Sprawozdanie

Implementacja algorytmu ACO dla problemu nr 4
Optymalizacja kombinatoryczna

1. Wprowadzenie

Treść problemu:

Dany jest spójny graf $G=(V, E)$ z wagami w_i przypisanymi do każdej krawędzi $v_i \in V$. Należy znaleźć ścieżkę łączącą wszystkie wierzchołki (każdy odwiedzony min. raz) taką, aby minimalizować jej koszt S. Koszt S obliczany jest w taki sposób, że stanowi ona sumę wszystkich wag krawędzi licząc od tej wychodzącej z pierwszego wierzchołka ścieżki do wagi krawędzi wchodzącej do ostatniego wierzchołka ścieżki, w taki jednak sposób, że jeśli właśnie dodawana do S waga w_i jest większa niż następna waga krawędzi na tej ścieżce: (tj. niż waga w_i+1), wtedy do sumy dodajemy nie samo w_i, ale jej x-krotność (czyli x * w_i). Przyjąć początkowo: |V| minimum 100, deg(v) = [1, 6], w*i* = [1, 100], x = minimum 5.

Ogólny opis algorytmu:

Algorytm użyty do rozwiązania tego problemu został zaimplementowany na podstawie metod probabilistycznych używanych w algorytmach ACO. Została użyta tutaj macierz feromonowa, która uaktualniana jest na podstawie ścieżek mrówek, które znalazły najlepsze rozwiązania. Feromony w macierzy po każdej iteracji algorytmu poddawane są operacjom parowania oraz wygładzania. Została wprowadzona również maksymalna wartość feromonu, której nie można przekroczyć w macierzy. Efektem działania tego algorytmu jest podanie rozwiązania, które odpowiada najmniejszej wartości funkcji celu S spośród wszystkich znalezionych rozwiązań. Rozwiązanie podawane jest osobno dla każdej wprowadzonej instancji. Jeżeli instancji jest więcej niż 1, to liczona jest także średnia wartość rozwiązań znalezionych w podanych instancjach.

Algorytm został zaimplementowany i skompilowany przy wykorzystaniu systemu operacyjnego Windows oraz środowiska MinGW. Do napisania kodu oraz kompilacji programu został użyty program CLion 2020.

2. Opis implementacji algorytmu

Indeksy w wektorach przechowujących informacje o wierzchołkach/krawędziach odpowiadają numerom tym wierzchołków pomniejszonym o jeden. Przykładowo, aby dostać się do listy sąsiedztwa (która posłużyła w tym algorytmie jako reprezentacja grafu) wierzchołka o numerze *v*, trzeba odwołać się do indeksu o numerze *v*-1 w odpowiednim wektorze.

2.1. Generator instancji

Generator instancji jest pierwszym z dwóch modułów, które składają się na opisywany w tym sprawozdaniu algorytm. Rozpoczyna on swoje działanie od losowania stopnia dla każdego wierzchołka w grafie. Gdy w grafie dopuszczalne są wierzchołki o stopniu 1, wierzchołki są dzielone na dwie podgrupy: jedna zawierająca tylko wierzchołki o stopniu 1, druga zawierająca pozostałe wierzchołki. W tej drugiej podgrupie kolejne wierzchołki zostają połączone ze sobą pojedynczą krawędzią. Ostatni wierzchołek z tej podgrupy łączony jest z pierwszym wierzchołkiem z tej samej podgrupy. W ten sposób powstaje prosty graf, w którym istnieje na razie tylko jedna ścieżka i jeden cykl. Aby zapewnić losową kolejność wierzchołków, które są łączone krawędzią, przed utworzeniem jakiegokolwiek połączenia wywoływana jest funkcja *random_shuffe*, która zmienia kolejność wierzchołków dodanych do tej podgrupy.

W kolejnym kroku do poprzednio utworzonego prostego grafu dodawane są krawędzie łączące wierzchołki grafu z wierzchołkami z pierwszej podgrupy, czyli tymi o stopniu 1. W tej podgrupie kolejność wierzchołków również została zmieniona względem oryginalnej kolejności za pomocą funkcji *random_shuffle*.

W ostatnim kroku dodawane są krawędzie tak, aby wylosowany wcześniej stopień wierzchołka zgadzał się z krawędziami wychodzącymi z tego wierzchołka.

Przy tworzeniu każdej krawędzi losowana jest od razu jej waga. Utworzone krawędzie zapisywane są do odpowiedniego wektora w taki sposób, że zapisywane są numery wierzchołków tworzących krawędź oraz waga tej krawędzi. W przypadku, gdy zostanie wierzchołek, od którego nie wychodzi jeszcze odpowiednia liczba krawędzi, a nie ma już innych wierzchołków, z którymi można utworzyć krawędź, stopień takiego wierzchołka zostaje zmieniany na liczbę krawędzi, która aktualnie z niego wychodzi. W grafie nie zostało dopuszczone tworzenie krawędzi wielokrotnych.

Tworzenie grafu w ten sposób ma zapewnić spójność w grafie.

