AGH
Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Fizyki i Informatyki Stosowanej

Praca magisterska

Yevhenii Natalenko

kierunek studiów: **Informatyka stosowana**

specjalność: **Przetwarzanie grafiki i analiza obrazów**

Wizualizacja grafów dynamicznych

Opiekun: **dr hab. inż. Małgorzata Krawczyk**

Kraków, listopad 2020

Oświadczenie studenta

Uprzedzony(-a) o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2018 r. poz. 1191 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony(-a) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.) „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelni przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. — Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu antyplagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

.....

(czytelny podpis)

Kraków, wrzesień 2020

Tematyka pracy magisterskiej i praktyki dyplomowej Yevhenia Natalenko, studenta drugiego roku studiów drugiego stopnia na kierunku Informatyka stosowana, specjalność grafika komputerowa i przetwarzanie obrazów

Temat pracy magisterskiej: **Wizualizacja i analiza grafów**

Opiekun pracy:	dr hab. Małgorzata Krawczyk
Recenzenci pracy:	
Miejsce praktyki dyplomowej:	WFiIS AGH, Kraków

Program pracy magisterskiej i praktyki dyplomowej

1. Omówienie realizacji pracy magisterskiej z opiekunem. Określenie wymaganych funkcjonalności.
2. Praktyka dyplomowa:
   - Analiza i dobór narzędzi(języków programowania, bibliotek, frameworków) umożliwiających implementację programu.
   - Przegląd istniejącego oprogramowania o zbliżonych możliwościach.
   - Konfiguracja środowiska programistycznego
   - Implementacja próbnej wersji programu oraz zapoznanie się z wybranymi narzędziami.
   - Planowanie dalszych działań
   - Sporządzenie sprawozdania z praktyki.
3. Rozwój powstałego wcześniej prototypu.
4. Ciągłe dodawanie kolejnych funkcjonalności.
5. Optymalizacja i roszerzenie architektury programu.
6. Testowanie oraz finalne modyfikacje.
7. Opracowanie redakcyjne pracy.

Termin oddania w dziekanacie:

....

(podpis kierownika katedry) (podpis opiekuna)

Merytoryczna ocena pracy przez opiekuna:

Merytoryczna ocena pracy przez recenzenta:

Serdeczne podziękowania kieruję do mojego promotora
dr hab. Małgorzaty Krawczyk,
której życzliwość, cenne uwagi i wyjątkowa cierpliwość
stały się kluczowym elementem podczas powstawania niniejszej pracy.

Spis treści

1 Wstęp

Graf jako abstrakcyjny model matematyczny można uznać za kwintesencję ludzkiego dążenia do systematyzacji i analizy otaczającego nas świata. Tworzenie, opisywanie i analiza układów złożonych, składających się z wielu oddziaływających elementów, rzadko może się obyć bez odwoływania się w jakikolwiek sposób do teorii grafów. W dużym stopniu do rozpowszechnienia jej zastosowania przyczynił się rozwój technologiczny. Jeszcze 30 lat temu praktyczne stosowanie grafów było mocno ograniczone. Dziś natomiast, codziennie mamy do czynienia z technologią, bazującą na tym właśnie modelu matematycznym. Co pociąga za sobą konieczność rozwoju metod i narzędzi do ich analizy. Wraz ze wzrostem złożoności istniejących układów, rośnie zapotrzebowanie na przedstawienie informacji w bardziej przejrzysty i zrozumiały człowiekowi sposób. Celem niniejszej pracy jest usprawnienie właśnie tego aspektu pracy z grafami. Interaktywna wizualizacja oraz analiza grafu są tematami kluczowymi, wokół których powstawało opracowane przez Autora narzędzie.

2 Wykorzystane technologie

2.1 Framework Qt

Będąc frameworkiem w szerokim tego słowa znaczeniu, Qt posiada w swoim arsenale szeroki zestaw funkcjonalności programistycznych, co pozwala ujednolicić i uprościć proces wytwarzania oprogramowania. Podchodząc do wyboru narzędzia dla języka C++ i uwzględniając wszystkie wymagania stawiane wobec aplikacji, Qt okazał się poza konkurencją. Warto wymienić szczególnie przydatne w ramach projektu moduły oraz możliwości tego frameworku, są to:

Wieloplatformowość Qt pozwala z minimalnym wysiłkiem w postaci ponownej kompilacji, używać napisanego kodu na dowolnej wspieranej przez niego platformie: od systemów wbudowanych aż do urządzeń mobilnych. Dlatego niniejszy projekt może być łatwo przeniesiony na dowolną platformę desktopową. Wystarczy skompilować kod dla platformy docelowej, a implementacja Qt zadba o poprawne działanie programu. Przykładowo, komunikacja w projekcie odbywająca z użyciem *QLocalSocket*. Na maszynach działających na systemie operacyjnym Windows klasa ta jest zaimplementowana, używając tzw. **kanałów nazwanych** (ang. named pipe). W systemach rodziny Unix z kolei, ta sama klasa korzysta z **lokalnych soketów** (ang. local domain socket) [1]. Warto jednak pamiętać, że próbując używać systemowych implementacji elementów interfejsu, wygląd aplikacji może się różnić w zależności od systemu docelowego. Szczególnie ważny jest ten aspekt przy wdrażaniu aplikacji na urządzeniach mobilnych.

Qt Designer Rozbudowane narzędzie do konstruowania interfejsu, które znacząco uprościło i przyśpieszyło projektowanie wyglądu oraz logiki interakcji użytkownika z programem. Qt Designer pozwala bezpośrednio łączyć elementy z odpowiednimi slotami[1] oraz dokonywać ich ogólnej konfiguracji.

Graphics View Framework Graphics View udostępnia dwuwymiarową płaszczyznę do obsługi i interakcji z dużą liczbą obiektów graficznych, a także widok do ich wizualizacji. Framework działa w oparciu o wzorzec model-widok. Kilka widoków mogą obserwować i wyświetlać jedną przestrzeń, odpowiedzialną za logikę zachowania obiektów. Dokładniej o tym w sekcji 4.5.

[1] Signały oraz sloty stanowią mechanizm obsługi zdarzeń w ramach Qt.

2.2 Boost Graph Library

BGL pochodzi z rodziny bibliotek Boost i jest uogólniona (generyczna) w znaczeniu analogicznym do generyczności STL[2] Oznacza to uogólnienie w kilku podstawowych obszarach. Jeden z nich to algorytmy uniezależnione od danych [2]. Oznacza to oddzielenie struktury grafu od funkcji, pozwalając uruchamiać niezbędne funkcje dla szerokiego zakresu struktur reprezentujących graf. Inny obszar generyczności polega na możliwości kojarzenia wielu atrybutów z wierzchołkami bądź krawędziami. Jest to podobne do parametryzacji kontenerów w STL, jednak bardziej skomplikowane w przypadku grafów. Warto podkreślić słowo *wielu* w ramach omawianej cechy. Każda właściwość (ang. property) jest z kolei parametryzowana jej typem oraz identyfikatorem. Możliwość dokonania takiej parametryzacji atrybutów wyróżnia BGL na tle pozostałych bibliotek, pozwalając definiować gotowe do użycia struktury reprezentujące graf przy analizie innych problemów, niż ten, dla którego początkowo zostały stworzone. Subiektywną wadą biblioteki można nazwać nie wyczerpującą dokumentację jak dla takiego rozbudowanego narzędzia. Małą społeczność korzystająca z BGL również utrudnia poszukiwanie rozwiązań nietypowych problemów, wymuszając analizę kodu źródłowego biblioteki, opartej o mechanizm szablonów języka $C++$.

2.3 Uzyskanie danych

Kluczowym etapem pracy była prezentacja poprawnego działania algorytmów pozycjonowania. Niezbędny więc jest zbiór grafów o różnych właściwościach oraz rozmiarach. Bazą do uzyskania grafów o dużych rozmiarach oraz klastrowej strukturze służyły odniesienia artykułów polskojęzycznej Wikipedii. Dane pochodzą z serwisu *DBpedia* [3]. Rozmiar pliku surowego wynosił powyżej 2GB, a liczba węzłów dorównywała odpowiednio liczbie artykułów w języku polskim (powyżej 1,4mln). W celu przetwarzania tego pliku został napisany osobny program konwertujący, umożliwiający podanie maksymalnego rozmiaru grafu, który użytkownik ma zamiar rysować. Niezbędna również jest funkcja parsowania w celu uzgodnienia danych z formatem *GraphMl*[4].

Grafy o regularnej strukturze zostały wygenerowane za pomocą osobnych programów pomocniczych lub stworzone ręcznie, jeśli rozmiar grafu na to pozwalał.

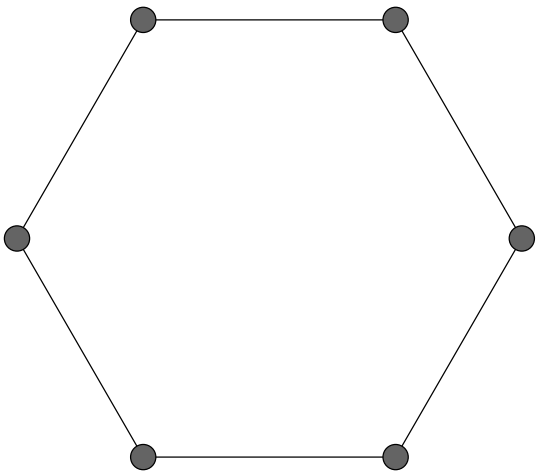
[2]STL(Standard Template Library) - biblioteka języka C++ zawierająca gotowe do użycia szablony różnego rodzaju kontenerów, funkcji lub innych obiektów.

3 Pozycjonowanie wierzchołków grafu

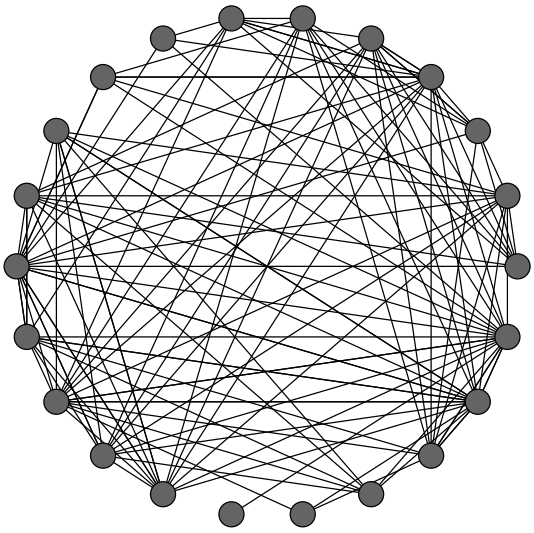
Rysowanie grafów jest zagadnieniem mającym na celu przedstawienie grafu w postaci wizualnej, nadającej się do wzrokowej percepcji przez człowieka. Dlatego wizualizacja powinna odpowiadać pewnym wymaganiom estetycznym, które, będąc charakterystyką postrzeganą wyłącznie subiektywnie, jest trudna do oszacowania. Ponadto, istnieją sytuacje, przy których jedna wizualizacja grafu będzie odbierana jako estetyczna i informatywna, a w innych okaże się bezużyteczna.

3.1 Metoda radialna

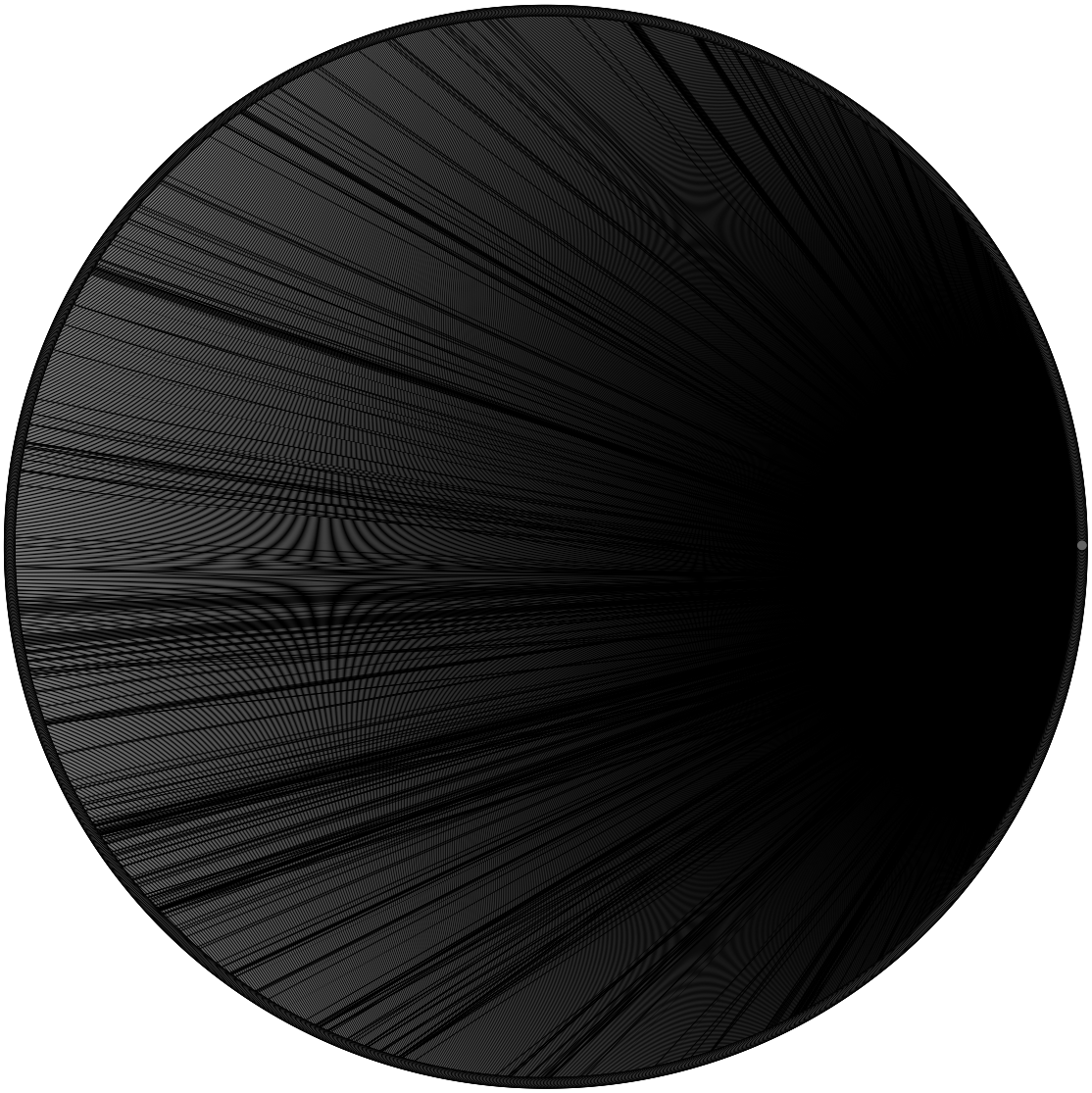
Radialne pozycjonowanie cechuje się ułożeniem wierzchołków w taki sposób, że one rozmieszczone zostają na okręgu. Metoda ta dobrze sprawdza się przy wizualizacji sieci o topologii pierścienia lub gwiazdy[5]. Innym przykładem użycia mogą być sieci społecznościowe lub bioinformatyka, dla których metoda radialna cechuje się neutralnością przy postrzeganiu grafu przez człowieka. Neutralność w tym kontekście oznacza, iż przy równoodległym położeniu wszystkich wierzchołków, żaden z węzłów nie ma "uprzywilejowanej" pozycji. Jest to ważne, ponieważ człowiek ma skłonność do postrzegania wierzchołków znajdujących się bliżej środka jako ważniejszych[6].



Rysunek 1: Wynik pozycjonowania radialnego: sześciokąt



Rysunek 2: Wynik pozycjonowania radialnego: 22 węzły



Rysunek 3: *Wygenerowane przez zaimplementowany wizualizator. Liczba węzłów: 3000+*

3.2 Algorytmy oparte o oddziaływania fizyczne

3.2.1 Algorytm Kamada-Kawai

Algorytm do rysowania grafów skierowanych oraz ważonych został przedstawiony przez T. Kamada oraz S. Kawai w 1989 roku (dalej **KK**). Autorzy zaproponowali w nim odmienne od często spotykanego spojrzenia na cały problem wizualizacji poprzez rezygnację z konieczności unikania krzyżujących się krawędzi. Będąc modyfikacją algorytmu Eadesa, KK naśladują model sprężynowy. Jednak zamiast dwóch różnych sił (odpychającej oraz przyciągającej) między poszczególnymi parami wierzchołków, wprowadzony zostaje układ sprężyn. Przy czym minimalizacja energii układu prowadzi do optymalnego pozycjonowania węzłów.

Ważnym założeniem algorytmu KK jest pojęcie *teoretycznej odległości grafowej* (ang. graph theoretic distance), która jest odpowiednikiem odległości geometrycznej (Euklidesowej) między dwoma węzłami narysowanego grafu[7]. Układ sprężynowy zaś jest tworzony w taki sposób, iż każda sprężyna ma długość początkową równą teoretycznej odległości grafowej. Mając układ n wzajemnie połączonych węzłów, można oznaczyć zbiór punktów na powierzchni dwuwymiarowej: $p_1, p_2, ..., p_n$, odpowiadający węzłom $v_1, v_2, ..., v_n$. Miara braku równowagi może być wyrażona jako energia układu:

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{i,j} \left(|p_i - p_j| - l_{i,j} \right)^2, \tag{1}$$

Gdzie l_{ij} jest początkową długością sprężyny między p_i a p_j i odpowiada ona pożądanej odległości pomiędzy tymi węzłami. Wartość $k_{i,j}$ jest współczynnikiem sprężystości między p_i a p_j. Całe wyrażenie (1) jest więc sumą kwadratów różnic między pożądanymi oraz aktualnymi odległościami dla wszystkich par wierzchołków. Wartości $l_{i,j}$ oraz $k_{i,j}$ są wyznaczane następująco:

$$l_{i,j} = L \times d_{i,j}, \tag{2}$$

$$L = \frac{L_0}{\max_{i<j} d_{i,j}}, \tag{3}$$

$$k_{i,j} = \frac{K}{d_{i,j}^2}. \tag{4}$$

W powyższych wyrażeniach L_0 symbolizuje wymiar przestrzeni, na której będzie odbywało się rysowanie. Wartość $\max_{i<j} d_{i,j}$, tzw. średnica grafu jest odległością między najbardziej oddalonymi od siebie wierzchołkami.

Współrzędne punktów w dwuwymiarowej przestrzeni Euklidesowej są podane jako pary x_i, y_i, co pozwala przepisać wyrażenie (1) następująco:

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{i,j} \left((x_i - x_j)^2 + (y_i - y_j)^2 + l_{i,j}^2 - 2 l_{i,j} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \right). \tag{5}$$

Poszukiwanie minimum globalnego funkcji energii $E(x_1, ...x_n, y_1, ...y_n)$ jest trudne. Dlatego obliczane jest minimum lokalne, którego warunkiem są zerowe pochodne cząstkowe dla każdej współrzędnej:

$$\frac{\partial E}{\partial x_m} = \frac{\partial E}{\partial y_m} = 0 \quad dla\ 1 \ \leq \ m \ \leq n \tag{6}$$

.

Poszukiwanie takiego minimum wprost wymagałoby rozwiązywania układu $2n$ równań nieliniowych, Dlatego KK stosują inne podejście. Jednocześnie położenie stabilne jest obliczane tylko dla jednego wybranego punktu p_m, zamrażając pozostałe węzły[7]. Innymi słowy, funkcja E jest postrzegana jako $E(x_m, y_m)$. Punkt p_m zostaje wybrany na podstawie największej wartości Δ_m:

$$\Delta_m = \sqrt{\left(\frac{\partial E}{\partial x_m}\right)^2 + \left(\frac{\partial E}{\partial y_m}\right)^2}. \tag{7}$$

Algorytm Pierwszym krokiem algorytmu jest obliczanie $d_{i,j}$ dla każdej pary wierzchołków w grafie. Autorzy, przedstawiając swój algorytm, wyznaczają najkrótsze ścieżki metodą Floyda[7]. Następnie są wyznaczane $l_{i,j}$ oraz $k_{i,j}$ według wzorów (2) i (4). Algorytm wymaga wcześniejszej inicjalizacji położeń wierzchołków. W niniejszej pracy do inicjalizacji wykorzystane zostało ułożenie radialne. W kolejnych iteracjach energia E zmniejsza się, prowadząc układ do równowagi. Została użyta implementacja biblioteki BGL.

