Optymalizacja kombinatoryczna - sprawozdanie

Sebastian Nawrot 145177 Bioinformatyka
sebastian.nawrot@student.put.poznan.pl

1 Problem

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 minimum raz) taką, aby zminimalizować jej koszt S. Koszt ten wyliczany jest dynamicznie w trakcie konstrukcji w taki sposób, że:

a. stanowi sumę wszystkich wag odwiedzonych do tej pory krawędzi, oraz

b. co x odwiedzonych łuków licząc od startu do aktualnej sumy S dodawana jest suma ostatnich $x/2$ odwiedzonych wag pomnożona przez podwójny stopień wierzchołka, w którym znajduje się algorytm przeszukiwania po przejściu x krawędzi.

Założenia dla instancji problemu: można przyjąć początkowo: $|V|$ minimum: 100, $deg(v) = [1, 6]$, $w_i = [1, 100]$, $x =$ minimum 5.

2 Opis algorytmu

2.1 Wyszukiwanie pierwszego rozwiązania:

Algorytm zaczyna od wyboru wierzchołka z najniższym indeksem, jest on dodawany do wektora przechowującego rozwiązanie. Następnie w pętli, z obecnego rozwiązania wybieramy ostatnio dodany wierzchołek i sprawdzamy czy sąsiaduje on z jakimkolwiek wierzchołkiem, którego jeszcze nie ma w rozwiązaniu.

- Jeżeli tak, dodajemy nieodwiedzony wierzchołek do rozwiązania i powtarzamy ten krok. Jeżeli istnieje kilka takich wierzchołków wybieramy ten, do którego prowadzi krawędź z najniższą wagą.
- Jeżeli nie, wybieramy z rozwiązania jeszcze wcześniejszy wierzchołek i powtarzamy krok. Cofając się po ścieżce, po natrafieniu na nieodwiedzonego wcześniej sąsiada, dodajemy go w obecnym miejscu rozwiązania. W tym wypadku dodajemy także wierzchołek który do niego prowadził.

Algorytm kończy działanie gdy wszystkie wierzchołki grafu znajdują się w rozwiązaniu.

2.2 Wyszukiwanie sąsiedztwa:

Wyszukiwanie sąsiedztwa dla obecnego rozwiązania odbywa się poprzez wybór 2 krawędzi i próbę zamiany ich kolejnośći, odwracając jednocześnie ścieżkę pomiędzy nimi.

Przykładowo dla rozwiązania:
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8

wybieramy krawędzie 3 i 6 i odwracamy fragment pomiędzy nimi, powstaje w ten sposób:
1 - 2 - 6 - 5 - 4 - 3 - 7 - 8

Jeżeli ścieżki nie da się utworzyć, tzn. dla powyższego przykładu nie ma krawędzi pomiędzy wierzchołkami 2 - 6 i 3 - 7, wyszukujemy połączenia pomiędzy nimi. Szukanie tej ścieżki jest ograniczone, dobudowane połączenie może zawierać maksymalnie 3 wierzchołki, używamy do tego algorytmu przeszukiwania w głąb.

Ilość sąsiedztw do wyszukania kontrolowana jest przez parametr wejściowy algorytmu. Na wstępie zwracane są sąsiedztwa, które nie wymagają dobudowy dodatkowej ścieżki, ponieważ to właśnie one w większości wypadków prowadzą do najlepszych rozwiązań. Jest ich na tyle mało (np. dla grafu ze 100 wierzchołkami i ilością krawędzi 6-30 jest ich około 200) że dopiero w drugiej kolejności uzupełniamy je rozwiązanami z dobudowanymi połączeniami.

Aktualizacja listy tabu odbywa się poprzez dodanie skrajnych wierzchołków zamienionego fragmentu, dla powyższego przykładu będzie to 3 i 6.

Nie jestem pewien czy jest to dobry sposób wyszukiwania sąsiedztw, zamiana kolejności fragmentów w istniejącej ścieżce wydaje mi się jedynym dobrym pomysłem. Niestety liczba takich możliwych podstawień jest dosyć mała a na laboratoriach była mowa o ograniczaniu ich ilości do kilku tysięcy, stąd próba zwiększenia ich ilości poprzez tworzenie losowych połączeń.