Przykładowa wizualizacja:

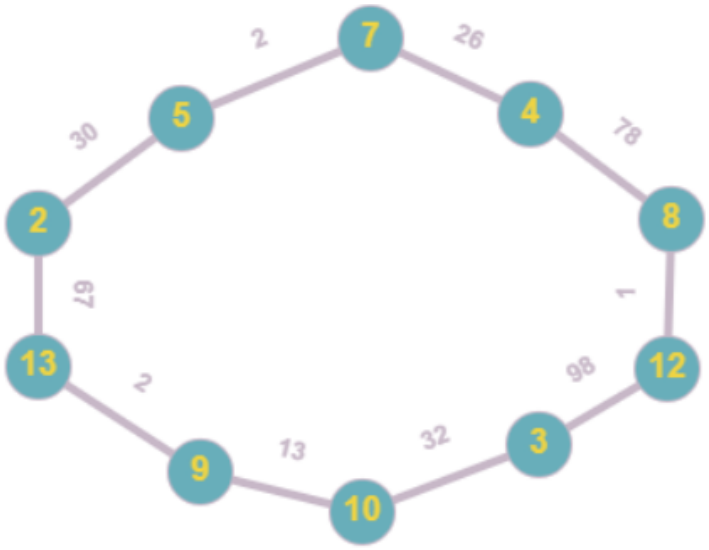
Wylosowane stopnie dla każdego wierzchołka. Pierwszy wiersz oznacza indeks wektora, a zarazem numer wierzchołka (zgodnie z zasadą zapisaną na początku tego segmentu, czyli wierzchołkowi o numerze *v* odpowiada indeks o numerze *v*-1).

0	1	2	3	4	5	6	7	8	9	10	11	12
1	3	3	4	5	1	2	3	4	5	1	3	6

Podgrupa zawierająca wierzchołki o stopniu większym niż 1:

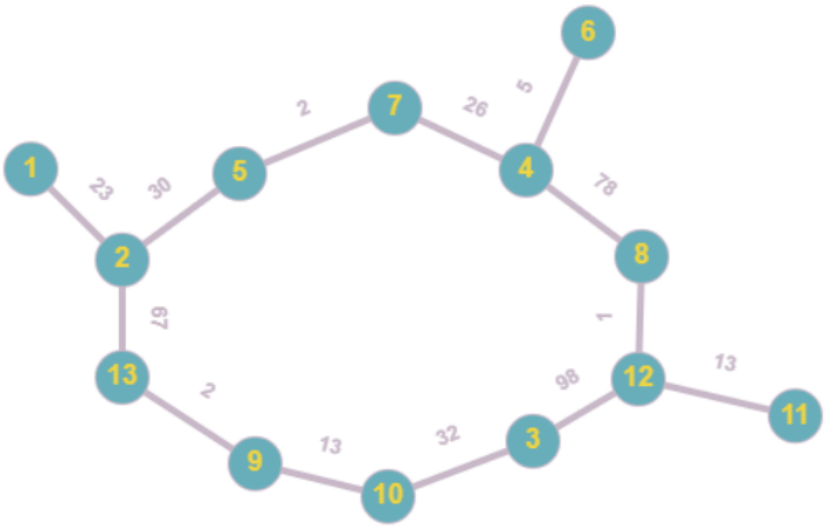
2	5	7	4	8	12	3	10	9	13

Powstały graf:

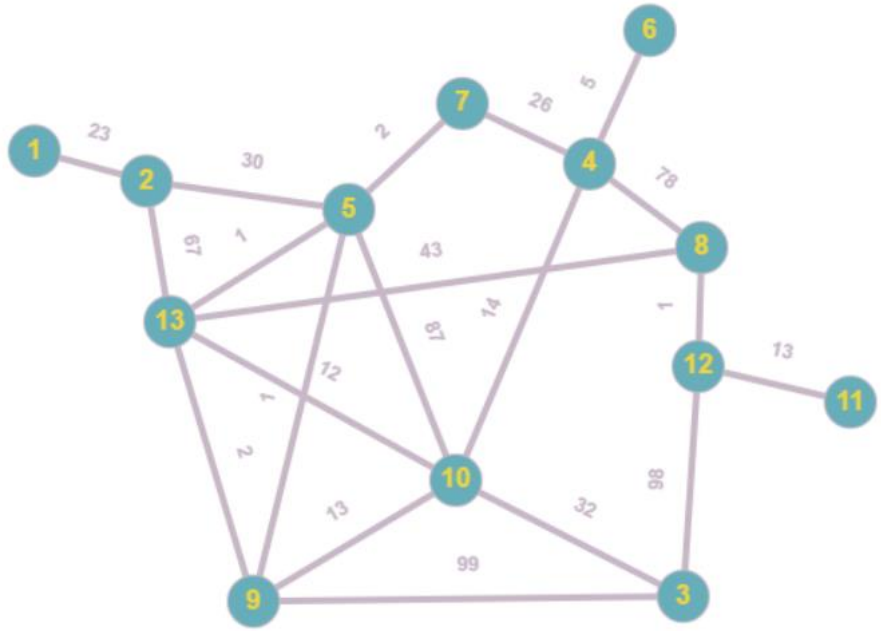


Podgrupa zawierająca wierzchołki o stopniu równym 1:

6	1	11



Łączenie pozostałych wierzchołków:



W tym przypadku nie udało się połączyć krawędzią wierzchołka o numerze 13 z innym wierzchołkiem tak, aby wychodzące z niego krawędzie zgadzały się z jego stopniem wylosowanym na początku. Dlatego taki stopień jest aktualizowany.