Algorytm 1 Pseudokod algorytmu Kamada Kawai

```
1: obliczanie d_{i,j} dla 1 ≤ i ≠ j ≤ n;
2: obliczanie l_{i,j} dla 1 ≤ i ≠ j ≤ n;
3: obliczanie k_{i,j} dla 1 ≤ i ≠ j ≤ n;
4: inicjalizacja p_1, ...p_n;
5: while(max_i Δ_i > ε):
6:     poszukiwanie p_m o największej wartości Δ_i;
7:     while(Δ_m > ε):
8:         obliczanie ∂x oraz ∂y;
9:         x_m+ = ∂x;
10:        y_m+ = ∂y;
```

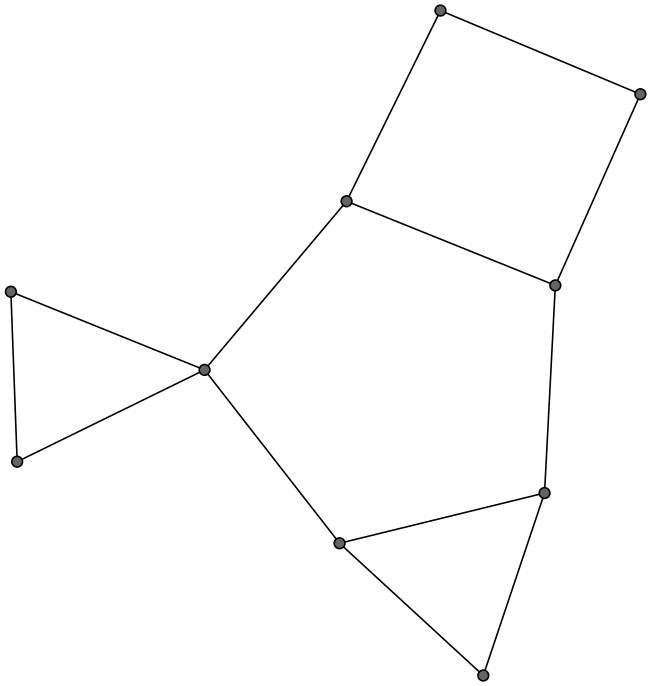
Złożoność obliczeniowa Algorytmu KK nie można nazwać obliczeniowo oszczędnościowym, ponieważ poza zagnieżdżonymi pętlami, wymagany jest dodatkowy narzut w postaci obliczania najkrótszych ścieżek dla każdej pary węzłów. Używając sugerowanego przez autorów algorytmu Floyda-Warshalla, narzut ten wynosi $O(n^3)$ [8]. Przy obliczaniu najkrótszych ścieżek mogą być użyte bardziej wydajne algorytmy. Przykładem może być algorytm Johnsona o złożoności $O(|V|^2 log|V| + |E||V|)$ [8]. Metoda Newtona-Raphsona do rozwiązywania układu równań daje narzut $O(n)$[7]. Zewnętrzna pętla wymaga analogicznego narzutu z uwagi na obliczanie największej Δ_i. Złożoność obu pętli jest trudna do określenia z powodu różnorodności danych wejściowych i wynosi $O(Tn)$, gdzie T jest liczbą wykonań pętli wewnętrznej. Warto wspomnieć o wymaganiach pamięciowych. Algorytm przechowuje odległości międzywęzłowe, co wymaga $O(n^2)$ zasobów pamięciowych.

Listing 1 przedstawia użycie zaimplementowanego w ramach BGL algorytmu KK[9]. Poszczególne przekazywane parametry to:

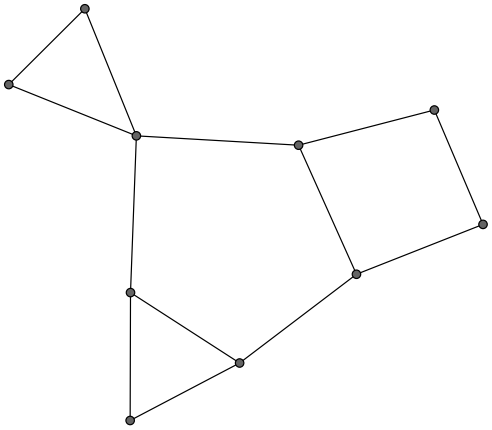
- `g` – obiekt grafu do pozycjonowania,
- `node_position_map` – struktura danych mapująca wierzchołki na ich położenie w ramach topologii. Finalne położenia wierzchołków są przechowywane w tym parametrze.
- `weight_property_map` – mapa wag krawędzi grafu. W programie wszystkie wagi są równe 1.
- `topology` – topologia przestrzeni (o czym mowa w sekcji 4.2),
- `boost::side_length<double>(dimension)` – wymiar planszy na której odbywa się pozycjonowanie.

```
1  void GraphClass::kamada_kawai_layout(double dimension, double strength) {
2    PositionMap node_position_map = boost::get(&VertexProperties::position, g);
3    WeightPropertyMap weight_property_map = boost::get(&EdgeProperties::weight, g);
4    boost::random::minstd_rand gen;
5    boost::square_topology<> topology(gen, dimension);
6
7    boost::circle_graph_layout(g, node_position_map, dimension / 2);
8    bool KKLayoutSuccess = boost::kamada_kawai_spring_layout(
9                g,
10               node_position_map,
11               weight_property_map,
12               topology,
13               boost::side_length<double>(dimension)
14   );
15
16   if (!KKLayoutSuccess) {
17       qDebug() << "KK layout could not be performed.";
18       return;
19   }
20
21 // Osobna metoda z prostym zadaniem: informowanie klas rysujacych biblioteki Qt o nowych p
22   this->setNodesCoordinatesFromPositionMap(node_position_map);
23 }
```

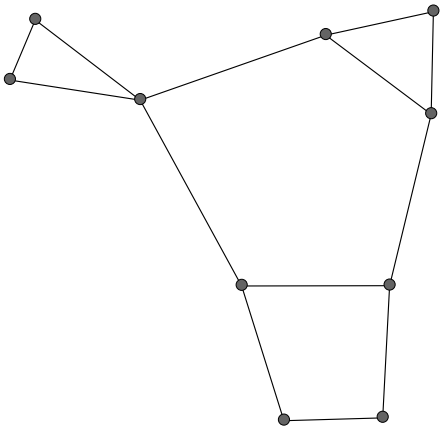
Listing 1: Użycie algorytmu KK z BGL



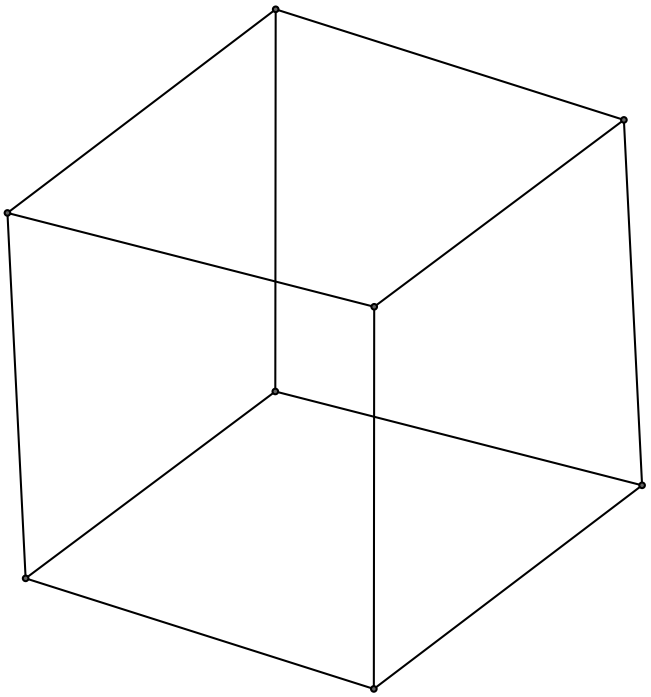
Rysunek 4: Wynik działania algorytmu KK



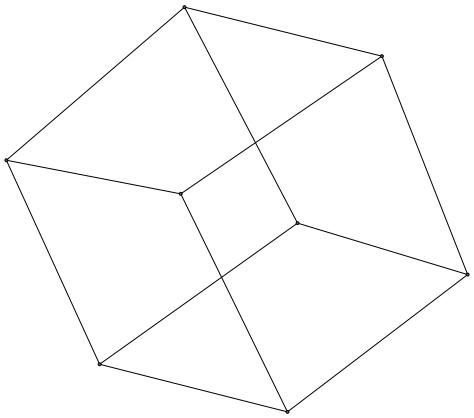
Rysunek 5: Wynik działania algorytmu FR



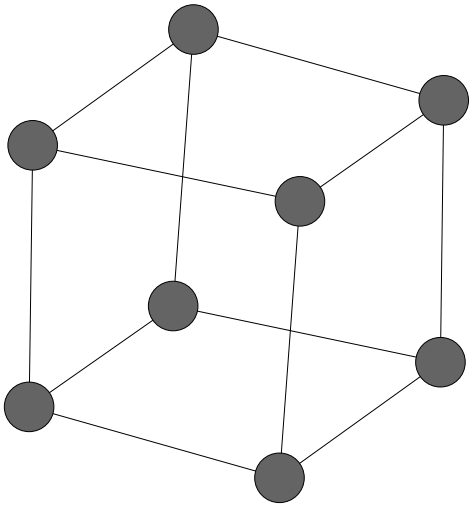
Rysunek 6: Wynik działania zaimplementowanego algorytmu



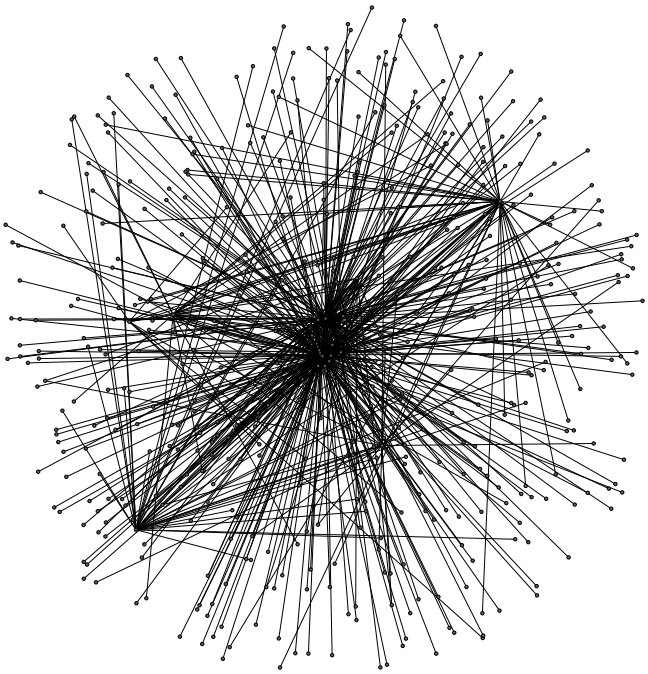
Rysunek 7: Wynik działania algorytmu KK



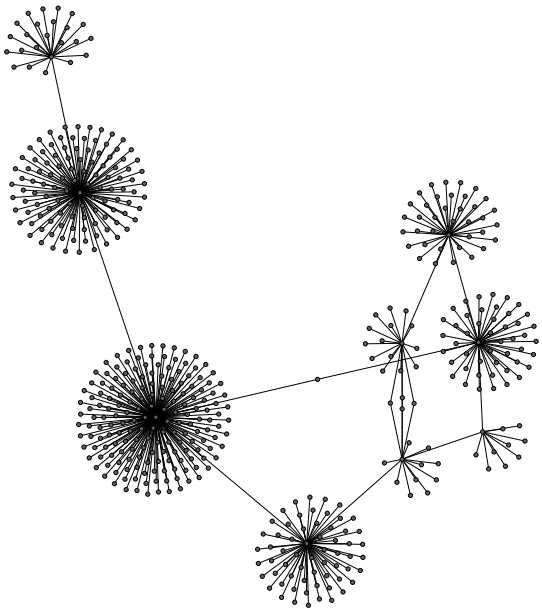
Rysunek 8: Wynik działania algorytmu FR



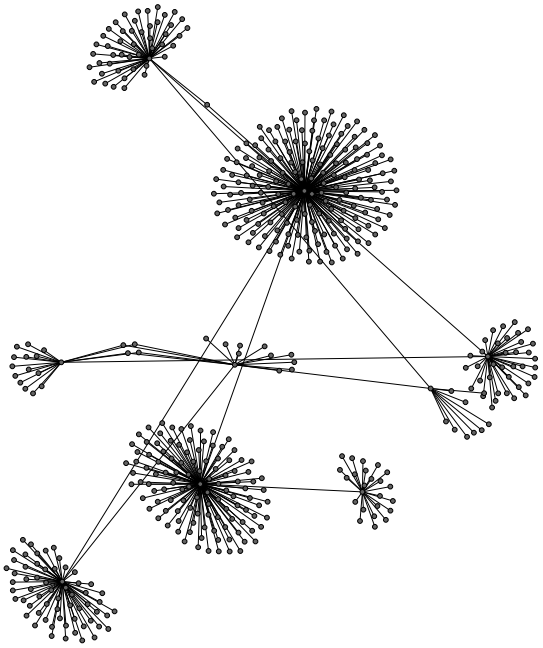
Rysunek 9: Wynik działania zaimplementowanego algorytmu



Rysunek 10: Wynik działania algorytmu KK



Rysunek 11: Wynik działania algorytmu FR



Rysunek 12: Wynik działania zaimplementowanego algorytmu

Rysunki 3.2.1-6 porównują działania wszystkich trzech zaimplementowanych algorytmów. Kolejne trójki przykładów znajdują się poniżej.

Podsumowanie Główną zaletą algorytmu KK jest intuicyjna definicja, czym jest "dobrze" narysowany graf, uzależniając odległości Euklidesowe od odległości między węzłami jako własności układu sprężyn. Jedną z wad jest możliwe utknięcie algorytmu w lokalnym minimum funkcji energii układu. Jedno z możliwych obejść tego problemu polega na dodatkowym uruchomieniu algorytmu Fruchtermana-Reingolda(dalej **FR**) na wyniku działania KK. Wysoka złożoność obliczeniowa oraz pamięciowa narzucają dodatkowe ograniczenia na wydajną wizualizację. Przykładowy zbiór danych połączeń artykułów polskiej Wikipedii ma około 1,4 mln węzłów. Pobrane dane odnośników zajmują powyżej 2GB przestrzeni dyskowej. Uniemożliwia to więc wizualizację "wprost" na większości komputerów personalnych. W zaawansowanych narzędziach do wizualizacji mogą być stosowane różne techniki optymalizacyjne, na przykład filtracja analizowanych wierzchołków, inaczej — zmniejszenie rozmiaru grafu, zachowując jego strukturę. W wyniku takiej operacji część informacji zostaje utracona. Dodatkową wadą KK jest brak obsługi grafów niespójnych. Do potrzeb wizualizacji graf oparty o dane z Wikipedii został sztucznie uspójniony. Po przeprowadzeniu licznych testów można stwierdzić, iż algorytm produkuje gorsze wyniki w przypadku grafów o nierównomiernej gęstości. Graf z Wikipedii demonstruje skutki braku sił odpychających. Algorytm, chociaż jest najszybszy z pośród innych używanych w programie, nie dostarcza satysfakcjonujących wyników dla grafów o klastrowej strukturze, które często występują w praktyce.

Problem Podczas konfiguracji algorytmu został zaobserwowany fakt tego, że zmiana parametrów algorytmu nie wpływa na generowane wyniki mimo tego, że debugger pokazywał poprawne przekazywanie parametru do środka biblioteki. Główna pętla algorytmu otrzymywała poprawne wartości, na przykład siły sprężyn, jednak nie miało to wpływu na końcowe ułożenie wierzchołków. Zmiany warunku zakończenia, topologii lub innych parametrów działały poprawnie. Przy niektórych znaczeniach parametru sił sprężyny, algorytm nigdy nie kończył swojego działania. Innym problemem było sporadyczne przepełnienie stosu w przypadku, kiedy węzły okazywały się zbyt blisko siebie. Została podjęta decyzja ograniczyć konfigurację algorytmu KK w celu zachowania stabilności kosztem gorszych wyników.

3.2.2 Algorytm Fruchtermana-Reingolda

Praca Fruchtermana i Reingolda została opublikowana w 1991 roku i rozwija ideę tego samego algorytmu, na którym bazowali KK. Podstawowy algorytm Eadesa traktował graf jako układ pierścieni połączonych sprężynami [10]. Autorzy zdecydowali się na wprowadzenie dodatkowego mechanizmu chłodzenia układu. Mechanizm ten jest podobny do symulowanego wyżarzania z wyjątkiem tego, iż wersja FR nie ma możliwości odrzucenia lokalnego minimum. Zawsze akceptuje tylko lepsze rozwiązania. Innymi słowy,

nie ma możliwości probabilistycznego "hill climbing". Jednak wyjście z lokalnego minimum jest możliwe w przypadku mijania jednego wierzchołka drugiego z wystarczającą prędkością. Warto podkreślić, iż przyjęte w algorytmie siły są "sztucznymi" nie tylko w kontekście ich obliczeń, lecz też zachowania. Obliczane siły definiują prędkość wierzchołków. W rzeczywistości siły indukują przyśpieszenie. Różnica ta jest ważna przy ustalaniu równowagi.

Ważnymi zasadami rysowania grafów są:

- Połączone wierzchołki przyciągają się nawzajem siłami skierowanymi wzdłuż krawędzi.
- Wierzchołki nie powinny być rysowane zbyt blisko siebie.
- Wierzchołki muszą być rozłożone możliwie równomiernie.
- Równe długości krawędzi.
- Narysowany graf powinien odzwierciedlać symetrię, jeśli dany graf ją wykazuje.

Algorytm wymaga wcześniejszej inicjalizacji położeń wierzchołków. Może temu służyć, Na przykład, zwykłe losowanie współrzędnych wierzchołków.

Algorytm zawiera trzy główne kroki dla każdej iteracji [10]:

1. Obliczanie sił przyciągających dla każdego wierzchołka.
2. Obliczanie sił odpychających.
3. Ograniczenie zmiany położeń węzłów na podstawie temperatury.

Punkt trzeci jest ważny z punktu widzenia stabilizacji układu. Chłodzenie układu powoduje ograniczenie możliwego ruchu węzłów. Coraz bardziej oddziaływające siły są potrzebne do znaczącej zmiany ich położeń.

Obliczanie sił z punktów 1-2 jest zależne od implementacji. Aplikacja używa wzoru na siłę przyciągającą proporcjonalną kwadratowej odległości. Siła odpychająca jest odwrotnie proporcjonalna odległości i jest skierowana w kierunku przeciwnym do $F_{attractive}$:

$$F_{attractive} = \frac{d^2}{k}; \tag{8}$$

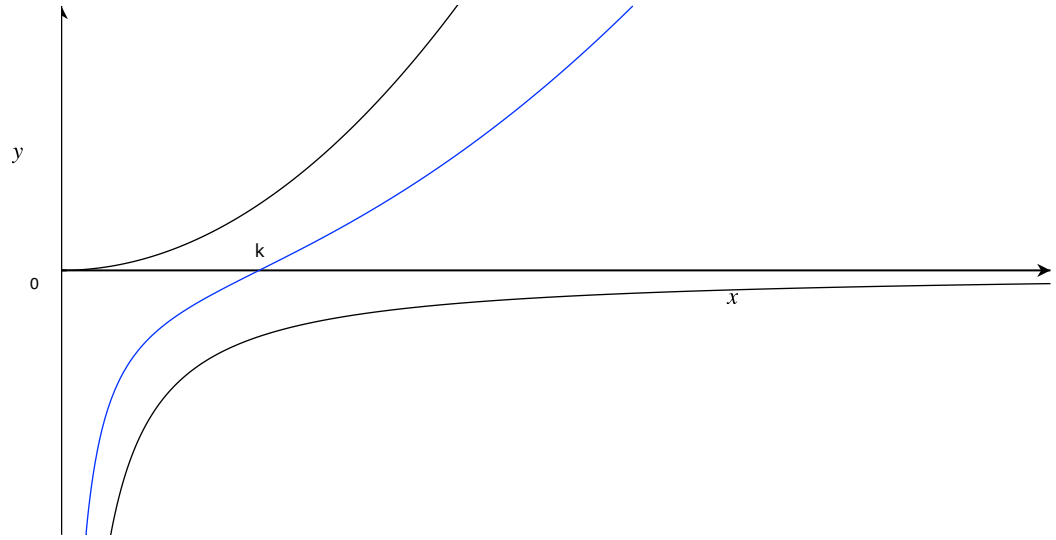
$$F_{repulsive} = -\frac{k^2}{d}; \tag{9}$$

Gdzie d jest aktualną odległością między dwoma węzłami. Wartość k jest odległością optymalną (docelową) i jest zdefiniowana jako:

$$k = C\sqrt{\left(\frac{area}{n}\right)}; \tag{10}$$

Gdzie n jest liczbą węzłów w grafie, a C stałą ustalaną eksperymentalnie.

Poniższy wykres 13 przedstawia wpływ sił odpychających (wykres dolny) oraz przyciągających (wykres górny) na osiąganie docelowej długości krawędzi k. Niebieski wykres demonstruje siłę wypadkową, która zeruje się w punkcie $(k, 0)$, czyli w stanie równowagi dla danej pary wierzchołków.



Rysunek 13: Osiąganie odległości docelowej k między węzłami

Poniżej został przedstawiony pseudokod algorytmu [10].

