**Problem kolorowania grafu na podstawie algorytmu wspinaczkowego**

**Wstęp**

Projekt został napisany w języku Java w IDE IntelliJ Idea. W celu ułatwienia zostały użyte dwie biblioteki: log4j w celu łatwiejszego logowania, oraz GraphStream (<http://graphstream-project.org/)> w celu szybszej wizualizacji tworzonych struktur. Zbiór testowy generowany był na podstawie generatorów z bilbioteki GraphStream oraz ze źródła: http://mat.gsia.cmu.edu/COLOR/instances.html

**Opis implementacji algorytmów**

**Algorytm wspinaczkowy**

Implementacja algorytmu znajduje się w klasie *pl.edu.pjatk.nai.algorithm.HillClimbingAlgorithmImpl.* Każdorazowe wykonanie algorytmu zaczyna analizę z losowego punktu. Każdy wierzchołek w którym algorytm zaczyna pracę jest „kolorowany” (oznaczany liczbą naturalną). Kolor wybierany jest na podstawie najniższego możliwego ze wszystkich sąsiadów. Następnie algorytm losowo sprawdza swoich sąsiadów do momentu szukając sąsiada z wyższym kolorem. Po jego odnalezieniu zaczyna swoją pracę od nowa, dla znalezionego sąsiada. Jeżeli algorytmowi nie uda się odnaleźć sąsiada z wyższym kolorem zakańcza swoją pracę.

**Algorytm genetyczny**

Implementacja algorytmu znajduje się w klasie *pl.edu.pjatk.nai.algorithm.GeneticAlgorithmImpl.* Algorytm zaczyna swoją pracę od wygenerowania listy populacji dla której będą kontynuowane dalsze kroki. Statystycznie, do listy populacji powinno zostać dodane ok 50% wierzchołków. Następnie, wylosowane wierzchołki poddawane są mutacji (ok 1%). Mutacja polega na losowej zmianie jednego z wylosowanych bitów. Jako kolejny krok algorytm realizuje krzyżowanie. W ramach tej akcji algorytm dobiera wierzchołki z populacji w losowe pary i zamienia ich wartości binarne w losowym punkcie.

Następnie algorytm rozpoczyna „ocenianie populacji”. Sprawdza, jaki procent ma wartość jego koloru w stosunku do sumy kolorów w całej populacji. Na tej podstawie wykonuje rozkład prawdopodobieństwa i wybiera losową ilość osobników z populacji (z tym warunkiem, że przed rozpoczęciem losowania, odwraca wartości rozkładu w celu wybrania wierzchołków z najniższymi wartościami – te które wg. mnie były bardziej wartościowe przy kolorowaniu grafu).

Końcowym etapem jest przypisanie wartości utworzonej populacji do istniejącego grafu.

