Uniwersytet Mikołaja Kopernika w Toruniu

Wydział Matematyki i Informatyki

Paweł Borkowski

Nr albumu: 280382

Informatyka

Praca Inżynierska

# Porównanie wydajności różnych wariantów algorytmu mrówkowego w zastosowaniu do problemu komiwojażera

Opiekun pracy dyplomowej

Doktor Marta Burzańska

Toruń 2021

# Spis treści

# Rozdział 1 - wstęp

Zadaniem niniejszej pracy było opisanie oraz porównanie różnych wariantów algorytmu mrówkowego w zastosowaniu problemu komiwojażera.

Travelling Salesman Problem, czyli problem komiwojażera, to jeden z bardziej znanych problemów w algorytmice, którego początki sięgają najprawdopodobniej roku 1832. Za sformalizowanie tego problemu oficjalnie uznaje się matematyka Karla Mengera, który zdefiniował ten problem w roku 1930 [14].

Komiwojażer to osoba, która podróżuje od miasta do miasta w celu sprzedania swoich towarów. Staje on często w swojej pracy przed następującym problemem: dane są miasta, które komiwojażer chce odwiedzić oraz odległości pomiędzy nimi. Celem jest znalezienie jak najkrótszej drogi łączącej wszystkie miasta, zaczynając i kończąc w tym samym mieście. W terminologii grafów problem ten polega na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Cykl Hamiltona to taki cykl w grafie, w którym każdy wierzchołek grafu zostaje odwiedzony dokładnie raz oprócz wierzchołka startowego, który jest również wierzchołkiem końcowym. Jest to ciekawe zagadnienie algorytmiczne, gdyż jest ono proste do opisania, jednak trudne do rozwiązania. Najbardziej intuicyjne rozwiązanie przeszukania wszystkich wierzchołków posiada za dużą złożoność obliczeniową rzędu $O\,(n!)$, natomiast kolejny prosty algorytm najbliższego sąsiada posiada dobrą złożoność $O\,(n^2)$, jednak nie daje gwarancji dobrego rozwiązania, a w skrajnych przypadkach może zwrócić najgorsze rozwiązanie.

Jednym z najefektywniejszych sposobów rozwiązania tego problemu są algorytmy mrówkowe, które są głównym tematem mojej pracy. Algorytmy te zostały zainspirowane zachowaniem mrówek w ich naturalnym środowisku. Przed podobnym problemem, jak komiwojażer mierzą się mrówki, poszukując pożywienia dla swojej kolonii. W wyniku badań na mrówkach odkryto, że do komunikacji między sobą mrówki używają specyficznych produktów chemicznych zwanych feromonami. Przeprowadzono kilka eksperymentów badających działanie feromonów. Jednym z ciekawszych eksperymentów, który przyczynił się do stworzenia algorytmów mrówkowych, jest “Double Bridge Experiments” [13] opisany przez Deneubourg, Aron, Goss i Pasteels w 1990.

Algorytmy mrówkowe znalazły szerokie zastosowanie w informatyce i algorytmice oraz są w stanie rozwiązać takie problemy jak np. problem marszrutyzacji, problem przydziału, problem harmonogramu.

W 2 rozdziale pracy zawarty jest szczegółowy opis 5 algorytmów mrówkowych tzn. opis funkcji przejścia, sposób nakładania feromonów oraz różnice w sposobie działania pomiędzy różnymi wersjami algorytmów.

W 3 rozdziale natomiast zawarty jest opis implementacji opisanych algorytmów oraz krótki opis języka C# w którym zaimplementowałem swoje rozwiązania, a rozwiązanie do nich zawarte jest na płytce dołączonej do pracy.

W 4 rozdziale opisałem proces testowania algorytmów. Testy oparłem o wyniki zawarte w plikach tekstowych odpowiadających wielkości badanego problemu, oraz obliczeniach wykonanych w plikach excel.

W ostatnim rozdziale pracy podsumowałem wyniki przeprowadzonych testów oraz opisałem zalety algorytmów mrówkowych.

# Rozdział 2 - opisy algorytmów

Istnieje kilka podstawowych rodzajów algorytmów mrówkowych, które różnią się między sobą między innymi sposobem odkładania śladu feromonowego, funkcją przejścia itp. Algorytmy mrówkowe są heurystycznym sposobem rozwiązania problemu komiwojażera, czyli takim, który nie daje gwarancji znalezienia najlepszego możliwego rozwiązania. Algorytmy te są do siebie podobne i w każdym z omawianych przeze mnie algorytmów występuje zjawisko odkładania śladu feromonowego, wybór kolejnej ścieżki za pomocą funkcji przejścia oraz odparowywanie feromonów. W poniższym rozdziale opiszę kilka podstawowych algorytmów, sposób ich działania, pseudokody algorytmów, twórców algorytmów oraz omówię podstawowe różnice w tych algorytmach. Wzory oraz opisy poszczególnych algorytmów zaczerpnąłem z książki głównych badaczy algorytmów mrówkowych Marco Dorigo oraz Thomasa Stutzle [1].

## 2.1 System Mrówkowy (AS)

Pierwszym algorytmem mrówkowym, który rozwiązuje problem wędrującego komiwojażera bądź też bardziej fachowo znajdowania cyklu Hamiltona w grafie ważonym jest System Mrówkowy (Ant System). System Mrówkowy jest to pierwszy algorytm mrówkowy wymyślony przez Marco Dorigo w 1991 roku oraz opublikowany w 1992 roku [2]. Marco Dorigo, który jest głównym inicjatorem badań nad algorytmami mrówkowymi, jestrównież jednym z głównych naukowców zajmujących się badaniem nad robotyką ławicy.

Na początku badań do algorytmu mrówkowego zaproponowane zostały trzy wersje odkładania śladu feromonowego: feromon stały, feromon średni i feromon cykliczny różniące się sposobem odkładania śladu feromonowego. W pierwszych dwóch wariantach ślad feromonowy jest aktualizowany w trakcie budowania rozwiązania przez danego insekta w momencie przejścia do kolejnego wierzchołka w grafie, z tym przypadkiem, że ilość zależy od heurystycznego wyboru. Optymalną wersją i wersją, którą zaimplementowałem, jest

wersja z odkładaniem śladu feromonowego po wykonaniu pełnego cyklu algorytmu, czyli feromon cykliczny. Algorytm ten można przedstawić w następujący sposób:

Przed rozpoczęciem głównej części algorytmu dodana jest początkowa wartość feromonów na wszystkich krawędziach grafu, by ułatwić początkowe iteracje algorytmu. Aby dobrze zoptymalizować początkowe wybory mrówek, powinno ustawić się minimalnie większe wartości feromonów na wszystkich krawędziach w odróżnieniu od tych wartości, jakie mrówki pozostawiłyby po pierwszym obrocie pętli. Początkowa wartość feromonów na wszystkich ścieżkach (i, j) należących do grafu, do odpowiedniego działania powinna wynosić następującą wartość:

(1) $$\tau_{ij} = \tau_0 = m/C^{nm},$$

gdzie $m$- liczba mrówek, $C^{nm}$- jest to długość cyklu Hamiltona znaleziona jednym z prostszych sposobów na znalezienie cyklu Hamiltona, np. metodą najbliższego sąsiada.

Po inicjalizacji feromonów mrówki wybierają losowo swoje startowe miasto z wszystkich możliwych opcji. Kolejne wybory mrówek dokonywane są z prawdopodobieństwem obliczonym za pomocą pseudolosowej funkcji przejścia, którą opiszę później. Po tym, gdy wszystkie mrówki z kolonii znajdą cykl Hamiltona, odtwarzają swoją trasę i pozostawiają określoną ilość feromonu w zależności od jakości danego rozwiązania (im lepsze rozwiązanie, tym więcej feromonu). W kolejnych iteracjach algorytmu prawdopodobieństwo wyboru wcześniej odwiedzonych ścieżek zwiększa się za każdym razem, gdy kolejna mrówka wybierze tę samą krawędź. W miarę postępu algorytmu sposób wyboru kolejnego wierzchołka w grafie jest już mniej przypadkowe ,niż w początkowych iteracjach, gdyż ilość feromonów na krótszych trasach będzie większa. Im większa ilość mrówek wybiera daną ścieżkę w grafie, tym większe prawdopodobieństwo wybrania tej ścieżki przez kolejne mrówki. Po powtórzeniu algorytmu określoną liczbę razy zależną od wielkości problemu dochodzimy do momentu, w którym wszystkie mrówki podążają tą samą ścieżką , powinno być to optymalne lub zbliżone optymalnego rozwiązania.

Wybór kolejnej krawędzi mrówki w systemie mrówkowym nie zależy tylko i wyłącznie od wartości feromonu, gdyż w wyniku początkowych losowych wyborów może powstać sytuacja stagnacji. Prowadzi ona do nieoptymalnych rozwiązań. W celu uniknięcia tego

problemu w algorytmie uwzględniania jest również informacja o długości krawędzi, wybór mrówki z danej pozycji do kolejnego wierzchołka w grafie jest obliczany za pomocą wcześniej wspomnianej funkcji przejścia.

Przed każdym krokiem mrówki k z węzła i do kolejnego węzła obliczane jest prawdopodobieństwo przejścia do węzła j:

(2) $$P_{i,j}^{k} = \frac{[\tau_{ij}]^{\alpha}[\mu_{ij}]^{\beta}}{\sum_{l \in N_i^k}[\tau_{ij}]^{\alpha}[\mu_{ij}]^{\beta}}$$ , jeżeli j ∈ $N_i^k$,

gdzie:

$P_{i,j}^{k}$ – Jest to Prawdopodobieństwo przejścia mrówki k z wierzchołka i do wierzchołka j,

$\tau_{ij}$- Jest to wartość feromonu z wierzchołka i do wierzchołka j,

$\mu_{ij}$- Jest to wartość heurystyczna określająca wartość przejścia z wierzchołka i do wierzchołka j. Wartość ta wynosi $1/C^n$, gdzie $C^n$ jest to koszt związany z przejściem z wierzchołka i do wierzchołka j,

$N_i^k$- Jest to zbiór decyzji, jakie mrówka k może podjąć, będąc w węźle i,

α- Parametr określający ważność wartości $\tau_{ij}$,

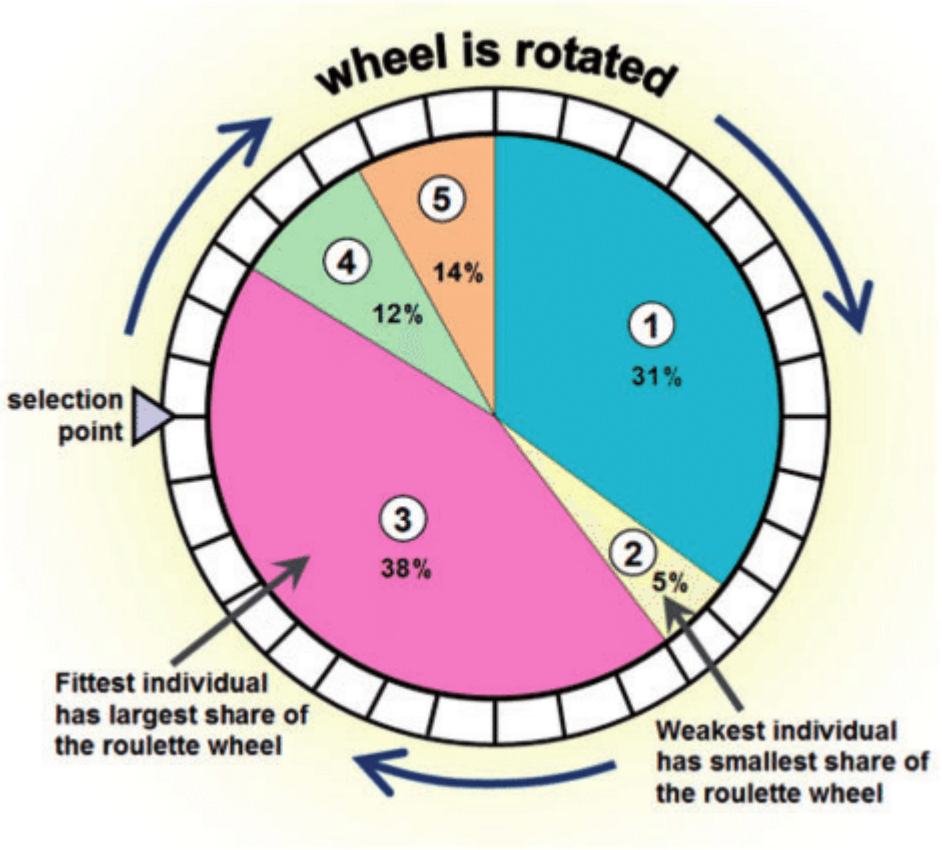
β- Parametr określający ważność wartości $\mu_{ij}$.

Parametry α i β działają w następujący sposób: jeżeli parametr α= 0, to pod uwagę brana jest tylko i wyłącznie krawędź o większej wartości heurystycznej, natomiast gdy β= 0 mrówka podejmuje decyzje tylko i wyłącznie na podstawie ilości feromonu.
W sytuacji, gdy ustawimy $\alpha > 1$ prowadzi to do nieoptymalnej sytuacji, w której mrówki szybko zastaną sytuację stagnacji, czyli taką, w której wszystkie mrówki podążają tą samą ścieżką, przez co znajdują gorsze rozwiązania. Analogicznie, gdy ustawimy wartość β na zbyt dużą, wtedy wpływ feromonów na rozwiązanie będzie za mały. Prowadzi to do sytuacji, w której algorytm będzie działał podobnie do naiwnego rozwiązania z najbliższymi sąsiadami. W wyniku początkowych testów przeprowadzonych przez twórców wychodzi na to, że optymalne rozwiązania są uzyskiwane dla wartości $\alpha = 1$ natomiast β wynosi od 2 do 5.

