorytmu w sekundach.

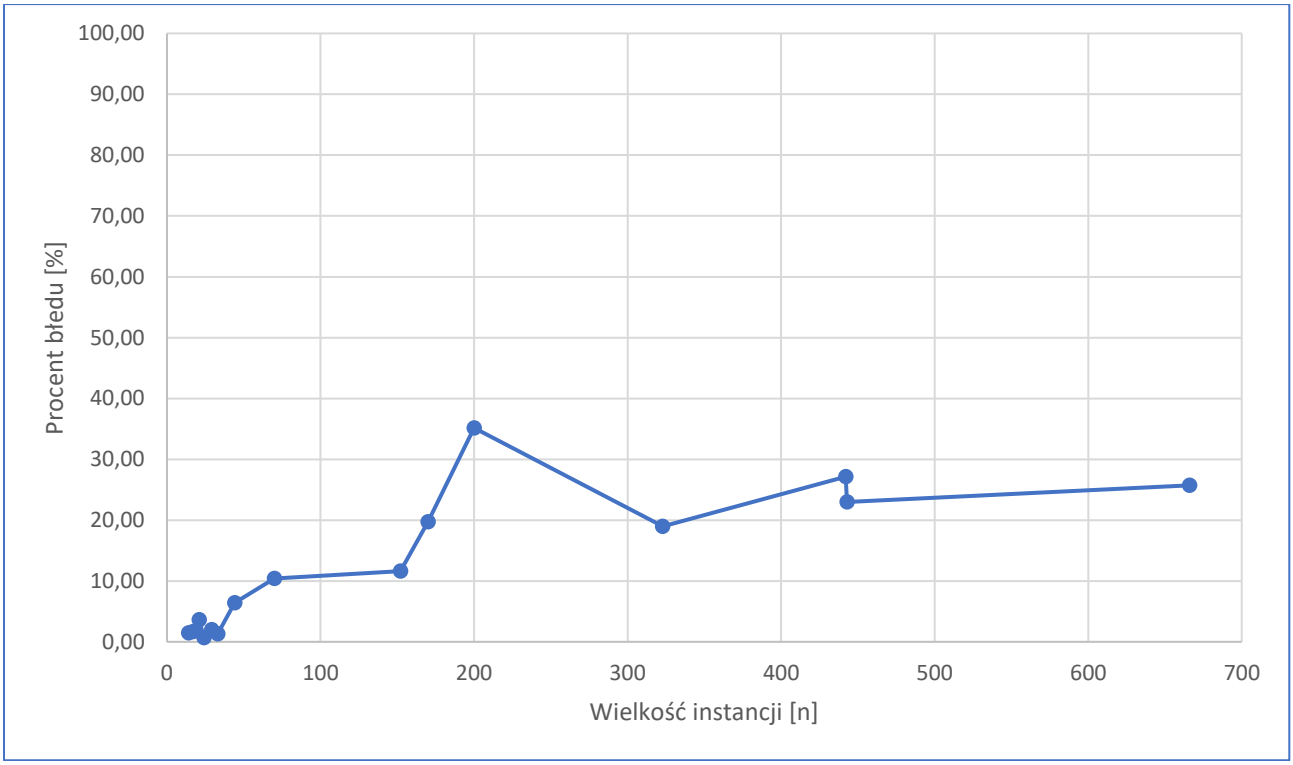
6.3 Prezentacja wyników

Tabela 1: Wyniki badania algorytmu "AOC"