Algorytm 2 Pseudokod algorytmu Fruchtermana-Reingolda

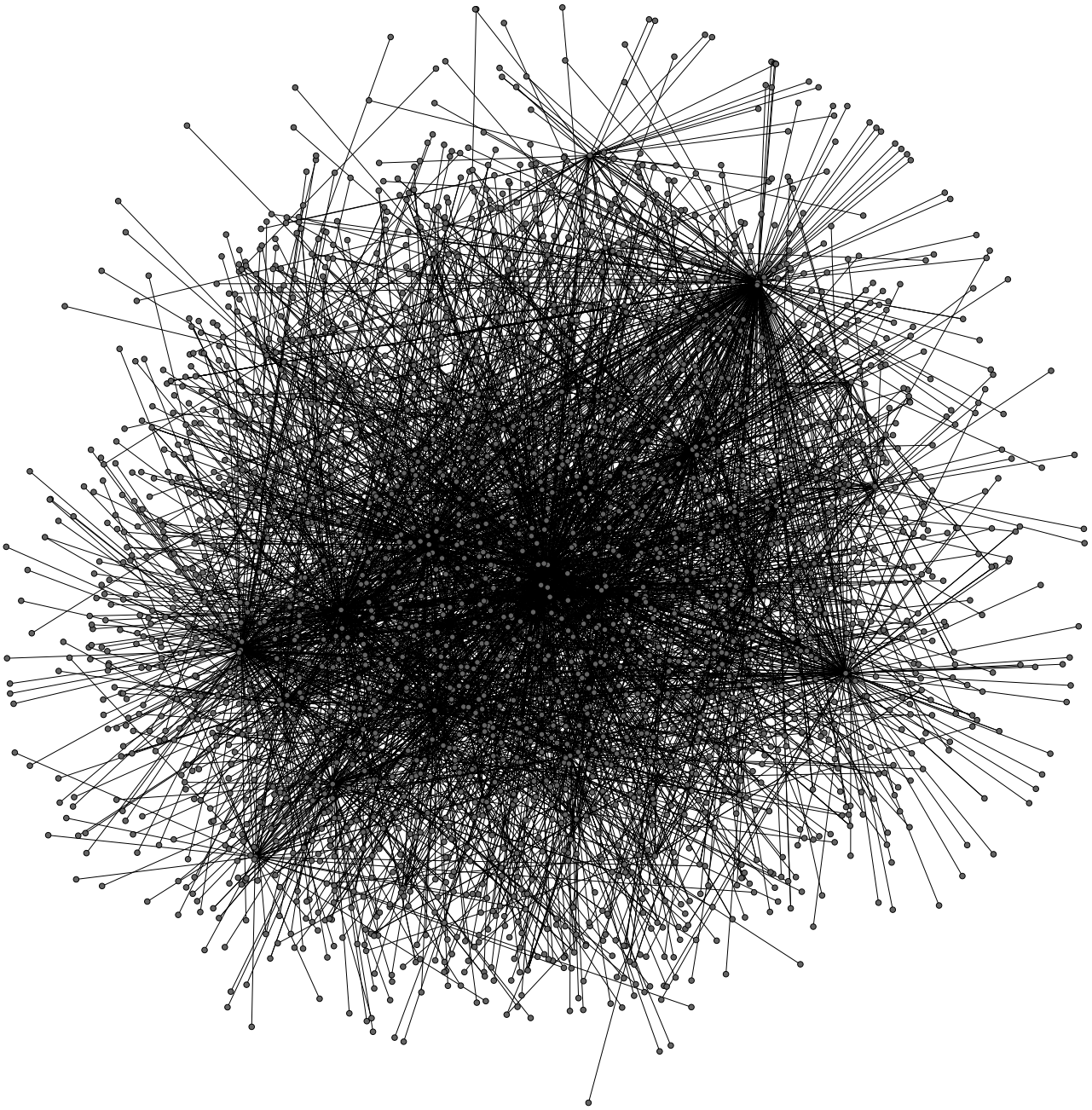
```
1: area := W * L;
2: G := (V, E);
3: k := C\sqrt{(area/n)};
4: function fa(x) := { return x^2/k } ;
5: function fr(x) := { return k^2/x} ;
6: for i := 1 to iterations do {
7:       // obliczanie sił odpychających
8:       for v in V do {
9:             // każdy wierzchołek ma dwa wektory: .pos i .disp
10:            v.disp := 0;
11:            for u in V do {
12:                  if (u ≠ v) then {
13:                        // Δ jest wektorem będącym różnicą położeń dwóch węzłów
14:                        Δ := v.pos − u.pos;
15:                        v.disp := v.disp + (Δ/|Δ|) * fr(|Δ|)
16:                  }
17:      }
18:      // obliczanie sił przyciągających
19:      for e in E do {
20:            // wierzchołki każdej krawędzi są oznaczone jako e.v oraz e.u
21:            Δ := e.v.pos − e.u.pos;
22:            e.v.disp := e.v.disp − (Δ/|Δ|) * fa(|Δ|);
23:            e.u.disp := e.u.disp + (Δ/|Δ|) * fa(|Δ|);
24:      }
25:      // ograniczenie możliwej zmiany położenia w zależności od temperatury układu.
   Zapobieganie wylatywania wierzchołków za granice obszaru rysowania.
26:      for v in V do {
27:            v.pos := v.pos + (v.disp/|v.disp|) * min(v.disp, t);
28:            v.pos.x := min(W/2, max(-W/2, v.pos.x));
29:            v.pos.y := min(L/2, max(-L/2, v.pos.y))
30:      }
31:      t := cool(t);
32: }
```

W powyższym pseudokodzie wartości W, L oznaczają odpowiednio szerokość oraz długość obszaru rysowania. Linijka 2 inicjalizuje początkowy stan grafu. Kolejne trzy pętli obliczają odpowiednio:

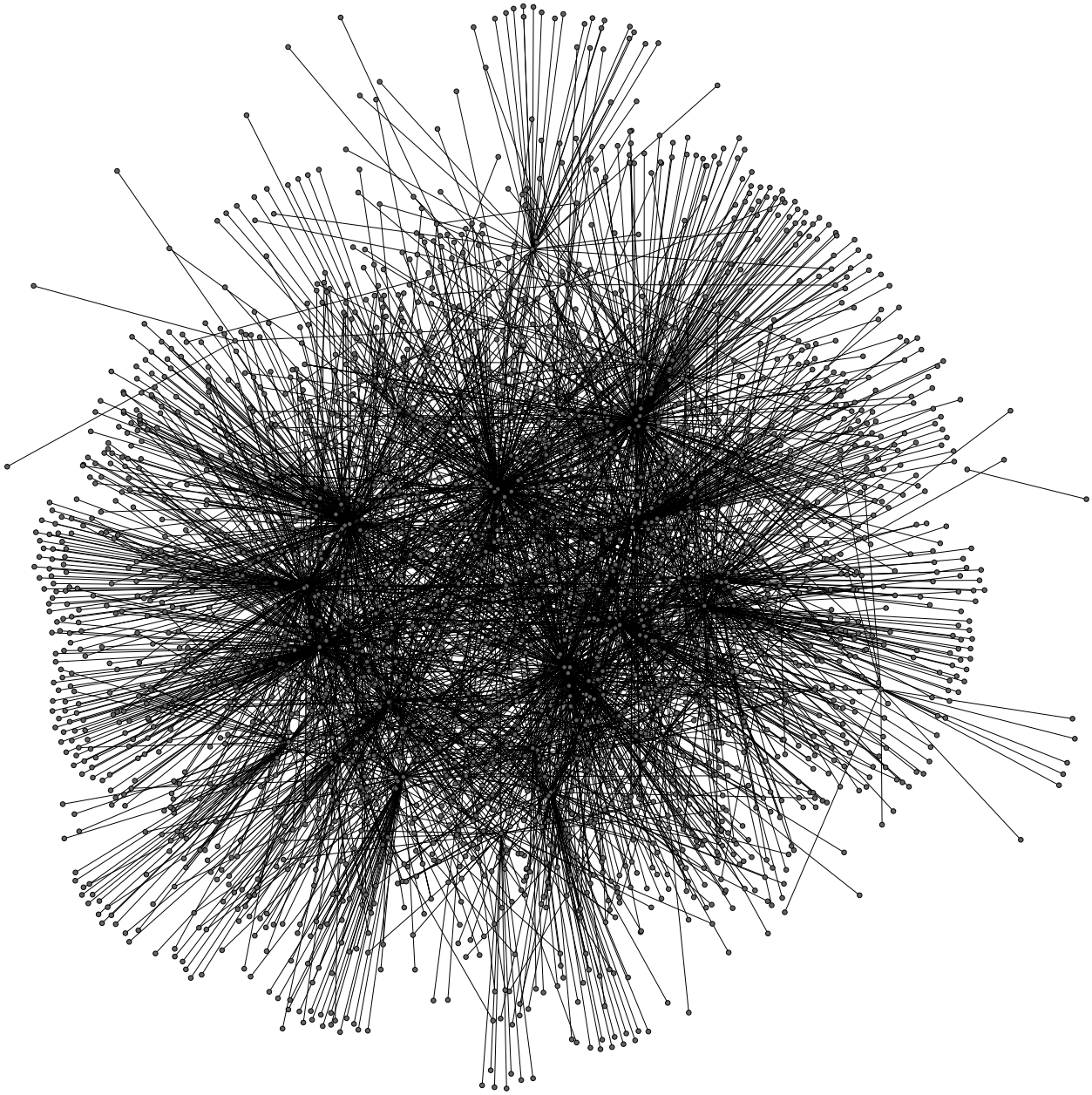
- siły odpychające;
- siły przyciągające;
- maksymalne możliwe przesunięcie uzależnione od temperatury układu;

Każda iteracja kończy się chłodzeniem. W aplikacji przyjęto liniowy charakter chłodzenia.

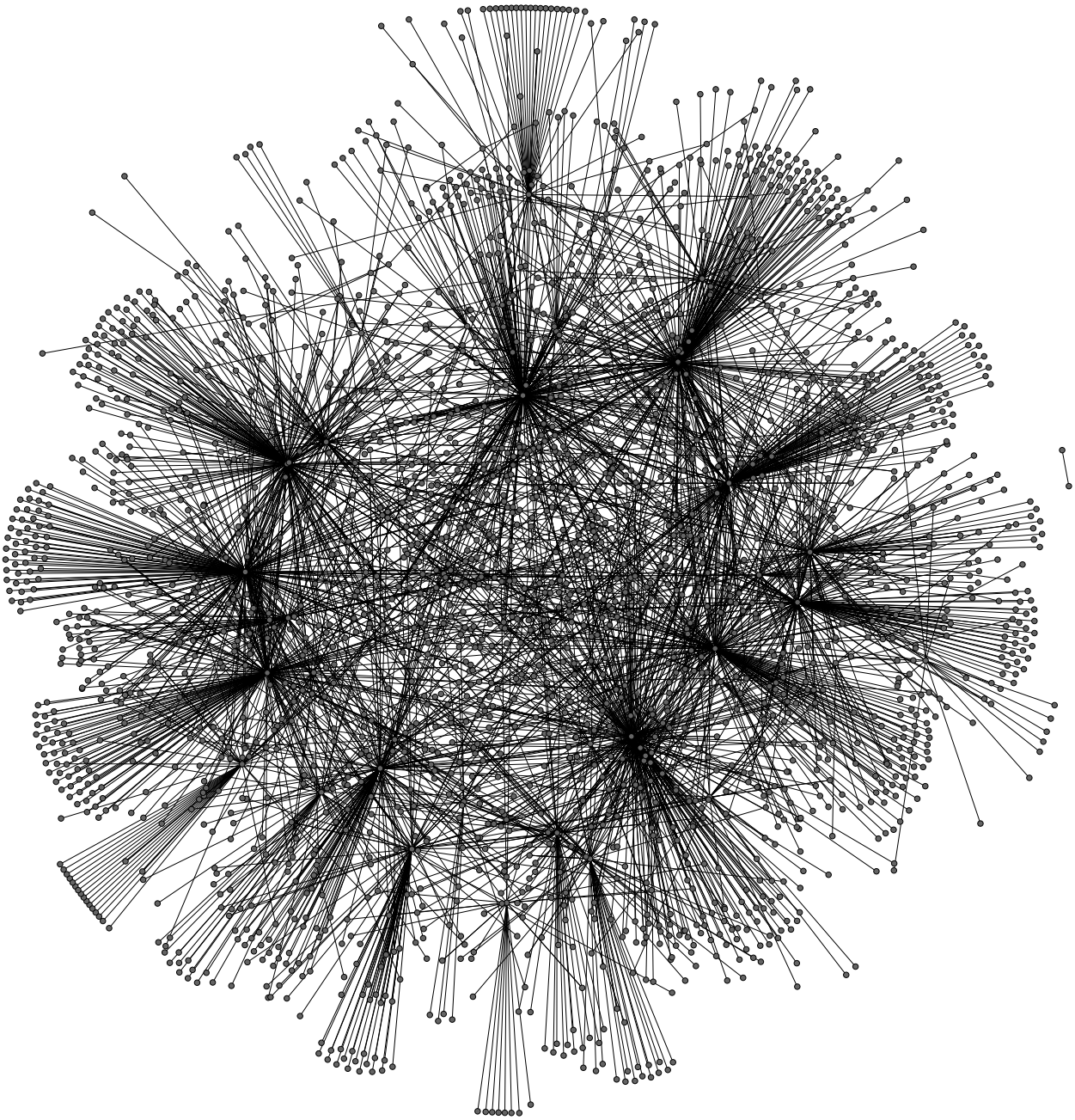
Kolejne strony zawierają wyniki działania algorytmu FR dla różnych temperatur początkowych. Można obserwować, jak wyższa temperatura, a więc wydłużony proces "chłodzenia" (cooling) powoduje coraz lepsze rozwinięcie grafu. Przykładowy graf składa się z 2400+ wierzchołków oraz 16800+ krawędzi. Została użyta implementacja biblioteki BGL.