Gdy obliczone zostały wszystkie prawdopodobieństwa, mrówka wybiera daną krawędź w pseudolosowy sposób. Istnieje kilka sposobów na implementacje podjęcia pseudolosowego wyboru z danym prawdopodobieństwem. W moich rozwiązaniach zastosowałem algorytm genetyczny Roulette wheel. Algorytm ten wziął swoją nazwę z koła ruletki. Działa on w następujący sposób [2]:

- Każda możliwa decyzja jest umieszczona w osobnej sekcji,
- Sekcje są różnych rozmiarów (rozmiar danej sekcji jest proporcjonalny do jej prawdopodobieństwa),
- Najbardziej prawdopodobna opcja posiada największą część „koła”,
- W przypadku algorytmu mrówkowego sekcje sumują się do 1.

Przykład działania algorytmu: Losujemy jedną liczbę zmiennoprzecinkową z zakresu od 0 do 1 włącznie. Następnie na podstawie prawdopodobieństw wyboru obliczamy sekcje dla danego rozwiązania w przypadku następujących prawdopodobieństw p1 = 40%, p2 = 10%, p3 = 50%. Jeżeli wylosowana liczba jest większa od 0,0 oraz mniejsza bądź równa 0,4 to wybrana zostanie opcja pierwsza, jeżeli jest większa od 0,4 oraz mniejsza bądź równa 0,5 to opcja druga, jeżeli liczba jest większa od 0,5 i mniejsza równa 1,0 to opcja 3.




**Rysunek 1**. Ilustracja działania algorytmu Roulette wheel, "[6]".

Po tym ,gdy mrówki podjęły wszystkie swoje decyzje i zbudowały swoje rozwiązania, następuje aktualizacja śladów feromonowych. Aktualizacja ta odbywa się w następujący sposób:

- Na początek odbywa się odparowywanie śladu feromonowego, które wyrażone jest następującym wzorem:

  (3) $\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} \forall (i,j) \in L,$

  gdzie:$\tau_{ij}$ – Jest to wartość feromonu z wierzchołka i do wierzchołka j, *L* – są to wszystkie wierzchołki w grafie.

  Stała p ustawia się w przedziale od 0 do 1 i jest to współczynnik odparowywania feromonu. Funkcja odparowywania pomaga uniknięcia skumulowania feromonu na danej ścieżce oraz zaniknięcia feromonu w nieoptymalnej trasie. Stała ρ w tym algorytmie powinna wynosić większe wartości niż w pozostałych algorytmach.

Twórcy sugerują ustawienie tej wartości na 0,5, wynika to z tego, że w tym rozwiązaniu każda mrówka odkłada ślad feromonowy.

- Po zakończeniu odparowywania w algorytmie wszystkie mrówki odkładają ślad feromonowy na ścieżkach, które przebyły podczas swojej podróży, odtwarzając swoje rozwiązania przy użyciu pamięci za pomocą następującego wzoru:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}, \ \forall (i,j) \in L, \tag{4}$$

gdzie $\Delta\tau_{ij}^{k}$ – oznacza to ilość feromonu, jaka mrówka k odkłada na drodze, którą odwiedziła i wartość ta jest obliczana w następujący sposób:

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{1}{C^k}, & jeżeli\ krawędź\ (i,j) należy\ do\ T^{bs}; \\ 0, & w\ przeciwnym\ wypadku; \end{cases}$$

gdzie$C^k$- Jest to długość trasy $T^k$, jaką zbudowała dana mrówka k na trasie należącej jej rozwiązania.

Każda mrówka k w algorytmie Ant System posiada pamięć $M^k$, która przechowuje wszystkie miasta w odpowiedniej kolejności, które zostały przez nią odwiedzone. Ta pamięć pozwala mrówkom na zrekonstruowanie swojej drogi. Dzięki tej rekonstrukcji możliwe jest pozostawienie śladów feromonowych na ścieżkach należących do rozwiązania danego insekta. Aktualizacja feromonów ma na celu zwiększenia prawdopodobieństwa wyboru krótszych tras. Działa to na zasadzie wzmocnienia lepszej trasy, czyli krawędź, która używana jest przez wiele mrówek, ma większą szansę na wybranie w kolejnych iteracjach od krawędzi rzadziej używanych.

W powyższym algorytmie przejście mrówek do kolejnego wierzchołka może zostać zrealizowane na dwa sposoby: równolegle lub sekwencyjnie. W wersji sekwencyjnej mrówki wyznaczają swoją trasę po kolei, czyli czekają aż, poprzednia mrówka wykona powierzone jej zadanie, by móc zacząć wykonywać swoje. Natomiast w wersji równoległej mrówki wykonują swoje zadanie w tym samym czasie, nie czekając na rozwiązanie poprzedniej. Zważywszy na fakt, że aktualizacja feromonów odbywa się po tym, gdy cała kolonia wykona swój pierwszy obrót pętli, wybór jednej z dwóch poniższych implementacji nie ma wpływu na jakość rozwiązania. Ze względu na prostszą implementację zdecydowałem się na wersję

sekwencyjną.

Opisane przeze mnie przejście mrówek to jeden przebieg pętli algorytmu, w zależności od złożoności danego problemu, aby znaleźć optymalną trasę, należy powtórzyć ten przebieg wielokrotnie.

Implementacje powyższego algorytmu mrówkowego można przedstawić poprzez następujący pseudokod “[5]”:

```
Ustaw:
     -nA: liczbę mrówek
     -nI: liczbę iteracji
     -iP: początkowe wartości feromonów
Utwórz kolonie mrówek
Ustaw początkowe wartości feromonów na krawędziach
while (obecna_iteracja <= nI ) do
     Losowo ustaw mrówki na startowych wierzchołkach
     Repeat
          Dla każdej mrówki:
               Wybierz kolejny wierzchołek zgodnie z wzorem(2)
     Until każda mrówka zbuduje rozwiązanie
          Zaktualizuj feromony
          Zaktualizuj najlepsze rozwiązanie
end While
Wyświetl najlepsze rozwiązanie
```

## 2.2 Elitarny Algorytm Mrówkowy (EAS)

Kolejnym algorytmem mrówkowym, który opisałem to Elitarny Algorytm Mrówkowy (Elitist Ant System), który został wymyślony również przez wspomnianego wcześniej Marco Dorigo w 1991 [11]. Jest to pierwsza modyfikacja Systemu mrówkowego, która wprowadza niewielkie zmiany w sposobie nakładania śladu feromonowego.

Do tego rozwiązania dodane zostało ulepszenie,które polega na wzmocnieniu wartości feromonowych na krawędziach należących do dotychczas najkrótszej znalezionej trasy w trakcie działania algorytmu. Najlepsza trasa do tej pory w opisach algorytmu oznaczona jest przez zmienną $T^{bs}$ (najlepsza do tej pory). To oznaczenie wykorzystywane zostało w tym algorytmie podczas aktualizacji śladów feromonowych. Podobnie jak w poprzednim rozwiązaniu, przed rozpoczęciem algorytmu następuje początkowa inicjalizacja śladów feromonowych. Inicjalizacja ta określona jest za pomocą wzoru:

(5) $$\tau_0 \leftarrow (e + m) / pC^{nm},$$

gdzie e jest liczbą elitarnych mrówek. W pierwszych eksperymentach z algorytmem opisanych w publikacji w 1991 r. [11] ustalona, że parametr *e* optymalnie powinien wynosić liczbę wierzchołków do odwiedzenia.

Aktualizacja dla najlepszej trasy odbywa się za pomocą następującego wzoru:

(6) $$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} + e\Delta\tau_{ij}^{bs},$$

Parametr $\Delta\tau_{ij}^{k}$ obliczany za pomocą równania opisanego podrozdział wyżej podczas opisu aktualizacji śladów feromonowych, natomiast wartość nowo zdefiniowanego parametru $\Delta\tau_{ij}^{bs}$ oblicza się za pomocą następującego wzoru:

$$\Delta\tau_{ij}^{bs} = \{1/C^{bs} Jeżeli\ ścieżka(i,j)\ należy\ do\ T^{bs}, 0\ \ W\ przeciwnym\ wypadku\}$$

Pozostałe mrówki odkładają ślad feromonowy w standardowy sposób opisany w pierwszym rozdziale z wykorzystaniem wzoru (4).

Pierwsze eksperymenty przeprowadzone na algorytmie wykazały, że niewielka zmiana w jego strukturze doprowadziła do znalezienia znacząco lepszych wyników oraz szybszym znajdowaniu optymalnych ścieżek w grafie.

## 2.3 Algorytm mrówkowy z rankingiem (ASrank)

Trzecim algorytmem, który opisałem, jest algorytm mrówkowy z rankingiem. Ta wersja algorytmu mrówkowego została opublikowana przez Bullnheimera, Hartla i Straussa w 1999 roku [9]. Wymyślili oni kolejne ulepszenie do algorytmu mrówkowego, które do oryginalnego rozwiązania dodaje listę rankingową. Lista Ta jest wykorzystywana potem podczas funkcji aktualizującej ślady feromonowe. Celem rankingu jest większe wzmocnienie prawdopodobieństwa wyboru krawędzi należących do ścieżek mrówek, które znalazły lepsze rozwiązania w pojedynczym cyklu algorytmu. Podobnie jak w poprzedniej wersji ulega modyfikacji inicjalizacja śladów feromonowych. W tym algorytmie na początku wartość feromonów na każdej krawędzi powinna wynosić:

$$\tau_0 \leftarrow 0.5r(r-1)pC^{nm}, \quad (7)$$

gdzie $r$ jest to ilość mrówek dopuszczona do odkładania feromonu.

W tym algorytmie przed aktualizacją śladów feromonowych następuje sortowanie mrówek na podstawie długości znalezionego przez nie Cyklu Hamiltona. W wyniku tego sortowania mrówki otrzymują odpowiednią pozycję w rankingu (im krótsza trasa, tym wyższa pozycja). Jeżeli dojdzie do sytuacji remisowej, czyli takiej, w której trasa znaleziona przez insekta jest równa trasie innego insekta, wówczas wyższa pozycja w rankingu zostaje przydzielona losowo.

Do posortowania mrówek można skorzystać z jednego z wielu algorytmów sortujących. Ze względu na łatwość implementacji oraz dobrą optymistyczną złożoność czasową $O(n \log n)$ zdecydowałem się na algorytm Quicksort [4].

Algorytm ten wykorzystuje technikę "dziel i zwyciężaj". Polega ona na tym, że wybieramy element w tablicy nazywany pivot. W moim rozwiązaniu zdecydowałem się na element środkowy. Nie jest to jednak konieczne, może być to dowolny element z tablicy. Następnie ustawiamy elementy w następujący sposób: nie większe od wybranego elementu na lewo a te nie mniejsze na prawo. Po podzieleniu zbioru sortujemy osobno lewą i prawą część tablicy. Rekursja algorytmu kończy się w momencie, gdy dzielenie problemu na mniejsze nie będzie możliwe, czyli kiedy mamy jeden element tablicy.

Po posortowaniu mrówek przechodzimy do funkcji aktualizującej ślady feromonowe. W każdej kolejnej iteracji algorytmu tylko (w-1) mrówek z najwyższym rankingiem jest dopuszczona do aktualizacji śladów feromonowych, przy czym w jest pewną stałą ustaloną

na początku rozwiązania. Ilość nałożonego feromonu przez daną mrówkę jest zależna od uzyskanej pozycji w rankingu, im wyższa pozycja danej mrówki w rankingu, tym więcej zostawia za sobą feromonów. Powyższa funkcja aktualizacyjna wyrażona jest następującym wzorem:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{r=1}^{w-1} (w - r)\Delta\tau_{ij}^{k} + w\Delta\tau_{ij}^{bs}, \quad (8)$$

gdzie:

- $r$- jest to ranga danej mrówki,
- $\Delta\tau_{ij}^{k} = 1/C^{r}$,
- $C^{r}$- jest to długość trasy, jaką przebyła mrówka z rankingiem $r$,
- $\Delta\tau_{ij}^{bs} = 1/C^{bs}$,
- $C^{bs}$- jest to droga mrówki z najlepszym rozwiązaniem do tej pory,
- $w$- ilość mrówek dopuszczona do pozostawienia feromonu.

Twórcy tego algorytmu sugerują, że algorytm ten powinien spisywać się minimalnie lepiej od poprzedniej wersji Elitarnego Algorytmu Mrówkowego a znacznie lepiej od podstawowej wersji algorytmu mrówkowego. Ze względu na to, że odkładany jest ślad feromonowy, odkładany jest tylko przez kilka mrówek, zalecane jest ustawienie parametru parowania na mniejszą wartość niż poprzednich algorytmów, sugerowana wartość wynosi 0,1. Sprawdzę te rozważania w czwartym rozdziale mojej pracy.

## 2.4 System mrówkowy Max-Min (MMAS)

System mrówkowy Max-Min jest czwartym omawianym algorytmem mrówkowym. Algorytm ten został opublikowany przez T.Stützle oraz H.Hossa w 1996 roku [8]. Opiera się on na czterech głównych modyfikacjach w stosunku do podstawowej wersji algorytmu mrówkowego.