2.3 Działanie algorytmu:

Na wstępie wyszukujemy pierwsze rozwiązanie używając funkcji opisanej w podpunkcie 2.1, oznaczamy je jako obecne rozwiązanie. Następnie przechodzimy do wykonywania głównej pętli algorytmu, wykonujemy ją przez ilość czasu podaną w parametrze wejściowym, domyślnie jest jedna minuta.

W pętli wyszukujemy sąsiedztwa dla obecnego rozwiązania, ich ilość także kontrolowana jest przez parametr wejściowy. Ze zbioru wygenerowanych sąsiedztw wybieramy te, które najbardziej polepsza obecne rozwiązanie czyli te, dla którego wartość funkcji celu jest najmniejsza. Ustawiamy je jako nasze obecne rozwiązanie.

Jeżeli żadne z sąsiedztw nie polepsza obecnej ścieżki przechodzimy w tryb pogarszania rozwiązania przez ilość następnych iteracji określoną w kolejnym parametrze wejściowym. Następnie z powrotem przełączamy w tryb normalny i zaczynamy szukać polepszeń.

2.4 Dodatkowe elementy:

W każdej iteracji pętli, zaraz po wybraniu najlepszego kandydata na następne rozwiązanie próbujemy polepszyć ścieżkę manualnie. Robimy to poprzez usunięcie z rozwiązania wierzchołków, które znajdują się w ścieżce dwa razy. Ponieważ generowanie sąsiedztwa poprzez dobudowanie fragmentów zwiększa długość ścieżki potrzebujemy czegoś co będzie w stanie ją skrócić. Algorytm ten jest bardzo prymitywny i pozostawia sporo do życzenia.

Działanie polega na wyszukaniu wszystkich powtarzających się wierzchołków. Następnie, dla każdego z nich sprawdzamy czy po jego wycięciu możliwe jest odtworzenie ścieżki w jego miejscu tzn. czy wierzchołki po lewej i prawej stronie mają wspólną krawedź. Jeżeli tak usuwamy go, powtarzamy te operację aż do momentu gdy każdy z nich będzie występował tylko raz, lub gdy pomiędzy żadymi z sąsiednich wierzchołków nie będzie krawędzi.

Naturalnym wydało mi się dodanie sprawdzenia czy po takiej operacji, nowo powstałe rozwiązanie jest lepsze od poprzedniego. Testy wykazały jednak, że prowadzi to do gorszych końcowych rozwiązań i jakiekolwiek skrócenie ścieżki jest lepsze od polepszenia wyniku funkcji celu.

Kolejnym sposobem na ulepszenie algoyrtmu było wykorzystanie wszystkich sąsiedztw wegenerowanych w obecnej iteracji i stworzenie z nich jednego, jeszcze lepszego rozwiązania. Jeżeli zmiany w sąsiedztwach nie pokrywały się, tzn. można było jednocześnie wykonać obydwa przekształcenia, sprawdzaliśmy czy generuje to lepsze rozwiązanie. Niestety, mimo tego, że w pierwszych kilku iteracjach wyniki były znacząco lepsze, to końcowe wyniki były gorsze. Z tego powodu zrezygnowałem z tej funkcjonalności.