0	1	2	3	4	5	6	7	8	9	10	11	12
1	3	3	4	5	1	2	3	4	5	1	3	5

Reprezentacją instancji jest zatem wektor zapisujący wszystkie powstałe krawędzie wraz z ich wagami. Każda taka instancja zapisywana jest do osobnego pliku.

2.2. Znajdowanie rozwiązania za pomocą algorytmu ACO

W drugim module następuje wczytywanie kolejnych instancji w pętli. Dzieje się to w taki sposób, że najpierw wczytana jest jedna instancja i dla niej uruchamiany jest algorytm, a po upłynięciu założonej ilości czasu algorytm jest zatrzymywany i podawane jest najlepsze znalezione rozwiązanie. Takie rozwiązanie jest zapisywane w odpowiednim wektorze, po czym następuje kolejna iteracja pętli i wczytana zostaje następna instancja.

2.2.1. Wczytywanie instancji z pliku, tworzenie listy sąsiedztwa oraz macierzy wagowej

Z pliku wczytywane są informacje o krawędziach oraz ich wagach. Na tej podstawie tworzona jest lista sąsiedztwa, która jest reprezentacją grafu, oraz macierz wagowa, która ma za zadanie przechowywać wagi krawędzi pomiędzy poszczególnymi wierzchołkami i zostanie wykorzystana przy liczeniu kosztu S utworzonej ścieżki.

2.2.2. Generator rozwiązań losowych – pierwsza iteracja algorytmu ACO

Pierwszą iteracją algorytmu jest generator rozwiązań losowych. W funkcji odpowiedzialnej za znalezienie stu takich rozwiązań zainicjowana jest pętla, która będzie wykonywać się sto razy. Iteracja każdej pętli zaczyna się od wylosowania wierzchołka *current*, od którego mrówka będzie zaczynać ścieżkę. Następnie w odpowiedniej pętli, w której przy każdej jej iteracji sprawdzane jest, czy wszystkie wierzchołki zostały już odwiedzone, losowany jest kolejny wierzchołek, do którego mrówka ma przejść. Wierzchołek ten jest oczywiście losowany spośród tych wierzchołków, z którymi wierzchołek *current* jest połączony krawędzią. Tak wylosowany wierzchołek staje się nowym wierzchołkiem *current*, a pętla ma swoją kolejną iterację. Wykonuje się ona, dopóki wszystkie wierzchołki nie zostaną przynajmniej raz odwiedzone. Z racji tego, że jest to generator rozwiązań losowych, mrówki nie są w żaden sposób kierowane, dlatego w niektórych miejscach mogą „kręcić się" w kółko, odwiedzając te same wierzchołki po kilka razy, podczas gdy sąsiednie wierzchołki nie zostały odwiedzone ani razu. W momencie, gdy mrówka odwiedzi wszystkie wierzchołki, pętla jest przerywana, a przebyta przez mrówkę ścieżka jest zapisywana do wektora *Paths*.

Po znalezieniu stu rozwiązań losowych następuje obliczenie kosztu S dla każdej ścieżki. Dzieje się to w sposób odwrotny do zapisanego w treści problemu, przy czym efekt pozostaje ten sam. W przypadku każdej ścieżki jej koszt całkowity liczony jest od ostatniego wierzchołka w ścieżce do pierwszego, czyli w odwrotnej kolejności – wierzchołek ostatni staje się wierzchołkiem pierwszym, a wierzchołek pierwszy ostatnim. W ten sposób, jeżeli poprzednia krawędź w tak zmienionej ścieżce miała mniejszą wagę niż obecna krawędź, to obecna krawędź jest przemnażana przez wartość kary *x*. Tak obliczony koszt S ścieżki jest zapisywany do odpowiedniego wektora. Następnie spośród wszystkich tych rozwiązań wybierane jest 30 najlepszych, które zostaną wykorzystane do zbudowania macierzy feromonowej. Rozwiązania te zostają ułożone w odpowiednim wektorze od najlepszego do najgorszego.

Ostatnim krokiem jest naniesienie feromonów na krawędzie, które znajdują się w najlepszych ścieżkach, przy czym najlepsza mrówka zostawia za sobą wartość feromonu równą 1, a pozostałe mrówki wartość odpowiednio pomniejszoną według następującego wzoru:

$$pheromoneValue = 1 - (i * bestAnts * 0.001),$$

gdzie *i* to numer indeksu rozwiązania w wektorze (najlepsza mrówka ma swoje rozwiązanie pod indeksem 0, druga najlepsza pod indeksem 1, itd.), a *bestAnts* to liczba najlepszych rozwiązań (w pierwszej iteracji jest to zawsze 30). Tak utworzona wartość feromonu dla odpowiedniej krawędzi zostaje dodana do wartości, która aktualnie znajduje się w macierzy feromonowej.

Po utworzeniu i wypełnieniu macierzy feromonowej mają miejsce kolejne iteracje algorytmu ACO.

2.2.3. Polepszanie rozwiązań – kolejne iteracje algorytmu ACO

Na początku każdej kolejnej iteracji algorytmu ma miejsce parowanie oraz wygładzanie feromonów. Funkcje odpowiedzialne za te operacje zostaną opisane w dwóch następnych podpunktach.

