Politechnika Śląska
Wydział Automatyki, Elektroniki i Informatyki

Mateusz Forczmański

Narzędzie do wizualizacji działań w sieciach przepływowych

projekt inżynierski

kierujący pracą: dr inż. Agnieszka Debudaj-Grabysz

Gliwice, 29 grudnia 2015

Spis treści

Rozdział 1

Wstęp

Sieć przepływowa jest abstrakcyjnym modelem danych, który jest wykorzystywany do rozwiązywania problemów związanych z przepływem produktów. Podobnie jak w klasycznym problemie producenta i konsumenta, istnieje w systemie źródło, gdzie produkty są tworzone, oraz ujście, gdzie są konsumowane. W sieci przepływowej zakłada się, że produkty są generowane i użytkowane w tym samym tempie, więc problemem nie jest synchronizacja, ale ilość produktów jakie można maksymalnie przesłać.

Intuicyjnie można wyobrazić sobie sieć przepływową jako układ hydrauliczny, gdzie krawędziami grafu są rury o stałej średnicy, wierzchołki jako połączenia rur, a dwa z nich są wyszczególnione jako miejsce wlewu oraz wylewu cieczy. Średnica każdej rury określa jej przepustowość, czyli maksymalną ilość cieczy jaka może przez nią przepłynąć. Połączenia są jedynie rozgałęzieniami - nie gromadzą płynu, jedynie przekazują go dalej. Innymi słowy, ilość cieczy jaka wpływa do rozgałęzienia musi być równa ilości jaka z niej wypływa. Jest to ta sama zasada co pierwsze prawo Kirchhoffa dla przepływu prądu w obwodzie elektrycznym.

Z siecią przepływową związany jest *problem maksymalnego przepływu*, czyli największej wartości produktów, jaka może przepłynąć w danej sieci ze źródła do ujścia. W swojej pracy inżynierskiej zająłem się implementacją trzech algorytmów służących do znajdowania maksymalnego przepływu w sieci: Forda-Fulkersona, Dinica oraz MKM (Malhortra, Kumar i Mahaswari). Zaprojektowana przeze mnie aplikacja ma mieć charakter edukacyjny: student, który będzie korzystał z programu powinien móc utworzyć własną sieć (wedle swojego zamysłu bądź odwzorować przykład z książki) oraz wykonać na niej jeden z algorytmów. Aplikacja ma na celu umożliwienie mu zrozumienie działania algorytmu krok po kroku, pracę na wielu sieciach oraz zapisanie swojego postępu.

- krótkie wprowadzenie (ok. 2 strony)
- przewodnik po pracy
- wstęp (i zakończenie) najłatwiej jest pisać na końcu

Rozdział 2

Analiza tematu

2.1 Podstawowe pojęcia

2.1.1 Definicja sieci przepływowej

Sieć przepływowa $G = (V, E)$ jest grafem prostym skierowanym; strukturą, która składa się z dwóch zbiorów:

- n-elementowego zbioru wierzchołków $V = \{v_1, v_2, ..., v_n\}$
- m-elementowwgo zbioru uporządkowanych łuków

$$E = \{e_1, e_2, ..., e_m\}, \quad \text{gdzie } e_i = (v_j, v_k); \quad i = 1, ..., m; \quad v_j, v_k \in V$$

  Każdy łuk jest parą wierzchołków należących do zbioru V.

Grafem prostym nazywany taki graf w którym nie istnieją *pętle*. Pętlą nazywamy łuk (v_j, v_k) dla którego $v_j = v_k$. W sieci przepływowej wyróżnia się dwa wierzchołki: źródło $s \in V$ oraz ujście $t \in V$. W sieci przepływowej definiuje się ponadto *funkcję przepustowości* $c : E \to R_{\geq 0}$ odwzorowującą zbiór łuków w zbiór liczb rzeczywistych. Dodatkowo, w celu uproszczenia dziedziny, zakłada się, że z każdego wierzchołka $v \in V \backslash \{s, t\}$ istnieje ścieżka prowadząca ze źródła do ujścia. Założenie to ma zapewnić, że w rozpatrywanych sieciach będzie istniała co najmniej jedna ścieżka ze źródła do ujścia. *Stopniem wejściowym* wierzchołka nazywamy sumę łuków, jakie do niego wchodzą, a *stopniem wyjściowym* sumę łuków, jakie z niego wychodzą.