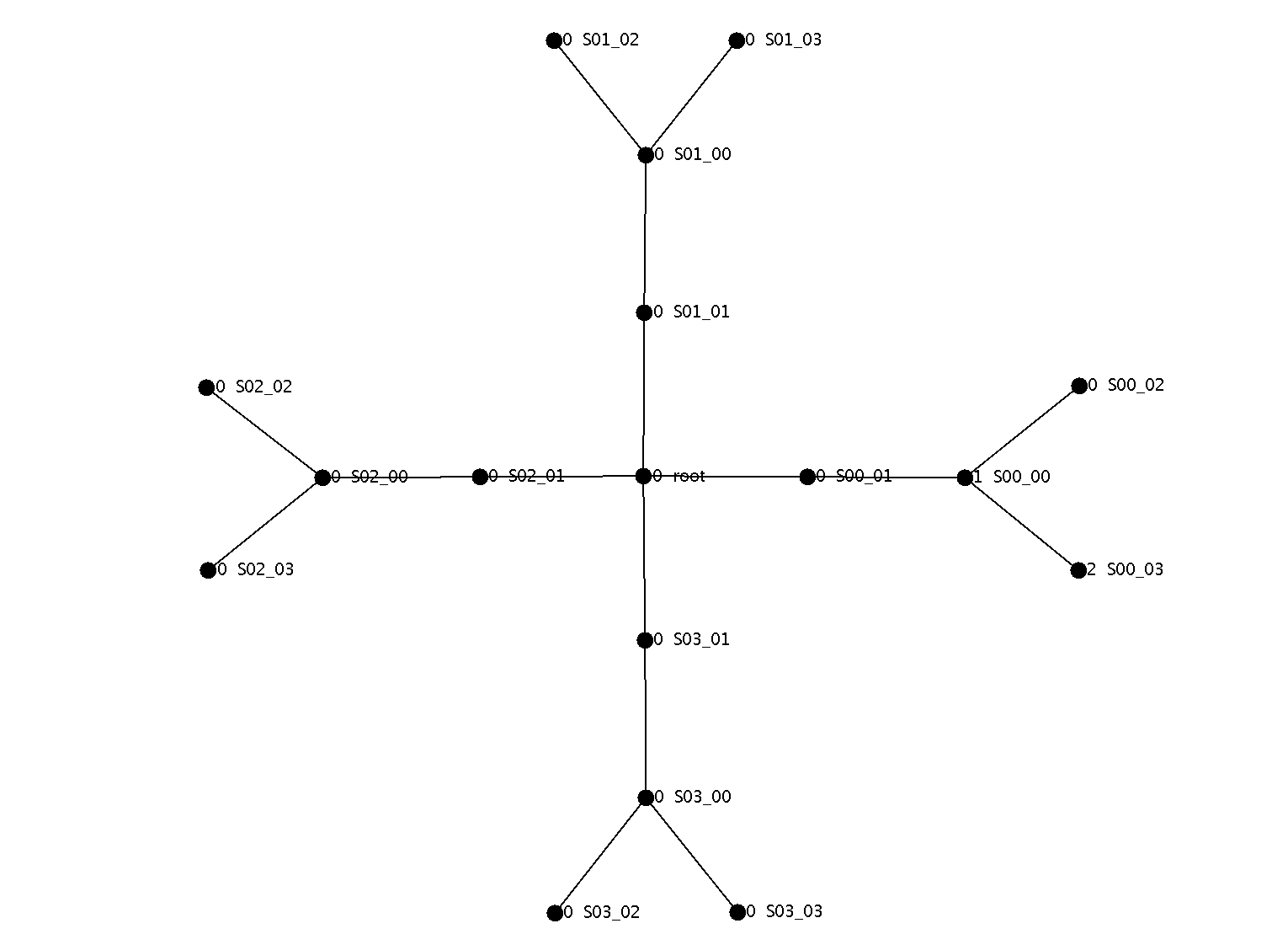
**Testowanie algorytmów**

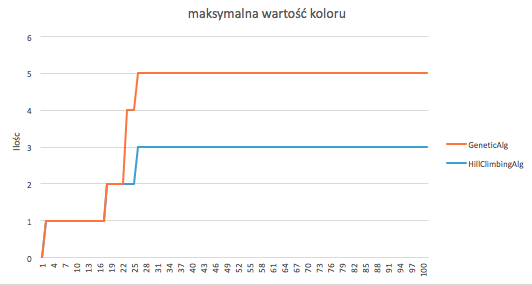
Algorytmy były poddawane n-krotnemu wykonaniu na podstawie grafów wygenerowanych przez bibliotekę GraphStream lub danych dostarczonych na stronie: <http://mat.gsia.cmu.edu>. Każdorazowe wykonanie algorytmu tworzyło raport który pozwalał na analizę i weryfikację:

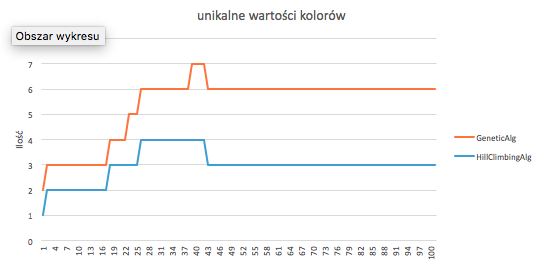
* Maksymalnej wartości kolorów wierzchołków w grafie
* Ilości różnych (unikalnych) wartości wierzchołków w grafie
* Wierzchołków z:
* Sąsiadami o tym samym kolorze
* Sąsiadami o innym kolorze

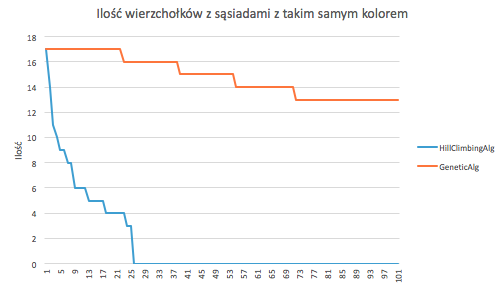
**BananaTree**

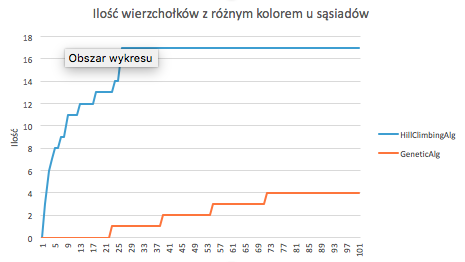
Drzewo BananaTree wygenerowane przez bibliotekę – każdy algorytm wykonany 100 razy.