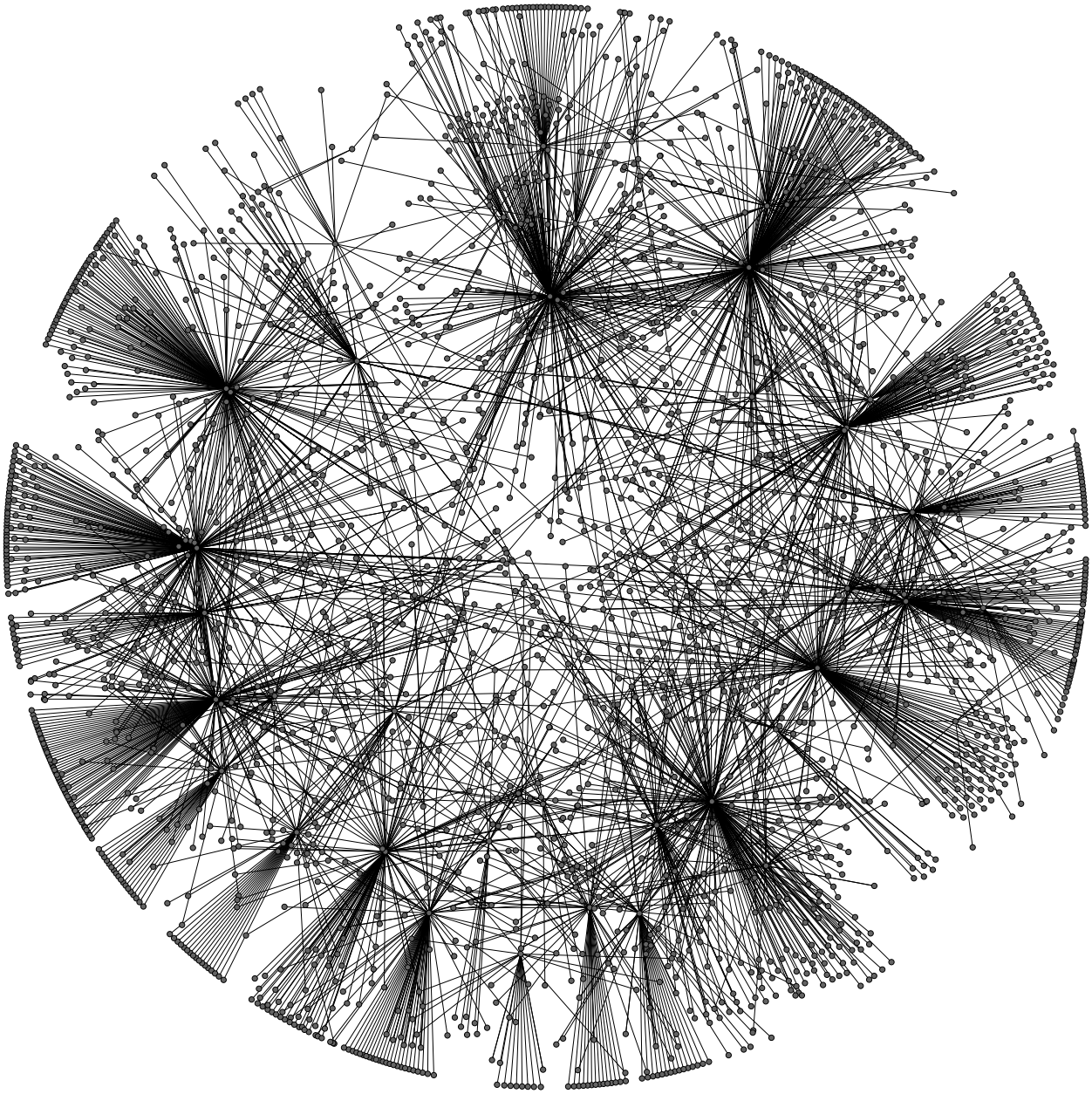
Rysunek 14: Temperatura początkowa: 100



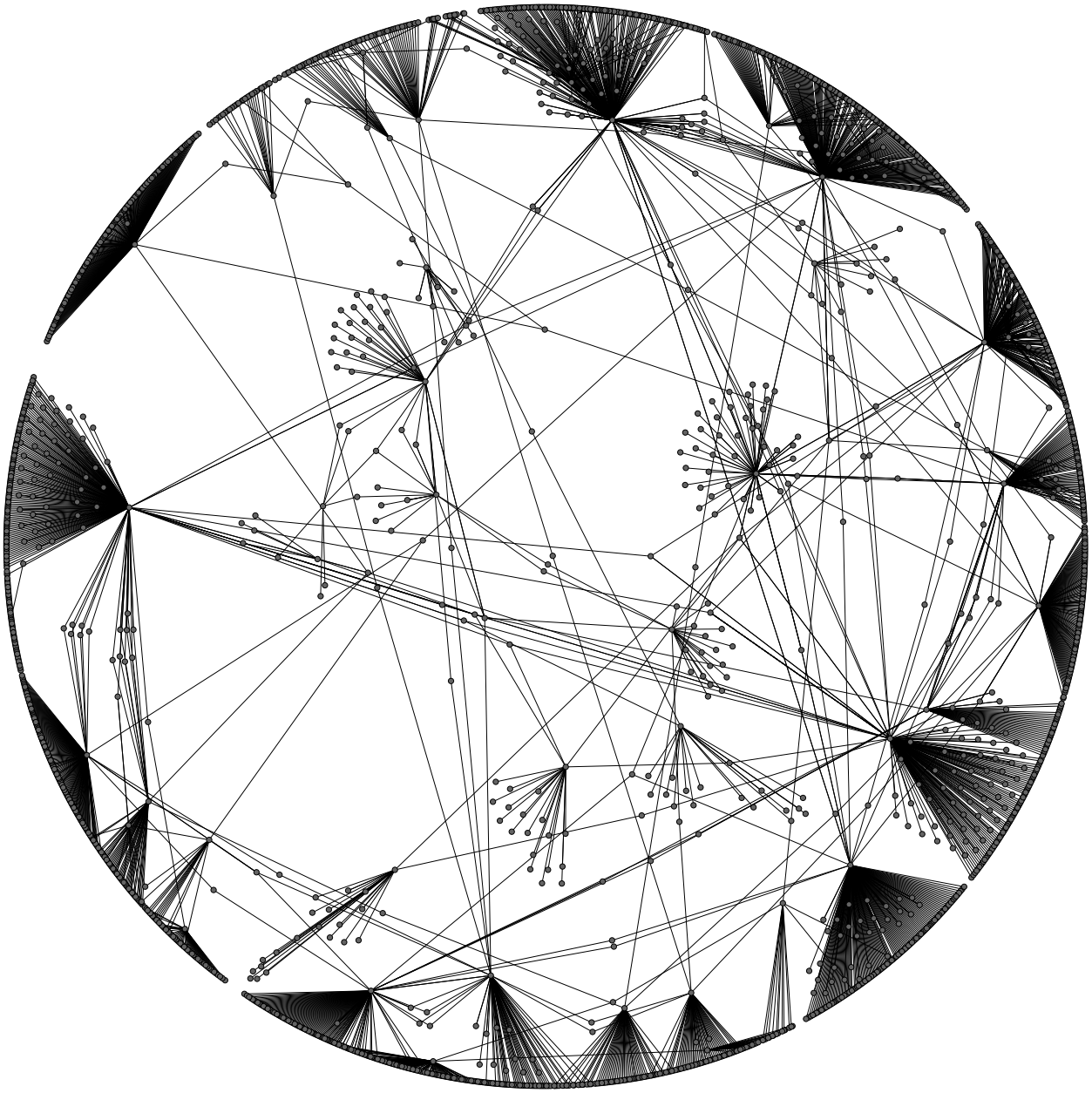
Rysunek 15: Temperatury początkowa: 150



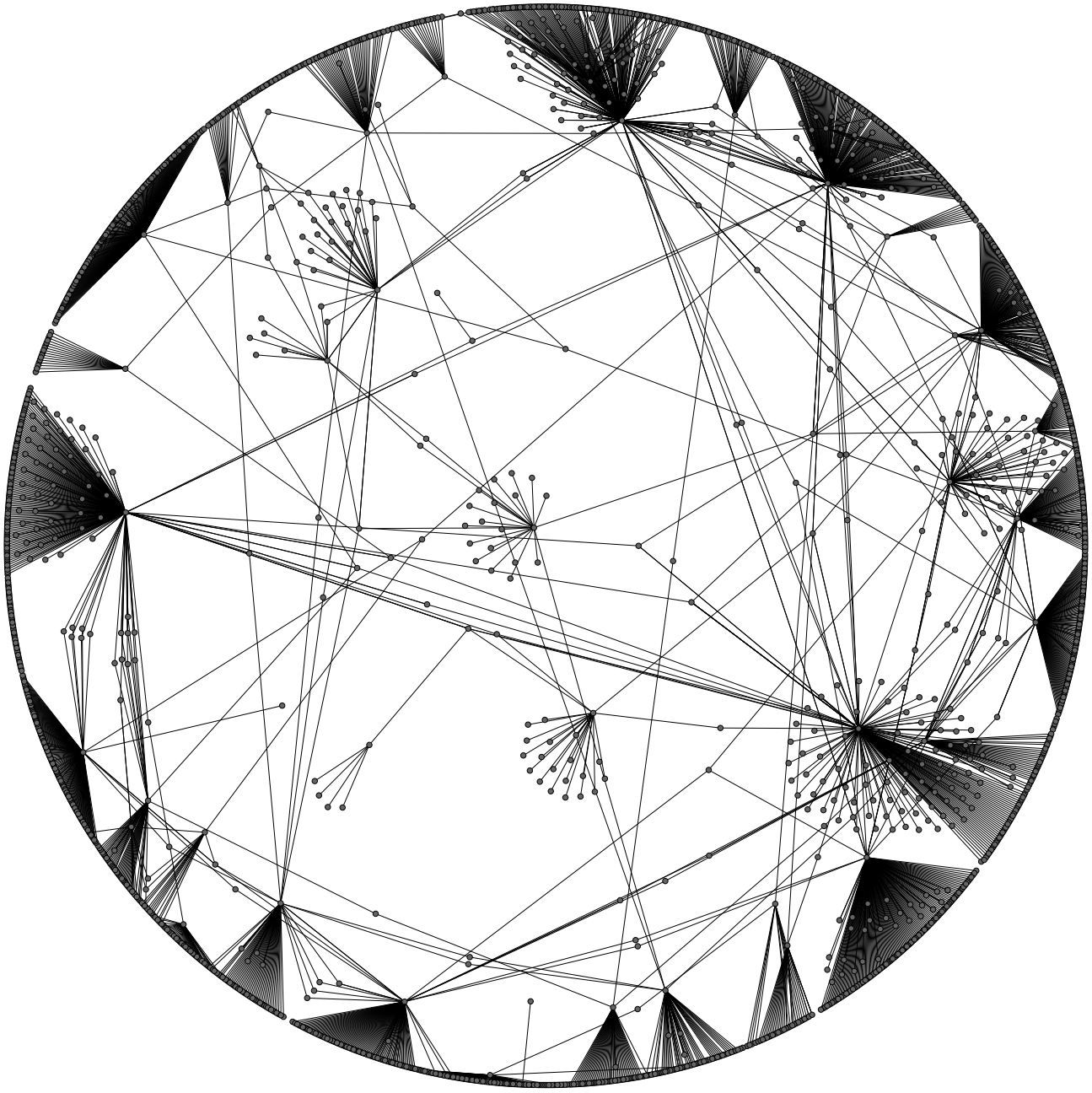
Rysunek 16: Temperatury początkowa: 175



Rysunek 17: Temperatury początkowa: 200

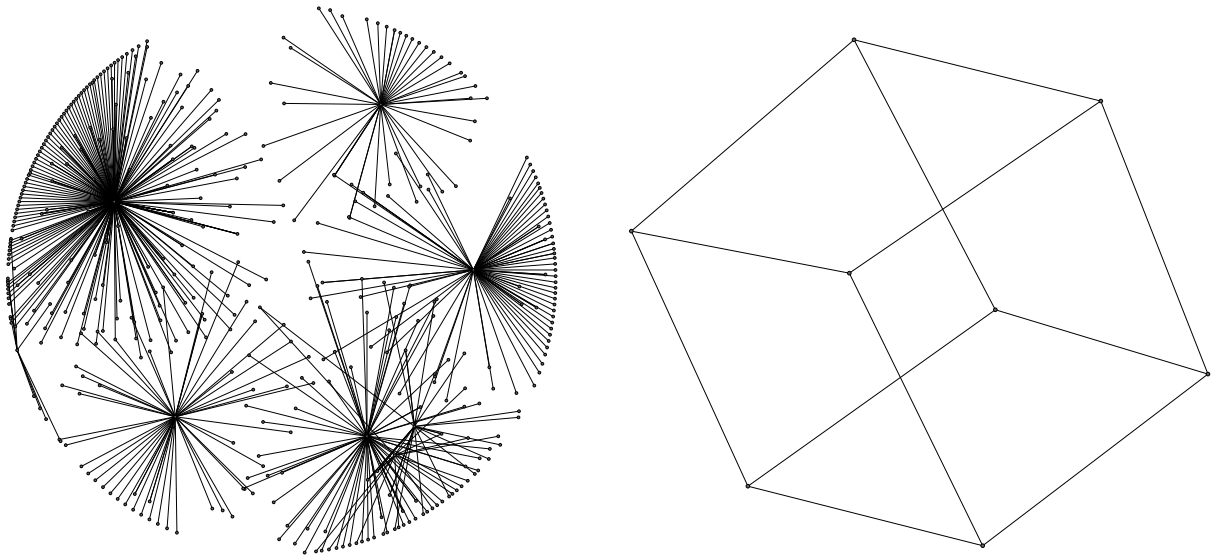


Rysunek 18: Temperatury początkowa: 300

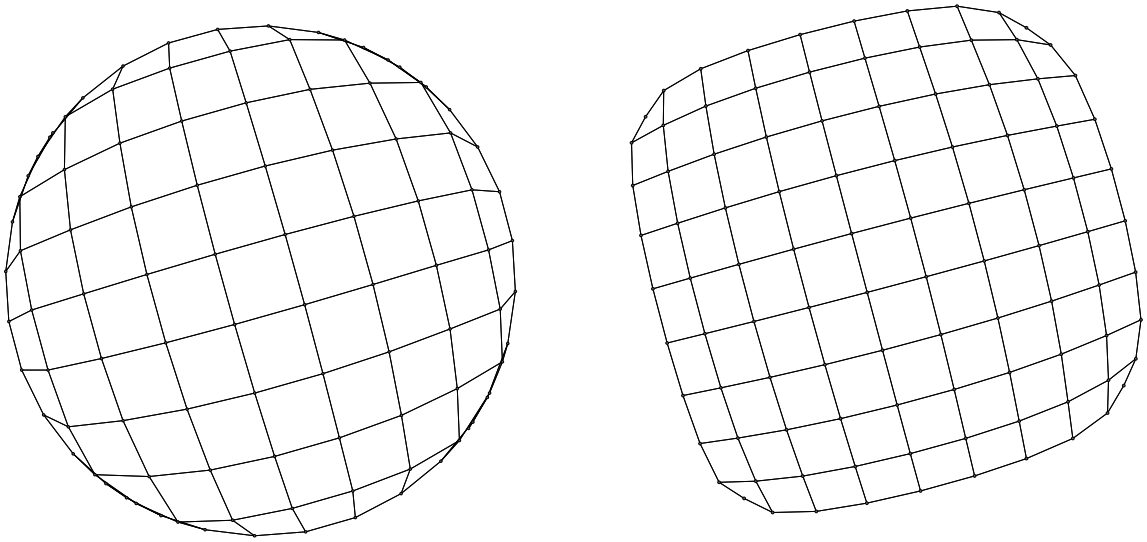


Rysunek 19: Temperatury początkowa: 500

Kolejne przykłady dotyczą mniejszych grafów. Rysunek 21 przedstawia wpływ sił odpychających na ułożenie grafu. Dobór tych sił jest kwestią indywidualną dla każdego grafu i zależy między innymi od jego gęstości, liczby wierzchołków oraz początkowej temperatury układu. Pierwsze grafy z rysunków 20, 21 oraz wszystkie wizualizacje dużego grafu (rysunki 14 - 19) demonstrują osiąganie algorytmem granic odgórnie założonego obszaru rysowania.

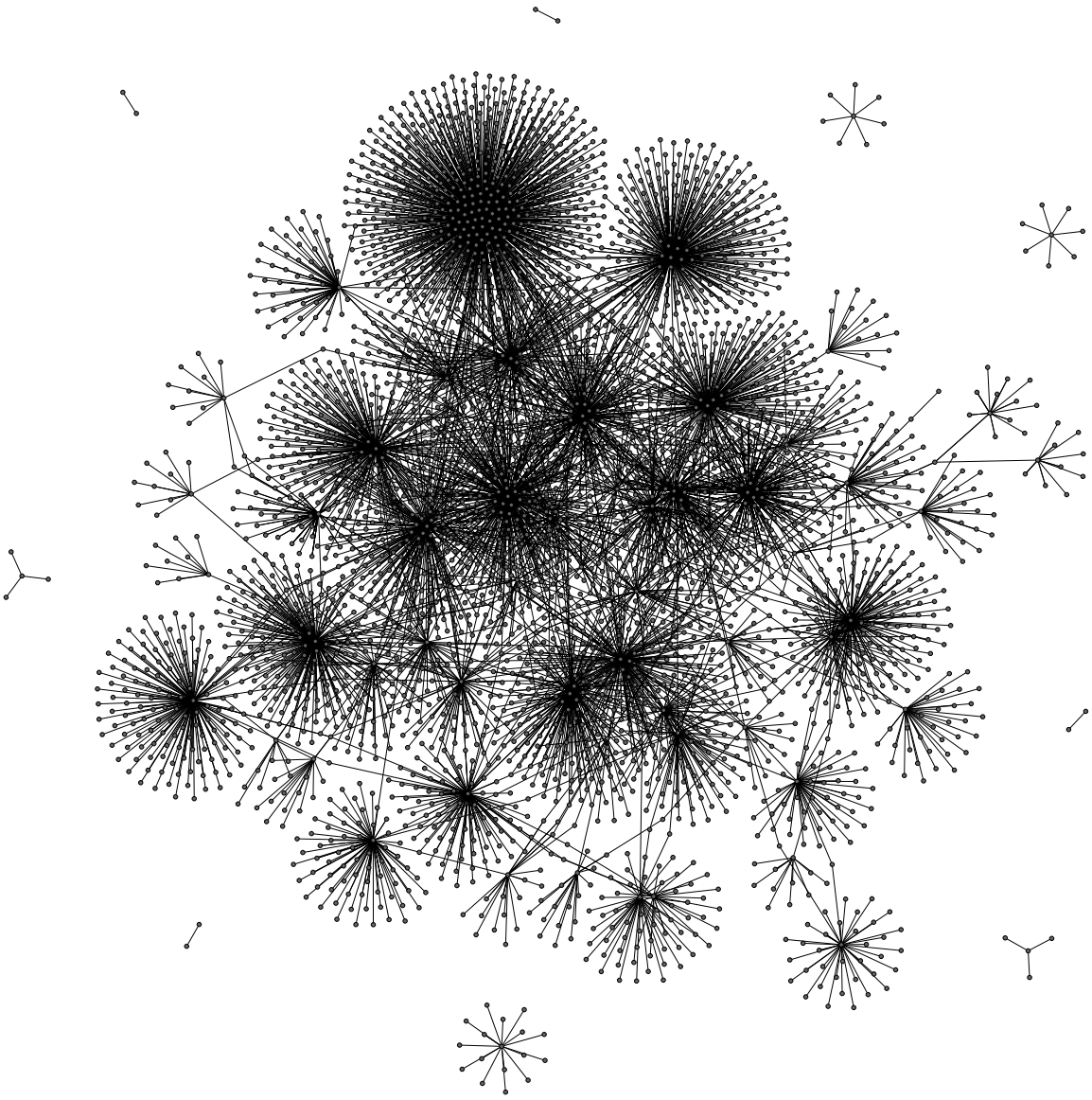


Rysunek 20: Przykłady wygenerowane przez wizualizator.



Rysunek 21: Wpływ sił odpychających

Dobierając parametry w sposób eksperymentalny, algorytmu można uzyskiwać dobre wyniki nawet dla dużych grafów.



Rysunek 22: Graf składający się z 4200+ wierzchołków oraz 32500+ krawędzi

Graf z rysunku 22 posiada 2 razy więcej wierzchołków oraz ponad dwukrotną liczbę krawędzi. Dobór parametrów odbywał się w sposób heurystyczny oraz metodą prób i błędów. Ze względu na rozmiar grafu, dobór parametrów okazał się bardzo czasochłonnym procesem. Analityczne spojrzenie pozwoliło ten czas skrócić, jednak łatwo zauważyć, iż otrzymany wynik jeszcze ma pole do usprawnień. Autor przyznał największy priorytet trzymaniu grafu w granicach rysowania. Drugim priorytetem były dobrze sformowane klastry.

Temperatura początkowa Obserwując zachowanie grafów z rysunków 14 - 19, można stwierdzić, iż znaczące zmniejszanie temperatury układu nie jest rozwiązaniem wystarczającym. Estetyczne odczucia Autora sugerują, iż bardziej rozwinięty graf z wyraźnymi klastrami jest bardziej pożądanym wynikiem. Dlatego temperatura zadana została wy-

brana w punkcie balansującym między dobrym rozwinięciem grafa a momentem, kiedy klastry zaczynają się rozjeżdżać, upierając się w granice obszaru rysowania.

Siła odpychająca W przypadku grafów o wyraźnej strukturze klastrowej konieczne jest zatrzymanie inflacji układu do granic planszy. W tym celu wzór na siłę odpychającą został poddany delikatnej modyfikacji:

$$F_{rep} = \frac{k^2}{d^3} \tag{11}$$

w stosunku do domyślnego:

$$F_{rep} = \frac{k^2}{d}. \tag{12}$$

W przypadku danego grafu zastosowany wzór pomógł zmniejszyć siłę odpychającą na większych odległościach, pozostawiając ją wystarczającą dla blisko położonych węzłów. Zachowanie silnego odpychania na małych odległościach jest ważne dla satysfakcjonujących rozmiarów klastrów. Dodatkowo opisane przejście od silnego do słabego oddziaływania powinno się odbywać szybko, żeby zmniejszająca temperatura nie stała na przeszkodzie początkowemu rozwijaniu grafu oraz formowaniu klastrów. W ostatnich etapach chłodzenia chcemy jedynie dokonywania korekt związanych z odległością międzyklastrową.

Siła przyciągająca Została minimalnie zmodyfikowana o mnożnik stały, żeby przyśpieszyć formowanie klastrów.

Listing 2 przedstawia użycie zaimplementowanego w ramach BGL algorytmu FR[11]. Poszczególne przekazywane parametry to:

- `g` – obiekt grafu do pozycjonowania.
- `node_position_map` – struktura danych mapująca wierzchołki na ich położenie w ramach topologii. Finalne położenia wierzchołków są przechowywane w tym parametrze.
- `topology` – topologia przestrzeni (o czym mowa w sekcji 4.2),
- `square_distance_attractive_force()` – struktura z przeciążonym operatorem (), definiująca siły przyciągające.
- `square_distance_repulsive_force()` – struktura z przeciążonym operatorem (), definiująca siły odpychające.
- `all_force_pairs()` lub `boost::make_grid_force_pairs` – definiuje sposób obliczania sił odpychających. Pierwsza opcja sprawia uwzględnianie oddziaływań między wszystkimi wierzchołkami. Druga – z użyciem optymalizacyjnej metody siatki.

- `linear_cooling<double>(initTemp)` − charakter chłodzenia (przyjęto liniowy) oraz początkowa temperatura układu.

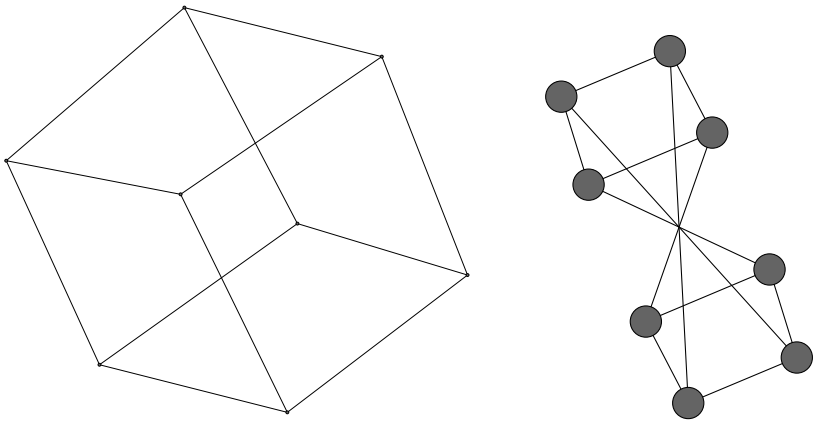
- `make_iterator_property_map()` − parametr nakładający ograniczenie na zmianę położeń poszczególnych wierzchołków. Nie jest aktywnie używany w ramach programu ze względu na potrzebę przeprowadzenia dużej liczby testów. Jest użyteczny podczas

```
void GraphClass::fruchterman_reingold_layout(
            double radius, double width,
            double height, int initTemp)
{
    PositionMap node_position_map = boost::get(&VertexProperties::position, g);
    WeightPropertyMap weight_property_map = boost::get(&EdgeProperties::weight, g);
    boost::minstd_rand generator;
    boost::circle_topology<> topology(generator, width);
    std::vector< Topology::point_difference_type > displacements(num_vertices(g));

    boost::random_graph_layout(g, node_position_map, topology);
    boost::fruchterman_reingold_force_directed_layout(
                g,
                node_position_map,
                topology,
                boost::square_distance_attractive_force(),
                boost::square_distance_repulsive_force(),
                boost::all_force_pairs(),
                boost::linear_cooling< double >(initTemp),
                make_iterator_property_map(displacements.begin(),
                            get(boost::vertex_index, g),
                                Topology::point_difference_type()
                                )
    );

// Osobna metoda informujaca klasy rysujace biblioteki Qt o nowych polozeniach wierzcholkow
    this->setNodesCoordinatesFromPositionMap(node_position_map);
}
```

Listing 2: Użycie algorytmu KK z BGL

Podsumowanie Zarówno algorytm FR, jak i KK wykazują problem zatrzymywania się w minimach lokalnych. Przykład takiego zachowania przedstawia rysunek 23. Aplikacja proponuje dwa obejścia tego problemu. Pierwsze polega na manualnej edycji wyniku. Drugie − na wielokrotnym uruchomieniu grafu dla różnych początkowych ułożeń węzłów. Randomizacja grafu w tym celu odbywa się poprzez wciśnięcie spacji.

Drugim problemem wykazywany przez oba algorytmy jest złożoność obliczeniowa. W przypadku FR wynosi ona $(|V|^2+|E|)$. Człon kwadratowy spowodowany jest obliczaniem sił odpychających (problem n-ciał)[10]. Człon liniowy wynika z obliczeń sił przyciągających. Możliwa optymalizacja w obu przypadkach polega na zaniedbywaniu dalekich wierzchołków przy obliczaniu sił odpychających. Proponowane przez Fruchtermana-Reingolda rozwiązanie polega na wprowadzeniu siatki. Dla danego wierzchołku znaczenie mają wyłącznie węzły znajdujące w tej samej lub sąsiadujących komórkach siatki[10]. BGL implementuje opisane podejście[11]. Dla dużych opisanych wcześniej grafów różnica pomiędzy metodą siatki a jej brakiem wynosi 4-5% na korzyść pierwszej. W przebadanych przykładach nie zaobserwowano jednak jej przewagi w kwestii wizualnej. Wytłumaczeniem podobnych obserwacji może służyć fakt, iż najlepsza optymalizacja jest osiągana przy równomiernej dystrybucji węzłów na siatce. Przykładowe duże grafy nie wykazują tej cechy.