Pierwszą z tych modyfikacji jest zwiększenie znaczenia najlepszej znalezionej trasy. Tylko

mrówka, która znalazła najlepszą trasę w danej chwili lub mrówka, która ma najlepszą wartość heurystyczną, jest dopuszczona do uaktualniania wartości feromonowych. Ta strategia prowadzi niestety do patowej sytuacji, w której tylko jedna trasa będzie zawierała feromon. To doprowadzi do tego, że wszystkie mrówki będą podążać tą samą drogą. By nie doprowadzić do powyższej sytuacji, twórcy algorytmu wprowadzili kolejną modyfikację polegająca na ograniczeniu wartości nakładania śladu feromonowego z ustalonego przedziału od $\tau_{min}$ do $\tau_{max}$. Dzięki tym ograniczeniom zwiększona jest eksploracja tras, co w konsekwencji prowadzi do lepszych końcowych rezultatów.

Po tym, gdy mrówki skonstruowały swoje rozwiązania, odbywa się globalna aktualizacja śladów feromonowych tak jak w pierwszym algorytmie wzorem:

(9) $$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} \forall (i,j) \in L.$$

Z uwagi na fakt, że tylko na jedną trasę w trakcie jednego przebiegu algorytmu zostaje nałożony feromon, stała ρdo optymalnego rozwiązania powinna wynosić niewielką wartość w granicy 0,02.

Po fazie odparowywania zostaje nałożony kolejny feromon za pomocą następującej funkcji:

(10) $$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}^{bs}.$$

Zmienna $\Delta\tau_{ij}^{bs}$ może wynosić dwie następujące wartości: $\Delta\tau_{ij}^{bs} = 1/C^{bs}$ lub $\Delta\tau_{ij}^{bs} = 1/C^{ib}$, gdzie $C^{bs}$ jest to trasa rozwiązania dotychczas najlepszego, natomiast $C^{ib}$ odnosi się do rozwiązania najlepszego w danej iteracji algorytmu.

Twórcy algorytmu sugerują, że dla małej ilości danych, gdy ustawimy zmienną $\Delta\tau_{ij}^{bs}$ na włącznie wybieranie trasy z najlepszej danej iteracji, może działać w lepszym czasie. Natomiast przy większej ilości danych wersja z najlepszym do tej pory rozwiązaniem jest optymalna.

W algorytmie tym nałożone są wcześniej wspomniane limity na zmienne $\tau_{min}$ oraz $\tau_{max}$, w których przechowywane są wartości feromonów. Wartości tych limitów są określone w taki sposób, iż prawdopodobieństwo wyboru może wynieść wartości z przedziału

$0 < p_{min} \leq p_{ij} \leq p_{max} \leq 1.$

Wartość $\tau_{max}$ oraz $\tau_{min}$ zmienia się w trakcie działania algorytmu i wynosi odpowiednio dla (11) $\tau_{max} = 1/pC^{bs}$ oraz dla (12) $\tau_{min} = \tau_{max}/a$, gdzie $p$ jest to stała określająca parowanie feromonów, $C^{bs}$ jest to najlepsza trasa odnaleziona do tej pory. Stała a została wyznaczona przez Strüzle jako:$a = ((avg - 1) * \sqrt[n]{pbest})/(1 - \sqrt[n]{pbest})$, gdzie $avg$ jest to średnia ilość decyzji, jaka może podjąć mrówka, $pbest$jest to prawdopodobieństwo wybrania najlepszej ścieżki. W praktyce dla uproszczenia obliczeń stosuje się przyjęcie dla zmiennej $\sqrt[n]{pbest}$ wartości 0,05 .Inicjalizujące wartości feromonów, są mniejsze niż w poprzednich algorytmach i zostały wyznaczone jako:

(13) $\tau_0 \leftarrow 1/pC^{mn}$

## 2.5 System Kolonii Mrówek (ACS)

Ostatnim algorytmem, który opisałem to System Kolonii Mrówek (Ant Colony System). Prace nad tym algorytmem zostały wykonane i opublikowane w 1996 roku przez Marco Dorigo i Luca Gambardella [7]. Jest to kolejne ulepszenie algorytmu mrówkowego, które różni się w trzech punktach od podstawowej wersji algorytmu.

Pierwsza z różnic polega na wyborze kolejnej krawędzi przez daną mrówkę, bardziej wykorzystywane są informacje zdobyte przez mrówki podczas podróży.

Druga zmiana polega na tym, że akcje odkładania i wyparowywania feromonu odbywają się tylko i wyłącznie na ścieżkach należących do najlepszej trasy do tej pory oraz że odkładanie śladu feromonowego odbywa się lokalnie w trakcie budowania rozwiązania danego insekta, a nie globalnie po tym, gdy wszystkie mrówki wykonają swoje zadanie.

Trzecia zmiana polega na tym,że gdy mrówka znajduje się w stanie (i, j) przemieszczenia z wierzchołka i do j usuwa pewną ilość feromonu w celu zwiększenia prawdopodobieństwa wyboru innych ścieżek.

Wybór kolejnej krawędzi w algorytmie ACO prezentuję się następująco: gdy mrówka znajduje się w mieście i prawdopodobieństwo przejścia do miasta j obliczane jest za pomocą następującego wzoru:

(14) $$j = \begin{cases} argmax_{l \in N_i^k} \left\{ \tau_{ij} [\mu_{ij}]^{\beta} \right\}, & Jeżeli\ q \leq q_0; \\ J, & w\ przeciwnym\ wypadku; \end{cases}$$

Gdzie q jest to zmienna, której wartość jest losowa i znajduję się w przedziale od 0 do 1, $q_0$ jest to zmienna stała również zawierająca się w przedziale od 0 do 1. Wartość stałej $q_0$ ma wpływ na strategie wyboru mrówki, jeżeli ustawimy ten parametr na 1, to kolejnym wyborem mrówki będzie nieodwiedzone miasto, do którego prowadzi krawędź z największym prawdopodobieństwem wyboru, natomiast gdy będzie wynosić 0, to decyzja ta jest podejmowana standardową pseudolosową metodą. Adekwatnie, gdy parametr ustawimy na 0,5, to następuje sytuacja, w której jest 50% szans na wybranie krawędzi potencjalnie najlepszej do tej pory i taką sama szansa eksplorowania trasy poprzez zastosowanie pseudolosowego wyboru opisanego wzorem (2).

Kolejną modyfikacją w stosunku do podstawowej wersji algorytmu mrówkowego jest zmiana aktualizowania śladów feromonowych. W odróżnieniu od podstawowej wersji algorytmu dodana została lokalna aktualizacja.

W trakcie lokalnego cyklu algorytmu mrówki odkładają ślad feromonowy na trasę, którą odbywają, czynność tą opisuje się za pomocą następującego wzoru:

(15) $$\tau_{ij} \leftarrow (1 - \varphi)\tau_{ij} + \varphi\tau_0,$$

gdzie $0 < \varphi < 1$ oraz $\tau_0$ to dwa stałe parametry.

Optymalne wartości dla poszczególnych stałych okazały się wynosić odpowiednio dla $\varphi$= 0,1, natomiast dla $\tau_0$ jest to wartość:

(16) $$\tau_0 \leftarrow 1/nC^{nm},$$

gdzie $n$ odpowiada ilości miast w problemie komiwojażera, $C^{nm}$ jest to długość trasy obliczona algorytmem najbliższego sąsiada, natomiast parametr $\varphi$ odpowiada za lokalne parowanie feromonów. Efekt lokalnego cyklu jest następujący. Za każdym razem, gdy dana mrówka przejdzie daną trasę (i, j) to wartość zmiennej feromonowej $\tau_{ij}$ jest zmniejszana.

Strategia ta zmotywowana jest większą eksploracją w algorytmie, czyli zwiększamy prawdopodobieństwo odwiedzenia tras, które nie zostały jeszcze odwiedzone.

Po tym, gdy mrówki wykonają swój lokalny cykl, następuje globalna aktualizacja feromonów. W tym algorytmie tylko najkrótszej dotychczas znalezionej trasie odkładana jest wartość feromonowa. Następująca aktualizacja jest wyrażona poniższym wzorem:

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \rho\Delta\tau_{ij}^{bs},\ \forall(i,j) \in T^{bs}, \quad (17)$$

gdzie $\Delta\tau_{ij}^{bs} = 1/C^{bs}$, ρ- jest to globalne parowanie. Wprowadzone w tym algorytmie zmiany odkładania śladów feromonowych, umożliwiły zaniechaniu dodania dodatkowej funkcji odpowiadającej za odparowywanie śladów feromonowych, poprawiając dzięki temu złożoność procesu aktualizacji feromonowej z $O(n * m)$w AS do $O(n)$ $w$ ACS.
Algorytm ten jest oparty na wcześniejszej wersji Ant-Q stworzonej przez tych samych twórców co System Kolonii Mrówek. Różnica pomiędzy dwoma wersjami algorytmu jest wyłącznie początkową wartością śladów feromonowych. W wersji Ant-Q wartość ta wynosi: $\tau_0 = y\, max\, j \epsilon N_i^k\{\tau_{ij}\}$. Wzór ten można rozumieć jako maksymalną wartość feromonu na trasie należącej do rozwiązania najbliższego sąsiada pomnożoną przez pewną wartość $y$.

# Rozdział 3 - implementacja algorytmów

W rozdziale tym opisuję swoje implementacje opisanych algorytmów mrówkowych. Algorytmu zaimplementowałem w języku C# przy użyciu środowiska Visual Studio 2019. Implementacje algorytmów oparłem na podstawie oficjalnego pseudokodu twórców [3] oraz opisów pomocniczych algorytmów potrzebnych do sortowania wyników, oraz symulacji pseudolosowości. Implementacja algorytmów nie odbiega znacząco od siebie, dlatego w opisach kolejnych algorytmów będę opisywał różnice w implementacji w stosunku do pierwszego Systemu Mrówkowego.

## 3.1 Język C#

Jak podaje "[10]": C# (wymowa "See sharp") jest prostym, nowoczesnym, obiektowym i bezpiecznym językiem programowania. C# ma swoje korzenie w języku c i będzie znany programistom C, C++ i Java. C# jest znormalizowany przez ECMA International jako standard ECMA-334 oraz przez ISO/IEC jako standard ISO/IEC 23270. Kompilator C# firmy Microsoft dla .NET Framework jest zgodną wersją obu tych standardów. Język ten powstawał w latach 1998-2001 pod zarządem Andersa Heisenberga dla firmy Microsoft.

## 3.2 System mrówkowy - implementacja

Implementacja pierwszego i kolejnych algorytmów opiera się o trzy klasy: Colony, Graph oraz Program.

W klasie Colony przechowuje się informacje o długości przebytej trasy przez mrówkę zapisaną w zmiennej distance oraz listę odwiedzonych przez nią wierzchołków zapisaną w zmiennej List.

W klasie Graph natomiast znajdują się wszystkie informacje dotyczące grafu takie jak: ilość wierzchołków w grafie, położenie wierzchołków w grafie, wartość feromonów i wartość heurystyczna pomiędzy krawędziami. Klasa ta zawiera również stałe parametry takie jak: alfa, beta oraz parowanie. Zawiera również informacje o najlepszej trasie (długość oraz lista odwiedzonych wierzchołków). W klasie tej znajdują się odpowiednio dwie metody:

ExampleFile oraz CreateT.

Metoda ExampleFile odpowiada za wczytanie podanego problemu z pliku i na podstawie zawartych w nim informacji o współrzędnych wierzchołków w przykładowym grafie zapisuje odległości pomiędzy wierzchołkami w grafie w macierzy edges (obliczam odległość między punktami na podstawie wzoru $|AB| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$). Jako argumenty metoda ta otrzymuje zmienne len oraz path2 zawierające odpowiednio ilość wierzchołków w przykładowym grafie, oraz nazwę pliku zawierającego współrzędne grafu.

```
public void ExampleFile(string path2)
{
    string dir = Directory.GetParent(Directory.GetCurrentDirectory()).Parent.Parent.FullName;
    string path = dir + "\\" + path2;
    string[] lines = File.ReadAllLines(path);
    int z = 0;
    x = new double[this.len];
    y = new double[this.len];
    foreach (var line in lines) // zapisanie wartosci do tablc na podstawie pliku
    {
        string firstValue = line.Split(new string[] { " ", "." }, StringSplitOptions.RemoveEmptyEntries)[1];
        string secondtValue = line.Split(new string[] { " ", "." }, StringSplitOptions.RemoveEmptyEntries)[2];
        x[z] = Convert.ToDouble(firstValue);
        y[z] = Convert.ToDouble(secondtValue);
        z++;
    }

    edges = new double[this.len][];

    for (int i = 0; i < this.len; i++) //obliczam odleglosc miedzy wierzcholkami
    {
        edges[i] = new double[this.len];
        for (int j = 0; j < this.len; j++)
        {
            double x1 = x[i];
            double x2 = x[j];
            double y1 = y[i];
            double y2 = y[j];

            double part1 = Math.Pow(x2 - x1, 2);
            double part2 = Math.Pow(y2 - y1, 2);


            edges[i][j] = Math.Sqrt((part1) + (part2)); //odleglosc miedzy punktami
        }
    }
}
```

**Rysunek 2.** Metoda ExampleFile.

Druga metoda, CreateT, odpowiada za stworzenie dwóch macierzy tau i eta. Macierz tau odpowiada za przetrzymywanie informacji o wartości feromonowej na danej krawędzi, natomiast eta za informacje o jakości przejścia. Dane te potrzebne są do obliczenia prawdopodobieństwa z wzoru (2).

```
public void CreateT(double c)
{
    tau = new double[len][];
    eta = new double[len][];
    for (int i = 0; i < len; i++)
    {
        tau[i] = new double[len];
        eta[i] = new double[len];
        for (int j = 0; j < len; j++) // inicjowanie macierzy feromonow i macierzy jakości przejscia do węzła
        {
            tau[i][j] = c;
            eta[i][j] = 1 / edges[i][j];
        }
    }
}
```

**Rysunek 3.** Metoda CreateT.

Klasa Colony zawiera również konstruktor, który przyjmuje jako argument numer badanego przykładu, następnie na podstawie numeru przykładu ustawiam ilość wierzchołków w grafie oraz uruchamiam metodę ExampleFile z odpowiednią nazwą pliku.