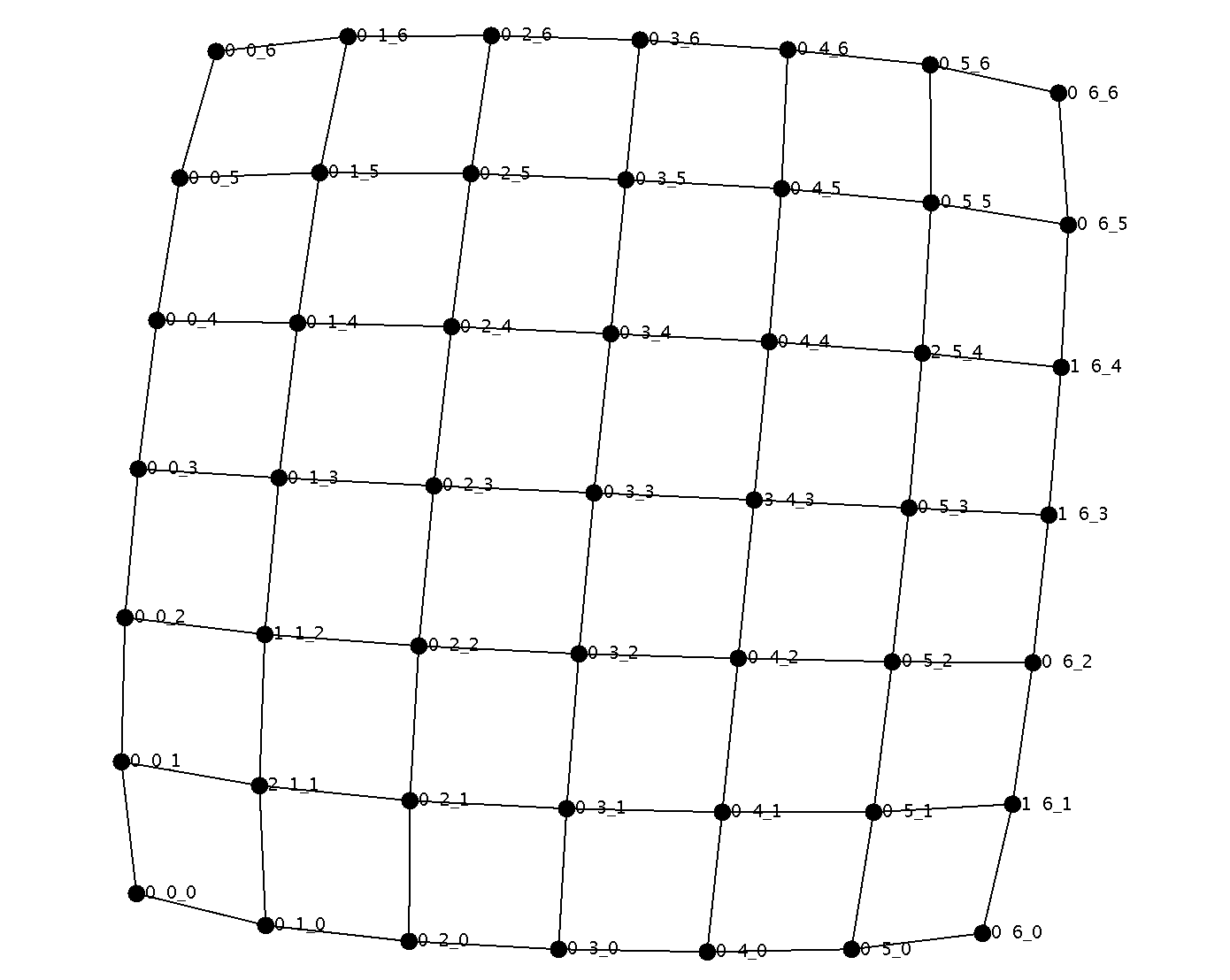


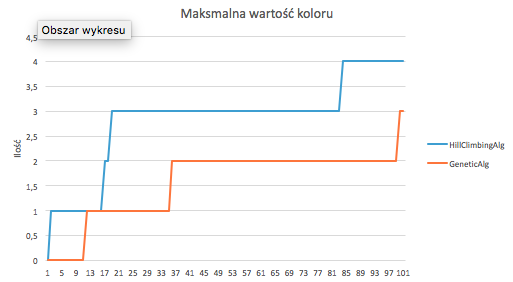


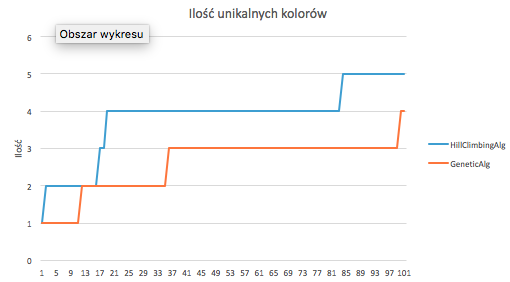


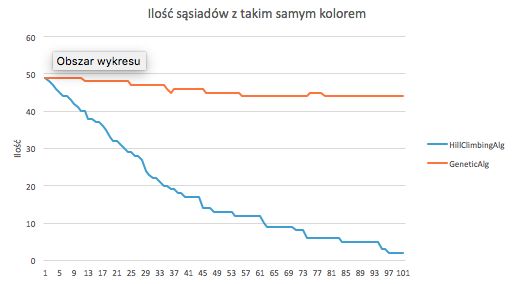
**Grid Graph**

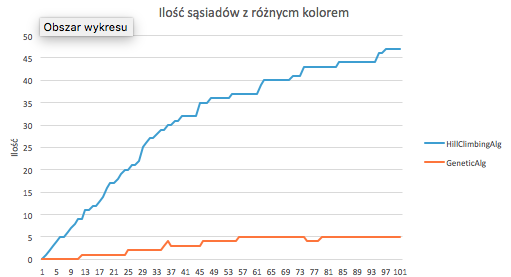
Graf wygenerowany na podstawie generatora z biblioteki. 100 iteracji każdego algorytmu.