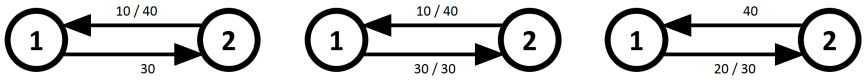
2.1.2 Definicja przepływu sieci

Przepływem w sieci przepływowej G jest funkcja f, która odwzorowuje zbiór łuków w zbiór liczb rzeczywistych $f : E \to R_{\geq 0}$ i spełnia następujące założenia:

- Warunek przepustowości: $\forall_{v_i, v_j \in V} : f(v_i, v_j) \leq c(v_i, v_j)$.
  Przepływ w żadnym z łuków nie może przekroczyć jego przepustowości.
- Warunek skośnej symetryczności: $\forall_{v_i, v_j \in V} : f(v_i, v_j) = -f(v_j, v_i)$.
  Przepływ w przeciwnym kierunku traktujemy jakby posiadał wartość ujemną.
- Warunek zachowania przepływu: $\forall_{v \in V \backslash \{s,t\}} : \sum_{u \in V} f(v, u) = \sum_{u \in V} f(u, v)$.
  Suma przepływów wpływających do wierzchołka musi być równa sumie przepływów wypływających z niego.

2.1.3 Przepływ netto

Przepływem netto nazywamy wartość funkcji przepływu $f(v_i, v_j)$ pomiędzy wierzchołkami v_i oraz v_j. Zgodnie z warunkiem skośnej symetryczności, wartość może być dodatnia lub ujemna. Dodatnia oznacza przepływ z wierzchołka v_i do wierzchołka v_j, natomiast ujemna - w przeciwnym kierunku. Sieć przepływowa jest grafem skierowanym, więc dla dwóch wierzchołków może posiadać parę sąsiednich łuków. Poniższy przykład przedstawia istotę przepływu netto, jeżeli w obu sąsiadach istnieje niezerowy przepływ.



(a) (b) (c)

Rysunek 1: Zobrazowanie przepływu netto

Na rysunku 1a istnieje przepływ z wierzchołka 2 do wierzchołka 1 o wartości 10, czyli $f(2,1) = 10$. Zgodnie z warunkiem skośnej symetryczności $f(1,2) = f(2,1) = -10$. Można powiedzieć, że wierzchołek 2 opuszcza 10 jednostek materiału, które wpływają do wierzchołka 1. Na rysunku 1b pojawia się drugi przepływ, z wierzchołka 1 do wierzchołka 2 przepływa 30 jednostek. W tym przypadku wartość przepływu w istocie jest równa $f(1,2) = -10 + 30 = 20$. Jest to równoznaczne sytuacji, w której istnieje tylko przepływ 20 jednostek z wierzchołka 1 do wierzchołka 2. Przepływ z wierzchołka 2 do wierzchołka 1 zostaje zmniejszony do $f(2,1) = 10 - 30 = -20$, co jest zgodne z warunkiem zachowania skośności.