Klasa Program jest natomiast główną klasą programu z metodą main, w której uruchamiany jest algorytm mrówkowy, oraz z pomocnymi metodami potrzebnymi do algorytmu takimi jak: GenerateSolution, Take_best_result, Update_pheromone, Evaporation, CalculateCostOfPath, NearestNeighbour i RouletWheel.

Metoda NearestNeighbour służy do znalezienia cyklu Hamiltona algorytmem najbliższego sąsiada i zwraca długość tego cyklu. Algorytm ten jest zachłanną metodą rozwiązania problemu komiwojażera. W algorytmie mrówkowym służy on jedynie do zainicjalizowania początkowych wartości feromonu według wzoru (1).

```
static double NearestNeighbour (Graph g)
{
    List<int> Tour = new List<int>(g.len);
    double distance = 0;
    Random rnd = new Random();
    int initialNode = rnd.Next(0, g.len); //losowanie poczatkowego indeksu
    Tour.Add(initialNode);
    int currentNode = initialNode;
    int nextNode = 0;
    for (int i = 0; i < g.len; i++)
    {
        double minval = Double.MaxValue;
        for (int y = 0; y < g.len; y++)
        {
            double distancetoCity = g.edges[currentNode][y];
            if (!Tour.Contains(y) && distancetoCity < minval)
            {
                minval = distancetoCity;
                nextNode = y;
            }
        }
        Tour.Add(nextNode);
        distance += g.edges[currentNode][nextNode];
        currentNode = nextNode;
    }
    distance += g.edges[currentNode][initialNode];
    return distance;
}
```

**Rysunek 4.** Metoda NearestNeighbour.

Metoda CalculateCostsOfPath zwraca długość cyklu Hamiltona podanej mrówki. Jako argumenty przyjmuje badany graf oraz mrówkę. Następnie na podstawie listy odwiedzonych przez mrówkę wierzchołków obliczana jest długość przebytej przez nią trasy korzystając z macierzy edges, która przetrzymuje informacje o długości wszystkich krawędzi w grafie.

```
static double CalculateCostOfPath(Graph g, Colony ant)
{
    double distance = 0;

    for (int i = 0; i < g.len; i++)
    {
        int currentNode = ant.Tour[i];
        int nextNode = ant.Tour[i + 1];
        distance += g.edges[currentNode][nextNode];
    }
    return distance;
}
```

**Rysunek 5.** Metoda CalculateCostOfPath.

Metoda RouleteWheel implementuje wybór danej mrówki w odpowiednim kroku algorytmu na podstawie tablicy prawdopodobieństw tNode. Dokładny opis tego algorytmu znajduje się w rozdziale 2.1.

```
static int RouleteWheel(double[] tNode, int nCities, List<int> list)
{
    double sum = 0;
    Random rnd = new Random();
    double rNumber = rnd.NextDouble();
    for (int i = 0; i < nCities; i++)
    {
        sum += tNode[i];
        if (!list.Contains(i) && rNumber < sum)
        {
            return i;
        }

    }
    return -1;
}
```

**Rysunek 6.** Metoda RouleteWheel

Metoda TakeBestResult jako argumenty przyjmuje tablice mrówek (obiektów z klasy Colony), graf oraz liczbę mrówek w kolonii. W wyniku działania aktualizuje zmienne należące do klasy Graph: bestDistance oraz bestPath odpowiadające odpowiednio za długość najkrótszego znalezionego cyklu Hamiltona, oraz za kolejność odwiedzanych wierzchołków w tym cyklu.

```
static void TakeBestResult(Graph g, Colony[] m, int nAnts)
{
    for (int i = 0; i < nAnts; i++)
    {
        if (m[i].distance < g.bestDistance)
        {
            g.bestDistance = m[i].distance;
            Console.WriteLine("Najkrótsza trasa do tej pory: " + g.bestDistance);

            for (int j = 0; j < m[i].Tour.Count - 1; j++)
            {
                g.bestPath[j] = m[i].Tour[j];
            }

            g.bestPath[g.len] = m[i].Tour.Last();
        }
    }
}
```

**Rysunek 7.** Metoda TakeBestResult.

Metoda GenerateSolution jest główną metodą odpowiedzialną za przebieg jednego cyklu kolonii mrówkowej. Jako argumenty przyjmuje Graf oraz tablice mrówek. Jest to najbardziej kosztowna czasowo metoda ze złożonością czasową $O\ (n^{\ 2} *\ m)$, gdzie $n$ to liczba wierzchołków natomiast $m$ to liczba mrówek. Główna pętla odpowiada za skonstruowanie rozwiązania dla pojedynczej mrówki, każdy przebieg pętli konstruuje rozwiązanie dla pojedynczego insekta. Na początku każda mrówka losuje swój wierzchołek startowy i zapisuje jego numer do listy odwiedzonych wierzchołków (do wylosowania wierzchołka skorzystałem z metody Random.Next zawartej w klasie Random). Następnie w kolejnej pętli obliczamy prawdopodobieństwo wyboru każdej krawędzi według wzoru (2) i zapisuję je w tablicy pAllNodes. Gdy obliczone zostają wszystkie prawdopodobieństwa, uruchomiona jest metoda RouletteWheel przyjmująca jako argumenty tablice pAllNodes w celu wybrania kolejnej krawędzi w grafie. Gdy mrówka za pomocą metody RouletteWhell dokona wszystkich wyborów, na końcu powraca do początkowego wierzchołka i zapisuje go w swojej liście. Metoda ta kończy się w momencie, gdy wszystkie mrówki w kolonii dokonają swojego rozwiązania.

```
static void GenerateSolution(Graph g, Colony[] a,int antNo)
{
    int nodeNo = g.len;
    for (int i=0;i<antNo;i++) // przechodzimy po wszystkich mrowkach
    {
        Random rnd = new Random();
        int initialNode =  rnd.Next(0, nodeNo); //losowanie poczatkowego indeksu
        a[i] = new Colony(nodeNo); // tworze mrowke
        a[i].Tour = new List<int>(nodeNo); // dodaje jej poczatkowe wartosci
        a[i].Tour.Add(initialNode); // wybranie dla danej mrowki miasto startowe

        for (int j=1;j<nodeNo;j++) // kolejne wybory mrowki
        {
            int currentNode = a[i].Tour.Last(); //
            double sum = 0; // suma tablicy tau i eta
            for (int x=0;x<nodeNo;x++) // przejsc po wszysykich krawedziach i obliczenie sumy
            {
                if (!(a[i].Tour.Contains(x))) // Jezeli nie odwiedzilismy
                {
                    double tij = g.tau[currentNode][x];
                    double nij = g.eta[currentNode][x];
                    sum += Math.Pow(tij, g.alpha) * Math.Pow(nij, g.beta);
                }
            }
            double [] pAllNodes = new double[nodeNo]; //lista prawdopodobienstw
```

**Rysunek 8a.** Metoda GenerateSolution.

```
            for (int y = 0; y < nodeNo; y++)
            {
                double pij;
                if (!(a[i].Tour.Contains(y))) // Jezeli nie odwiedzilismy
                {
                    double tij = g.tau[currentNode][y];
                    double nij = g.eta[currentNode][y];
                    pij = Math.Pow(tij, g.alpha) * Math.Pow(nij, g.beta) / sum;

                }
                else // jezli odwiedzilismy wierzcholek to p=0
                {
                    pij = 0;
                }
                pAllNodes[y] = pij;
            }
            int selectedNode = RouleteWheel(pAllNodes, nodeNo,a[i].Tour);
            a[i].Tour.Add(selectedNode);
        }
        a[i].Tour.Add(initialNode); //powracamy do poczatkowego punktu
    }
}
```

**Rysunek 8b**.  Metoda Generate Solution.

Metoda UpdatePheromone służy do uaktualnienia wartości feromonowych w grafie zgodnie ze wzorem (4). Jako argumenty przyjmuje tablice mrówek oraz graf. Na podstawie znalezionych Cykli Hamiltona  uaktualnia wartości feromonowe na krawędziach

odwiedzonych przez mrówki. Zwiększenie wartości feromonowych w moim przypadku odpowiada zwiększeniu odpowiednich elementów w macierzy tau.

```
static void UpdatePheromone(Graph g, Colony[]m, int nAnts)
{
    for(int i=0;i<nAnts;i++)
    {
        for(int j=0;j<g.len-1;j++) //dodawanie sladu przez dana mrowke
        {
            int currentNode = m[i].Tour[j];
            int nextNode = m[i].Tour[j + 1];
            g.tau[currentNode][nextNode]+= 1.0/m[i].distance;
            g.tau[nextNode][currentNode]+= 1.0/ m[i].distance;
        }
    }
}
```

**Rysunek 9.** Metoda UpdatePheromone.

Metoda Evaporation jest odpowiedzialna za wyparowywanie śladów feromonowych na wszystkich krawędziach należących do grafu. Zmniejsza ona wartość macierzy tau zgodnie ze wzorem (3).

```
static void Evaporation(Graph g)
{
    for(int i=0;i<g.len;i++)
    {
        for(int j=0;j<g.len;j++)
        {
            g.tau[i][j] = g.tau[i][j]* (1.0 - g.evaporation);
        }
    }
}
```

**Rysunek 10.** Metoda Evaporation.

Metoda Main jest główną metodą programu i odpowiada za uruchomienie algorytmu. Na początku metody tworzę instancję klasy Graph, podając jako argument numer badanego przykładu. Następnie ustalam ilość mrówek i iteracji, obliczam długość cyklu Hamiltona algorytmem najbliższego sąsiada i na podstawie długości tego cyklu inicjalizuje wartości feromonów według wzoru (1). Po początkowych ustawieniach włączam zegar w celu testowania, korzystam z klasy Stopwatch zawartej w bibliotece System.Diagnostics.

```
//Tworzenie grafu
Graph g = new Graph(3);


int maxinter = 500;
int nAnts = g.len;
double iVal = NearestNeighbour (g);
double T0;
T0 = nAnts / (g.len*iVal); // poczatkowe wartosci feromonow
g.CreateT(T0); // tworzenie macierzy eta i tau


Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();
```

**Rysunek 11.** Przygotowanie grafu, ustawienie parametrów i włączenie zegara - System mrówkowy.

Po przygotowaniu grafu następuje główna pętla algorytmu mrówkowego. Pętla ta wykonuje się tyle razy, ile zadeklarowałem w zmiennej maxinter. Jeden przebieg pętli odpowiada za jeden przebieg kolonii mrówkowej. Na początku pętli tworze tablice instancji klasy Colony reprezentująca kolonię mrówek. Następnie uruchamiam metodę GenerateSolution, która znajduje Cykl Hamiltona dla każdej mrówki w kolonii. Po utworzeniu rozwiązań aktualizuje najlepszą znalezioną trasę, wykorzystując do tego metodę TakeBestResult. Po utworzeniu rozwiązań uruchamiam metody Evaporation oraz UpdatePheromone odpowiedzialne za uaktualnienie wartości feromonowych według wzoru (3) dla Evaporation i (4) dla UpdatePheromone.

```
for (int t=0;t<maxinter;t++) // Główna pętla dla algorytmu
{
    Colony[] m = new Colony[nAnts]; //tworzymy kolonie
    GenerateSolution(g, m, nAnts);

    for (int i=0;i<nAnts;i++)
    {
        m[i].distance=CalculateCostOfPath(g, m[i]);
    }
    TakeBestResult(g,m,nAnts); // Znalezienie mrowki z najlepszym rozwiazaniem
    Evaporation(g); // Odparowanie sladow feromonowych
    UpdatePheromone(g,m,nAnts); // Zaktualizowanie sladow feromonowych

}
```

**Rysunek 12.** Główna pętla algorytmu mrówkowego.

Po wykonaniu algorytmu zatrzymuje zegar oraz wypisuje wyniki algorytmu na ekranie. Otrzymane rezultaty zapisuje do pliku tekstowego w celu testowania wydajności algorytmu. Do zapisania rezultatów do pliku korzystam z klasy StreamWriter zawartej w bibliotece System.IO, natomiast do uzyskania informacji o ścieżce folderu z rozwiązaniem wykorzystałem metody Directory.GetParent oraz Directory.GetCurrentDirectory. Rezultaty zapisuje w pliku txt o nazwie zależnej od ilości wierzchołków w badanym problemie komiwojażera.

```
stopWatch.Stop();

Console.WriteLine("Najlepsze rozwiazanie :");
Console.WriteLine(g.bestDistance);
for(int q=0;q<g.len+1;q++)
{
    Console.Write(g.bestPath[q]+1+"->");
}

Console.WriteLine("Czas wykonania: ", stopWatch.Elapsed);

string dir = Directory.GetParent(Directory.GetCurrentDirectory()).Parent.Parent.FullName;
string result = Convert.ToString(g.bestDistance);
result += " ";
result += stopWatch.Elapsed;
string tmppath = Convert.ToString(g.len);
string path = dir + "\\" + "result" + tmppath + ".txt";
using (System.IO.StreamWriter file =
new System.IO.StreamWriter(path, true))
{
    file.WriteLine(result);
}
```

**Rysunek 13.** Wyświetlenie oraz zapisanie wyników algorytmu.

## 3.3 Elitarny Algorytm Mrówkowy - implementacja

Implementacja Elitarnego Algorytmu mrówkowego nie odbiega znacząco od podstawowej wersji algorytmu, wynika to z niewielkiej modyfikacji w sposobie nakładania śladów feromonowych. Klasy Colony oraz Graph wyglądają tak samo, jak w systemie mrówkowym. Zmiany występują w metodzie UpdatePheromone oraz Main znajdujących się w klasie Program.