Po operacjach parowania oraz wygładzania feromonów zostaje wywołana funkcja odpowiedzialna za szukanie lepszych ścieżek, tym razem z możliwością skorzystania z informacji zawartych w macierzy feromonowej. Możliwość ta związana jest z prawdopodobieństwem *probability*, które jest pomnożoną obecną iteracją algorytmu przez liczbę 10.

Następnie tak jak wcześniej ma miejsce losowanie pierwszego wierzchołka, którym rozpoczynać ma się ścieżka. Inicjowana jest pętla, która będzie się wykonywać, dopóki każdy wierzchołek w grafie nie zostanie przynajmniej raz odwiedzony. Na początku każdej jej iteracji losowana jest wartość w zakresie od 0 do 999. W przypadku, gdy wylosowana wartość jest większa od *probability*, to następny wierzchołek jest wybierany w taki sposób, że najpierw sprawdzane jest, czy z sąsiadów obecnego wierzchołka jakiś nie został jeszcze odwiedzony. Jeżeli tak, to następnym wierzchołkiem jest nieodwiedzony jeszcze sąsiad wierzchołka obecnego. Jeżeli jednak wszystkie wierzchołki, z którymi połączony jest wierzchołek *current*, zostały już odwiedzone, to kolejny wierzchołek w ścieżce losowy analogicznie do tego, w jaki był wybierany w generatorze rozwiązań losowych. W tej funkcji wprowadzono jednak kolejną zasadę: następny wierzchołek nie może być wierzchołkiem odwiedzonym w iteracji bezpośrednio poprzedzającą iterację obecną (oczywiście zasada ta nie obowiązuje w przypadku, kiedy jedynym sąsiadem wierzchołka *current* jest właśnie ten poprzedni wierzchołek, tj., kiedy wierzchołek *current* jest stopnia 1). Ma to na celu uniknięcia sytuacji, w której mrówka „kręciłaby się" w kółko po tych samych wierzchołkach. Jeżeli tak wylosowany wierzchołek jest wierzchołkiem, który poprzednio odwiedziła mrówka, to wybierany jest inny wierzchołek, sąsiadujący przy nim na liście sąsiedztwa.

W przypadku, kiedy wylosowana wartość jest mniejsza od *probability*, to następny wierzchołek zostanie wybrany na podstawie macierzy feromonowej. Liczone są zatem przedziały, które będą odpowiadały prawdopodobieństwu wybrania danego wierzchołka. Jest to robione w sposób następujący: najpierw inicjowana jest zmienna *sum*, która przyjmuje wartość 0. Następnie dla każdej krawędzi, którą tworzy wierzchołek *current* z wierzchołkami ze swojej listy sąsiedztwa, z macierzy feromonowej odczytywana jest wartość feromonu dla tej krawędzi, a potem mnożona przez 100. Dolna wartość przedziału odpowiadać będzie wartości *sum* przed dodaniu do niej pomnożonej wartości feromonu, a górna – wartości *sum* po dodaniu do niej wartości feromonu. W ten sposób odjęcie od górnej wartości przedziału dolnej wartości tego przedziału da wynik równy wcześniej wymnożonej wartości feromonu. Po określeniu przedziałów dla każdej krawędzi zmienna *sum* będzie miała wartość równą sumie wartości wszystkich przemnożonych feromonów. Aby określić, który wierzchołek będzie następnym w ścieżce, losowana jest liczba z przedziału [0, *sum*). Następnie sprawdzane jest, w przedziale dla którego wierzchołka się ona mieści. Jest to następny wierzchołek na ścieżce.

Przykładowa wizualizacja

Numer wierzchołka: **29**

Lista sąsiedztwa wierzchołka: **15, 1, 87, 39, 43**

Wartości feromonów po przemnożeniu przez 100:

- Krawędź: **29, 15**: 13852
- Krawędź: **29, 1**: 1841
- Krawędź: **29, 87**: 23198
- Krawędź: **29, 39**: 932
- Krawędź: **29, 43**: 19732

Lista sąsiedztwa wierzchołka numer 29:

	28
0	15
1	1
2	87
3	39
4	43

Przedziały odpowiadające kolejnym wierzchołkom z listy sąsiedztwa:

0	1	2	3	4
0	13852	15423	38621	39553
13852	15423	38621	39553	59285

Przedziały dla wierzchołków są lewostronnie domknięte oraz prawostronnie otwarte, np. dla wierzchołka 15: [0, 13852).

Wylosowana została liczba **32746**. Mieści się ona w przedziale dla wierzchołka 87, dlatego to on będzie następnym wierzchołkiem w ścieżce.

Dalsze kroki (obliczanie kosztu S dla każdej ścieżki, wybieranie najlepszych rozwiązań oraz uaktualnianie macierzy feromonowej) są przeprowadzane analogicznie do tego, jak to zostało opisane wcześniej.

2.2.4. Parowanie oraz wygładzanie feromonów

Dla parowania feromonów została zaimplementowana funkcja, która w macierzy feromonowej każdą wartość *y* większą od zera nadpisuje wartością *y* zmniejszoną o pewien jej procent, np. w przypadku parowania ustawionego na 15% nowa wartość będzie równa 0,85 * *y*.