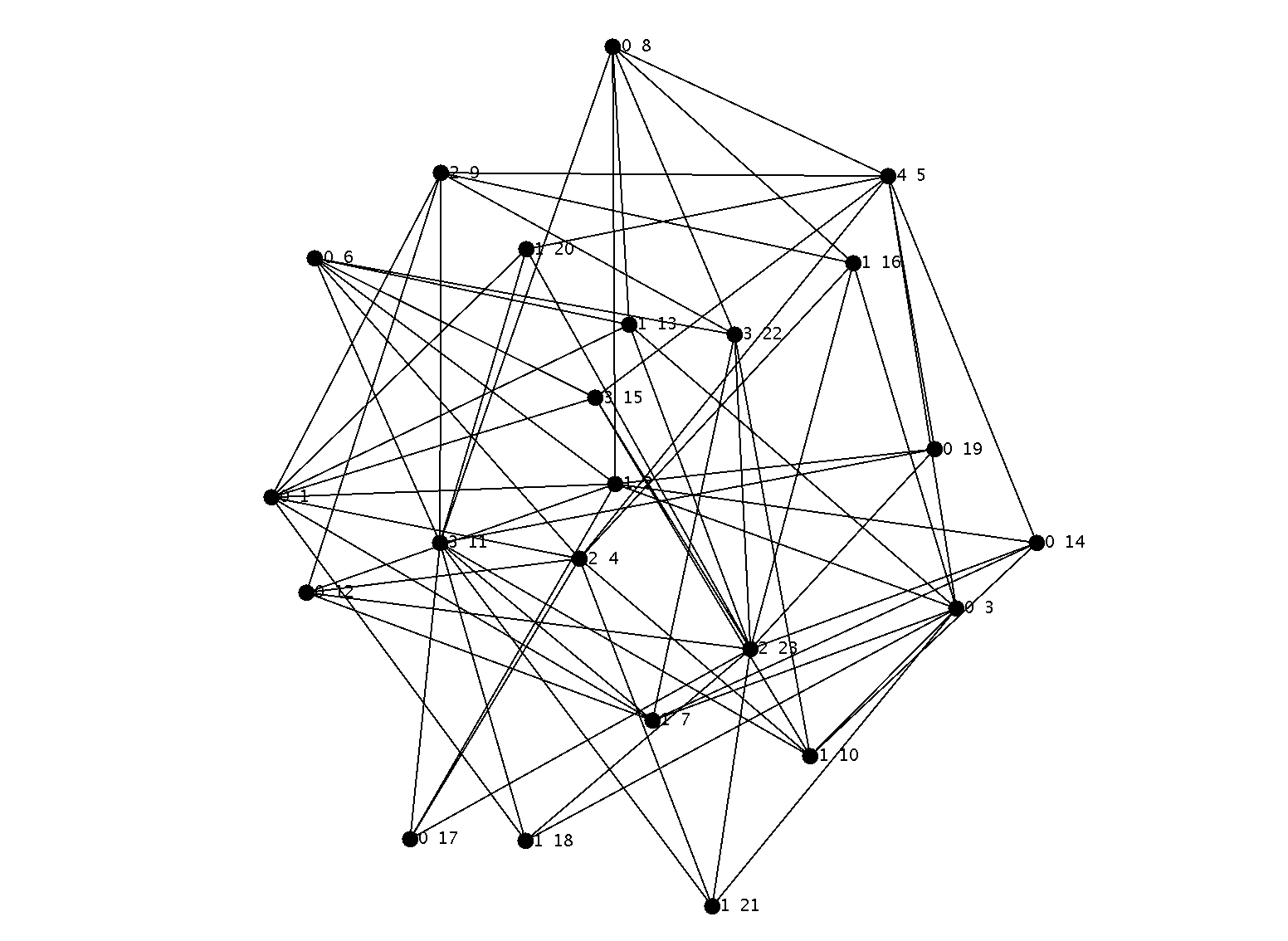


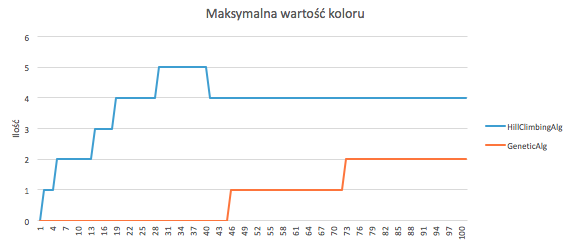


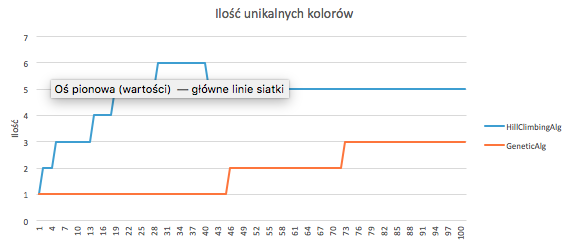


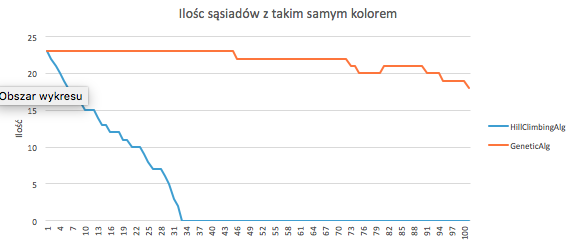
**Graf przykładowy myciel4.col**

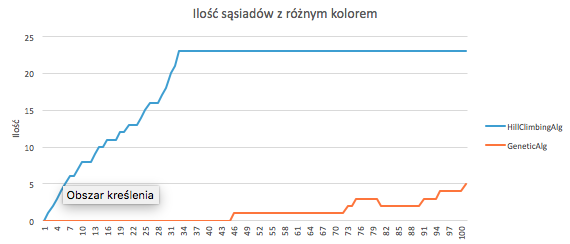
Graf z załączonego pliku. Optymalna ilość kolorów to 5. 100 iteracji każdego algorytmu