W Metodzie UpdatePheromone dodana została nowa pętla, która odpowiada za wzmocnienie feromonów na najkrótszej znalezionej trasie według wzoru (6).

```
static void UpdatePheromone(Graph g, Colony[] m, int nAnts,double e)
{
    for (int i = 0; i < nAnts; i++)
    {
        for (int j = 0; j < g.len - 1; j++) //dodawanie sladu przez dana mrowke
        {
            int currentNode = m[i].Tour[j];
            int nextNode = m[i].Tour[j + 1];
            g.tau[currentNode][nextNode] += 1.0 / m[i].distance;
            g.tau[nextNode][currentNode] += 1.0 / m[i].distance;
        }
    }

for (int j = 0; j < g.len - 1; j++) //dodatkowy slad feromonowy
{
    int currentNode = g.bestPath[j];
    int nextNode = g.bestPath[j + 1];
    g.tau[currentNode][nextNode] += 1.0 / g.bestDistance + e / g.bestDistance;
    g.tau[nextNode][currentNode] += 1.0 / g.bestDistance + e / g.bestDistance;
}
```

**Rysunek 14.** Metoda UpdatePheromone - Elitarny Algorytm Mrówkowy.

W metodzie Main dodaje nową zmienną e odpowiedzialną za wzmocnienie najlepszej trasy podczas aktualizowania feromonów. Zmianie ulega zmienna T0 odpowiedzialna za początkową wartość feromonów na trasie.

```
//Tworzenie grafu
Graph g = new Graph(3);

int maxinter = 500;
int nAnts = g.len;
double iVal = NearestNeighbour (g);
double T0;
double e = g.len;
T0 = (nAnts +e) / (g.evaporation * iVal);
g.CreateT(T0); // tworzenie macierzy eta i tau

Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();
```

**Rysunek 15.** Przygotowanie grafu, ustawienie parametrów i włączenie zegara - Elitarny Algorytm Mrówkowy.

W głównej pętli algorytmu jedyna zmiana, jaka następuje w stosunku do pierwszego algorytmu to dodanie nowego parametru e podczas uruchomienia metody UpdatePheromone.

```
for (int t = 0; t < maxinter; t++) // Główna pętla dla algorytmu
{
    Colony[] m = new Colony[nAnts]; //tworzymy kolonie
    GenerateSolution(g, m, nAnts);

    for (int i = 0; i < nAnts; i++)
    {
        m[i].distance = CalculateCostOfPath(g, m[i]);
    }
    TakeBestResult(g, m, nAnts); // Znalezienie mrowki z najlepszym rozwiazanie
    Evaporation(g); // Odparowanie sladow feromonowych
    UpdatePheromone(g, m, nAnts,e); // Zaktualizowanie sladow feromonowych

}
```

**Rysunek 16.** Główna pętla Elitarnego Algorytmu Mrówkowego

Wyświetlanie oraz zapisanie rezultatów algorytmu przebiega tak samo, jak w Systemie Mrówkowym.

## 3.4 Algorytm Mrówkowy z rankingiem (ASrank)- implementacja

Algorytm mrówkowy z rankingiem podobnie jak poprzedni algorytm różni się od bazowego sposobem odkładania śladu feromonowego. Do poprawnej aktualizacji śladu feromonowego wymagane jest sortowanie mrówek na podstawie jakość ich rozwiązania. W celu posortowania mrówek trzeba skorzystać z jednego z wielu algorytmów sortujących. W moim rozwiązaniu zastosowałem Algorytm Quicksort [4] ze względu na dobrą optymistyczną złożoność obliczeniową $O\ (n\ log\ n)$ oraz łatwość w implementacji.

Implementacja tego algorytmu jest w metodzie QuickSort. Jako argumenty przyjmuje ona tablice instancji klasy Colony reprezentującą kolonie mrówek, indeks początkowy oraz końcowy tablicy. Algorytm Quicksort dokładnie opisałem w rozdziale 2.3.

```
static void QuickSort(Colony[] m, int left, int right) // Sortowanie mrówek względem przebytej trasy algorytmem QuickSort
{
    var i = left;
    var j = right;
    double pivot = m[(left + right) / 2].distance;
    while (i < j)
    {
        while (m[i].distance < pivot) i++;
        while (m[j].distance > pivot) j--;
        if (i <= j)
        {
            // swap
            var tmp = m[i];
            m[i++] = m[j];  // ++ and -- inside array braces for shorter code
            m[j--] = tmp;
        }
    }
    if (left < j) QuickSort(m, left, j);
    if (i < right) QuickSort(m, i, right);
}
```

**Rysunek 17.** Metoda QuickSort.

Zmieniona została również implementacja metody UpdatePheromone. Przyjmuje ona dodatkowy argument, który wykorzystywany jest we wzorze na nakładanie nowego śladu feromonowego (8).

```
static void UpdatePheromone(Graph g, Colony[] m, int nAnts,int w)
{
    for (int i = 0; i < nAnts; i++)
    {
        for (int j = 0; j < g.len - 1; j++) //dodawanie sladu przez dana mrowke
        {
            int currentNode = m[i].Tour[j];
            int nextNode = m[i].Tour[j + 1];
            g.tau[currentNode][nextNode] += (w - i + 1) *(1.0 / m[i].distance) + w *(1.0/g.bestDistance);
            g.tau[nextNode][currentNode] += (w - i + 1) * (1.0 / m[i].distance) + w * (1.0 / g.bestDistance);
        }
    }
}
```

**Rysunek 18.** Metoda UpdatePheromone - Algorytm Mrówkowy z rankingiem.

W metodzie Main  w fazie przygotowania grafu dodane zostały dwa dodatkowe parametry $r$ oraz $w$ odpowiedzialne za początkową wartość feromonu a także ilość mrówek dopuszczonych do odkładania śladu feromonowego.

```
//Tworzenie grafu
Graph g = new Graph(3);


int maxinter = 500;
int nAnts = g.len;
double iVal = NearestNeighbour (g);
double T0;
int r = 6;
T0 = 0.5 *r*(r-1) / 0.1* (g.evaporation * iVal);
int w=6; // liczba mrówek dopuszczona do odkładania feromonu
g.CreateT(T0); // tworzenie macierzy eta i tau

Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();
```

**Rysunek 19**. Przygotowanie grafu - Algorytm Mrówkowy z rankingiem.

W głównej pętli algorytmu przed aktualizacją śladów feromonowych sortuję mrówki na podstawie jakość znalezionej przez nice trasy, uruchamiając metodę QuickSort. Metoda UpdatePheromone przyjmuje jako argumenty graf, tablicę mrówek, liczbę mrówek odkładającą ślady feromonowe.

```
for (int t = 0; t < maxinter; t++) // Główna pętla algorytmu
{
    Colony[] m = new Colony[nAnts]; //tworzymy kolonie
    GenerateSolution(g, m, nAnts);

    for (int i = 0; i < nAnts; i++)
    {
        m[i].distance = CalculateCostOfPath(g, m[i]);
    }
    TakeBestResult(g, m, nAnts); // Znalezienie mrowki z najlepszym rozwiazaniem
    Evaporation(g); // Odparowanie sladow feromonowych
    QuickSort(m, 0, nAnts - 1); // Posortowanie mrówek
    UpdatePheromone(g, m, w-1,w); // Zaktualizowanie sladow feromonowych

}
```

**Rysunek 20.** Główna pętla Algorytmy Mrówkowego z rankigiem.

Wyświetlenie oraz zapisanie wyników pracy algorytmu nie ulegają zmianie.

## 3.5 Algorytm Mrówkowy Max-Min (MMAS) - implementacja

Implementacja Algorytmu Max-Min, podobnie jak w przypadku poprzedniego algorytmu, różni się zmianą metody odpowiedzialną za odkładania śladów feromonowych, zmianą początkowych oraz dodaniem dodatkowej metody. Nowa metoda w tym algorytmie odpowiada za nałożenie limitu górnego i dolnego śladów feromonowych.

Metodą odpowiedzialną za nałożenie limitów jest AddLimits. Metoda ta jako argumenty przyjmuje graf oraz zmienną Cmn. Zmienna ta przetrzymuje początkowo wartość feromonów według wzoru (9). Na początku ustawiam zakresy w zmiennych maxLimit oraz minLimit korzystając ze wzorów (11) oraz (12). Metoda ta sprawdza wszystkie wierzchołki w grafie i jeżeli na którymś wartość feromonu przekracza limit, to zostaje zwiększona do dolnego limitu, jeżeli była za mała, lub do górnego, jeżeli była za duża.

```
static void AddLimits(Graph g)
{
    double maxLimit = 1 / ( g.evaporation * g.bestDistance);
    double avg = g.len / 2.0;
    double pdec = Math.Pow(0.05, 1.0 / g.len);
    double a = ((avg - 1.0) * pdec) / ( 1.0 - pdec);
    double minLimit = maxLimit / a;
    for(int i=0;i<g.len ;i++)
    {
        for(int j = 0; j < g.len; j++)
        {
            if (g.tau[i][j] > maxLimit &i!=j)
            {
                g.tau[i][j] = maxLimit;
            }
            if (g.tau[i][j] < minLimit &i!=j)
            {
                g.tau[i][j] = minLimit;
            }
        }

    }
}
```

**Rysunek 21.** Metoda AddLimits.

Metoda UpdatePheromone jako argumenty przyjmuje Graf oraz pojedynczą mrówkę. Metoda ta w tym rozwiązaniu posiada tylko jedną pętlę, w której aktualizuje feromony należące do trasy mrówki podanej w argumencie, stosując wzór (10).

```
static void UpdatePheromone(Graph g,Colony a)
{
        for (int j = 0; j < g.len ; j++) //dodawanie sladu przez dana mrowke
        {
            int currentNode = a.Tour[j]; // bład wychodzi za zakres
            int nextNode = a.Tour[j + 1];
            g.tau[currentNode][nextNode] += 1.0 / a.distance;
            g.tau[nextNode][currentNode] += 1.0 / a.distance;
        }
}
```

**Rysunek 22.** Metoda UpdatePheromone - Algorytm Mrówkowy Max-Min.

W metodzie TakeBestResult jako argumenty dodałem dwie instancje klasy Colony bIntAnt oraz bAnt, które przetrzymują najlepszą trasę znalezioną przez mrówki w pojedynczym przebiegu kolonii (bIntAnt) a także najlepszą znalezioną trasę od początku działania algorytmu (bAnt).

```
static void TakeBestResult(Graph g, Colony[] m, int nAnts,Colony bAnt,Colony bIntAnt)
{
    double bIntDistance = double.MaxValue;

    for (int i = 0; i < nAnts; i++)
    {
        if (m[i].distance < g.bestDistance)
        {
            g.bestDistance = m[i].distance;
            Console.WriteLine("Najkrótsza trasa do tej pory : "+ g.bestDistance);

            for (int j = 0; j < m[i].Tour.Count - 1; j++)
            {
                g.bestPath[j] = m[i].Tour[j];
            }
            g.bestPath[g.len] = m[i].Tour.Last();
            bAnt.Tour = m[i].Tour;
            bAnt.distance = m[i].distance;
        }

        if (m[i].distance < bIntDistance)
        {
            bIntAnt.Tour = m[i].Tour;
            bIntAnt.distance = m[i].distance;
        }
    }
}
```

**Rysunek 23.** Metoda TakeBestResult - Algorytm Mrówkowy Max-Min.

W głównej metodzie Main w fazie przygotowania grafu zmieniona zostaje zmienna T0, której wartość obliczana jest na podstawie wzoru (8). Utworzone są również dwie

instancje klasy Colony bAnt oraz bIntAnt. Reprezentują one odpowiednio mrówkę, która znalazła najlepsze rozwiązanie do tej pory oraz mrówkę z najlepszym rozwiązaniem w danej iteracji głównej pętli algorytmu. Dodana jest również zmienna probability, która określa prawdopodobieństwo wyboru jednego z dwóch sposobów odkładania śladu feromonowego.

```
//Tworzenie grafu
Graph g = new Graph(4);


int maxinter =500;
int nAnts = g.len;
double iVal = NearestNeighbour (g);
double T0;
T0 = 1.0 / (g.evaporation * iVal); // poczatkowe wartosci feromonow
g.CreateT(T0); // tworzenie macierzy eta i tau

Colony bAnt = new Colony(g.len);
Colony bIntAnt = new Colony(g.len);
double probability = 0.9;

Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();
```

**Rysunek 24.** Przygotowanie grafu Algorytm Mrówkowy Max-Min.

W głównej pętli algorytmu dodałem losowanie wyboru jednej z dwóch opcji odkładania śladu feromonowego. Wybór ten oparty jest o zmienną probability, która zmniejsza się co 100 iteracji, aby w miarę postępu algorytmu zwiększyć wpływ najlepszej znalezionej trasy w trakcie działania algorytmu, kosztem najlepszej trasy w jednym cyklu. Po zaktualizowaniu śladów feromonowych uruchamiam metodę AddLimits w celu nałożenia limitów na feromony.

```
for (int t = 0; t < maxinter; t++) // Główna pętla dla algorytmu
{
    Colony[] m = new Colony[nAnts]; //tworzymy kolonie
    GenerateSolution(g, m, nAnts);

    for (int i = 0; i < nAnts; i++)
    {
        m[i].distance = CalculateCostOfPath(g, m[i]);
    }

    TakeBestResult(g, m, nAnts,bAnt,bIntAnt); // Znalezienie mrowki z najlepszym rozwiazaniem w interacji i do tej pory
    Evaporation(g); // Odparowanie sladow feromonowych
    if(t%100==0) // zmieniam prawdopodobienstwo co 100 obrotów pętli
    {
            probability -= 0.1;

    }
    Random rnd = new Random();
    double tmp = rnd.NextDouble();

    if (tmp < probability)
    {
        UpdatePheromone(g, bIntAnt); }
    else
    {
        UpdatePheromone(g, bAnt);
    }
    AddLimits(g);
}
```

**Rysunek 25.** Główna pętla Algorytmu Mrówkowego Max-Min

## 3.7 System Kolonii Mrówek (ACS) - implementacja

Implementacja tego algorytmu różni się od podstawowej wersji nie tylko sposobem odkładania śladu feromonowego, lecz także funkcją przejścia. W związku z tym zmianom uległa metoda odpowiedzialna za wygenerowanie rozwiązania przez kolonie GenerateSolution. Niewielkiej zmianie uległa również klasa Graph, nowe zmienne to lokalEvaporation oraz $q_0$. Zmienna lovalEvaporation odpowiada za lokalne parowanie feromonu, natomiast zmienna $q_0$ określa prawdopodobieństwo wyboru jednej z dwóch opcji wyboru kolejnej krawędzi według wzoru (14).