Rysunek 23: Alternatywne wizualizacje jednego grafu

3.2.3 Implementacja własna

Najbardziej interaktywną częścią programu jest własne podejście do wizualizacji. Idea oraz implementacja powstały na bazie sekcji 3.2.2. W gruncie własnego opracowania leży prosta idea podobna do algorytmu FR. Istnieją siły odpychające między każdą parą węzłów. Tylko połączone wierzchołki się przyciągają. Algorytm jest uruchamiany przy każdej rejestracji zdarzenia licznika czasu. Licznik z kolei jest ustawiany przy każdej iteracji, dopóki układ się nie ustabilizuje. Częstotliwość uruchomiania algorytmu może być zmieniana za pomocą odpowiedniego slidera z poziomu interfejsu użytkownika.

```
1    timer = new QTimer(this);
2    connect(timer, SIGNAL(timeout()), this, SLOT(timerEvent()));
3    timer->start(400);
```

Listing 3: Przykład konfiguracji licznika czasu

Powyższy fragment kodu pokazuje przykładową konfigurację licznika czasu. Zadaniem obiektu typu `QTimer` jest dekrementacja wartości podanej w linijce trzeciej. Czas jest podany w mikrosekundach. Druga linijka zapewnia wykonanie funkcji `timerEvent()` kiedy licznik skończy swoje działanie. Musi zostać ponownie ustawiony w celu kontynuacji wizualizacji. Odpowiada za to funkcja `itemMoved()`, ustawiając licznik w zależności od preferencji użytkownika.

```
1     void GraphWidget::itemMoved() {
2         timer->start(1000.0 / configPtr->getFps());
3     }
```

Listing 4: Ustawianie licznika

Opisane podejście bazujące na licznikach umożliwia na wysoki stopień interakcji z programem, pozwalając zatrzymać wizualizację w dowolnym momencie, zmienić graf według własnych potrzeb, a później ją wznowić.

```
1  void GraphWidget::timerEvent(){
2      QVector<Node *> nodes;
3      const QList<QGraphicsItem *> items = getScene()->items();
4
5      for (QGraphicsItem *item : items) {
6          if (Node *node = qgraphicsitem_cast<Node *>(item))
7              nodes << node;
8      }
9
10     for (Node *node : qAsConst(nodes)) {
11             node->calculateForces();
12     }
13
14     bool nodesMoved = false;
15     for (Node *node : qAsConst(nodes)) {
16         if (node->changePosition())
17             nodesMoved = true;
18     }
19
20     if (!nodesMoved) {
21         timer->stop();
22     }
23
24 }
```

Listing 5: Uruchamianie algorytmu

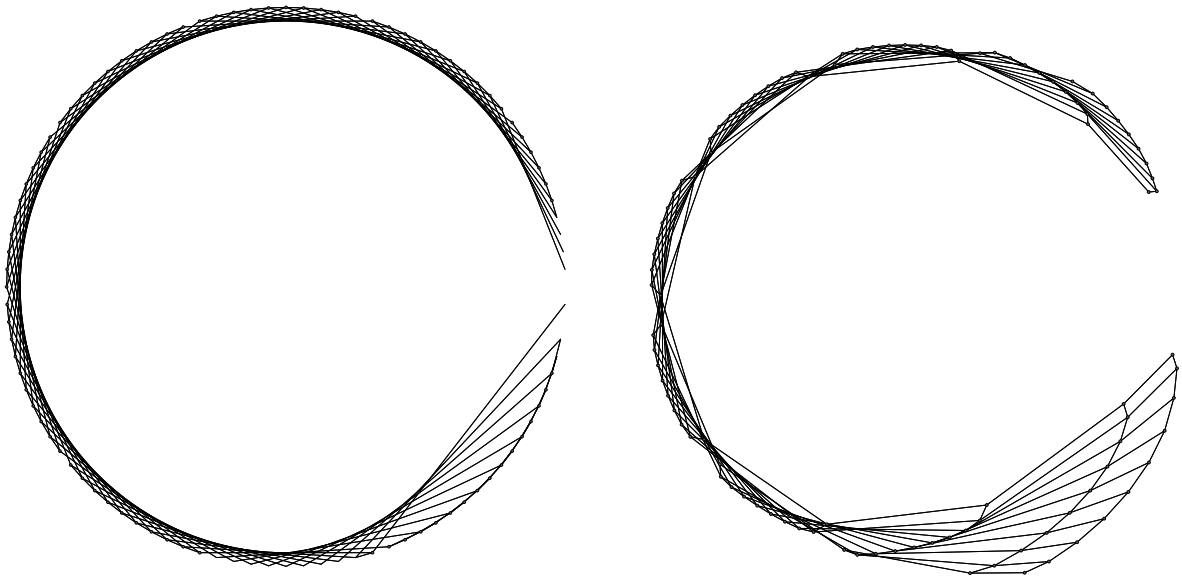
Linijki 3-8 są odpowiedzialne za aktualizację listy obiektów obecnych na planszy oraz selekcji samych wierzchołków. W linijkach 11-13 odbywa się obliczanie sił między wierzchołkami. Kolejnym krokiem jest próba zmiany położeń węzłów. Jeśli zmiana kończy się powodzeniem, informujemy program, że symulacja powinna się kontynuować. Ostatni etap polega na zatrzymaniu licznika w przypadku braku zmian w położeniach węzłów.

Siła odpychająca dla danego wierzchołka obliczana jest jako wypadkowa sił wychodzących od pozostałych węzłów. Algorytm akumuluje wszystkie siły odpychające dla każdej z dwóch współrzędnych. Wynikowa siła przyciągająca jest uzależniona od stopnia wierzchołka, czyli liczby jego sąsiadów. Im więcej krawędzi wychodzą z niego, tym mniejszy udział każdej z nich w sile wypadkowej.

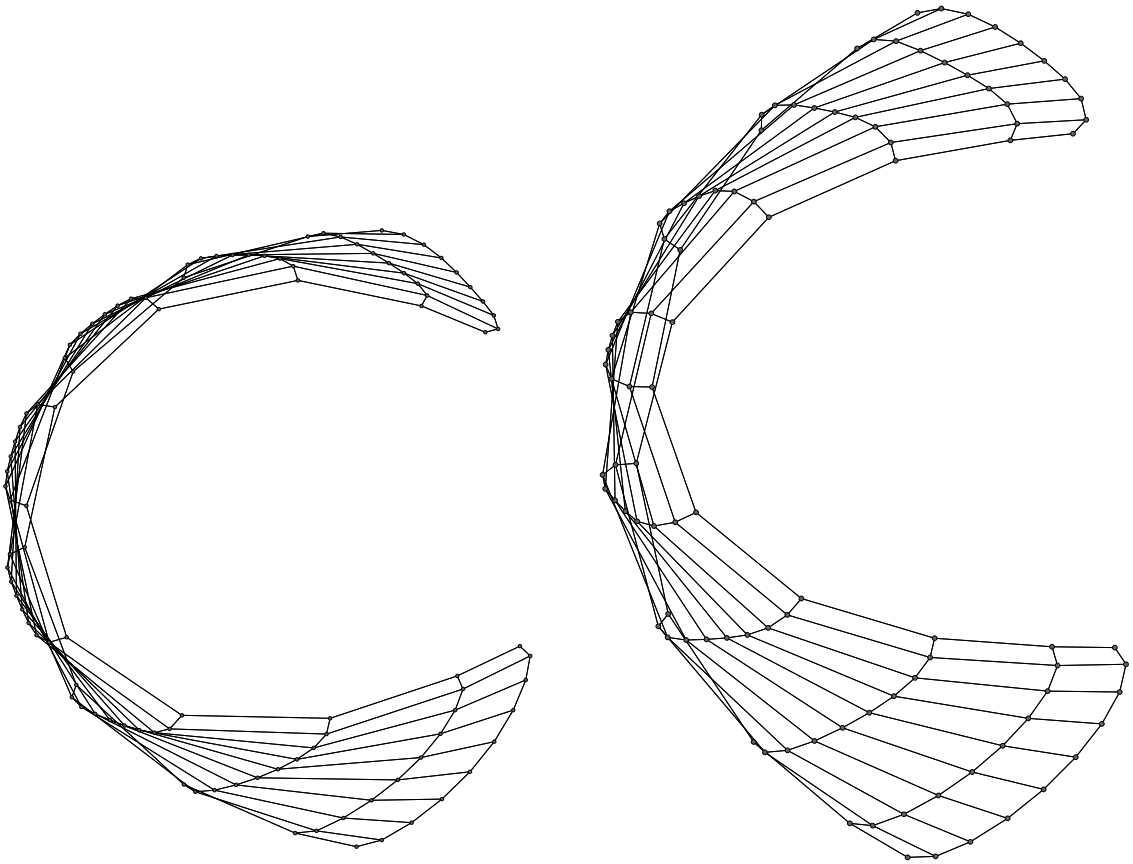
$$F_{repulsive} = \frac{k}{l}; \tag{13}$$

$$F_{attractive} = \frac{l}{b}; \tag{14}$$

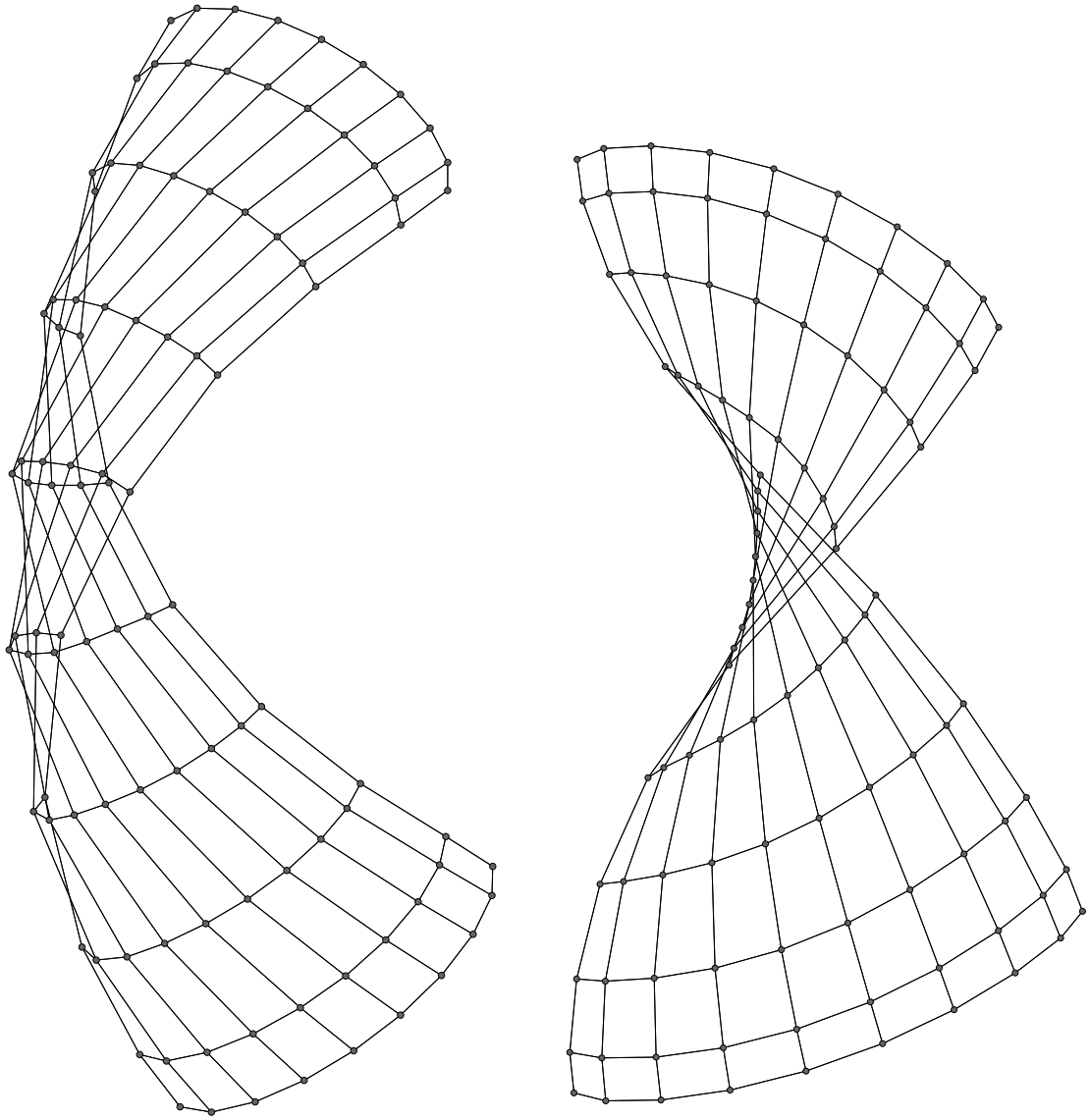
k oraz b są stałymi ustalanymi eksperymentalnie.



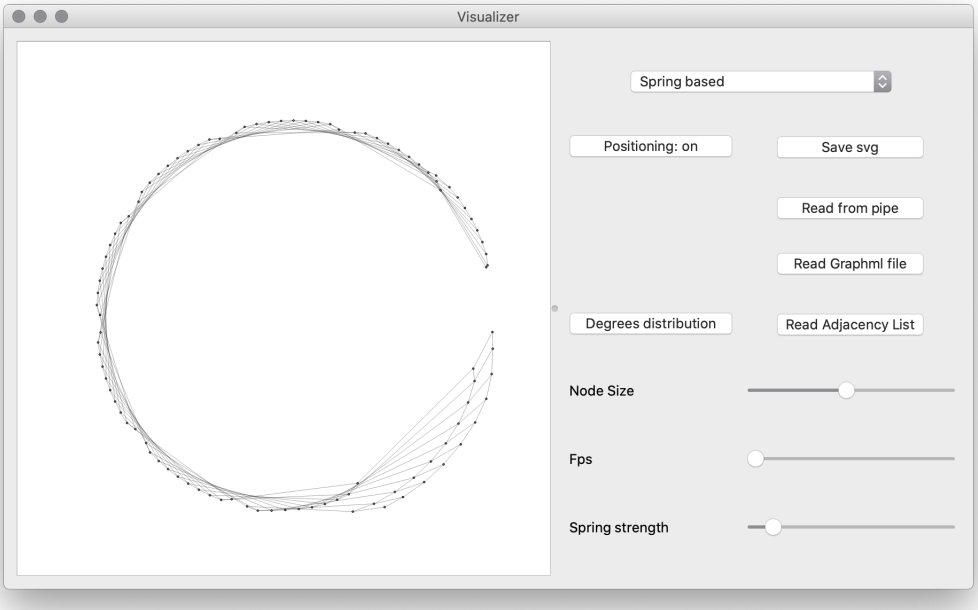
Rysunek 24:



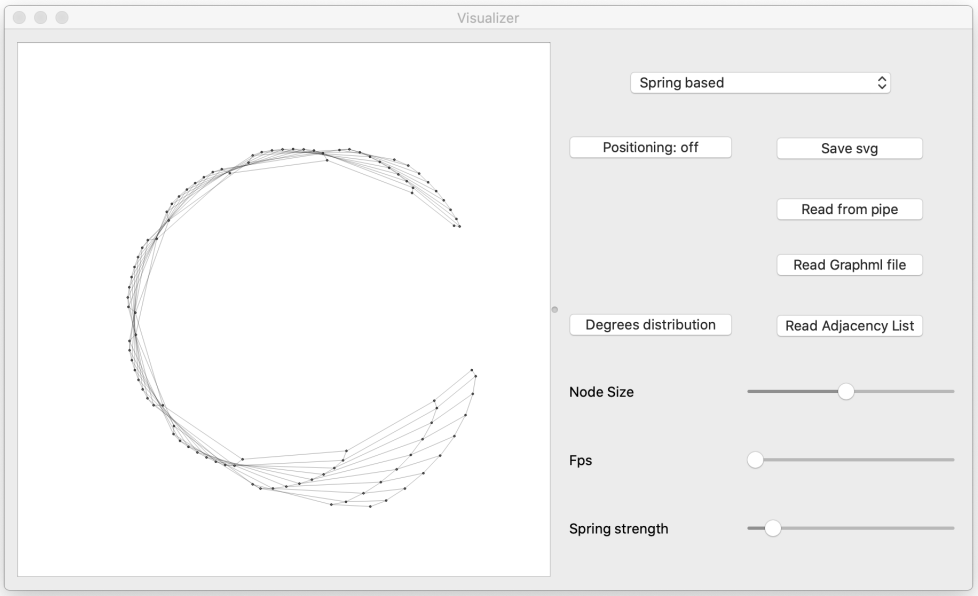
Rysunek 25:



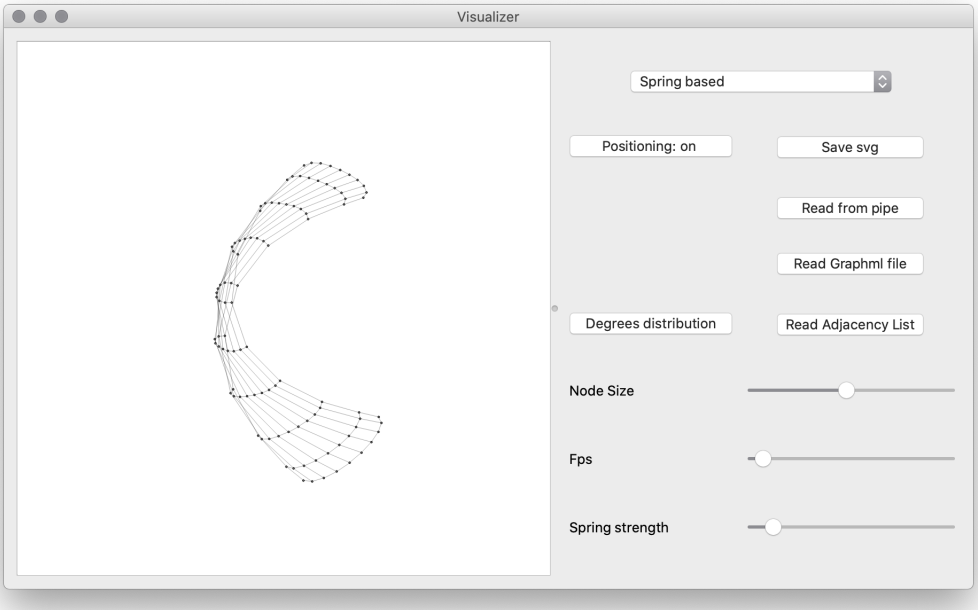
Rysunek 26: Proces rozwijania grafu w kształcie siatki



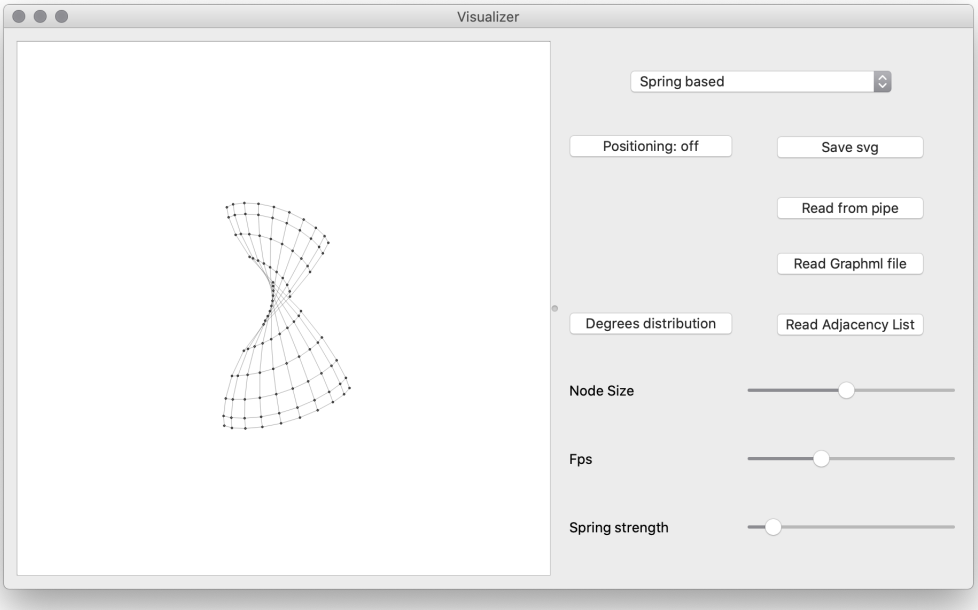
Rysunek 27: Przebieg wizualizacji w aplikacji