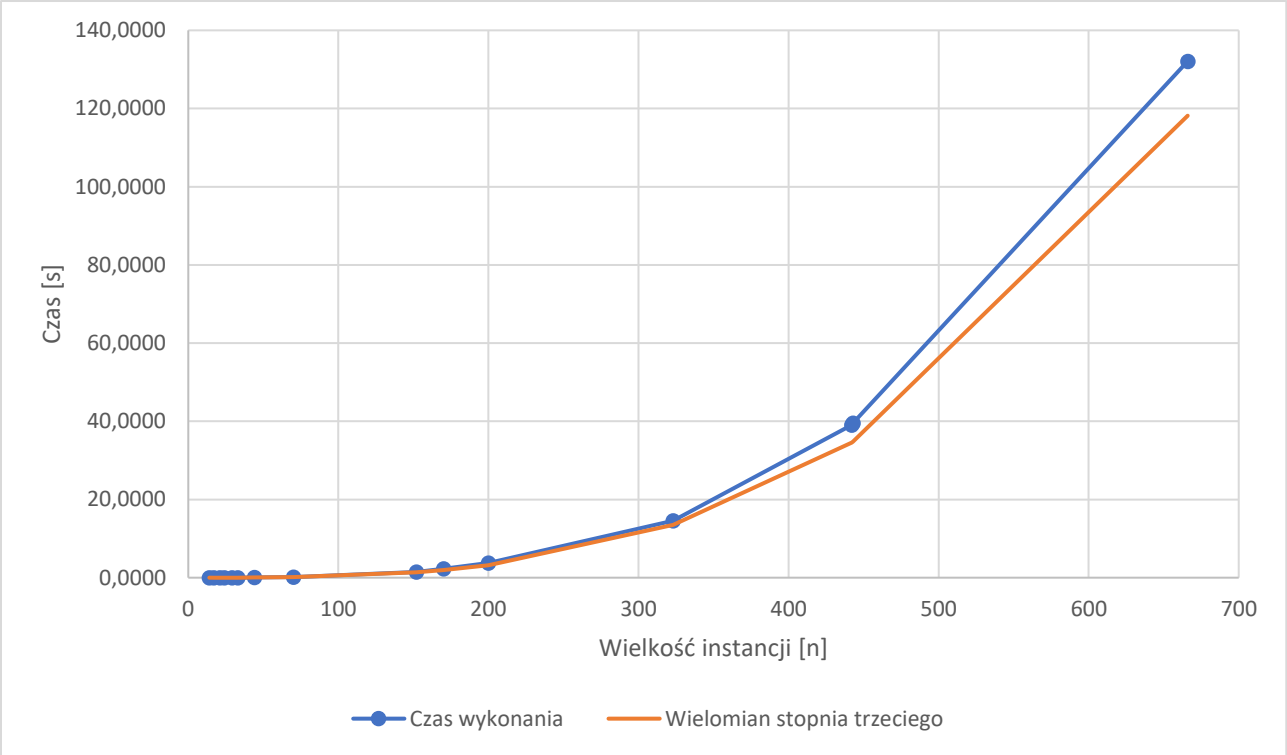
n	Nazwa pliku	Czas [s]	Czas/iter [s]	Sredni koszt	Optymalny koszt	Błąd bezwzględny [%]	Iteraz	Próby
14	burma14,tsp	0,1313	0,0013	3371	3323	1,45	100	10
17	gr17,txt	0,2230	0,0022	2120	2085	1,66	100	10
21	gr21,tsp	0,4095	0,0041	2805	2707	3,62	100	10
24	gr24,tsp	0,6035	0,0060	1281	1272	0,69	100	10
29	bays29,tsp	1,0771	0,0108	2061	2020	2,01	100	10
33	ftv33,txt	1,7444	0,0174	1303	1286	1,31	100	10
44	ftv44,atsp	4,0608	0,0406	1717	1613	6,42	100	10
70	ftv70,atsp	3,7492	0,1500	2152	1950	10,38	25	3
152	pr152,tsp	14,5497	1,4550	82252	73682	11,63	10	3
170	ftv170,atsp	23,0644	2,3064	3298	2755	19,72	10	3
200	kroB200,tsp	18,6338	3,7268	35318	26130	35,16	5	3
323	rbg323,atsp	72,8696	14,5739	1578	1326	18,98	5	3
442	pcb442,tsp	195,0573	39,0115	64566	50778	27,15	5	3
443	rbg443,atsp	197,6384	39,5277	3346	2720	23,00	5	3
666	tsp666,tsp	660,5103	132,1021	370123	294358	25,74	5	3
1002	p1002.tsp	1372,6454	457,5485	246882004	259045	95204,68	3	1