Metoda UpdatePheromene działa podobnie jak w Algorytmie Mrówkowym Max-Min, czyli ogranicza się do jednej pętli, w której feromony są odkładane na pojedynczą trasę. Różnica polega na tym, że nakładanie nowych śladów połączone jest z ich parowaniem według wzoru (17).

```
static void UpdatePheromone(Graph g)
{
    for (int j = 0; j < g.len ; j++)
    {
        int currentNode = g.bestPath[j];
        int nextNode = g.bestPath[j + 1];
        g.tau[currentNode][nextNode] = (1 - g.evaporation) * g.tau[currentNode][nextNode] + g.evaporation * 1.0 / g.bestDistance;
        g.tau[nextNode][currentNode] = (1 - g.evaporation) * g.tau[nextNode][currentNode] + g.evaporation * 1.0 / g.bestDistance;
    }
}
```

**Rysunek 26.** Metoda UpdatePheromone.

Metoda GenerateSolution przyjmuje dodatkowy parametr tau0, który jest wykorzystywany podczas lokalnej aktualizacji śladu feromonowego. Podczas tworzenia rozwiązania przez mrówkę dodaję losowanie liczby z zakresu od 0 do 1, zapisuję ją pod zmienną q a potem sprawdzam, czy jest ona mniejsza od zmiennej $q_0$. Następnie stosując wzór (14), dokonuje kolejnych wyborów. Po każdym dokonanym wyborze przez mrówkę aktualizowana jest wartość feromonów według wzoru (15). Złożoność czasowa tej metody zależy od ustawienia parametru $q_0$, dokładne działanie tego parametru na czas działania algorytmu opisałem w rozdziale 4.5.

```
static void GenerateSolution(Graph g, Colony[] a, int antNo, double tau0)
{
    int nodeNo = g.len;
    for (int i = 0; i < antNo; i++) // przechodzimy po wszystkich mrowkach
    {
        Random rnd = new Random();
        int initialNode = rnd.Next(0, nodeNo); //losowanie poczatkowego indeksu
        a[i] = new Colony(nodeNo); // tworze mrowke
        a[i].Tour = new List<int>(nodeNo); // dodaje jej poczatkowe wartosci
        a[i].Tour.Add(initialNode); // wybranie dla danej mrowki miasto startowe

        int selectedNode = 0;

        for (int j = 1; j < nodeNo; j++) // kolejne wybory mrowki
        {
            int currentNode = a[i].Tour.Last(); //aktualny wierzcholek danej mrowki
            double q = rnd.NextDouble(); // losuje sposob wyboru
            if (q < g.q0)
            {
                double argmax = double.MinValue;
                for (int x = 0; x < nodeNo; x++)
                {
                    if (!(a[i].Tour.Contains(x))) // Jezeli nie odwiedzilismy
                    {
                        double tmp = g.tau[currentNode][x] * Math.Pow((g.eta[currentNode][x]),g.beta);
                        if (argmax < tmp)
                        {
                            selectedNode = x;
                            argmax = tmp;
                        }
                    }
                }
            }
```

**Rysunek 27a.** Metoda GenerateSolution - System Kolonii Mrówek.

```
            else //wybor funkcja pseudolowa w przypadku gdy q>q0
            {
                double sum = 0; // suma tablicy tau i eta
                for (int x = 0; x < nodeNo; x++) // przejsc po wszysykich krawedziach i obliczenie sumy
                {
                    if (!(a[i].Tour.Contains(x))) // Jezeli nie odwiedzilismy
                    {
                        double tij = g.tau[currentNode][x];
                        double nij = g.eta[currentNode][x];
                        sum += Math.Pow(tij, g.alpha) * Math.Pow(nij, g.beta);

                    }
                }
                double[] p_allnodes = new double[nodeNo]; //lista prawdopodobienstw
                for (int y = 0; y < nodeNo; y++)
                {
                    double pij;
                    if (!(a[i].Tour.Contains(y))) // Jezeli nie odwiedzilismy
                    {
                        double tij = g.tau[currentNode][y];
                        double nij = g.eta[currentNode][y];
                        pij = Math.Pow(tij, g.alpha) * Math.Pow(nij, g.beta) / sum;
                    }
                    else // jezli odwiedzilismy wierzholek to p=0
                    {
                        pij = 0;
                    }
                    p_allnodes[y] = pij;
                }
                selectedNode = RouleteWheel(p_allnodes, nodeNo, a[i].Tour);
            }

            a[i].Tour.Add(selectedNode); //dodaje wybor mrowki do listy
            g.tau[currentNode][selectedNode] = (1.0 - g.lokalEvaporation) * g.tau[currentNode][selectedNode] + tau0 * g.lokalEvaporation;// dodanie lokalne sladow feromonowych
            g.tau[selectedNode][currentNode] = (1.0 - g.lokalEvaporation) * g.tau[selectedNode][currentNode] + tau0 * g.lokalEvaporation;
        }
        g.tau[selectedNode][initialNode] = (1.0 - g.lokalEvaporation) * g.tau[selectedNode][initialNode] + tau0 * g.lokalEvaporation;// dodanie sladu do sciezki powrotnej
        g.tau[selectedNode][initialNode] = (1.0 - g.lokalEvaporation) * g.tau[selectedNode][initialNode] + tau0 * g.lokalEvaporation;

        a[i].Tour.Add(initialNode); //powracamy do poczatkowego punktu
    }
}
```

**Rysunek 27b.** Metoda GenerateSolution - System Kolonii Mrówek.

W głównej metodzie Main w fazie przygotowania grafu do rozwiązania ponownie zmieniona zostaje zmienna T0, której wartość określa wzór (16).

```
//Tworzenie grafu
Graph g = new Graph(3);


int maxinter =5000;
int nAnts = 10;
double iVal = NearestNeighbour (g);
double T0;
T0 = 1.0 / (g.len * iVal); // poczatkowe wartosci feromonow
g.CreateT(T0); // tworzenie macierzy eta i tau

Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();
```

**Rysunek 28.** Przygotowanie grafu - System Kolonii Mrówek.

W głównej pętli algorytmu, w odróżnieniu od poprzednich rozwiązań nie ma metody Evapotarion. Brak jej wynika z tego, że parowanie zostało uwzględnione już podczas nakładania nowych śladów w metodach GenerateSolution oraz UpdatePheromone.

```
for (int t = 0; t < maxinter; t++) // Główna pętla dla algorytmu
{
    Colony[] m = new Colony[nAnts]; //tworzymy kolonie
    GenerateSolution(g, m, nAnts, T0);

    for (int i = 0; i < nAnts; i++)
    {
        m[i].distance = CalculateCostOfPath(g, m[i]);
    }
    TakeBestResult(g, m, nAnts); // Znalezienie mrowki z najlepszym rozwiazaniem
    UpdatePheromone(g);
}
```

**Rysunek 29.** Główna pętla Systemu Kolonii Mrówek

# Rozdział 4 - testowanie algorytmów

W tym rozdziale opisałem, jak poszczególne algorytmy mrówkowe radzą sobie z problemem komiwojażera. Porównałem tutaj czas działania poszczególnych algorytmów, złożoność czasową, oraz sprawdziłem jak poszczególne algorytmy, działają w zależności od ustawionych parametrów. W tabelach opisanych w tym rozdziale podałem średnie długości cyklu Hamiltona oraz średnie czasy wykonania algorytmu uruchomionego dziesięć razy na poszczególnych ustawieniach. Jako wyniki w poniższych tabelach prezentuje średnie długości Cyklu Hamiltona na 10 uruchomieniach w danym problemie.

Do testowania algorytmu wykorzystałem problemy ze strony: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/. Na tej stronie znaleźć można różne problemy komiwojażera opisane za pomocą punktów na osi kartezjańskiej. Algorytmy testowałem na laptopie Lenovo G70 o parametrach: Intel core i7 5500u 2.40 ghz, 8 GB RAM.

## 4.1 System Mrówkowy (AS)

Pierwszym algorytmem, który testowałem to System Mrówkowy. Na początku testów sprawdziłem jak algorytm spisuje się w zależności od ustawień parametrów $\alpha$, $\beta$, wartości parowania oraz ilości iteracji. W pierwszej tabeli porównałem wyniki w zależności ustawień $\alpha$oraz $\beta$.

**Tabela 1.**

Sprawdzenie optymalnych wartości $\alpha$oraz $\beta$na ustawieniach: $\rho = 0,5$, liczba iteracji = 500, $m = n$ (liczba mrówek równa liczbie miast) w przykładach dantzig 42 oraz pr 76.

| $\beta$ / $\alpha$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1<br>dantzig 42<br>pr 76 | 722,54<br>118565,16 | **708,13**<br>116185,87 | 716,86<br>**115761,08** | 728,55<br>115967,02 | 729,035<br>116576,5025 |

| | | | | | |
|---|---|---|---|---|---|
| 2<br>dantzig 42<br>pr 76 | 779,73<br>121766,3676 | 754,27<br>118710,77 | 756,76<br>1120205,23 | 760,88<br>119781,30 | 755,92<br>- |

Jak widać na powyższej tabelce, najlepsze wyniki algorytm osiągnął dla wartości $\alpha = 1$. Gdy zwiększamy wartość $\alpha$, dochodzimy wówczas szybciej do sytuacji stagnacji, czyli takiej, w której mrówki poruszają się głównie jedną trasę. A to ogranicza eksploatowanie nowych możliwie lepszych tras w późniejszych iteracjach algorytmu. Co do wartości $\beta$, z powyższej tabeli wynika, że algorytm spisuje najgorzej w przypadku $\beta = 1$, przy takim ustawieniu wpływ długości krawędzi jest zbyt duży w stosunku do wpływu feromonów na kolejne wybory. W zależności od przypadku do optymalnego rozwiązania należy wybrać ten parametr na równy od 2 do 5. W badanych przeze mnie przykładach najlepsze wyniki wychodzą dla $\beta = 2$ i $\beta = 3$.

**Tabela 2.**

Działanie algorytmu w zależności od ustawień parametru $\rho$, na przykładzie berlin 52 oraz eli 101. Pozostałe ustawienia: $\alpha = 1, \beta = 3, m = n$, liczba iteracji 500.

| $\rho$ | 0,1 | 0,3 | 0,5 | 0,6 | 0,7 |
|---|---|---|---|---|---|
| berlin 52 | 7630,20 | 7611,99 | **7527,90** | 7585,34 | 7667,37 |
| eli 101 | 697,86 | **695,06** | 696,19 | 700,04 | 699,10 |

Z tabeli można zauważyć, że najlepsze rezultaty algorytm osiąga w okolicach $\rho = 0,5$. Taki rezultat wynika z tego, że w tej wersji algorytmu każda mrówka odkłada ślad feromonowy, natomiast odparowywanie następuje po jednym cyklu algorytmu. Ustawienie parametru $\rho$ odpowiada za ustawienie dobrego balansu pomiędzy eksploracją a eksploatacją ścieżek w grafie. Zbyt mała wartość $\rho$ powoduje za małą eksplorację i za dużą eksploatację tras, natomiast zbyt duża wartość $\rho$ prowadzi do odwrotnej sytuacji.

**Tabela 3**

Sprawdzenie działania algorytmu w zależności od ilości iteracji na przykładach: dantzig 42, berlin 52, kroA 100 na ustawieniach : $\alpha$=1, $\beta$=3, $\rho$ = 0.5, z uwzględnieniem czasu działania.

| | 100 | 300 | 500 | 700 | 1000 |
|---|---|---|---|---|---|
| dantzig 42 | 729,78<br>02,037s | 718,302<br>05,11s | 716,31<br>08,13s | 710,80<br>11,16s | 715,14<br>15,75s |
| berlin 52 | 7687,40<br>03,38s | 7642,50<br>08,87s | 7572,90<br>14,33s | 7616,17<br>19,86s | 7573,54<br>28,17s |
| kroA 100 | 23010,66<br>22,87s | 22708,74<br>1m 06,141s | 22717,87<br>1m 49,23 s | 22598,58<br>2m 34,76s | 22652,11<br>3m 37,10s |

Z powyższej tabeli można wywnioskować, że algorytm znajduje najkrótsze trasy w okolicach 500 iteracji. W momencie gdy zwiększam liczbę iteracji, różnica w rezultatach wynika z początkowych wyborów mrówek. Widać też, że złożoność czasowa znacząco zwiększa się wraz z trudniejszym problemem. Jest to wynikiem zwiększenia ilość mrówek oraz liczby wierzchołków do odwiedzenia wraz z każdym trudniejszym przykładem. Natomiast każdy pojedynczy wybór mrówki również wymaga więcej obliczeń. Badania przeprowadzone przez Marco Dorigo [3] wykazały, że złożoność obliczeniowa algorytmu wynosi $O\ (m\ *\ n\ +\ n^3)$, m- liczba mrówek, n-liczba miast.