Do wygładzania feromonów została zaimplementowana kolejna funkcja. W każdym wierszu macierzy sprawdzane są wartości w nim występujące. Jeżeli w takim wierszu znajdzie się wartość, która jest większa od ustalonej wartości maksymalnej dla feromonów, uruchamiany zostaje proces wygładzania dla tego wiersza. Każda wartość w tym wierszu zostaje zatem zastąpiona nową wartością wyliczoną zgodnie ze wzorem:

$$\tau_{ij}^{*}(t) = \tau_{ij}\,(t) + \delta\left(\tau_{max}(t) - \tau_{ij}(t)\right),$$

gdzie $\tau_{ij}^{*}(t)$ to nowa wartość feromonu dla krawędzi *i*,*j* w iteracji *t* algorytmu, $\tau_{ij}\ (t)$ to wartość tego feromonu przed zmianą, δ jest stałą liczbową, którą wartość trzeba ustalić, natomiast $\tau_{max}(t)$ to maksymalna wartość feromonu w tej iteracji, która w tej implementacji algorytmu zawsze będzie równa jednej liczbie. Wzór ten został wspominany w kilku artykułach poświęconych algorytmom ACO. W niektórych artykułach zmienna δ określona została w przedziale [0, 1], natomiast w innych stwierdzono, że zmienna ta może przyjmować wartości powyżej 1. W implementacji tego algorytmu jednak zmienna δ przyjmuje wartość z przedziału [0, 1] i zawsze jest to stała wartość liczbowa, tj. nie zmienia się w trakcie działania algorytmu, podobnie jak wartość τ_{max}.

3. Testy

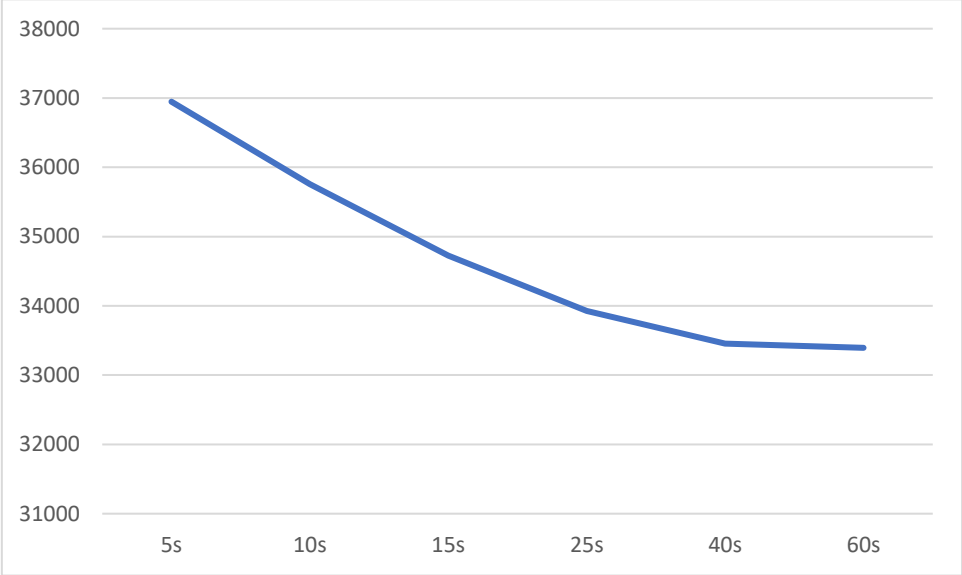
Wszystkie testy przeprowadzone zostały na grafach o 100 wierzchołkach.

3.1. Testy parametrów algorytmu

Testy te przeprowadzane były na jednym zestawie 100 instancji. Wynikiem dla każdego testowanego parametru jest średnia wartość funkcji celu.

3.1.1. Czas

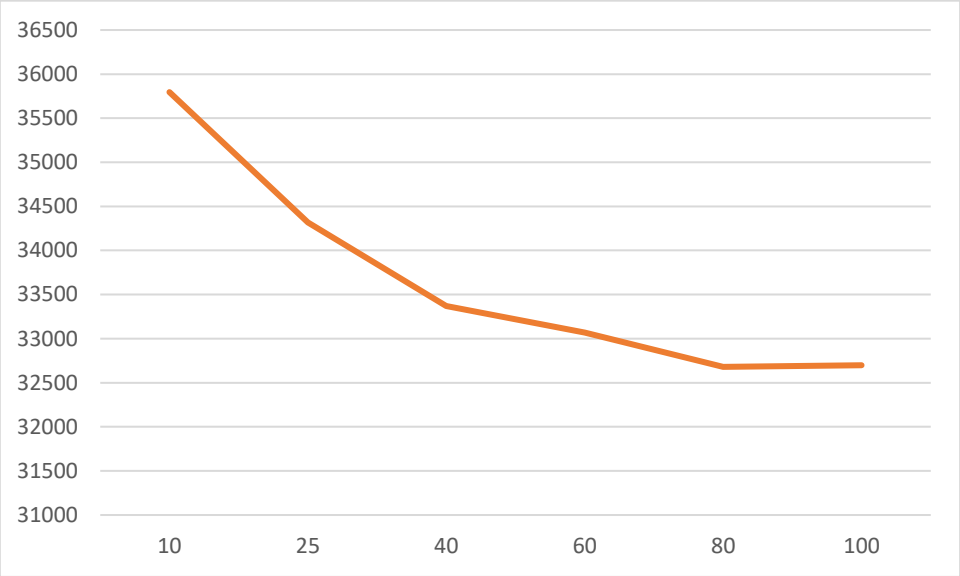
Liczba mrówek od drugiej iteracji: 20
Parowanie: 15%
δ= 0,2
Wartość *x*: 5
deg(v) = [1, 6]
w*i* = [1, 100]