2.1.4 Maksymalny przepływ w sieci

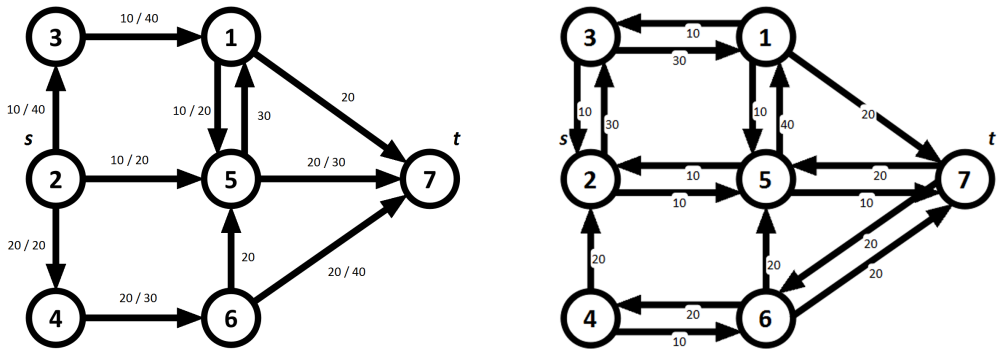
Wartość przepływu $|f|$ w sieci jest definiowana jako suma przepływów netto wychodzących ze źródła $\sum_{v \in V} f(s,v)$. Zgodnie z warunkiem zachowania przepływu, jest ona równa sumie wartości przepływów netto wpływających do ujścia $\sum_{v \in V} f(v,t)$. Problem maksymalnego przepływu polega na znalezieniu maksymalnej wartości $|f|$ w sieci przepływowej.[Agn12]

2.1.5 Sieć residualna

W celu wyznaczenia maksymalnego przepływu wprowadza się model *sieci residualnej*. To dodatkowa sieć przepływowa utworzona na podstawie pierwotnej. Aby zwiększyć przepływ w sieci obserwuje się stan łuków - jeżeli przepustowość łuku nie jest zapełniona jego przepływem netto, to oznacza, że przepływ może być zwiększony. Wprowadza się w tym celu pojęcie *przepustowości residualnej* $c_f(v_i, v_j)$, definiowanej jako różnica przepustowości i przepływu netto.

$$c_f(v_i, v_j) = c(v_i, v_j) - f(v_i, v_j)$$

Sieć residualną tworzy się w następujący sposób: dla każdego łuku (v_i, v_j) z sieci pierwotnej o przepływie f i przepustowości residualnej c_f tworzy się w sieci residualnej łuk (v_j, v_i) o przepustowości równej f oraz łuk (v_i, v_j) o przepustowości równej c_f. Jeżeli istnieje łuk sąsiedni dokonuje się ujednolicenia przepływu netto zgodnie z 2.1.3. W utworzonej tym sposobem sieci można zwiększyć przepływ jeżeli istnieje ścieżka ze źródła s do ujścia t.



(a) (b)

Rysunek 2: Przykładowa sieć przepływowa i utworzona na jej podstawie sieć residualna

Może się zdarzyć, że w sieci residualnej będą istnieć łuki, które nie istnieją w sieci pierwotnej lub będą posiadać przepustowość większą niż łuki odpowiadające w sieci pierwotnej. Należy wówczas w trakcie zwiększania przepływu wykorzystać *przepływ zwrotny*, wynikający z warunku skośnej symetryczności, i zmniejszyć wartość przepływu netto w danym łuku.

Ścieżką powiększającą p w sieci residualnej nazywamy zbiór łuków, który stanowi drogę ze źródła s do ujścia t, musi zawierać co najmniej jeden łuk. Maksymalną wartością o jaką można zwiększyć przepływ dzięki niej jest najmniejsza przepustowość residualna spośród wszystkich łuków należących do tej ścieżki.

$$c_f(p) = \min\{c_f(v_i, v_j) : \text{łuk } (v_i, v_j) \text{ należy do ścieżki } p\}$$

Jeżeli ścieżki powiększającej nie da się wyznaczyć w sieci residualnej, to oznacza, że aktualny przepływ f sieci pierwotnej jest maksymalny.