## 4.2 Elitarny Algorytm mrówkowy (EAS)

Kolejnym testowanym algorytmem jest Elitarny Algorytm mrówkowy, Podobnie jak w podstawowej wersji algorytmu sprawdzałem działanie algorytmu na pięciu przygotowanych grafach.

**Tabela 4.** Sprawdzenie działania Elitarnego Algorytmu Mrówkowego w zależności od ustawień parametrów αoraz β. Pozostałe ustawienia: $\rho = 0.5$, liczba iteracji 500, $m\ =\ n$.

| β<br>α | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1<br>dantzig 42<br>pr 76 | 709,84<br>113822,13 | 692,61<br>**111839,26** | 690,35<br>111944,93 | 690,47<br>112397,55 | **688,37**<br>112635,88 |

| | | | | | |
|---|---|---|---|---|---|
| 2<br>dantzig 42<br>pr 76 | 843,06<br>141360,77 | 737,12<br>120463,10 | 726,51<br>117233,89 | 711,34<br>115170,64 | 712,48<br>- |

Jak widać na powyższej tabelce, najlepsze wyniki osiągane przez tę wersję algorytmu są znacząco lepsze od podstawowej. Podobnie jak w pierwszej wersji najlepsze rezultaty uzyskujemy dla wartości $\alpha = 1$ oraz dla wartości βzawierającej się w przedziale od 2 do 5. W badanych problemach uzyskałem najlepsze średnie rezultaty dla ustawień odpowiednio: $\alpha = 1, \beta = 2$ oraz $\alpha = 1, \beta = 5$. Podobne najlepsze ustawienia jak w pierwszym algorytmie wynikają z niewielkich zmian w sposobie odkładania feromonu. Gorsze rezultaty w porównaniu do systemu mrówkowego uzyskałem tylko przy ustawieniach: $\alpha = 2, \beta = 1$, wynika to z większej ilości nakładanego feromonu, co prowadzi do małej eksploracji danych.

**Tabela 5.**

Sprawdzenie działania algorytmu w zależności od ustawień parametru ρ. Pozostałe parametry:$\alpha = 1, \beta = 3, m = n$ oraz liczba iteracji 500.

| ρ | 0,1 | 0,3 | 0,5 | 0,6 | 0,7 |
|---|---|---|---|---|---|
| berlin 52 | 7579,45 | 7579,35 | **7548,28** | 7555,80 | 7624,30 |
| eli 101 | 652,05 | 652,66 | **650,48** | 655,33 | 654,55 |

W obydwu badanych przypadkach najlepsze wyniki uzyskano dla $\rho = 0,5$. Parametr ρ do optymalnego działania podobnie jak w systemie mrówkowym powinien zawierać się w okolicach 0,5. Przy czym wartość ta do optymalnego działania powinna wynosić minimalnie więcej aniżeli w poprzednim przypadku, ze względu na dodatkowy feromon nakładany na najkrótszej trasie w danym cyklu algorytmu.

**Tabela 6.**

Działanie algorytmu w zależności od ilości iteracji głównej pętli algorytmu na ustawieniach: $\rho = 0,5, \alpha = 1, \beta = 3, m = n$, liczba iteracji 500.

| | 100 | 300 | 500 | 700 | 1000 |
|---|---|---|---|---|---|
| dantzig 42 | 698,78<br>2,14s | 692,96<br>5,13s | 685,62<br>8,05s | 685,32<br>11,00s | 691,10<br>15,45s |
| berlin 52 | 7605,43<br>3,39s | 7631,69<br>8,96s | 7548,28<br>14,55s | 7598,76<br>20,07s | 7566,53<br>28,58s |
| kroA 100 | 22029,23<br>22,65s | 21881,94<br>1m 05,58s | 21754,38<br>1m 50,87s | 21792,77<br>2m 32,36s | 21623,31<br>3m 35s |

Podobnie jak w podstawowej wersji algorytmu można zauważyć, że najlepsze trasy znajdowane są w okolicach 500 iteracji i są one nieznacznie lepsze od wyników po 100 iteracjach. Widać również że już po 100 iteracjach uzyskuje lepsze wyniki niż najlepsze rozwiązania w podstawowej wersji algorytmu.A to wynika z szybkiego zbiegania do optymalnych rozwiązań. Algorytm działa w praktycznie takim samym czasie co podstawowa wersja.

Na podstawie przeprowadzonych testów, mogę stwierdzić, że jest to znacząco lepsza wersja algorytmu, która przy odpowiednich ustawieniach uzyskała lepsze rozwiązania we wszystkich badanych przeze mnie grafach.

## 4.3 Algorytm mrówkowy z rankingiem (ASrank)

Trzecim algorytmem, który testowałem to algorytm mrówkowy z rankingiem, podobnie jak w poprzednich sprawdziłem najpierw optymalne ustawienia dla stałych $\alpha$ oraz $\beta$ . W przypadku tego algorytmu ustawiłem mniejszą wartość parowania ze względu na mniejszą ilość mrówek odkładających feromony. Liczba mrówek równa się liczbie miast, liczba iteracji 500, parowanie 0,1.

**Tabela 7.**

Sprawdzenie optymalnych wartości dla parametrów $\alpha$i $\beta$. Dla ustawień: $m = n, \rho = 0,1$, $w = 6$ (ilość dopuszczonych mrówek do nałożenia śladu feromonowego), liczba iteracji 500.

| β<br>α | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1<br>dantzig 42<br>pr 76 | 693,79<br>112447,67 | **690,20**<br>111336,21 | 696,39<br>111399,08 | 697,67<br>**110889,56** | 700,32<br>111504,52 |
| 2<br>dantzig 42<br>pr 76 | 720,37<br>115700,65 | 709,78<br>113873,59 | 708,79<br>113253,84 | 710,30<br>113836,22 | 707,18<br>114136,24 |

Z powyższej tabeli można wywnioskować podobnie jak w pierwszych dwóch rozwiązaniach, że najlepsze ustawienie dla $\alpha$ powinno wynosić 1, jednak wyniki dla $\alpha = 2$, nie różnią się znacząco od wyników dla $\alpha = 1$. Wynika to z tego, że w algorytmie tylko wybrana liczba mrówek nakłada ślad feromonowy. W przypadku $\beta$ w zależności od rozpatrywanego przypadku najlepsze rezultaty uzyskuje się w przedziale od 2 do 5, co potwierdzają wyniki, w których wychodzi, że optymalna wartość $\beta$ dla przykładu dantzig 42 to $\beta = 2$, natomiast dla przykładu pr 76 $\beta = 4$. Wyniki są nieznacznie gorsze dla przykładu dantzig 42 od Elitarnego algorytmu mrówkowego. Uzyskałem jednak znacznie lepsze wyniki od poprzednich dwóch algorytmów w przypadku bardziej skomplikowanego problemu pr 76.

**Tabela 8.**

Sprawdzenie działania algorytmu w zależności od ustawień parametru $\rho$, przy pozostałych ustawieniach: $\alpha = 1, \beta = 2, m = n, w = 6$, liczba iteracji 500.

| ρ | 0,05 | 0,1 | 0,2 | 0,3 | 0,4 |
|---|---|---|---|---|---|
| berlin 52 | 7544,42 | **7544,36** | 7590,92 | 7630,75 | 7654,84 |
| eli 101 | 682,89 | **651,82** | 652,15 | 657,23 | 663,97 |

Jak można zauważyć, najlepsze wyniki w obydwu przypadkach osiągnąłem dla wartości $\rho = 0,1$. Optymalna wartość $\rho$ zależy głównie od ilości mrówek dopuszczonych do nakładania nowego śladu feromonowego oraz od wielkości badanego grafu (im więcej

mrówek dopuszczonych do odkładania śladu oraz im więcej wierzchołków w grafie, tym większą wartość powinien wynosić parametr ρ).

**Tabela 9.** Działanie Algorytmu Mrówkowego z Rankingiem w zależności od ilości iteracji na ustawieniach: $\alpha = 1, \beta = 2, \rho = 0,1, m = n, w = 6$, liczba iteracji 500.

| | 100 | 300 | 500 | 700 | 1000 |
|---|---|---|---|---|---|
| dantzig 42 | 1003,48<br>1,96s | 689,14<br>5,09s | 684,95<br>8,01s | 687,60<br>10,89s | 685,79<br>15,42s |
| berlin 52 | 11726,34<br>3,31s | 7549,60<br>9,02s | 7555,81<br>14,58s | 7544,42<br>20,12s | 7544,36<br>28,23s |
| kroA 100 | 44690,41<br>21,64s | 22156,27<br>1m 04,12s | 21612,88<br>1m 46,79s | 21635,20<br>2m 29,15s | 21573,38<br>3m 30,31s |

Jak widać w powyższej tabeli wyniki dla 100 iteracji, są znacznie gorsze niż w poprzednich rozwiązaniach, natomiast pomiędzy integracją 100 a 300 algorytmu następuje gwałtowna poprawa znalezionych tras. W przypadku dwóch mniej skomplikowanych problemów dantzig 42 oraz berlin 52 najlepsze trasy znajdowane są w okolicach 500 iteracji natomiast w przypadku bardziej skomplikowanego problemu kroA 100 w okolicach 1000 iteracji.

Z testów mogę wywnioskować, że dla najlepszych ustawień parametrów algorytm będzie spisywał się nieznacznie lepiej od poprzednich wersji. Natomiast gdy ustawimy zbyt małą liczbę iteracji, będzie się spisywał gorzej od poprzednich wersji algorytmu. Wraz ze zwiększaniem liczby iteracji algorytmu uzyskałem lepsze wyniki od poprzednich dwóch wersji.

## 4.4 Algorytm Mrówkowy Max-Min (MMAS)

Kolejnym algorytmem, który testowałem to algorytm Mrówkowy Max-Min. Podobnie jak w poprzednich wersjach badałem optymalne ustawienia dla parametrów α i β a także jak algorytm spisuje się w zależności od ilości iteracji. W tym algorytmie można również modyfikować sposób odkładania feromonu przez mrówki poprzez ustawienie wybierania

odkładania śladu feromonowego na najlepszej trasie znalezionej w danej iteracji lub najlepszej trasy do tej pory znalezionej w trakcie działania algorytmu.

**Tabela 10.**

W pierwszej tabeli sprawdziłem optymalne ustawienia αoraz βna ustawieniach: $\rho = 0,02$, liczba iteracji 500, $m = n$.

| β<br>α | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1<br>dantzig 42<br>pr 76 | 701,16<br>117870,72 | 689,45<br>112191,45 | 683,11<br>111415,29 | **679,34**<br>**111349,65** | **679,34**<br>111493,27 |
| 2<br>dantzig 42<br>pr 76 | 711,66<br>117729,43 | 691,52<br>112432,35 | 689,09<br>112613,97 | 685,45<br>112759,34 | 684,77<br>112035,72 |

Jak można zauważyć z powyższej tabeli, ponownie najlepsze wyniki algorytm uzyskuje się dla wartości α równiej 1. Ponownie ustawienie parametru β zależy od badanego przykładu i powinno zawierać się w zakresie od 2 do 5. Dla przykładu dantzig 42 uzyskałem najlepsze wyniki w przypadku $\beta = 4$ i $\beta = 5$, natomiast dla przykładu pr 76 dla $\beta = 4$.

**Tabela 11.**

Działanie algorytmu w zależności od ustawień parametru ρ, na ustawieniach: $\alpha = 1$, $\beta = 4, m = n$, liczba iteracji 500.

| ρ | 0.01 | 0.02 | 0.05 | 0.1 | 0.2 |
|---|---|---|---|---|---|
| berlin 52 | 7577,82 | **7544,36** | **7544,36** | **7544,36** | 7561,59 |
| eli 101 | 704,09 | 664,64 | **653,64** | 655,20 | 655,03 |

Jak można wywnioskować z tabeli, wartość parowania w algorytmie tym powinna wynosić mniejszą wartość niż w poprzednich trzech wersjach algorytmu mrówkowego. W badanych przykładach otrzymałem najlepsze wyniki w zakresach od $\rho = 0,02$ do $\rho = 0,1$.

Mniejsza optymalna wartość ρ wynika z tego, że w wersji tej nowe feromony są nakładane na pojedynczą ścieżkę po jednym cyklu algorytmu w odróżnieniu od poprzednich wariantów, gdzie feromony odkładane były na kilku ścieżkach.

**Tabela 12.**

W poniższej tabeli prezentuje wyniki algorytmu w zależności od ilości iteracji od 100 do 1000, ustawienia parametrów to : α=1, $\beta = 4$,$\rho = 0,02, m = n$.

| | 100 | 300 | 500 | 700 | 1000 |
|---|---|---|---|---|---|
| dantzig 42 | 738,55<br>2,01 s | 693,72<br>4,97s | 679,34<br>8,14s | 679,34<br>10,83s | 679,34<br>18,55s |
| berlin 52 | 8389,70<br>4,38 s | 7604,25<br>9,00 s | 7544,36<br>14,40s | 7545,29<br>19,79s | 7544,36<br>28,03s |
| kroA 100 | 25591,49<br>21,517s | 22605,97<br>1m 03,58s | 21851,52<br>1m 45,21s | 21513,46<br>2m 27,12s | 21403,84<br>3m 30,95s |

Jak można zauważyć z przeprowadzonych testów, złożoność czasowa algorytmu jest bardzo podobna do poprzednich wersji. Algorytm w badanych przykładach spisuje się lepiej od poprzednich wersji, w szczególności od podstawowej wersji algorytmu. Podobnie jak wersja algorytmu mrówkowego z rankingiem. W przypadku dwóch prostszych problemów dantzig 42 oraz berlin 52 potrzebował około 500 iteracji do znalezienia najlepszych rozwiązań, a w przypadku trudniejszego problemu kroA 100 był w stanie znaleźć lepsze rozwiązania nawet w okolicach 1000 iteracji.