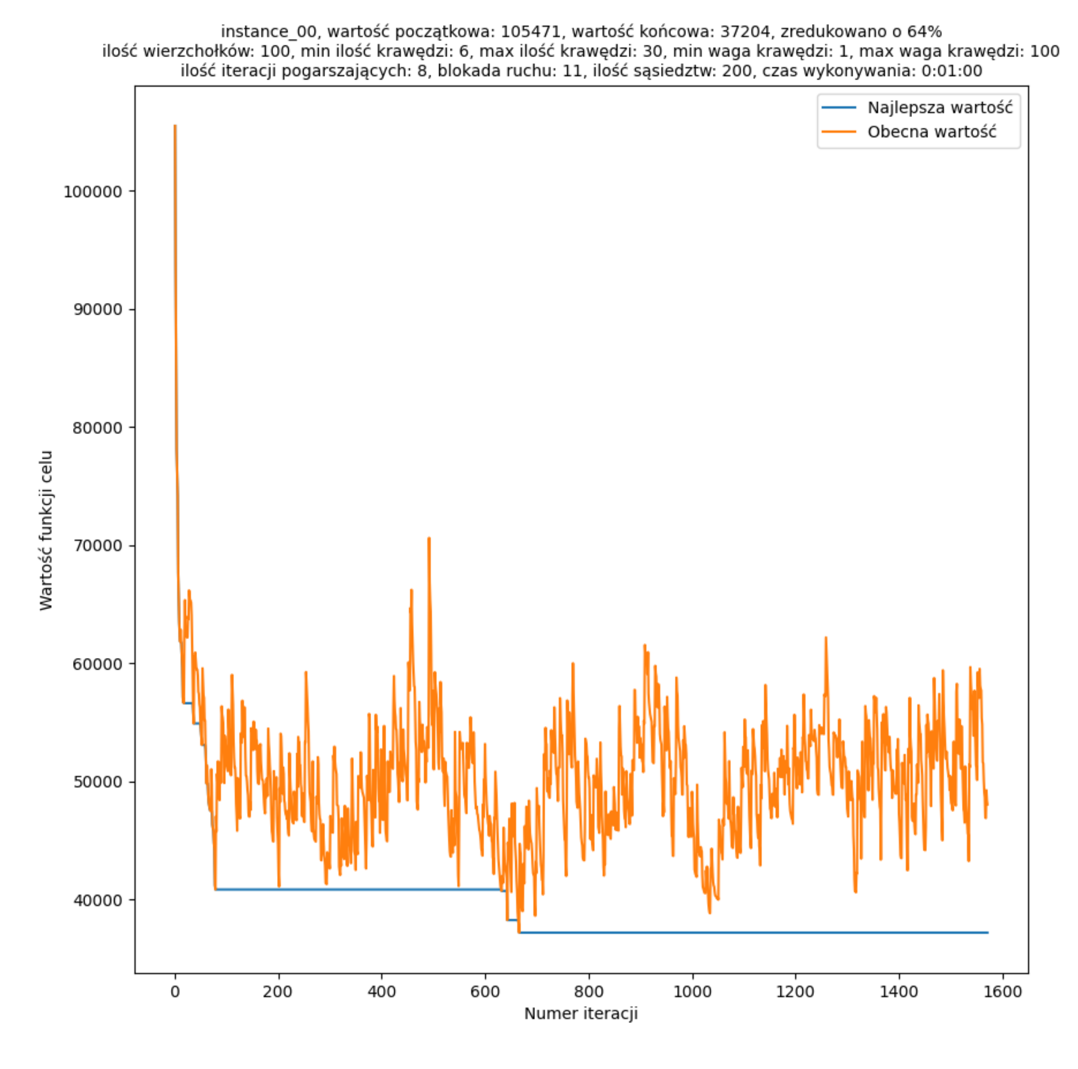
3 Testowanie metaheurestyki

W algorytmie, poza oczywistymi parametrami grafu, możemy przetestować następujące parametry:

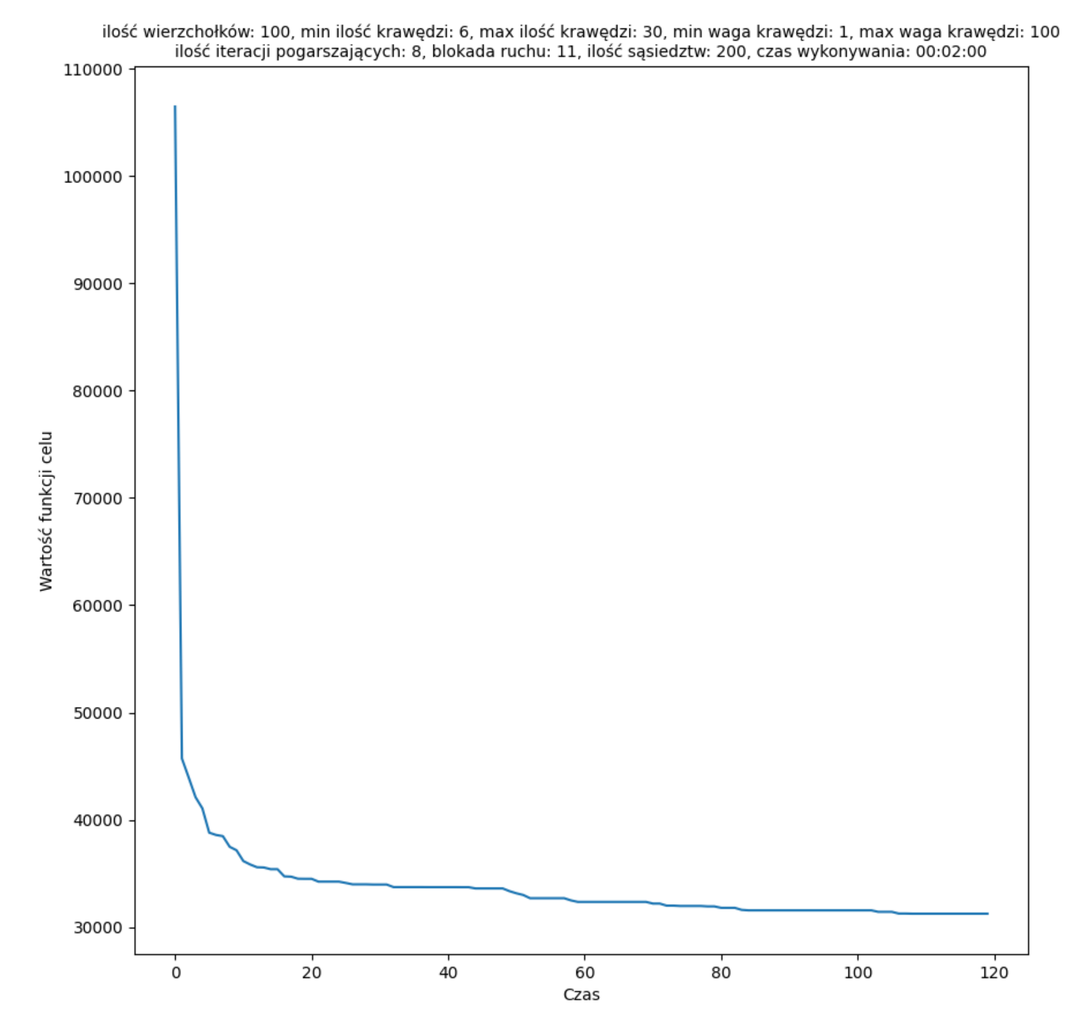
- Ilość iteracji, na którą ruch jest blokowany (dodawany do listy tabu)
- Ilość iteracji algorytmu w fazie pogarszającej rozwiązania
- Ilość sąsiedztw do wyszukania
- Czas wykonywania algorytmu

Podczas tworzenia algorytmu domyślnymi parametrami dla powyższych właściwości było 11 iteracji blokady, 8 iteracji pogarszania, 300 sąsiedztw, i 60 sekund trwania. Dla przykładowego grafu 100 wierzchołków 6-30 krawędzi, końcowe rozwiązanie było średnio o 70% lepsze względem pierwszego. Jest to dobry punkt odniesienia i względem tych wartości będą wykonywane testy.