Rysunek 28: Przebieg wizualizacji w aplikacji



Rysunek 29: Przebieg wizualizacji w aplikacji



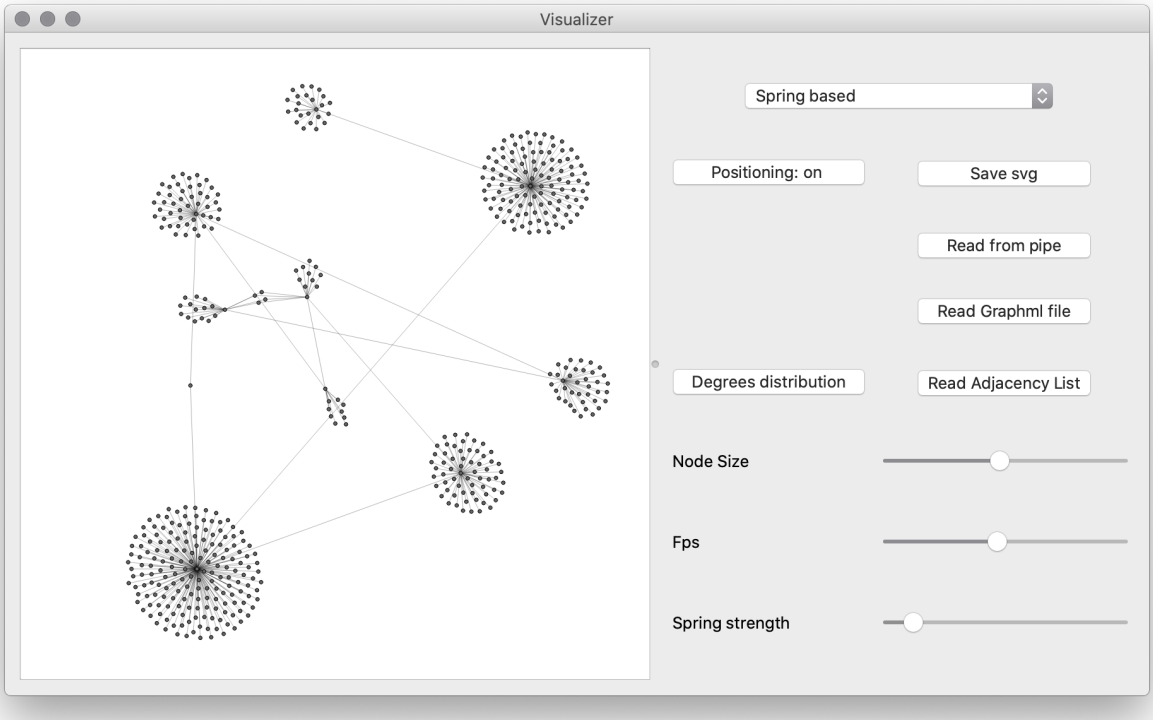
Rysunek 30: Przebieg wizualizacji w aplikacji

Podsumowanie Trudna do oszacowania jest efektywność algorytmu w porównaniu do zaimplementowanych w bibliotece BGL z uwagi na różny mechanizm działania oraz efekty, które Autor chciał uzyskać. Głównym priorytetem była interaktywność algorytmu. Umożliwiło to śledzenie każdej iteracji, sterowanie częstotliwością odświeżania pozycji wierzchołków, oraz wstrzymywanie lub wznawianie wizualizacji. Złożoność każdej

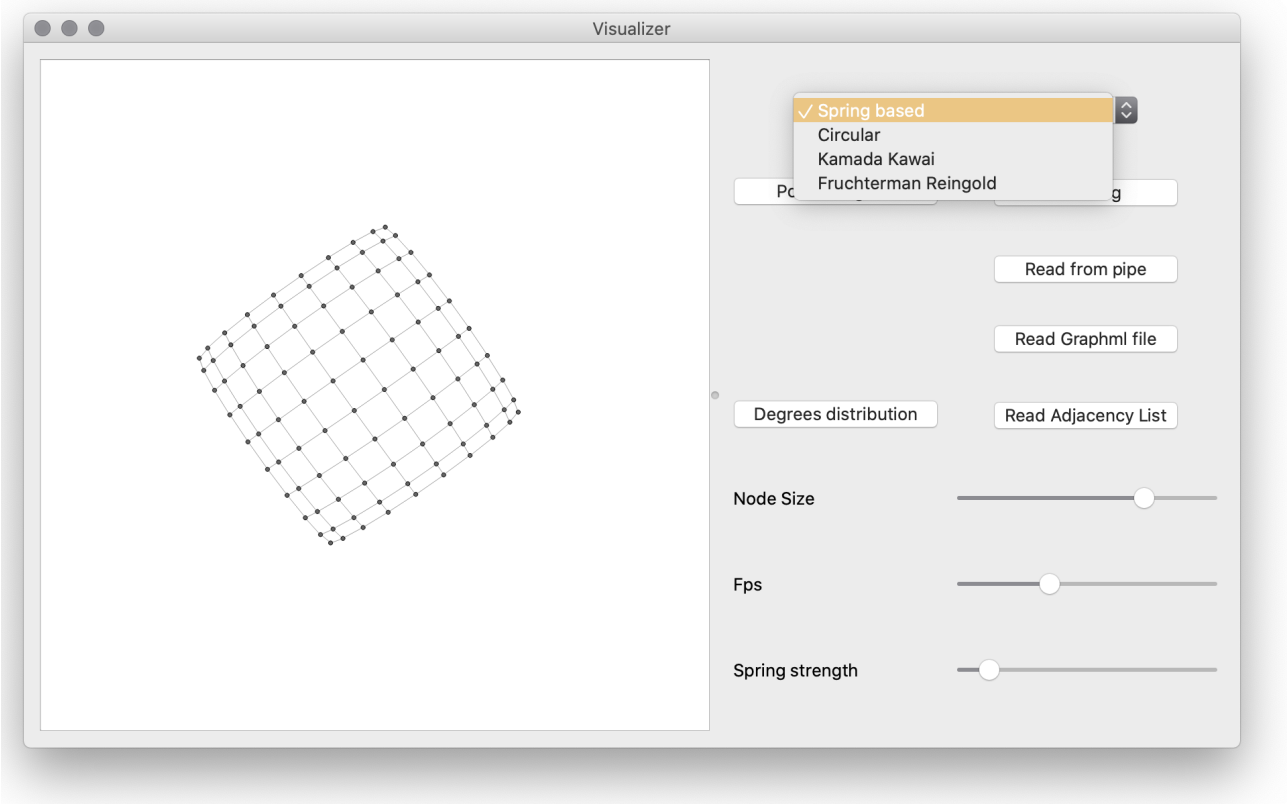
iteracji składa się z $O(|V|^2)$ potrzebnego do obliczania sił odpychających oraz czynnika $O(|E|)$ odpowiadającego za obliczanie sił przyciągających (iteracje po wszystkich krawędziach grafu). Dodatkowy narzut w postaci $O(|V| + |E|)$ pojawia się przy konieczności odświeżania obecnych na planszy obiektów (linijki 5-8 listingu 5).

Wyniki, otrzymywane za pomocą własnej implementacji są podobne do generowanych przez algorytm FR. Jednak ostatni zachowuje bardziej równomierny rozkład wierzchołków i jest odporny na podanie błędnych parametrów. Istnieją przypadki, przy których zbyt duże lub małe wartości parametrów algorytmu własnego prowadzą do destabilizacji układu. Jednak w większości przetestowanych przypadków wyniki są satysfakcjonujące, a interaktywność wizualizacji sprawia estetyczną przyjemność.

4 Opis programu



Rysunek 31: *Interfejs użytkownika*



Rysunek 32: *Interfejs użytkownika z rozwiniętym elementem wyboru algorytmu*

4.1 Konfiguracja

Możliwa jest wstępna konfiguracja działania programu za pomocą pliku w formacie *json*. Obsługa konfiguracji została przeniesiona do osobnej klasy **ConfigReader**. W programie istnieje tylko jedna instancja tej klasy, której pola zostają zainicjalizowane na początku działania programu. Klasy potrzebujące uzyskać dostęp do danych konfiguracyjnych otrzymują wskaźniki typu `std::shared_ptr<ConfigReader>`. Wykorzystane podejście jest wzorcem projektowym o nazwie **Dependency injection**[12]. Podejście różni się od znanego wzorca **Singleton**, ponieważ nie ma żadnych dodatkowych warunków na istnienie pozostałych instancji. Technicznie, każda inna klasa może tworzyć instancje **ConfigReader**. Singleton nie pozwala na takie zachowanie. Parsowanie formatu *json* odbywa się za pomocą klasy **QJsonDocument**:

```
1            QFile inFile("config.json");
2            inFile.open(QIODevice::ReadOnly|QIODevice::Text);
3            QByteArray data = inFile.readAll();
4            inFile.close();
5
6            QJsonParseError errorPtr;
7            QJsonDocument doc = QJsonDocument::fromJson(data, &errorPtr);
8            if (doc.isNull()) {
```

```
9                 qDebug() << "Parse failed";
10              }
11              QJsonObject json = doc.object();
12              if (json.contains("fps")) {
13                fps = json["fps"].toInt();
14              }
```

Listing 6: Przykład obsługi konfiguracji aplikacji.

Po poprawnym wczytaniu surowych danych z pliku funkcja *QJsonDocument::fromJson()* dokonuje próby parsowania wczytanego ciągu bitów jako *json*. Następnie, w przypadku odnalezienia szukanego klucza (w przykładzie szukany klucz to "fps"), pole klasy o nazwie klucza zostaje zainicjalizowane wczytanymi danymi. Poniżej znajduje się przykładowy plik konfiguracyjny (listing 7). Ze względu na przyjętą koncepcję, dodawanie nowych kluczy do konfiguracji jest bardzo łatwe. Polega ono na deklaracji odpowiedniego pola klasy **ConfigReader** oraz doklejaniu odpowiedniej instrukcji warunkowej (linijki 12-14).

```
1 {
2   "fps": 1,
3   "nodeRadius": 10,
4   "sourceFilePrefixPath": "/Users/user/Desktop/graphFiles/",
5   "layout": "Circular"
6 }
```

Listing 7: Przykładowy plik konfiguracyjny.

gdzie:

fps Określa częstotliwość odświeżania położeń wierzchołków. Większe fps wiąże się z koniecznością uruchomienia wybranego algorytmu pozycjonującego więcej razy na sekundę. Przy dużych grafach powoduje to spowolnione działanie programu. Dlatego w przypadku wymagających algorytmów (KK oraz FR) zostaje całkowicie wyłączona opcja odświeżania. Aktualizacja stanu grafu otrzymana od aplikacji-źródła spowoduje automatyczne odświeżenie położeń węzłów.

nodeRadius Opcja rysowania. Określa rozmiar rysowanych wierzchołków.

sourceFilePrefixPath Określa ścieżkę, gdzie muszą się znajdować wszystkie pliki do wczytywania.

layout Określa domyślny sposób rysowania grafu. Możliwe opcje to:

- Spring based
- Circular

- Kamada Kawai
- Fruchterman Reingold

gdzie opcja **Spring based** jest implementacją własną, o której mowa w sekcji 3.2.3.

4.2 Graf

Klasa `GraphClass` reprezentuje graf i definiuje metody do pozycjonowania wierzchołków.

```
1     BoostGraph g;
2     QVector<Node*> nodes;
3     QVector<Edge*> edges;
```

Listing 8: Główne pola klasy `GraphClass`.

Z listingu 8 pole `BoostGraph g` jest używane do potrzeb algorytmicznych i struktura `BoostGraph` jest zgodne z wymaganiami Biblioteki BGL. Z kolei pola `QVector<Node*> nodes` i `QVector<Edge*> edges` wykorzystywane są przy rysowaniu. Klasy `Node` oraz `Edge` dziedziczą po `QGraphicsItem` i są bezpośrednio umieszczane na `QGraphicsScene`. Klasa `Edge` zawiera dane wierzchołków, które dana krawędź łączy. Klasa `Node` przechowuje listę krawędzi wychodzące z danego wierzchołka, graf, którego częścią jest oraz zmienne potrzebne do aktualizacji jego położenia.

```
1  typedef boost::_topology<>::point_type Point;
2
3  struct VertexProperties {
4      std::string name;
5      Point position;
6  };
7  struct EdgeProperties {
8  };
9
10 typedef boost::adjacency_list<boost::vecS,
11                                 boost::vecS,
12                                 boost::undirectedS,
13                                 VertexProperties,
14                                 EdgeProperties> BoostGraph;
```

Listing 9: Struktura grafu BGL `BoostGraph`.

Biblioteka BGL definiuje wymagania wobec struktur danych, na których mogą działać jej algorytmy[13]. Listing 9przedstawia graf używany w programie. Struktura `VertexProperties` zawiera atrybuty każdego wierzchołka[13]. Pole`name` jest wypełniane tylko w przypadku wczytywania grafu w formacie *GraphMl*. `Position` z kolei definiuje położenie wierzchołka

w ramach topologii boost. Topologia opisuje właściwości przestrzeni, w której jest opisana plansza. `Square_topology<>` zachowuje się jako zwykła przestrzeń dwuwymiarowa[14]. Graf jest przechowywany jako lista sąsiedztwa (`adjacency_list<>`). Parametry `vecS` definiują kontener, który będzie przechowywał odpowiednio wierzchołki oraz krawędzie. Kolejne parametry są strukturami opisującymi atrybuty składowych grafu.

4.3 Wczytywanie grafu

Program wspiera cztery podstawowe sposoby na wczytywanie grafów oraz dwa formaty parsowania. Wspierane formaty to

- GraphMl (listing 10),
- Lista sąsiedztwa (listing 11).

```
1     <?xml version="1.0" encoding="UTF-8"?>
2 <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
5       http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
6 <graph id="G" edgedefault="undirected">
7 <node id="A"/>
8 <node id="B"/>
9 <node id="C"/>
10 <node id="D"/>
11 <node id="E"/>
12 <node id="F"/>
13 <node id="G"/>
14 <node id="H"/>
15
16 <edge source="A" target="B"/>
17 <edge source="A" target="D"/>
18 <edge source="A" target="E"/>
19 <edge source="B" target="C"/>
20 <edge source="B" target="F"/>
21 <edge source="C" target="D"/>
22 <edge source="C" target="G"/>
23 <edge source="D" target="H"/>
24 <edge source="E" target="F"/>
25 <edge source="E" target="H"/>
26 <edge source="F" target="G"/>
27 <edge source="G" target="H"/>
28
29 </graph>
30 </graphml>
```

Listing 10: Przykład grafu z rys 8 zapisanego w formacie GraphMl;

```
# 6
0: 5
1: 0
2: 1
3: 2
4: 3
5: 4
```

Listing 11: Przykład grafu z rys 1 zapisanego w formacie listy sąsiedztwa;

Zaimplementowane zostały następujące sposoby wczytywania:

- Z pliku tekstowego:
  - Rozszerzenie *.graphml* dla formatu GraphMl,
  - rozszerzenie *.txt* lub *.dat* dla formatu listy sąsiedztwa,
- Z użyciem kanału nazwanego (ang. named pipe)[15],
- Odbieranie danych od programu-źródła(opisany w sekcji 4.7.2).

Przykładowe użycie kanału nazwanego jest przedstawione w listingu 12. Ten sposób wczytywania jest dostępny dla systemów Unix. Kanały powinny być tworzone w katalogu `/tmp`.

```
cd /tmp
mkfifo my_pipe
cat file > my_pipe
```

Listing 12: Tworzenie kanału nazwanego

Wczytywanie zwykłych plików jest proste i nie wymaga przedstawiania kodu. Natomiast obsługa formatu GraphMl jest realizowane za pomocą biblioteki BGL i jest przedstawione w listingu 13. Zmienna `dp` umożliwia mapowanie atrybutów spotkanych pofczas parsowania pliku *.graphml* na atrybuty biblioteki BGL[16].

```
GraphClass GraphClass::fromGraphMl(
                        const char* filename, GraphWidget *graphWidget,
                        std::shared_ptr<ConfigReader> configPtr)
{
  std::ifstream inFile;
  inFile.open(filename, std::ifstream::in);
  typedef boost::adjacency_list<boost::vecS,
                                boost::vecS,
```

```
                                    boost::undirectedS,
                                    VertexProperties,
                                    EdgeProperties,
                                    GraphData> GraphMl;

    GraphMl gg;
    boost::dynamic_properties dp(boost::ignore_other_properties);
    dp.property("name", boost::get(&VertexProperties::name, gg));
    boost::read_graphml(inFile, gg, dp);

    // Konstruowanie struktur niezbednych do rysowania: krawedzie oraz wierzcholki
    // jako obiekty QGraphicsItem*

    QVector<Node*> nodeVec(0);
    QVector<Edge*> edgeVec(0);

    for (int i = 0; i < boost::num_vertices(gg); ++i) {
        nodeVec.append(new Node(graphWidget, configPtr));
    }

    auto es = boost::edges(gg);
    for (auto eit = es.first; eit != es.second; ++eit) {
       int srcId = boost::source(*eit, gg);
       int dstId = boost::target(*eit, gg);
       edgeVec.append(new Edge(nodeVec.at(srcId),
                       nodeVec.at(dstId),
                       srcId, dstId));
    }

    return GraphClass(nodeVec, edgeVec, gg);
}
```

Listing 13: Wczytywanie formatu GraphMl[16]

4.4 Eksport

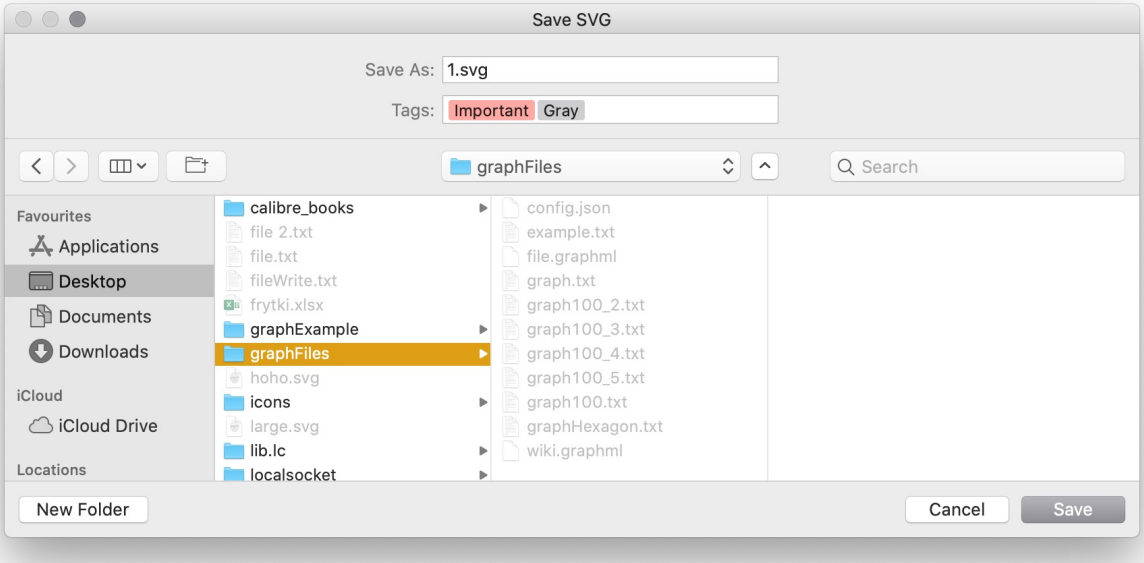
Ważnym elementem programu było umożliwienie zapisu otrzymanych grafów w formatach umożliwiających ich umieszczenie w różnego rodzaju artykułach, prezentacjach czy innych dokumentach. Program umożliwia eksport wyświetlanego grafu w grafice wektorowej. W tym celu zostało zaimplementowane wygenerowanie pliku w formacie *svg* (listing 14). Pierwszym krokiem jest podanie ścieżki docelowej oraz nazwy pliku docelowego. Następnie dokonywana jest konfiguracja generatora. Między innymi, funkcja *setViewBox* definiuje część **QGraphicsScene**, która zostanie wyeksportowana. W przypadku niniejszej

pracy jest to cały wyświetlany graf. Funkcja *render()* służy do uchwycenia zawartości **QGraphicsScene** na potrzeby robienia zrzutów ekranu lub druku.

```
void MyMainWindow::on_saveButton_clicked(){
    QString newPath = QFileDialog::getSaveFileName(this,
                                                   "Save SVG",
                                                   path,
                                                   "SVG files (*.svg)");

            if (newPath.isEmpty())
                return;

            path = newPath;
            int h = static_cast<int>(ui->graphWidget->getScene()->height());
            int w = static_cast<int>(ui->graphWidget->getScene()->width());
            QSvgGenerator generator;
            generator.setFileName(path);
            generator.setSize(QSize(w, h));
            generator.setViewBox(ui->graphWidget->getScene()->sceneRect());
            generator.setTitle("SVG Example");

            QPainter painter;
            painter.begin(&generator);
            ui->graphWidget->getScene()->render(&painter);
            painter.end();
}
```

Listing 14: generowanie pliku *svg* za pomocą klasy **QSvgGenerator**.

Użytkownik ma możliwość wskazania docelowej lokalizacji oraz nazwy pliku. Przykłady wygenerowanych grafów zostały wklejone w różnych sekcjach niniejszej pracy. Na rys 33przedstawiony został wygląd okna dialogowego dla systemu Mac OS. W innych systemach operacyjnych wygląd będzie się różnił. Funkcjonalność pozostaje jednak niezmienna.



Rysunek 33: *Interfejs zapisywania pliku svg*

4.5 Rysowanie

Kluczowym elementem frameworku Graphics View jest klasa **QGraphicsScene**, która zapewnia przestrzeń, na której umieszczane są obiekty graficzne (wierzchołki wraz z krawędziami w przypadku niniejszej pracy). Jej zakres odpowiedzialności stanowią:

- interfejs obsługujący dużą liczbę obiektów,
- kontrola stanu każdego obiektu,
- propagowanie sygnałów do każdego z obiektów.

.