Na wykresie widać, że w przypadku małej ilości czasu dla każdej instancji średnia wyników jest dużo wyższa niż w przypadku, kiedy na znalezienie rozwiązania w każdej instancji poświęci się więcej czasu. W przypadku 40s oraz 60s wyniki były bardzo podobne, co świadczy o tym, że krzywa zaczyna się wypłaszczać.

3.1.2. Liczba mrówek od drugiej iteracji

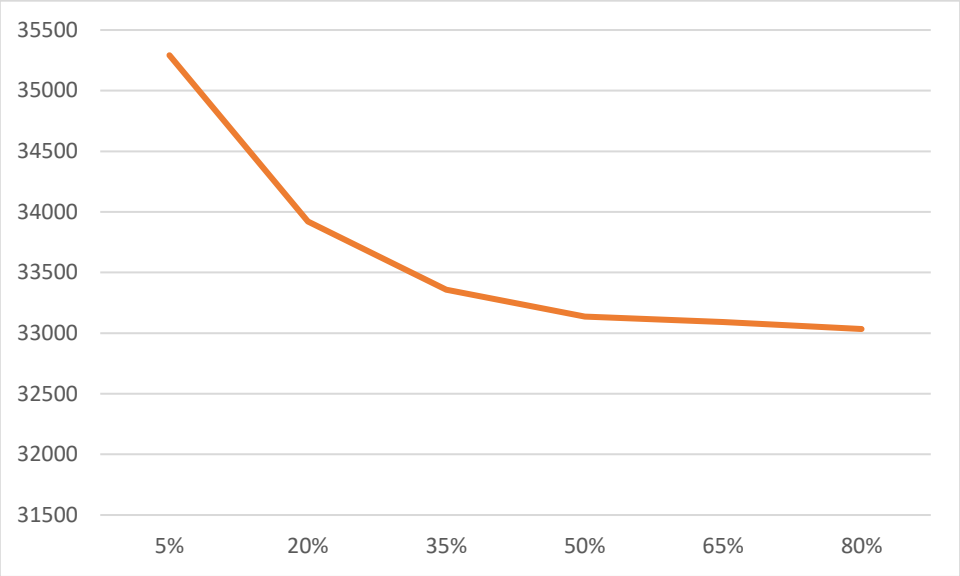
Czas na każdą instancję: 25s
Parowanie: 15%
δ= 0,2
Wartość x: 5
deg(v) = [1, 6]
wi = [1, 100]



Wykres można zinterpretować w sposób analogiczny, jak w przypadku, gdy testowanym parametrem był czas – gdy od drugiej iteracji zostało puszczonych mało mrówek, średnia ich rozwiązań była wyższa niż w przypadku, kiedy od drugiej instancji zostało puszczonych więcej mrówek. W tym przypadku krzywa również zaczęła się wypłaszczać od pewnej wartości mrówek – średnie wyniki w przypadku 80 oraz 100 mrówek są bardzo do siebie zbliżone.

3.1.3. Parowanie feromonów

Czas na każdą instancję: 25s
Liczba mrówek od drugiej iteracji: 20
δ= 0,2
Wartość *x*: 5
deg(v) = [1, 6]
w*i* = [1, 100]



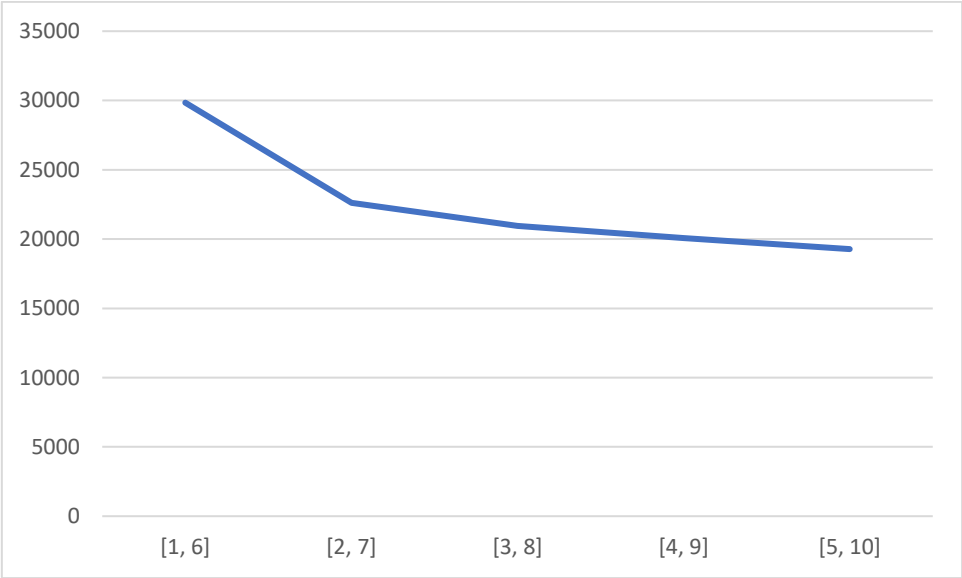
Wykres wygląda bardzo podobnie jak w dwóch poprzednich przypadkach – wraz ze zwiększaniem parametru parowania feromonów malała średnia wartość rozwiązania. Jest to dość ciekawa zależność, która może wskazywać na małe znaczenie macierzy feromonowej w procesie znajdowania ścieżki o coraz mniejszym koszcie.