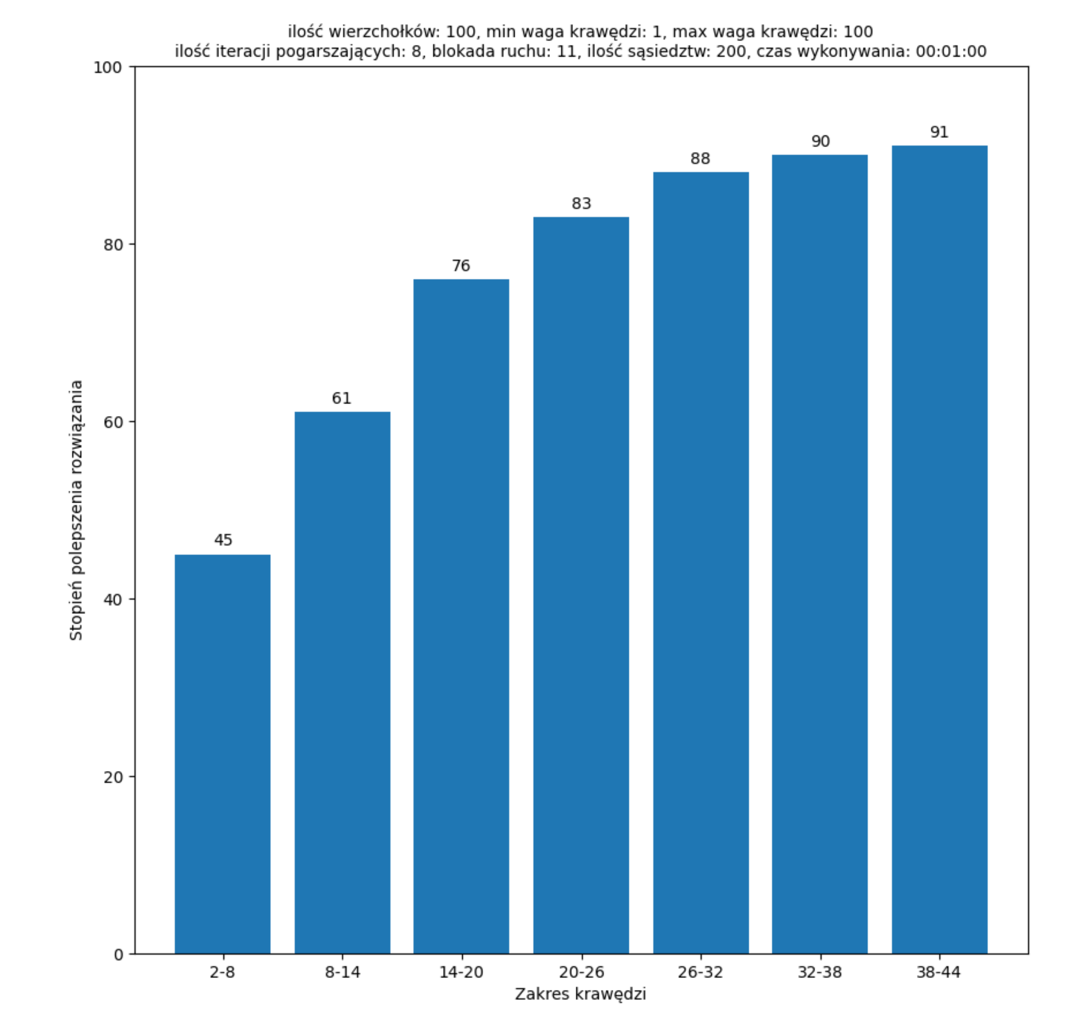
Wykres przedstawia wykonanie algorytmu dla powyższych wartości, wykonanie całości trwało minutę. Możemy zauważyć, że algorytm działa poprawnie, tzn. polepsza wartość funkcji celu i nie zapętla się wokół jednego rozwiązania. Widać także, że w miarę upływu czasu efektywność spada i coraz ciężej znaleźć rozwiązanie polepszające.