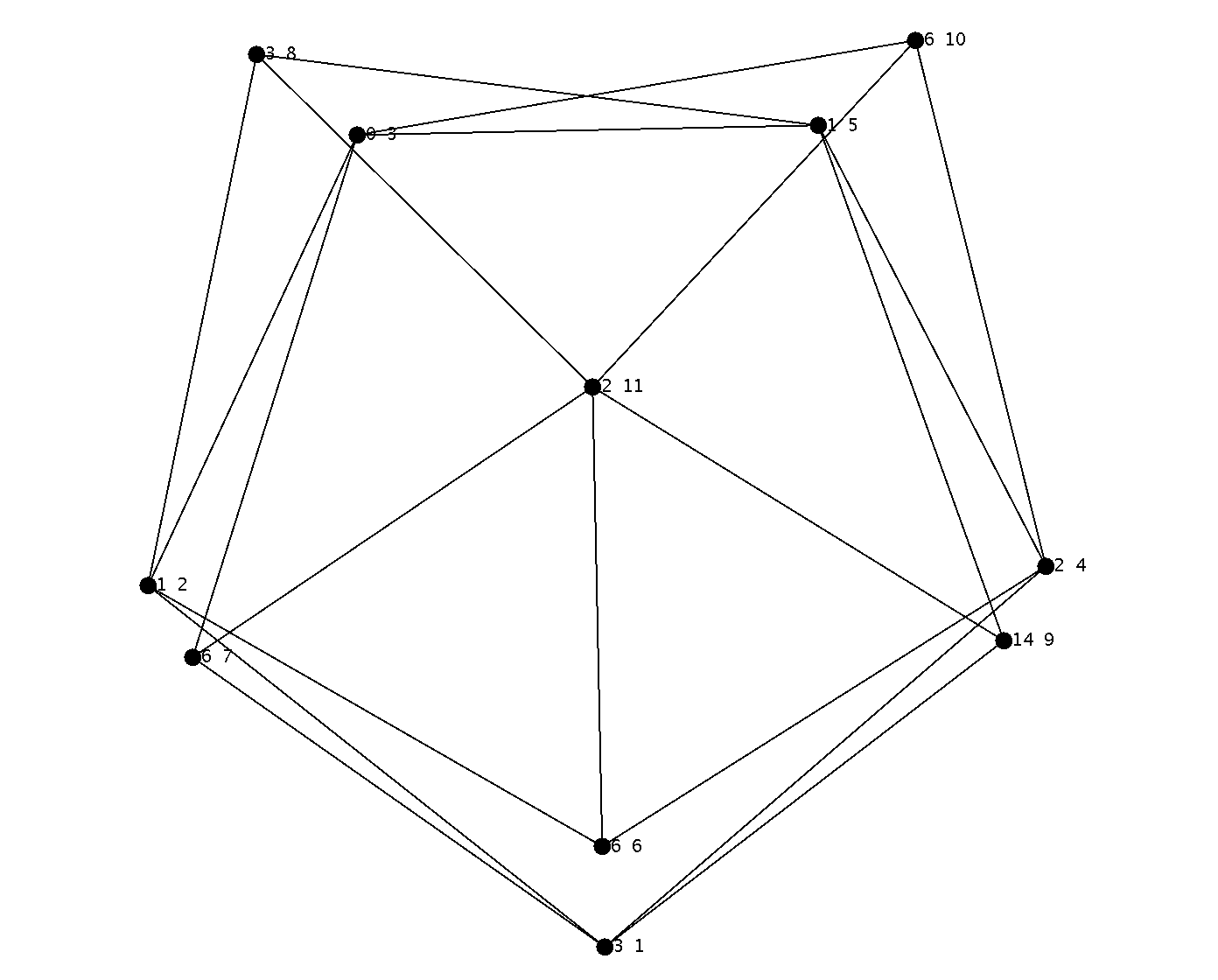


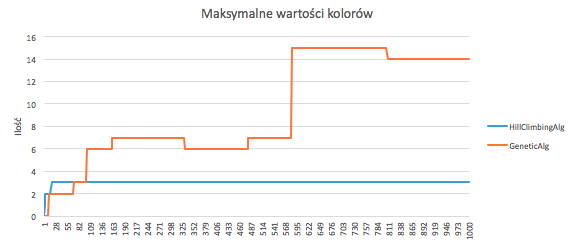


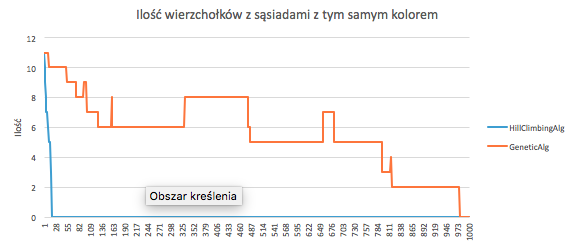


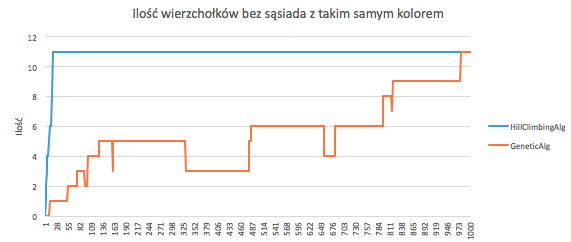
**Graf myciel3.col**

Optymalna ilość różnych kolorów to 4. W tym wypadku wykorzystałem 1000 iterację algorytmów.