3.2. Testy parametrów instancji

W tych testach za każdym razem generowanych było 10 nowych instancji w taki sposób, aby pasowały one do zmienianych parametrów.

3.2.1. Stopień wierzchołka

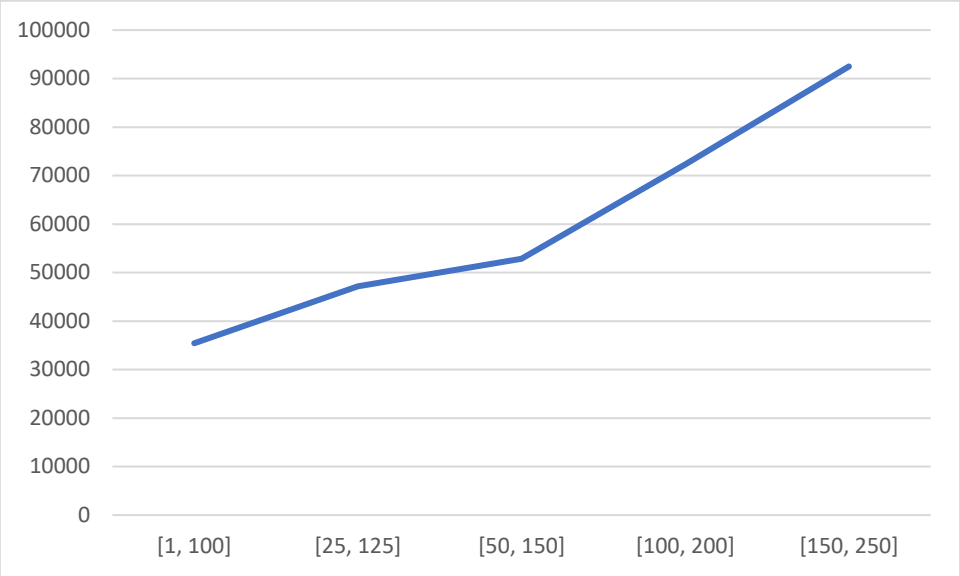
Czas na każdą instancję: 25s
Liczba mrówek od drugiej iteracji: 20
Parowanie: 15%
δ= 0,2
Wartość x: 5
wi = [1, 100]



Zgodnie z oczekiwaniami, im większe stopnie wierzchołków, tym gęściejsze grafy powstają, a co za tym idzie – więcej możliwości przejścia z jednego wierzchołka do drugiego. Stąd łatwiej jest znaleźć ścieżkę o mniejszym koszcie.

3.2.2. Waga krawędzi

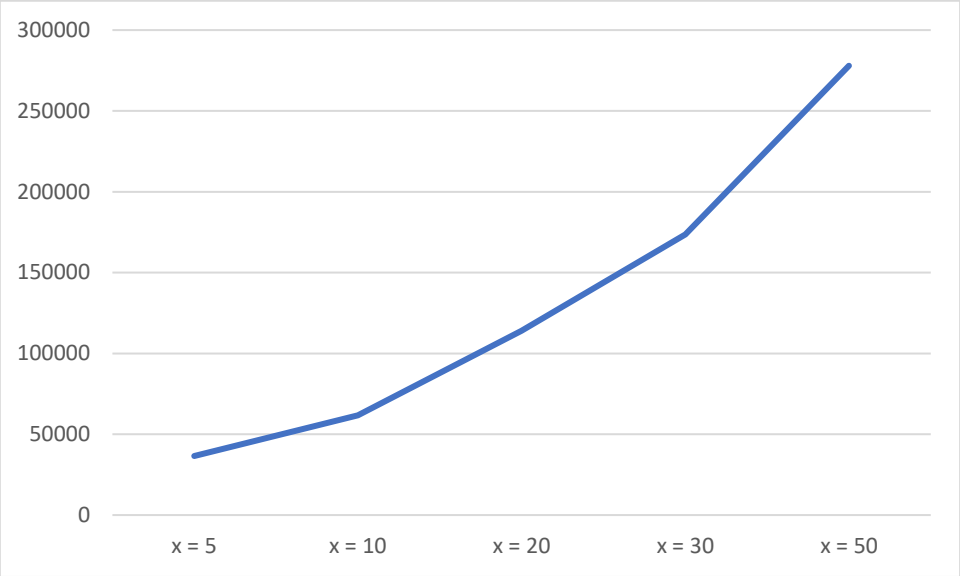
Czas na każdą instancję: 25s
Liczba mrówek od drugiej iteracji: 20
Parowanie: 15%
δ= 0,2
Wartość x: 5
deg(v) = [1, 6]



W przypadku zwiększenia przedziału wag krawędzi będzie rósł koszt S ścieżek, a więc również średnia wartość rozwiązania.