Kod przedstawiony na listingu 15 tworzy obiekt **QGraphicsScene**, ustawia jego wymiary oraz wyłącza indeksowanie obiektów. Indeksowanie ma służyć przyśpieszonemu wyszukiwaniu obiektów przy interakcjach. Nakłada jednak dodatkowy narzut obliczeniowy przy animacjach lub częstych zmianach wyświetlanych obiektów. Operacja wyszukiwania wyświetlanych obiektów nie jest aktywnie używana, dlatego zdecydowano się na wyłączenie tej opcji. Ostatnim krokiem konfiguracji jest wskazanie widokowi **QGraphicsView**, zawartość jakiej sceny powinien on wizualizować[17].

```
1 scene = new QGraphicsScene(this);
2 scene->setItemIndexMethod(QGraphicsScene::NoIndex);
3 scene->setSceneRect(-200, -200, 400, 400);
```

```
4 setScene(scene);
```

Listing 15: Uproszczona konfiguracja **QgraphicsScene**.

Dodawanie obiektów graficznych jest bardzo proste. Musimy wyczyścić obszar przed rysowaniem, tak aby nowy stan grafu nie był po prostu "doklejany" do poprzedniego stanu. Oba typy obiektów są przechowywane w wektorach, a klasy **Node** oraz **Edge** dziedziczą po **QGraphicsItem**. Jest to klasa bazowa, dostarczająca następujące funkcjonalności klasom potomnym:

- przeciąganie obiektów (Drag and drop);
- obsługa zdarzeń klawiturowych oraz myszy;
- wykrywanie kolizji;

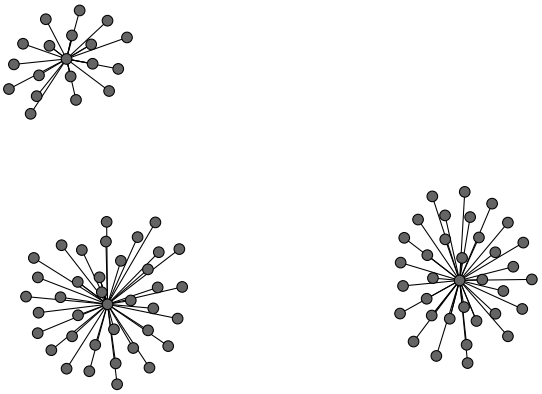
We frameworku Graphics View mamy do czynienia z trzema układami współrzędnych. **QGraphicsItem** ma swoje lokalne współrzędne, jednocześnie będąc osadzonym w układzie **QGraphicsScene**. **QGraphicsView** ma natomiast własny układ obowiązujący dla konkretnego widoku. Framework dostarcza mechanizm wzajemnego mapowania tych trzech układów współrzędnych.

```
1  void GraphWidget::addNodesEdgesToScene(const QVector<Node*>& nodeVec,
2                                         const QVector<Edge*>& edgeVec) {
3      scene->clear();
4
5      for (int i = 0; i < nodeVec.length(); ++i) {
6          scene->addItem(nodeVec.at(i));
7       }
8
9      for (int i = 0; i < edgeVec.length(); ++i) {
10         scene->addItem(edgeVec.at(i));
11     }
12 }
```

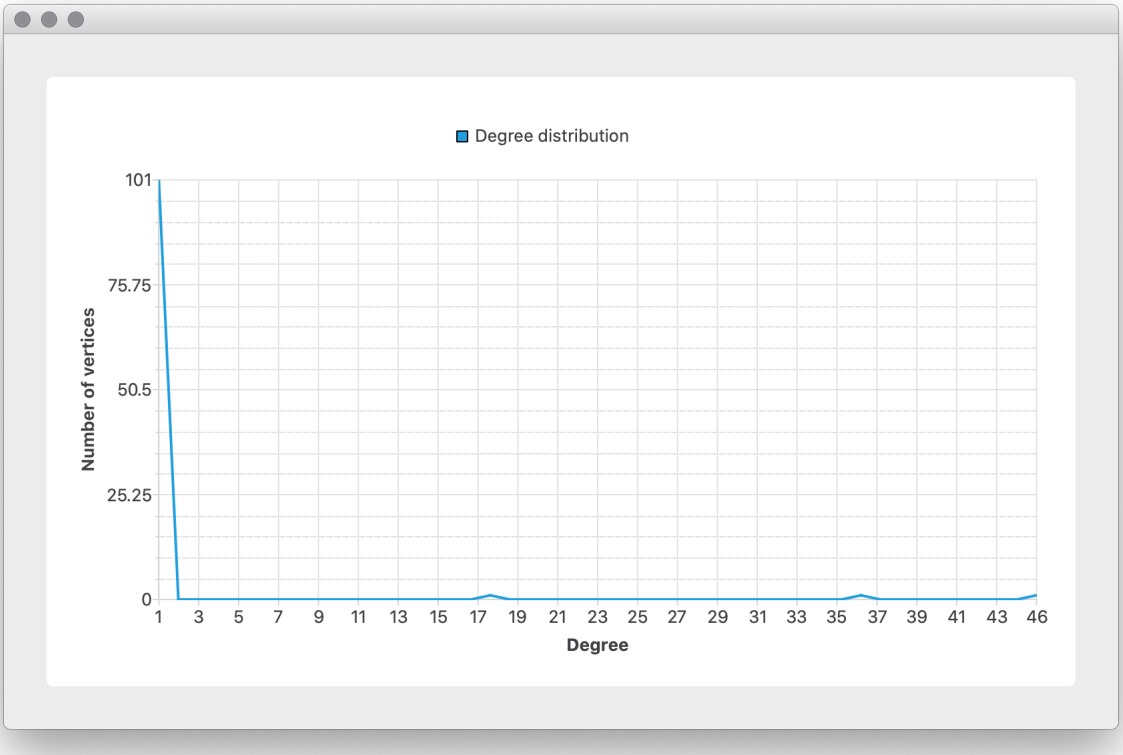
Listing 16: dodawanie obiektów do **QgraphicsScene**.

4.6 Rozkład stopni wierzchołków

Dodatkowym narzędziem do analizy cech grafu jest rozkład stopni wierzchołków. Program umożliwia generowanie wykresu dla wcześniej wczytanego grafu. Rysunki 35, 36 i 37 przedstawiają rozkłady dla grafów połączeń między artykułami Wikipedii.



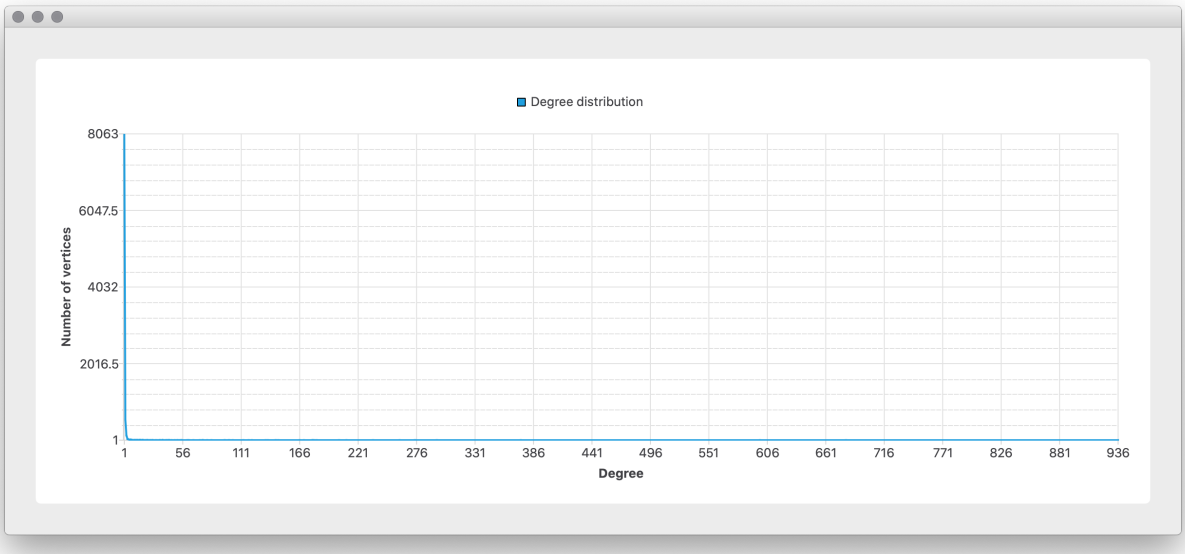
Rysunek 34: Wizualizacja wygenerowana przez zaimplementowany wizualizator.



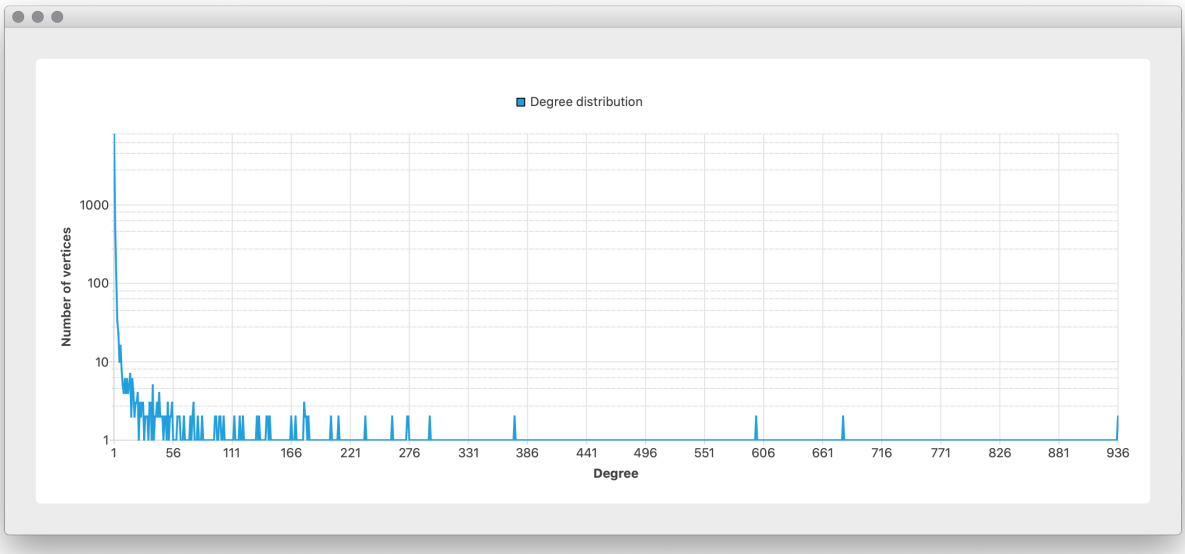
Rysunek 35: Rozkład stopni wierzchołków grafu z rys. 34

Nawet dla takiego małego grafu z nierównomierną gęstością widać, że wykres traci na czytelności. Z tego powodu została zaimplementowana możliwość wyświetlania wykresu w skali logarytmicznej. Rysunki 36 37 przedstawiają różnicę pomiędzy dwiema skalami dla grafu składającego się z około 9000 wierzchołków. Rys. 36 nie pozwala na wzrokowe wykrycie największych klastrów poza największym, w którym węzeł centralny ma 936 sąsiadów. Skala logarytmiczna natomiast umożliwia określenie liczby oraz liczności naj-

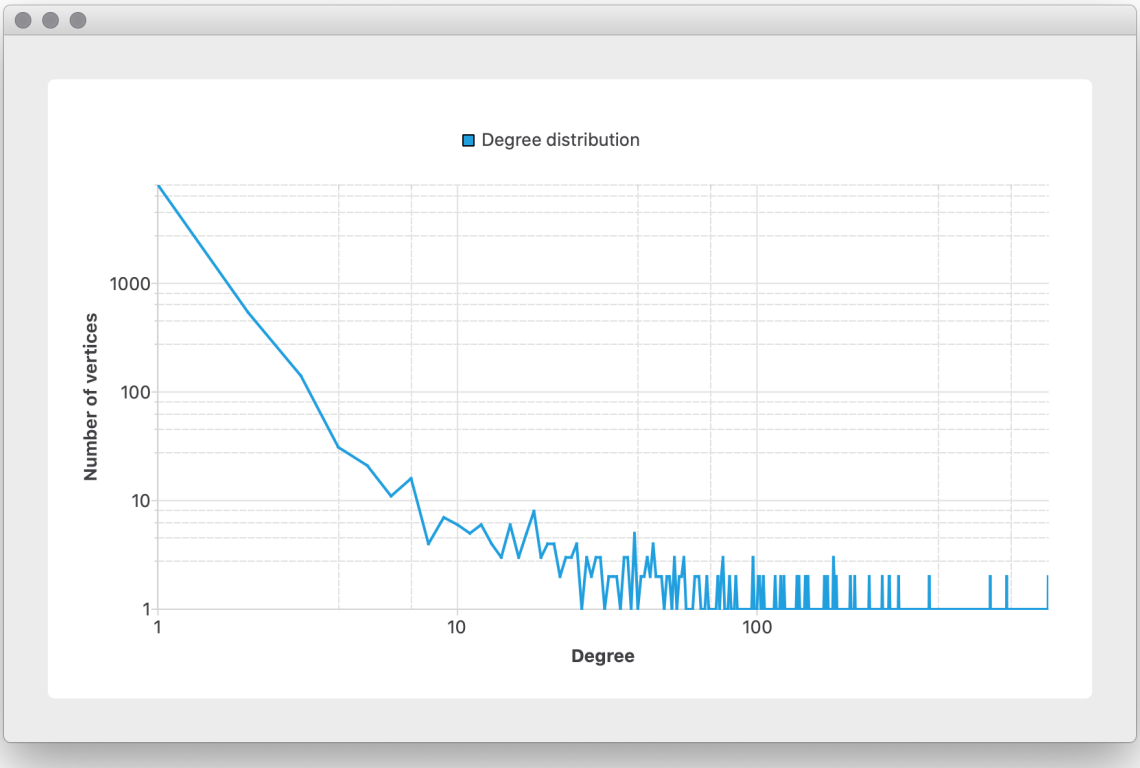
większych klastrów (rys. 37). Skala logarytmiczna dla obu osi (rys. 38) pozwala wyciągać wnioski o wykładniczym charakterze rozkładu stopni.



Rysunek 36: *Rozkład sąsiedztwa w skali liniowej*



Rysunek 37: *Rozkład sąsiedztwa w skali logarytmicznej dla osi Y*



Rysunek 38: *Rozkład sąsiedztwa w skali logarytmicznej dla obu osi*

Listing 17 przedstawia konfigurację wykresów za pomocą obiektu `QChartView* chartView`. Podobnie do klasy `QGraphicsView`, `QChartView` jest widokiem wyświetlającym obiekt wykresu typu `QChart`. Dla poprawnej konfiguracji diagramów niezbędne jest dostarczenie danych (`QLineSeries`) oraz konfiguracja osi (`QValueAxis`).

```
1  QLineSeries *series = new QLineSeries();
2  for (auto const& x : distribution){
3          series->append(x.first, x.second);
4  }
5  QChart *chart = new QChart();
6  chart->legend();
7  chart->addSeries(series);
8
9  QValueAxis *axisX = new QValueAxis();
10 axisX->setTitleText("Degree");
11 axisX->setLabelFormat("%i");
12 axisX->setTickCount(ticksNumber);
13 chart->addAxis(axisX, Qt::AlignBottom);
14 series->attachAxis(axisX);
15
```

```
16 QValueAxis *axisY = new QValueAxis();
17 axisY->setTitleText("Number of vertices");
18 axisY->setLabelFormat("%g");
19 axisY->setMinorTickCount(minorTicksCount);
20 chart->addAxis(axisY, Qt::AlignLeft);
21 series->attachAxis(axisY);
22
23 this->chartView->setChart(chart);
```

Listing 17: konfiguracja wykresów.

Dane Zmienna `distribution` jest obiektem typu `std::map<long, long>`, gdzie kluczami są posortowane stopni wierzchołków, a wartościami – liczba wierzchołków o danym stopniu. Listing 18 przedstawia proces uzyskania danych o stopniach wierzchołków. Biblioteka BGL dostarcza funkcję `degree()`, która zwraca stopień danego wierzchołka.

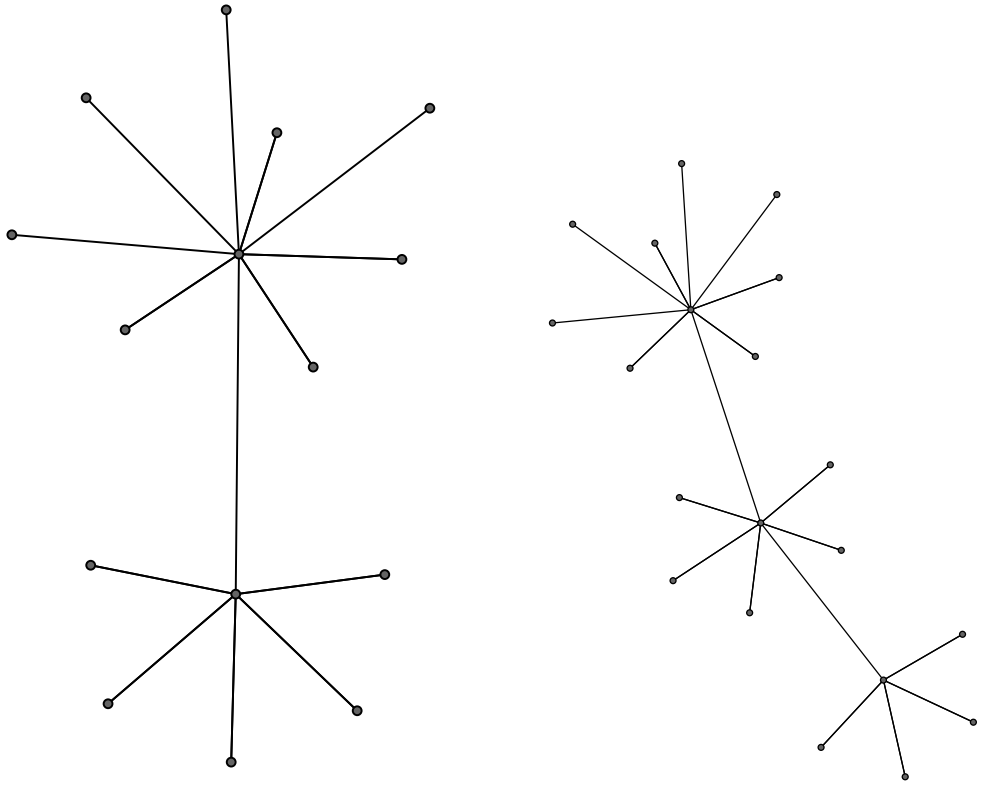
```
1  std::vector<long> graph_degrees;
2  boost::graph_traits<BoostGraph>::vertex_iterator i, end;
3  for (boost::tie(i, end) = boost::vertices(g);  i != end; ++i) {
4      graph_degrees.push_back(boost::degree(*i, g));
5  }
6
7  // sortowanie upraszcza analize wizualna wykresu
8  sort(degrees.begin(), degrees.end());
9
10 // zakres wszystkich stopni wystepujacych w grafie
11 auto bounds = std::minmax_element(degrees.begin(), degrees.end());
12 std::map<long, long> distribution;
13
14 // inicjalizacja slownika
15 for (long i = *bounds.first; i <= *bounds.second; ++i){
16     distribution[i] = 1;
17 }
18
19 // uzupelnianie rozkladu
20 for(std::vector<long>::iterator it = degrees.begin(); it != degrees.end(); ++it) {
21     qDebug() << it - degrees.begin() << " " << *it;
22     distribution[*it]++;
23 }
```

Listing 18: dane do wykresu

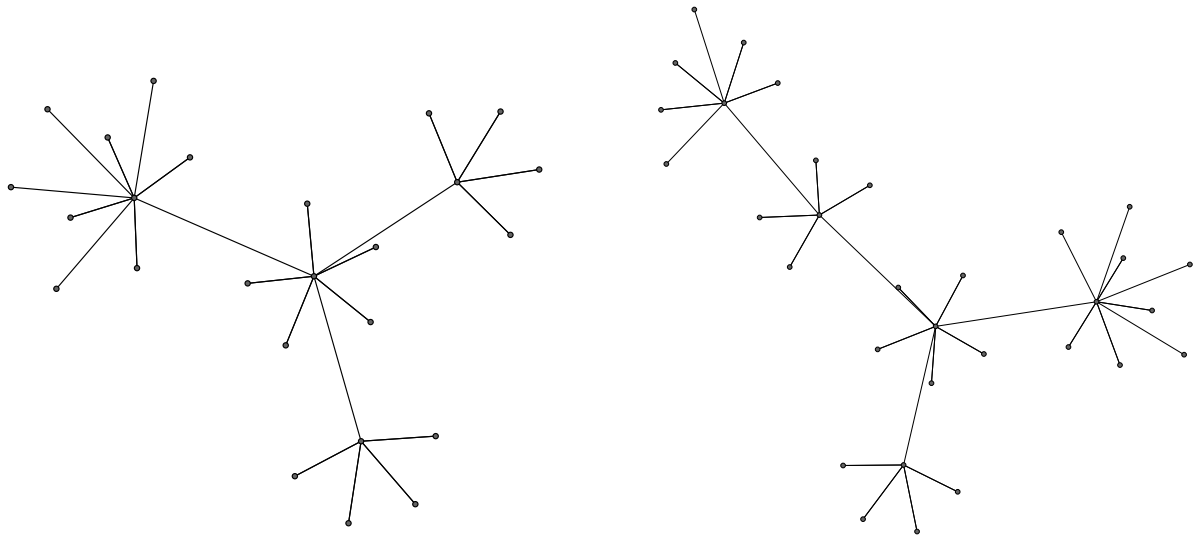
4.7 Grafy dynamiczne

W praktyce grafy często mają charakter dynamiczny. Innymi słowy, ich struktura może ulegać zmianie wraz z upływem czasu. Wizualizator umożliwia wczytywane ta-

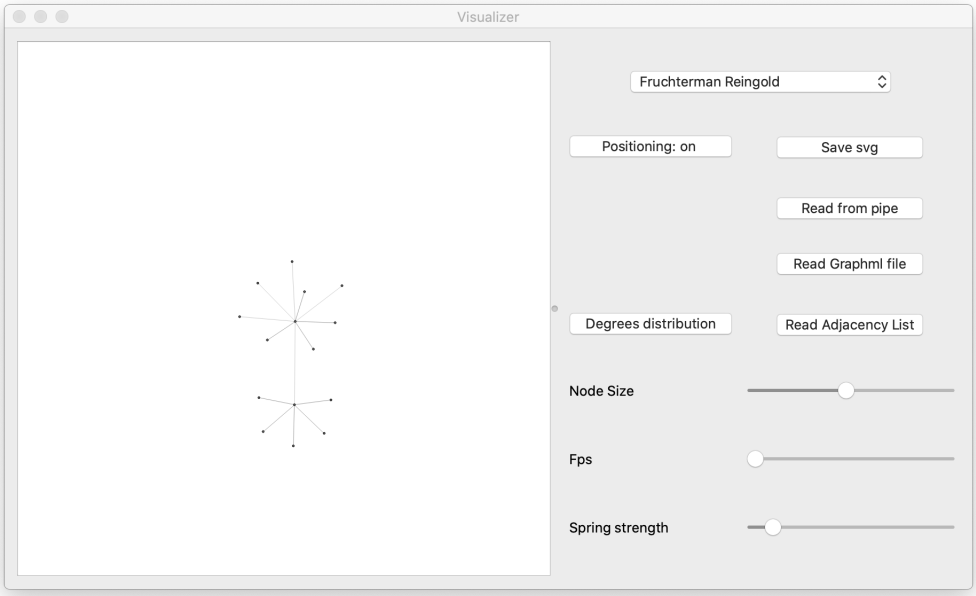
kich grafów w czasie rzeczywistym używając w tym celu komunikacji między procesami (ang. IPC[3]). Jest to nazwa zbiorcza sposobów komunikacji procesów na poziomie systemu operacyjnego[18]. Rozpowszechnioną praktyką przy IPC jest działanie aplikacji w rolach serwera lub klienta. Klient wysyła zapytania do serwera, otrzymując od ostatniego odpowiedzi. W celu symulacji dynamiki grafu został napisany program klient, który łączy się z serwerem i wysyła aktualny stan grafu ze zdefiniowaną częstotliwością. Serwer odczytuje wysłane dane jako ciąg bajtów, parsuje je do formatu listy sąsiedztwa, i wykonuje cały proces wizualizacji dla każdej otrzymanej ramki. Rysunek 39 przedstawia cztery kolejne stany grafu otrzymane od programu-źródła. Z koli rysunki 40 - 43 obrazują wygląd tego procesu z punktu widzenia użytkownika.