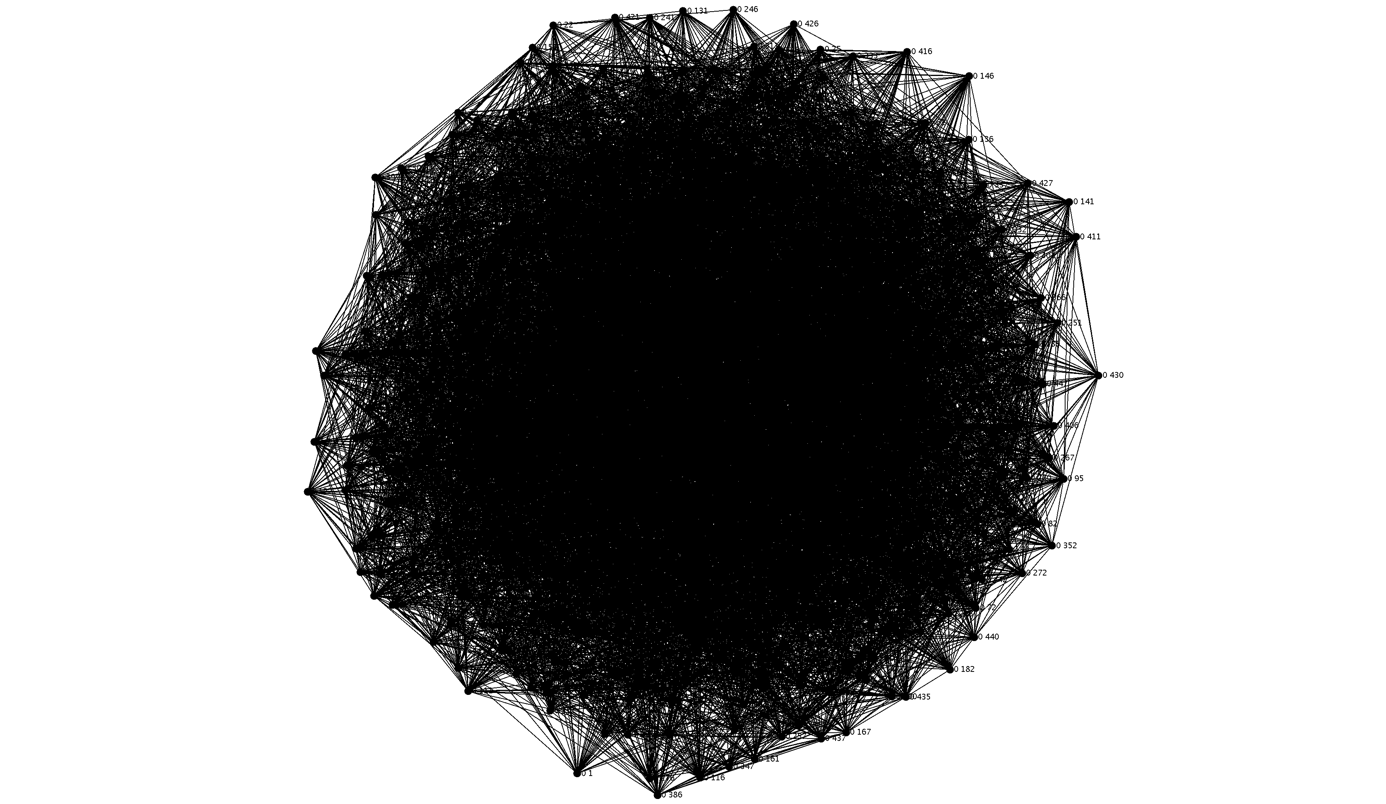


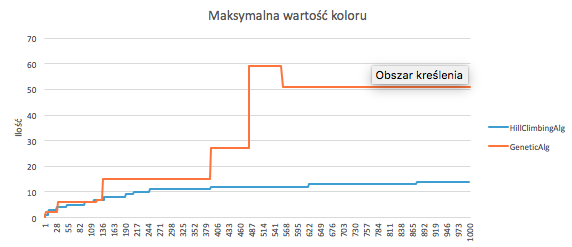


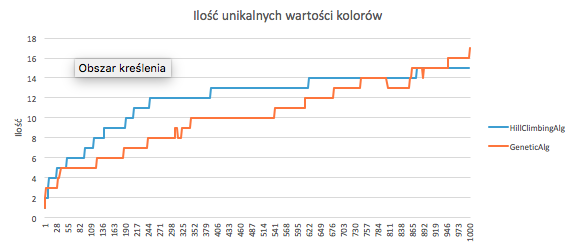


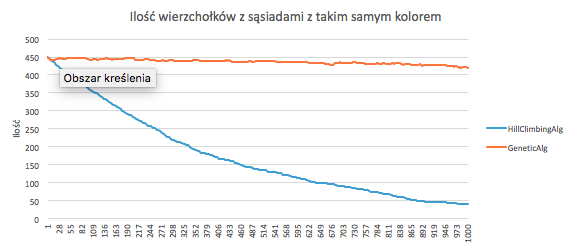
**Graf le450\_5c.col**

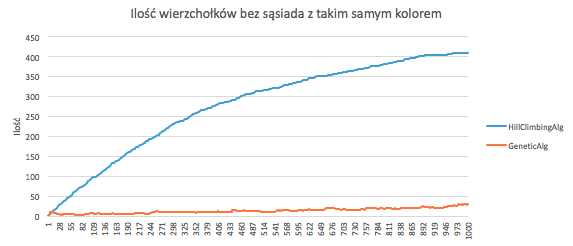
Graf z 5 optymalnymi kolorami. 1000 iteracji algorytmu.