Rysunek 1: Czas wykonania algorytmu AOC



Rysunek 2: Otrzymany błąd bezwzględny



Rysunek 3: Znormalizowane wyniki czasu wykonania

6.4 Opracowanie wyników

Sposób wyboru wierzchołka przez mrówkę jest probabilistyczny, znaczy to, że w niektórych przypadkach nawet mimo wysokiego prawdopodobieństwa wyboru dobrej ścieżki, mrówki wybiorą tą złą. Objawia się to możliwością otrzymania błędnego wyniku w przypadku małych instancji, gdzie w poprzednich opracowanych przeze mnie programach ten błąd zawsze wynosił 0%. Widać to dobrze na Rysunku 2, gdzie kropki dla małych instancji powinny znajdować się idealnie na osi x.

Rysunek 3 jest znormalizowaną formą przedstawienia wyniku badania. Dla różnych wielkości instancji stosowałem różną ilość iteracji jakie mrowisko musi wykonać, aby podać ostateczny wynik. Dlatego też aby lepiej pokazać złożoność algorytmu, podzieliłem czas wykonania przez ilość przepracowanych iteracji.

W Tabeli 1 na samym dole widnieje instancja powyżej 1000 miast, która nie została przedstawiona na żadnym wykresie, gdyż byłby one wtedy nieczytelne. Dana instancja została zbadana tylko raz i dla bardzo małej ilości iteracji, nie pozwalającej podania dobrego wyniku (Inaczej, istnieje bardzo małe prawdopodobieństwo podania dobrego wyniku).

Jak widać na Rysunku 4, metoda AOC niemal dwukrotnie zwiększyła zasięg dostępnych do rozwiązania instancji. Dodatkowo jej złożoność pamięciowa jest niewielka, ze względu na małą ilość struktur przetrzymywanych przez program. Czas wykonania jest także lepszy w porównaniu do poprzedniego algorytmu, czyli SA (Symulowanego wyżarzania).

Rysunek 4: Porównanie wszystkich zaimplementowanych algorytmów

7. Wnioski

Po analizie zaimplementowanego algorytmu stwierdzam, że jest on wykonany poprawnie. Błędy procentowe dla średnich i dużych instancji nie przekraczają nawet 40%. Wyjątkiem tutaj jest instancja 1000, jednak wykonanie dodatkowych iteracji mogących zwiększyć precyzję algorytmu jest czasochłonne.

Parametry użyte w zadaniu były opisane jako optymalne dla większości zbadanych instancji przez autora tej metody i rzeczywiście przysporzyły się do osiągnięcia złożoności $O = (\ CC * n^3)$, gdzie CC to liczba cykli (w moim przypadku odnosiłem się do tego jako iteracje). Poza przeprowadzonym badaniem, sprawdzałem algorytm na innych instancjach, głównie mniejszych podczas budowy programu i zauważyłem, że w przypadku małych instancji (np. 6), liniowa zależność między ilością mrówek a miast nie daje najlepszych efektów, a dodanie dodatkowych 2-3 mrówek pozwala na obniżenie tego błędu do 0%.

Metoda probabilistyczna jaką jest algorytm mrówkowy na pewno pozwala nam na szybsze i bardziej rozległe eksperymenty na większych instancjach niż miało to w przypadku innych algorytmów, jednak dla mniejszych instancji gdzie inne algorytmy potrafią bezbłędnie podać wynik, mrówki mogą sobie z tym nie poradzić aż tak dobrze.