3.2.3. Wartość *x*

Czas na każdą instancję: 25s
Liczba mrówek od drugiej iteracji: 20
Parowanie: 15%
δ= 0,2
deg(v) = [1, 6]
w*i* = [1, 100]



Tak samo jak w przypadku zwiększonego zakresu wag krawędzi, tak w przypadku zwiększania wartości *x* rośnie średnia wartość rozwiązania przy czym tutaj zmiana ta wydaje się być o bardziej zauważalna, gdyż średnia wartość rośnie znacząco już od *x* = 10.

4. Wnioski

Założeniem algorytmu metaheurystycznego jest polepszanie swoich własnych wyników. Na podstawie przeprowadzonych testów widać jednak, że gdy algorytmowi dane zostanie zbyt wiele czasu na znalezienie rozwiązania, to ostateczne rozwiązanie wcale nie musi różnić się znacząco jakością od takiego rozwiązania, które zostało znalezione w krótszym czasie. Analogicznie będzie w sytuacji, kiedy rozszerzy się zbiór rozwiązań, w których poszukiwane jest to najlepsze.

W najlepszym przypadku algorytmowi w takich sytuacjach uda się znaleźć rozwiązanie nieco lepsze od takiego, które zostało znalezione w krótszym czasie lub wybrane spośród mniejszej puli rozwiązań. W najgorszym wypadku takie rozwiązanie będzie gorsze od poprzednich rozwiązań. Przeważnie jednak rozwiązania takie są na tyle do siebie zbliżone, że na ogół dalsze wydłużanie czasu bądź zwiększanie puli rozwiązań nie będzie opłacalne, gdyż nie będzie dawało znacząco polepszonych wyników.

Taka zależność dla algorytmów implementowanych zgodnie z metodami ACO może wynikać z tego, że przy każdej iteracji wybierana jest pewna część najlepszych mrówek, które pozostawią po sobie feromony. Pomimo implementacji takich funkcji jak parowanie czy wygładzanie feromonów, nie uniknie się jednak zostawiania jakichś śladów feromonowych przez mrówki wybierające rozwiązania co najwyżej średnie w skali wszystkich rozwiązań dla danej instancji.

Ciekawym natomiast okazał się wynik testów dotyczący parametru parowania feromonów – mogłoby się wydawać, że mrówki będą dawały gorsze wyniki, kiedy częściej będą szły w sposób zupełnie losowy. Otrzymany wynik może jednak wskazywać na to, że macierz feromonowa nie miała zbyt wielkiej wartości dla mrówek, aby te mogły nadal uzyskiwać dobre rozwiązania. Zależność taka najprawdopodobniej wynika z tego, że w przypadku generatora instancji losowych mrówki nie były kierowane w żaden sposób, a zatem nawet najlepsze rozwiązanie z takiego generatora może nadal nie być najgorszym rozwiązaniem w przypadku mrówek, które szukały rozwiązań od drugiej iteracji. Dlatego też ślady feromonowe zostawione przez mrówki z pierwszej iteracji mogą być nieprzydatne, a w najgorszym wypadku nawet i niepożądane, gdyż fałszywie wskazują rzekomo opłacalne do wybrania krawędzie. W przypadku ustawienia wartości parowania na coraz większe wartości, szybciej parowaniu ulegały fragmenty ścieżek, które nie zawierały się w najlepszych rozwiązaniach. To mogło zapewniać częstsze wybieranie takich ścieżek, które były bardziej opłacalne i zawierały się w najlepszych rozwiązaniach.

W przypadku pozostałych testowanych parametrów uzyskane wyniki były raczej spodziewane i w pewnym sensie oczywiste – im więcej możliwości dojścia do danego wierzchołka, tym więcej możliwości wyboru krawędzi o najmniejszej wadze, przez co maleje koszt S ścieżki. Nietrudno było też przewidzieć, że w przypadku zwiększenia przedziału wag krawędzi oraz wartości *x* koszt S ścieżek znacząco wzrośnie.

Przeprowadzenie testów okazało się kluczowe, aby nabrać lepszego poglądu na to, jak naprawdę działa zaimplementowany algorytm i jakiej jakości rozwiązania daje. Wyniki takich testów są niezastąpione w przypadku, kiedy chce się polepszyć rozwiązania podawane przez metaheurystykę. O wiele łatwiej jest w taki sposób zauważyć, które zmiany parametrów mają rzeczywisty wpływ na poprawę lub pogorszenie jakości rozwiązania.

Źródła

- *Ant Colony Optimization - Optimal Number of Ants*, Ch. L. Johansson, L. Pettersson, https://www.diva-portal.org/smash/get/diva2:1214402/FULLTEXT01.pdf
- *Ant colony optimization for routing and load-balancing: Survey and new directions*, https://www.researchgate.net/publication/3412306_Ant_colony_optimization_for_routing_and_load-balancing_Survey_and_new_directions
- http://www.cs.put.poznan.pl/mradom/teaching/laboratories/OptKomb/CI_wyklad_ewoluc_4.pdf