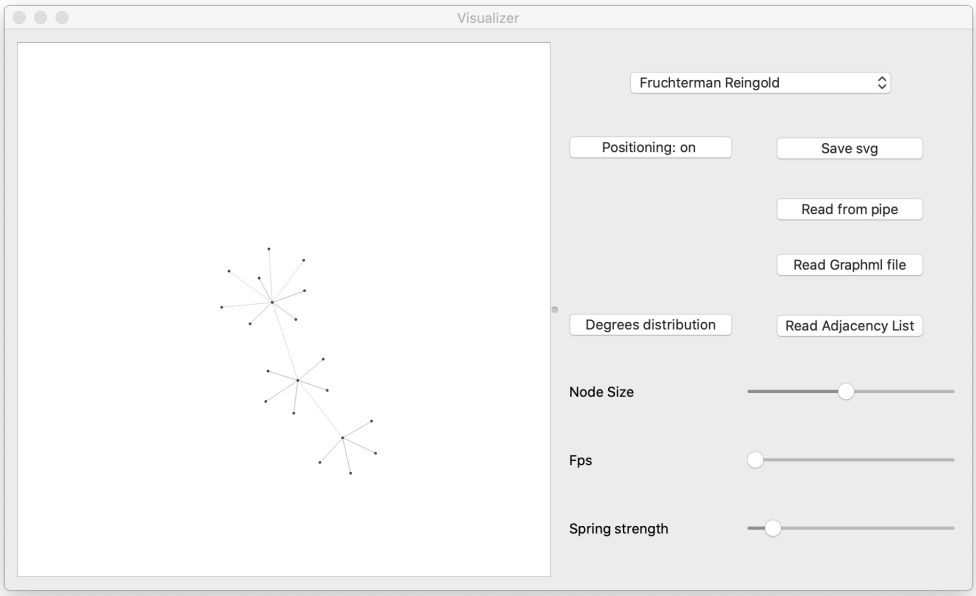
[3]Inter-Process Communication



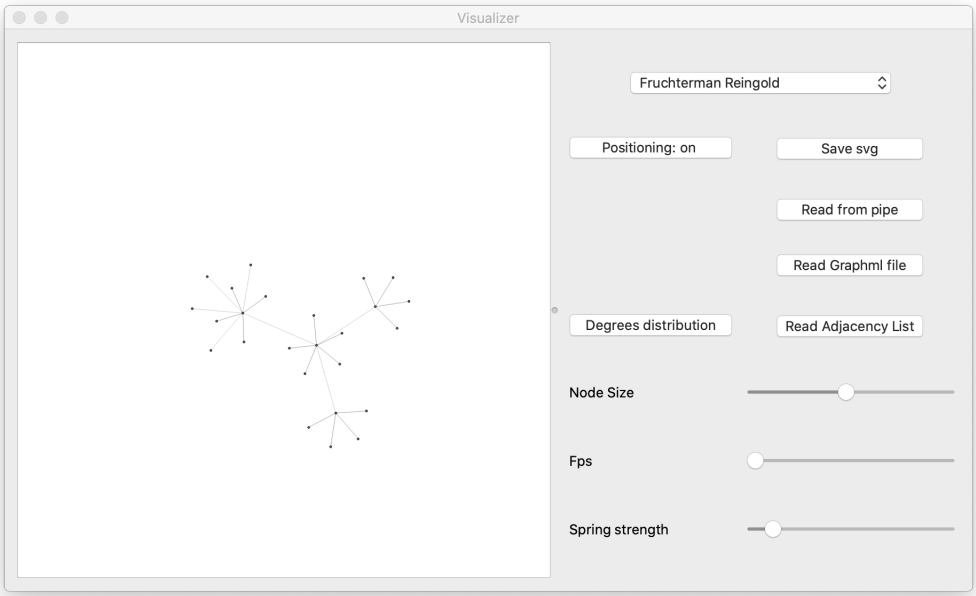
Rysunek 39: Wizualizacja grafu dynamicznego. Wygenerowane przez zaimplementowany wizualizator



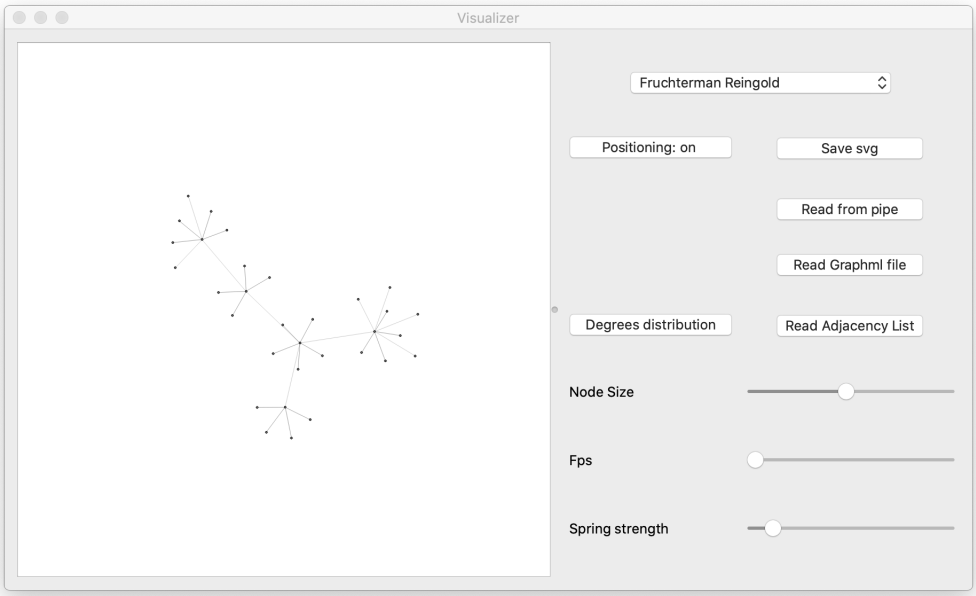
Rysunek 40: Kolejne stany grafu. Wygląd aplikacji



Rysunek 41: Kolejne stany grafu. Wygląd aplikacji



Rysunek 42: Kolejne stany grafu. Wygląd aplikacji



Rysunek 43: Kolejne stany grafu. Wygląd aplikacji

4.7.1 Serwer

Główny program pełni funkcje serwera, otrzmując zapytania oraz dane ze stanem grafu od klienta. Serwer powinien być uruchomiony wcześniej od programu źródłowego. Funkcjonalność zaimplementowana została za pomocą klasy `QLocalServer`, która umożliwia odbieranie połączeń za pomocą socketów[19]. Metoda `listen()` rozpoczyna nasłuchiwanie połączeń do wcześniej zdefiniowanego adresu serwera. Każde nowe zapytanie wyzwala zdarzenie `newConnection()`. Listing 19 przedstawia proces uruchomienia serwera. Metoda `connect()` odpowiada za wiąanie zdarzenia nowego połączenia oraz obsługującej go funkcji `handleNewConnection()`. Obiekt `server` jest polą głównej klasy aplikacji `MyMainWindow`, dziedziczącej po `QMainWindow`.

```
1     server = new QLocalServer(this);
2     QString serverName = "vizualizer";
3     if (!server->listen(serverName)) {
4         std::cout << "Unable to start the server: "
5                   << server->errorString().toStdString();
6     }
7     connect(server,
8             &QLocalServer::newConnection,
9             this,
10            &MyMainWindow::handleNewConnection);
```

Listing 19: Konfiguracja lokalnego serwera

```
1 void MyMainWindow::handleNewConnection() {
2     QLocalSocket* clientConnection = new QLocalSocket(this);
```

```
3       clientConnection = server->nextPendingConnection();
4       connect(clientConnection,
5               &QLocalSocket::readyRead,
6               this,
7               &MyMainWindow::readGraph);
8   }
```

Listing 20: Obsługa połączenia

Listing 20 przedstawia podstawową obsługę połączenia ze strony klienta. Metoda `handleNewConnec` tworzy obiekt wchodzącego połączenia oraz zleca konstruowanie grafu z otrzymanych danych. `nextPendingConnection` zwraca oczekujące połączenie. Ostatnia linijka wyzwala metodę tworzenia grafu w momencie dostępności nowych danych w kanale odczytu[1].

```
1   void MyMainWindow::readGraph(){
2       QLocalSocket* clientSocket = (QLocalSocket*)sender();
3       QByteArray receivedByteArr;
4       receivedByteArr = clientSocket -> readAll();
5
6       QDataStream in(&receivedByteArr, QIODevice::ReadOnly);
7       ui->graphWidget->readStdString(receivedByteArr.toStdString());
8       clientSocket -> flush();
9   }
```

Listing 21: Wczytywanie danych

Kluczowym krokiem przy konstruowaniu grafu jest poprawna obsługa danych wchodzących. Listing 21 rozpoczyna się z funkcji `sender()`, która zwraca obiekt generujący zdarzenie. W danym przypadku jest to obiekt połączenia w momencie dostępności danych otrzymanych od klienta. Metoda `readAll()` wczytuje całą dostępną zawartość kanału jako tablicę bajtów. Na podstawie tej tablicy powstaje obiekt strumieniowy `QDataStream`, a następnie wywoływana jest metoda parsująca `readStdString()`. Na końcu zawartość kanału zostaje wyczyszczona za pomocą metody `flush()`.

4.7.2 Program-źródło

Zaimplementowana została aplikacja, pełniąca rolę klienta. Odpowiada ona za regularne wysyłanie danych do serwera. Obecny scenariusz wysyłania polega na kolejnym wczytywaniu plików zawierających stan grafu w określonym momencie. W zależności od konkretnego przypadku, program może być odpowiednio dostosowany do działania na jednym pliku lub kanałach nazwanych (ang. named pipe), co pozwoli zaoszczędzić zasoby obliczeniowe oraz pamięciowe. Jednak uwzględniając złożoność obliczeniową algorytmów pozycjonujących, scenariusz rysowania dużych grafów w czasie rzeczywistym pozostaje trudny do realizacji.

```
Client::Client(): socketWrite(new QLocalSocket(this))
{
  connect(socketWrite,
          QOverload<QLocalSocket::LocalSocketError>::of(&QLocalSocket::error),
          this, &Client::displayError);
  QString serverName = "vizualizer";
  socketWrite->connectToServer(serverName);
}
```

Listing 22: Konfiguracja po stronie klienta

```
void Client::sendGraph(){
  QDataStream out(socketWrite);
  out.setVersion(QDataStream::Qt_5_10);

  std::ifstream t("filename");
  std::stringstream buffer;
  buffer << t.rdbuf();
  buffer.flush();

  const QString &message = QString::fromStdString(buffer.str());

  out << message.toUtf8();
  socketWrite->flush();
}
```

Listing 23: Wysyłanie danych

Fragmenty kodu 22 oraz 23 przedstawiają uproszczone procesy konfiguracji i wysyłania jednej ramki danych.

Konfiguracja w listingu 22 polega na inicjalizacji pola `QLocalSocket writeSocket`, reprezentującego połączenie. W konstruktorze głównej klasy aplikacji klienckiej(`Client`) programowane jest zachowanie aplikacji w przypadku błędu oraz próba nawiązywania komunikacji z serwerem o nazwie `"vizualizer"`.

Zanim dojdzie do bezpośredniej komunikacji, ustawiena jest wersja Qt w celu poprawnego kodowania strumienia danych (metoda `setVersion`). Nie jest to krok wymagany, jednak pozwala uniknąć trudno wykrywalnych błędów w przypadku niezgodności wersji serwera z klientem[20]. Wysyłanie danych polega na wczytywaniu danych z pliku do obiektu strumieniowego `std::ifstream`, a następnie wczytywanie danych do `std::stringstream`. Jest to potrzebne ze względu na kompatybilność typu `QString`, który nie posiada konwersji bezpośrednio z typu `std::ifstream`. Ostatnim krokiem jest wysyłanie danych do strumienia wychodzącego `out`.

5 Podsumowanie

głównym celem pracy było umożliwienie interaktywnej wizualizacji grafów. W tym celu zostały przeanalizowane trzy algorytmy z biblioteki BGL: metoda radialna, algorytm Kamada-Kawai oraz algorytm Fruchtermana-Reingolda. Przeanalizowano różnice produkowanych wyników i wymagań obliczeniowych. Poza tym, w ramach aplikacji zaimplementowany został uproszczony algorytm, który, mając ograniczenia algorytmiczne oraz konfiguracyjne, umożliwił wysoki stopień interaktywności wizualizacji poprzez płynne zmiany położeń wierzchołków. Użytkownik ma możliwość wstrzymania animacji. Wszystkie cztery sposoby wizualizacji umożliwiają dalszą edycję manualną poprzez przeciąganie wierzchołków i modyfikacji struktury grafu według własnych preferencji użytkownika. Zrealizowany eksport narysowanego grafu do plików z rozszerzeniem *.svg* pozwala na dalszą edycję wyników wizualizacji w programach, wspierających obsługę grafiki wektorowej. Przykładami takiego oprogramowania są: Adobe Illustrator, CorelDRAW, Inkscape oraz inne. Adobe Illustrator został użyty w ramach niniejszej pracy do dopasowywania rozmiaru, grubości krawędzi, położeń spójnych składowych w przypadkach grafów niespójnych. Autor zaleca wykorzystanie podobnego podejścia przed umieszczaniem obrazków w artykułach lub innych dokumentach. Profesjonalne oprogramowanie umożliwia dopasowywanie stylistyczne wygenerowanych grafów do dokumentów docelowych. Kwestia grafów dynamicznych została zaimplementowana z użyciem komunikacji międzyprocesowej. Umożliwia to szybki transfer danych między programem-źródłem a serwerem, natychmiastowe połączenia oraz oszczędność pamięciową. Wizualizator wspiera wczytywanie grafu w formacie *GraphMl*, co zwiększa uniwersalną użyteczność programu dla innych użytkowników.

głównym wyzwaniem podczas pisania pracy był dobór stosu technologicznego oraz pisanie kodu w technologii, z którą Autor nie ma regularnej styczności. Niektóre powstałe problemy wymagały analizy kodu źródłowego z powodu niewystarczającego udokumentowania biblioteki BGL. Większość z tych problemów została rozwiązana. Jednym z nierozwiązanych problemów jest ograniczona możliwość konfiguracji algorytmu KK.

Podczas tworzenia aplikacji, ciągle pojawiały się pomysły w dotyczące rozwoju oraz pożądanych funkcjonalności aplikacji. Istnieje wiele potencjalnych kierunków rozwoju powstałej aplikacji. Wyróżnić wśród nich można optymalizację do obsługi grafów składających z liczby węzłów rzędu 10^6 i wyżej, rozbudowanie bazy dostępnych algorytmów, ulepszenie interfejsu użytkownika oraz inne. Na chwilę pisania niniejszego manuskryptu Autor wyraża chęć dalszego rozwoju Wizualzatora w ramach hobbistycznego projektu.

Pisanie pracy wiązało się z implementacją licznych funkcjonalności, będących częściami różnych dziedzin wiedzy. Jednak największą satysfakcję oraz zgłębianie wiedzy dotyczyło

algorytmiki, szczególnie całego zagadnienia wizualizacji grafów. Dodatkową przyjemnością stało znalezienie zastosowanie zdobytego doświadczenia na temat wizualizacji grafów w pracy zawodowej.

Literatura

[1] Qt. Qt documentation: Qlocalsocket. https://doc.qt.io/qt-5/qlocalsocket.html [dostęp 20.11.2020].

[2] Boost. Boost graph library documentation. https://www.boost.org/doc/libs/1_74_0/libs/graph/doc/index.html [dostęp 25.10.2020].

[3] DBpedia. Dbpedia datasets. https://wiki.dbpedia.org/develop/getting-started [dostęp 20.11.2020].

[4] GraphML. The graphml file format. http://graphml.graphdrawing.org [dostęp 08.09.2020].

[5] Uğur Doğrusöz, Brendan Madden, and Patrick Madden. Circular layout in the graph layout toolkit. In Stephen North, editor, *Graph Drawing*, pages 92–100, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[6] Weidong Huang, Seok-Hee Hong, and Peter Eades. Effects of sociogram drawing conventions and edge crossings in social network visualization. *Journal of Graph Algorithms and Applications*, 11(2):397–429, 2007.

[7] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7 – 15, 1989.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[9] Boost. Bgl kamada-kawai algorithm. https://www.boost.org/doc/libs/1_74_0/libs/graph/doc/kamada_kawai_spring_layout.html [dostęp 20.11.2020].

[10] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991.

[11] Boost. Bgl fruchterman-reingold algorithm. https://www.boost.org/doc/libs/1_74_0/libs/graph/doc/fruchterman_reingold.html [dostęp 20.11.2020].

[12] Mark Seemann. Dependency injection is loose coupling. https://blog.ploeh.dk/2010/04/07/DependencyInjectionisLooseCoupling/ [dostęp 19.11.2020].

[13] Boost. Bgl graph concepts. https://www.boost.org/doc/libs/1_66_0/libs/graph/doc/graph_concepts.html [dostęp 21.11.2020].

[14] Boost. Bgl topologies for graph drawing. https://www.boost.org/doc/libs/1_74_0/libs/graph/doc/topology.html [dostęp 21.11.2020].

[15] GNU. Fifo special files. https://www.gnu.org/software/libc/manual/html_node/FIFO-Special-Files.html [dostęp 20.11.2020].

[16] Boost. Boost graph library: read_graphml. https://www.boost.org/doc/libs/1_74_0/libs/graph/doc/read_graphml.html [dostęp 08.09.2020].

[17] Qt. Qt documentation: Graphics view. https://doc.qt.io/qt-5/graphicsview.html [dostęp 10.11.2020].

[18] Microsoft. Interprocess communications. https://docs.microsoft.com/en-us/windows/win32/ipc/interprocess-communications?redirectedfrom=MSDN [dostęp 21.11.2020].

[19] Qt. Qt documentation: Qlocalserver. https://doc.qt.io/qt-5/qlocalserver.html [dostęp 20.11.2020].

[20] Qt. Qt documentation: Qdatastream. https://doc.qt.io/qt-5/qdatastream.html [dostęp 21.11.2020].