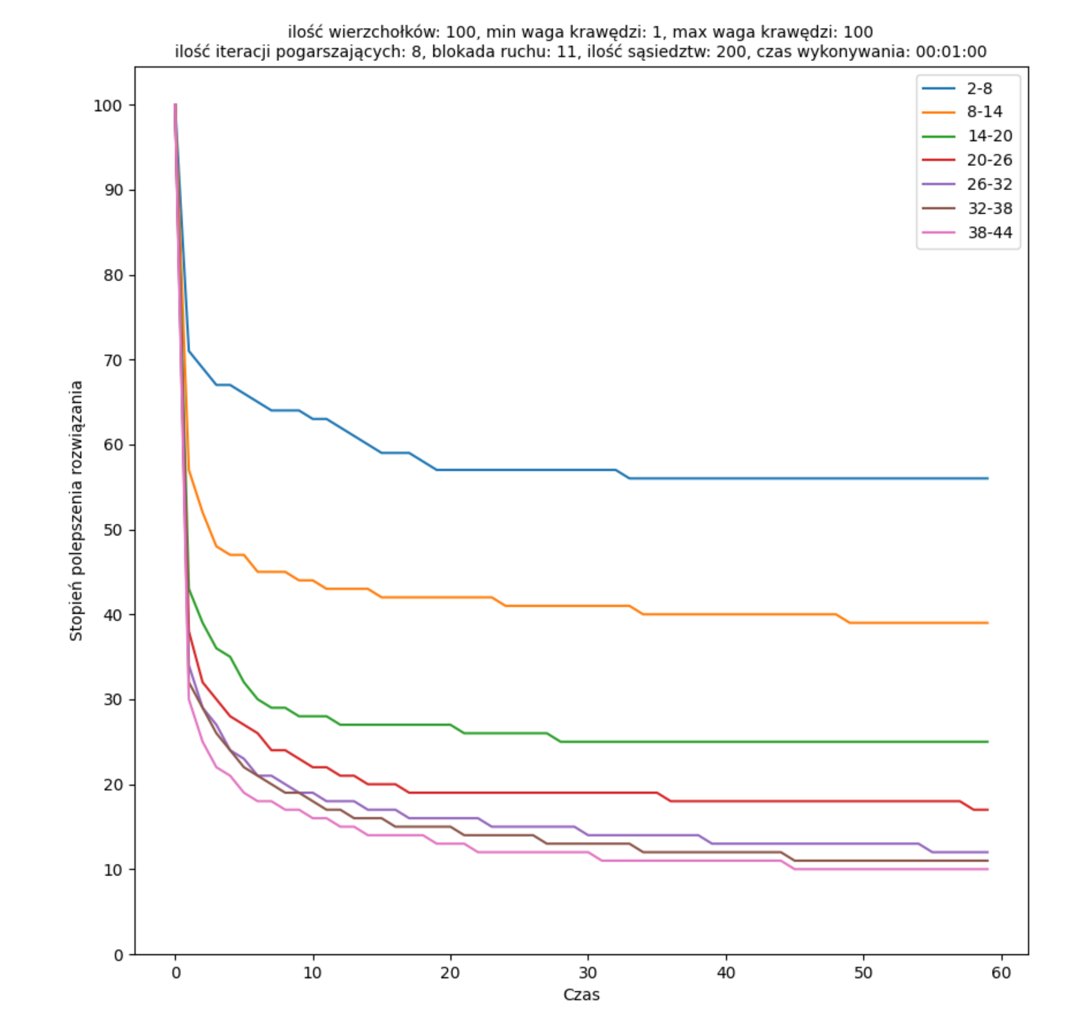


Czas działania algorytmu:
Na poniższym wykresie przedstawiona jest uśredniona wartość funkcji celu dla 100 instancji. Jak można zauważyć, po minucie poprawa jest znikoma. Dla kolejnych testów jako czas wykonania przyjmiemy 60 sekund.