## 4.5 System Kolonii Mrówek (ACS)

Ostatnim algorytmem, który testowałem to System Kolonii Mrówek (Ant Colony System). W odróżnieniu od poprzednich algorytmów liczbę mrówek ustawiłem na stałą równą 10. Ilość iteracji ze względu na mniejszą złożoność czasową pojedynczej iteracji algorytmu spowodowaną szybszym wyborem kolejnej ścieżki przez mrówkę oraz mniejszą liczbę mrówek zwiększyłem do 5000 interakcji. Do algorytmu dodane zostały dwa nowe parametry $q_0$wykorzystywany we wzorze (14) oraz φwykorzystywany we wzorze (15).

**Tabela 13.**

Działanie algorytmu w zależności od ustawienia α i β. Ustawienia pozostałych parametrów: $\rho = 0,1$, φ=0,1, $m = 10$, $q_0 = 0,9$.

| β<br>α | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1<br>dantzig 42<br>pr 76 | 686,14<br>110663,37 | 685,37<br>**110066,85** | **680,87**<br>110467,66 | **680,87**<br>110758,26 | 681,02<br>111464,54 |
| 2<br>dantzig 42<br>pr 76 | 687,82<br>111336,74 | 686,01<br>110627,51 | 681,92<br>110984,74 | 684,09<br>110481,95 | 683,38<br>111288,96 |

Podobnie jak w poprzednich przypadkach najlepsze rezultaty uzyskałem dla $\alpha = 1$ , zbliżone wyniki uzyskano dla βzawierającej się w przedziale od 2 do 5. Przy czym najlepszy rezultat uzyskał dla $\beta = 2$. Dla przykładu pr 76 uzyskałem najlepsze rezultaty, natomiast dla przykładu dantzig 42 gorsze tylko od Algorytmu Mrówkowego Max-Min. Ponownie w przypadku $\alpha = 2$ algorytm spisuje się nieznacznie gorzej niż w przypadku $\alpha = 1$, co wynika z mniejszej eksploracji potencjalnie lepszych tras. Natomiast wartość βw zależności od algorytmu skutkuje najlepszymi rezultatami w przypadku βzawierającego się w przedziale od 2 do 5. Można zauważyć również że ustawienia α wpływa w mniejszym stopniu na działanie algorytmu niż w poprzednich algorytmach.

**Tabela 14.**

Działanie Systemu Kolonii Mrówek w zależności od ustawień parametrów parowania lokalnego i globalnego przy ustawieniach: $m = 10$, $q_0 = 0,9$, α=1 , β= 3, liczba iteracji 5000.

| ρ, φ | 0,05 | 0,1 | 0,2 | 0,3 | 0,4 |
|---|---|---|---|---|---|
| berlin 52 | **7590,28** | 7640,25 | 7653,64 | 7613,07 | 7601,61 |
| eli 101 | 653,87 | **649,69** | 655,11 | 650,42 | 649,81 |

Jak można zauważyć wartości parowania w Systemie Kolonii Mrówek, przy takich ustawieniach spisuje się bardzo podobnie w zakresie od 0,05 do 0,4. W tym algorytmie połączone zostało parowanie wraz z nakładaniem nowych śladów feromonowych według wzorów (15) i (17). W badanych problemach komiwojażera najlepsze wyniki uzyskano dla $\rho, \varphi = 0,1$w przypadku problemu eli 101 oraz 0,05 w przypadku problemu berlin 52. Parametrami $\rho$ i $\varphi$ można regulować wpływ najkrótszej trasy znalezionej podczas danej iteracji poprzez stałą $\rho$ oraz wpływ wyborów pojedynczej mrówki modyfikując parametr $\varphi$. Mniejsze parametry $\rho$ i $\varphi$ skutkują początkowo szybkim znajdowaniem lepszych rozwiązań, wiąże się to jednak z kosztem mniejszej eksploatacji w późniejszych fazach algorytmu.

**Tabela 15.**

Działanie Systemu Kolonii Mrówek w zależności od ilości iteracji, na ustawieniach: $m = 10$, $q_0 = 0,9 \alpha = 1, \beta = 3, \rho = 0,1, \varphi = 0,1$.

| | 1000 | 5000 | 10000 | 15000 | 20000 |
|---|---|---|---|---|---|
| dantzig 42 | 686,02<br>01,94s | 680,10<br>07,64s | 680,46<br>14,70s | 680,10<br>14,47s | 679,34<br>28,94s |
| berlin 52 | 7641,86<br>2,73s | 7686,45<br>11,71s | 7642,14<br>22,89s | 7670,53<br>34,30s | 7669,22<br>45,07s |
| kroA 100 | 21647,88<br>10,00s | 21354,60<br>47,43s | 21326,28<br>1m 34,92s | 21357,48<br>2m 21,37s | 21371,03<br>3m 08,02s |

Jak można zauważyć z powyższej tabeli poprzez zmniejszenie ilości mrówek oraz sposób wyboru kolejnej ścieżki przez daną mrówke, zmniejszyło złożoność pojedynczej iteracji o kilkanaście razy w stosunku do poprzednich. Wyniki w przypadku dwóch prostszych problemów są nieznacznie gorsze od poprzedniej wersji algorytmu MIN-MAX. Jednakże w przypadku problemu dantzig 42 podobnie jak w przypadku poprzedniego algorytmu znajdował najlepsze rozwiązania, jednak potrzebne do tego była większa ilość czasu. W sytuacji bardziej skomplikowanego problemu kroA 100 algorytm ten spisuje się zdecydowanie najlepiej ze wszystkich omawianych.

System Koloni Mrówek wprowadza dodatkowy parametr $q_0$, który określa prawdopodobieństwo wyboru jednej z dwóch strategii wyboru mrówki, jedna to wybór

krawędzi o największym prawdopodobieństwie a druga to pseudolosowa metoda stosowana w poprzednich algorytmach.

**Tabela 16.**

Działanie algorytmu w zależności od ustawień parametru $q_0$, z uwzględnieniem czasu działania. Na ustawieniach: parowanie lokalne oraz globalne na 0,1, $\alpha$=1, $\beta$= 3, m = 10.

| $q_0$ | 0,5 | 0,65 | 0,8 | 0,9 | 0,95 |
|---|---|---|---|---|---|
| pr76 | 111143,26<br>39,65s | 110544,81<br>29,40s | 110446,64<br>29,17s | 110586,41<br>25,63s | **109600,40**<br>23,93s |
| kroa 100 | 21573,46<br>1m 13,16s | 21373,04<br>1m 03,42s | 21391,82<br>53,66s | **21354,60**<br>47,43s | 21490,31<br>44,01s |

Z powyższej tabeli wynika, że wraz ze wzrostem wartości $q_0$ algorytm wykonuje się szybciej. Wynika to z tego, że zwiększając wartość $q_0$, zwiększa się prawdopodobieństwo wyboru mniej kosztownej metody wyboru kolejnej krawędzi w grafie. Metoda ta wybiera krawędź z największym prawdopodobieństwem obliczonym według wzoru (14). Drugi sposób wyboru kolejnej krawędzi opiera się o wzór (2), który jest bardziej kosztowny, ponieważ wymaga obliczenia sumy wszystkich prawdopodobieństw, ponadto w tym sposobie wykorzystywany jest algorytm Roulette Wheel. Z testów przeprowadzonych można wywnioskować, że najlepiej ustawić parametr $q_0$ na wartość około 0,9, co pozwala na osiągnięcie lepszych wyników w krótszym czasie od innych ustawień parametru.

# Rozdział 5 - podsumowanie

W wyniku przeprowadzonych testów wynika, że najskuteczniejsze wersje omawianych przeze mnie algorytmów to Algorytm Mrówkowy MAX-MIN (MMAS) oraz System Kolonii Mrówek (ACS).

**Tabela 17.** Najlepsze wyniki algorytmów w przykładowych problemach komiwojażera.

| | dantzig 42 | berlin 52 | pr76 | kroA 100 | eil 101 |
|---|---|---|---|---|---|
| AS | 710,80 | 7573,54 | 115761,08 | 22598,58 | 695,06 |
| EAS | 685,32 | 7548,28 | 111839,26 | 21623,31 | 650,48 |
| ASrank | 684,95 | 7544,36<br>28,23s | 110889,56 | 21573,38 | 651,82 |
| MMAS | **679,34<br>8,14s** | **7544,36<br>14,40s** | 111349,65 | 21403,84 | 653,64 |
| ACO | 679,34<br>28,94s | 7642,14 | **110066,85** | **21326,28** | **649,69** |

Algorytm MMAS uzyskał najepsze rezultaty dla problemu dantzig 42 i berlin 52 (algorytm ASrank i ACO również znajdowały optymalne rozwiązania, jednak MMAS potrzebował znacznie mniej czasu), natomiast ACS w przykładach: pr 76, kroA 100 i eli 101. Przewaga tych algorytmów nad resztą wynika z limitów w Algorytmie MAX-MIN oraz ze specyficzności wzorów (15) i (17) w algorytmie ACO. Powodują one to, że w późniejszych iteracjach algorytmu wciąż istnieje możliwość znalezienia krótszych Cykli Hamiltona, co daje im przewagę nad poprzednimi wersjami, w których nie są odporne na sytuację stagnacji. Z tego względu MMAS oraz ACS są wciąż wykorzystywanymi algorytmami w różnych problemach, aplikacjach i modyfikacjach nowych algorytmów mrówkowych. Przykładem nowo utworzonych algorytmów mrówkowych jest np. algorytm HUMANT [12], który opiera się o kilka algorytmów np. Algorytm Mrówkowy MAX-MIN.

Algorytmy mrówkowe są w stanie rozwiązać również inne problemy takie jak np. Problem Marszrutyzacji (Routing Problem), problem przydziału (Assignment Problem), Problem z Harmonogramem (Subset Problem), Problem z sumą podzbiorów (Subset sum

problem), Eksploracja Danych (Data Mining). Algorytmy mrówkowe mają szerokie zastosowanie. Dzięki temu wiedza zdobyta poprzez implementację oraz analizę działania poszczególnych algorytmów może okazać się użyteczna dla mnie w potencjalnych projektach obejmujących takie zagadnienia, jak np. nawigacja, przydzielanie zasobów, wydobywanie oraz eksploracja danych, uczenie maszynowe.

Algorytmy mrówkowe są bardzo ciekawym zagadnieniem algorytmicznym wykorzystującym obserwacje świata zwierząt. Dzięki tej wiedzy możemy uświadomić sobie piękno oraz skomplikowanie otaczającej nasz rzeczywistości.

# Bibliografia

[1]: M. Dorgio, T. Stutzle, “Ant Colony Optimization." The MIT Press, 2004. ISBN-13: 978-0262042192

[2]: Encyklopedia Wikipedia EN, Fitness proportionate selection (wersja z: 2020, wrzesień 21). https://en.wikipedia.org/wiki/Fitness_proportionate_selection

[3]: M. Dorigo, Optimization, Learning and Natural Algorithms, PhD thesis, Politecnico di Milano, Italy, 1992.

[4]: Encyklopedia Wikipedia PL, sortowanie szybkie (wersja z: 2021, luty 16). https://pl.wikipedia.org/wiki/Sortowanie_szybkie

[5]: Turabieh, Hamza & Mafarja, Majdi & Li, Xiaodong. (2018). Iterated Feature Selection Algorithms with Layered Recurrent Neural Network for Software Fault Prediction. Expert Systems with Applications. 122. 10.1016/j.eswa.2018.12.033.

[6]: Jun J, Kang J, Jeong D, Lee H. An efficient approach for optimizing full field development plan using Monte-Carlo simulation coupled with Genetic Algorithm and new variable setting method for well placement applied to gas condensate field in Vietnam. Energy Exploration & Exploitation. 2017;35(1):75-102. doi:10.1177/0144598716680307.

[7]: L.M. Gambardella and M. Dorigo, "Solving Symmetric and Asymmetric TSPs by Ant Colonies", Proceedings of the IEEE Conference on Evolutionary Computation, ICEC96, Nagoya, Japan, May 20–22, pp. 622-627, 1996

[8]: Stützle Thomas, and Holger H. Hoos. "Improving the Ant System: A detailed report on the MAX–MIN Ant System." *FG Intellektik, FB Informatik, TU Darmstadt, Germany, Tech. Rep. AIDA–96–12* (1996).

[9]:Bullnheimer, B., Hartl, R. & Strauss, C. An improved Ant System algorithm for theVehicle Routing Problem. Annals of Operations Research 89, 319–328 (1999)

[10]: Specyfikacja języka C#. 2017 (dostęp: 28.03.2021) https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction

[11]: M. Dorigo, V. Maniezzo, A. Colorni, et al. Positive feedback as a search strategy, 1991.

[12]: Marko Mladineo & Ivica Veza & Nikola Gjeldum, 2017. "Solving partner selection problem in cyber-physical production networks using the HUMANT algorithm," International Journal of Production Research, Taylor & Francis Journals, vol. 55(9), pages 2506-2521, May.

[13]: Deneubourg, J.L., Aron, S., Goss, S. *et al.* The self-organizing exploratory pattern of the argentine ant. *J Insect Behav* 3, 159–168 (1990). https://doi.org/10.1007/BF01417909

[14]: K. Menger, "Das botenproblem", in *Ergebnisse eines Mathematischen Kolloquiums* **2** (K. Menger, editor), Teubner, Leipzig, pages 11-12