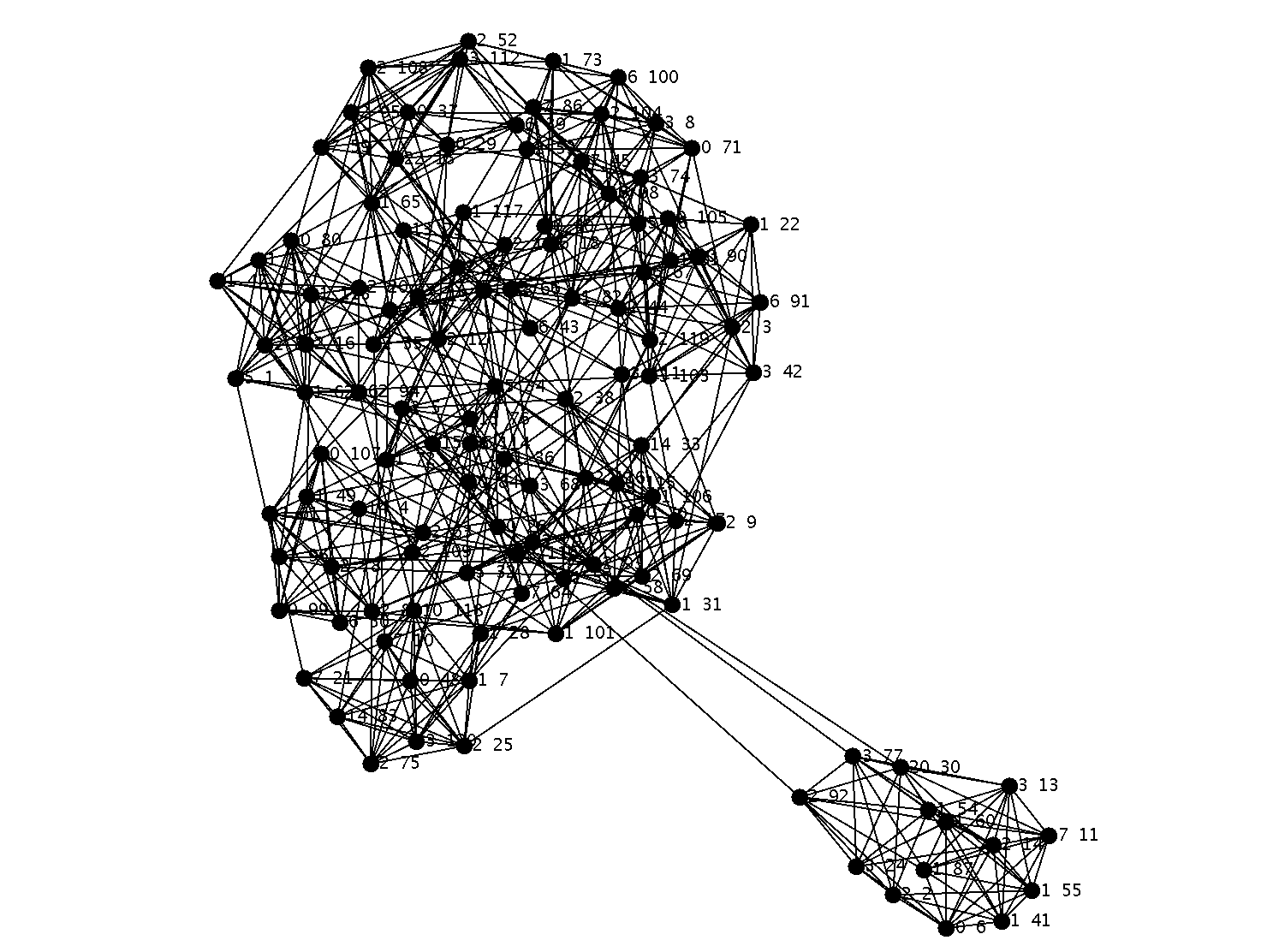


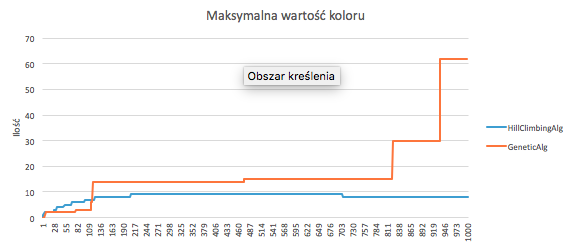


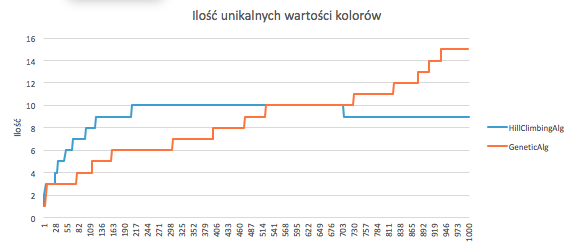


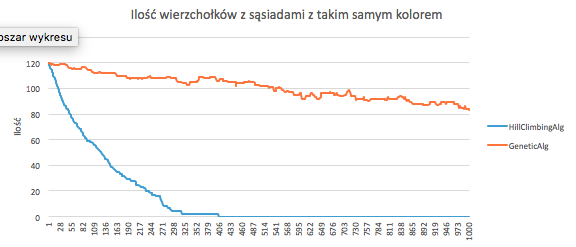
**Graf games120.col**

1000 iteracji. 9 optymalnych kolorów.









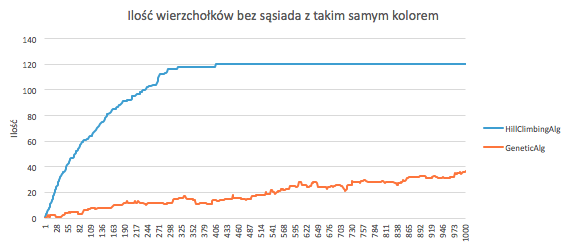
**Wnioski**

Algorytm HillClimbing w przedstawionych przeze mnie przykładach w większości generował wynik optymalny lub do niego bardzo zbliżony. Dużo mniej dokładny był algorytm genetyczny – w jego przypadku na standardowych wartościach wynik w większości przypadków znacząco odbiegał od wartości optymalnych.

Testując jego wartości początkowe – szanse osobnika populacji na mutację oraz jego szanse na jego pozytywną ocenę udało mi się nieco poprawić jego działanie.

Poniższe wnioski zostały przedstawione na podstawie grafu: games120.col

Poniższy wykres przedstawia większenie szansy osobnika na mutację:



Jak widać – nie przyniosło to żadnych efektów. Zapewne jest to związane z tym, że w przypadku mojej implementacji mutacja jedynie zwiększa lub zmniejsza wartość osobnika przed jej oceną

Natomiast stopniowe zwiększanie szansy i ilości osobników na bycie wytypowanym do dołożenia do końcowego grafu przyniosło zdecydowany efekt. Poniżej wykresy prezentujące zwiększenie szansy i ilości osobników które trafią do końcowego grafu. Algorytm ten nadal odbiegał już od optymalnego wyniku – jednak znacząco się do niego zbliżał