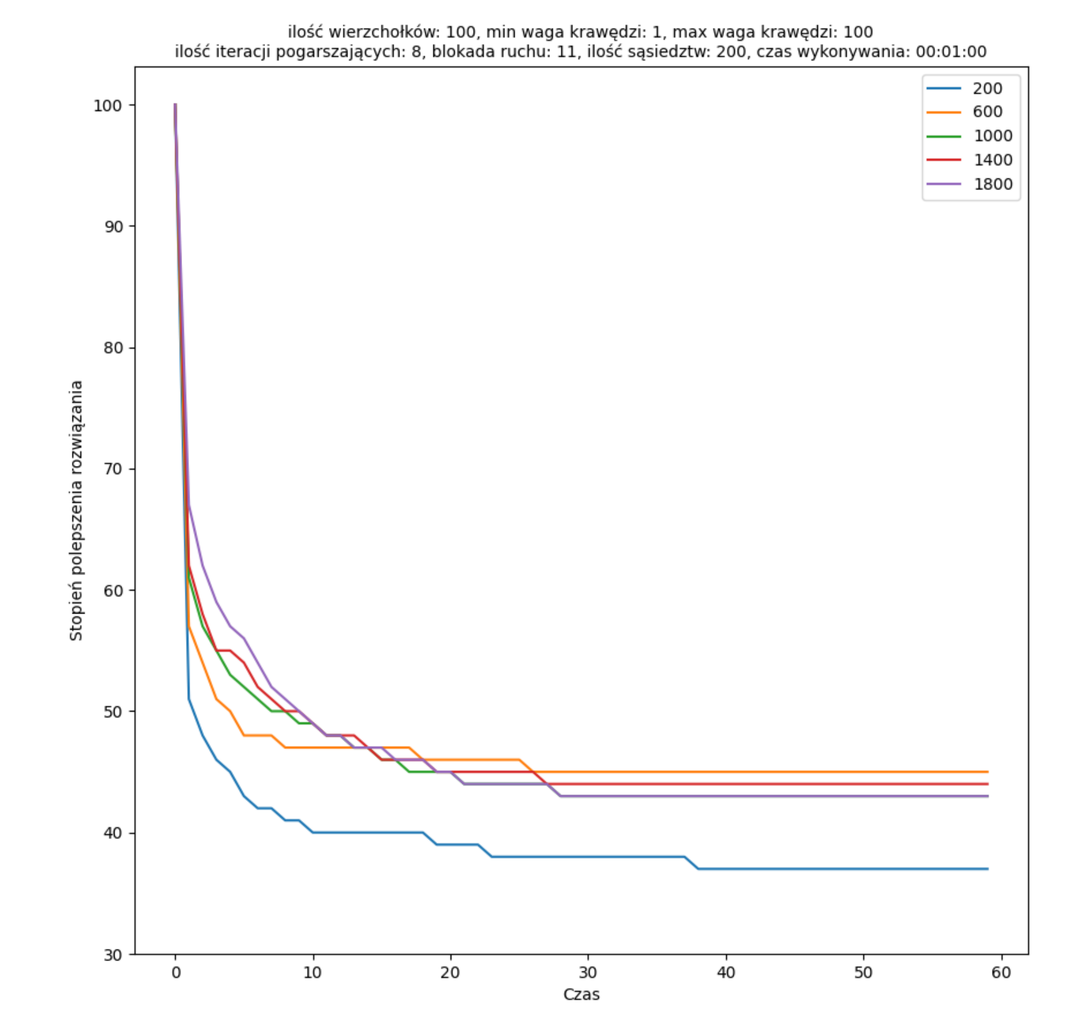


Wpływ ilości krawędzi na końcowe rozwiązanie:
Na poniższym wykresie przedstawiony jest uśredniony stopień polepszenia wartości rozwiązania końcowego względem pierwszego wyrażony w procentach, tzn. dla zakresu 38-44 początkowa wartość funkcji celu została zredukowana o 93%. Dla każdego zakresu krawędzi wygenerowano 100 instancji. Możemy zauważyć, że istnieje zależność pomiędzy ilością krawędzi i stopniem poprawy rozwiązania. Im więcej krawędzi, tym łatwiej znaleźć polepszające sąsiedztwo co prowadzi do lepszych wyników końcowych.

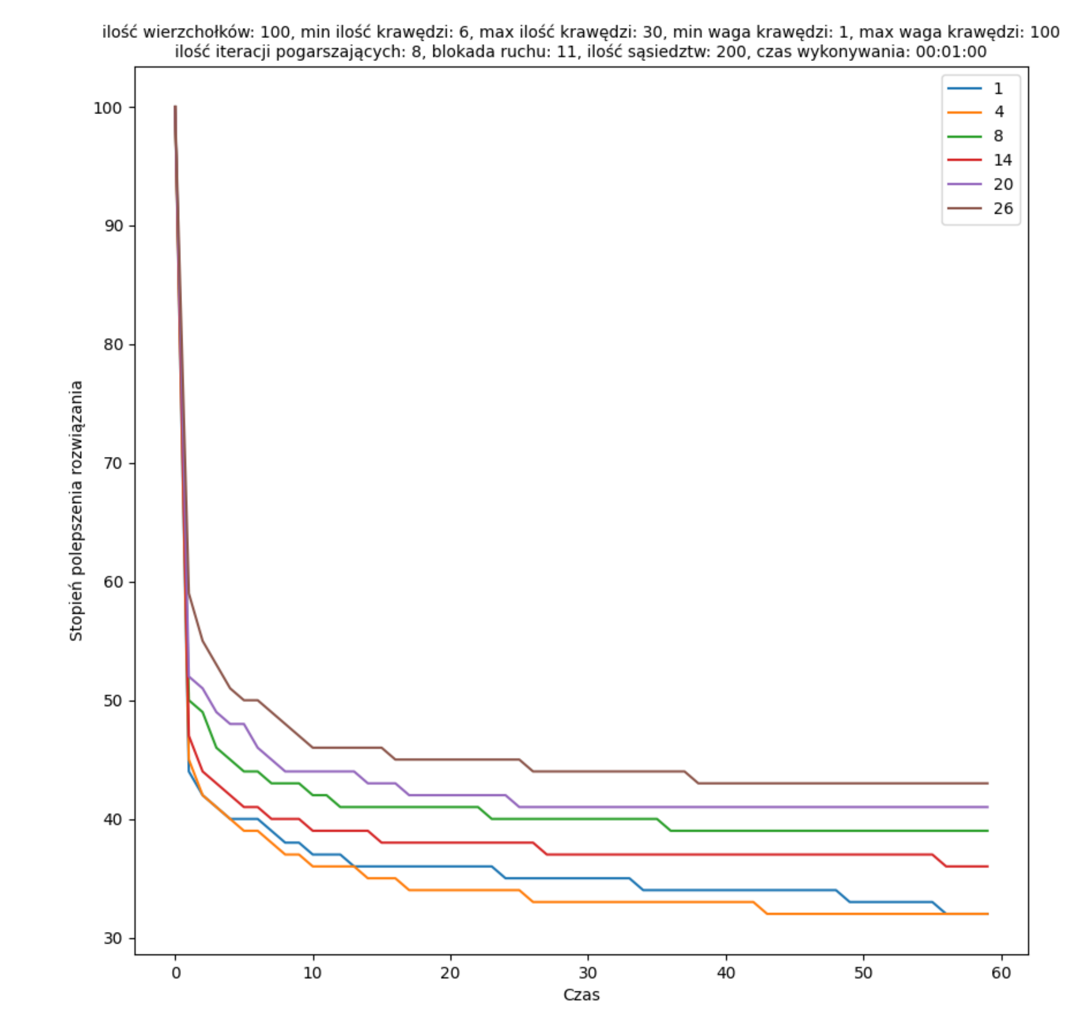




Wpływ ilości wyszukiwanych sąsiedztw na końcowe rozwiązanie:
Wykres przedstawia grupy instancji z różną ilością sąsiedztw generowanych w jednej iteracji i ich wpływ na końcową wartość fukncji celu. Dla wartości 200 sąsiedztw uzyskujemy najlepsze rezultaty. Wynika to z że, dla parametrów wygenerowanych grafów (100 wierzchołków, 6-30 krawędzi) ilość sąsiedztw uzyskiwanych bez dobudowywania ścieżki do właśnie około 200. Wszystkie sąsiedztwa powyżej tej wartości to ścieżki z losowo wybranymi wierzchołkami i dobudowanymi połączeniami. Bardzo rzadko są w stanie coś polepszyć i z racji 60 sekund na wykonanie algorytmu ich generowanie jest marnowaniem czasu na dodatkowe iteracje. Ze szczegółowych grafów poszczególnych instancji możemy odczytać że przy 200 sąsiedztwach program wykonuje średnio około 600 iteracji przez 60 sekund, dla 600 sąsiedztw jest to 150 iteracji, dla 1000 100, dla 1400 80 i dla 1800 około 70. Nie przekłada się to jednak całkowicie na wyniki na wykresie, sąsiedztwa od 600 do 1800 znajdują się mnie więcej na tym samym poziomie. Nie jestem w stanie określić dlaczego, aczkolwiek ukazuje to, że moduł dobierania losowych ścieżek jest całkowicie bezużyteczny. Najlepszym sposobem na ustawienie tego parametru jest jego całkowite usunięcie i wykorzystanie wszystkich rozwiązań z pierwszej fazy szukania sąsiedztw.



Wpływ ilości iteracji pogarszających rozwiązanie na końcowe rozwiązanie:
Wykres przedstawia zależność pomiędzy ilością iteracji pogarszających rozwiązanie i stopniem polepszenia wartości funkcji celu. Wartość w okolicach 4 wydaje się najlepszym wyborem dla tego parametru. Wartości dla testu mają zbyt duży rozkład, warto byłoby przeprowadzić kolejny test z wartościam od 1 do 8 aby poznać najlepszą wartość.



4 Podsumowanie

No jest wszystko w porządku, jest dobrze, dobrze robi, dobrze wszystko jest w porządku. Jest git pozdrawiam całą bioinformatykę, dobrych chłopaków i niech sie to trzyma. Dobry kod leci.