2.1.6 Warstwowa sieć residualna

Zoptymalizowana forma sieci residualnej, zapisujemy ją jako $G_f^w = (V_f^w, E_f^w)$. Idea tej sieci wychodzi z założenia, że wyszukiwanie ścieżki powiększającej powinno odbywać się po najkrótszych możliwych ścieżkach, więc pewne wierzchołki i łuki nie będą do sieci należały. Niech $d_{min}(v, u)$ będzie najkrótszą możliwą odległością między wierzchołkami v i u w sieci residualnej $G = (V, E)$ $(v, u \in V)$, liczoną w liczbie łuków pośredniczących. Z sieci residualnej G zostają usunięte:

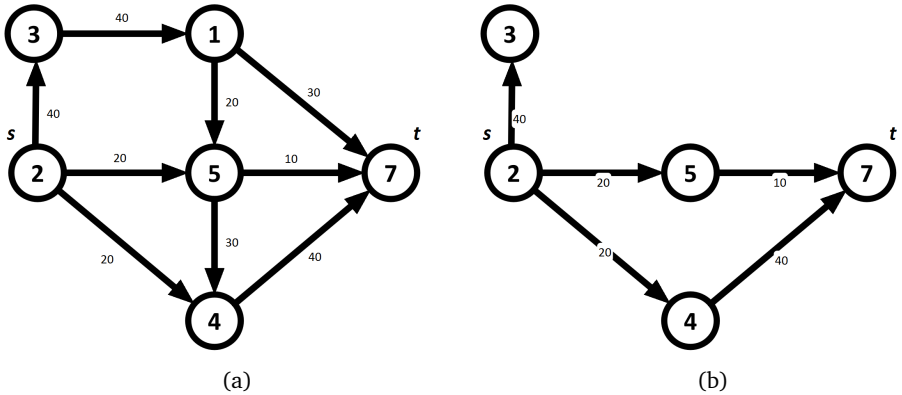
- Wszystkie wierzchołki inne niż źródło, dla których minimalna odległość ze źródła jest większa niż odległość między źródłem, a ujściem.

$$V_f^w = V \backslash \{v \in V \backslash \{t\} : d_{min}(s, t) \leq d_{min}(s, v)\}$$

- Łuki, dla których ścieżka ze źródła s do wierzchołka docelowego v_i jest mniejsza lub równa ścieżce ze źródła s do wierzchołka początkowego v_j.

$$E_f^w = E \backslash \{(v_j, v_i) \in E : d_{min}(s, v_i) \leq d_{min}(s, v_j)\}$$

Powyższe warunki zapewniają, że każda ścieżka ze źródła s do dowolnego innego wierzchołka jest najkrótszą możliwą ścieżką. Pierwszy warunek gwarantuje nadmiarowych wierzchołków, a drugi łuków, które nie są częścią żadnej z najkrótszych ścieżek.



(a) (b)

Rysunek 3: Zobrazowanie różnicy między siecią residualną, a warstwową siecią residualną

Rysunek 3 prezentuje jak wygląda utworzenie warstwowej sieci residualnej (3b). W sieci residualnej (rys. 3a) $d_{min}(s, t) = 2$, ponieważ najkrótszą drogą jest $s \rightarrow 5 \rightarrow t$. Zgodnie z pierwszym warunkiem został usunięty wierzchołek 1, ponieważ $d_{min}(s, 1) = d_{min}(s, t) = 2$, a wraz z nim wszystkie łuki, które z niego wychodzą lub do niego prowadzą. Usunięty został również łuk $(5, 4)$, ponieważ $d_{min}(s, 4) = 1$ oraz $d_{min}(s, 5) = 1$, czyli droga ze źródła s do wierzchołka 4 nie będzie najkrótsza, gdy będzie przechodzić przez wierzchołek 5.

2.2 Algorytmy znajdowania maksymalnego przepływu

2.2.1 Algorytm Forda-Fulkersona

Algorithm 1 Wyznaczenie maksymalnego przepływu algorytmem Forda-Fulkersona

procedure ZNAJDŹ MAKSYMALNY PRZEPŁYW(Sieć przepływowa $G = (V, E)$)
  Wykonaj sieć residualną G_f dla sieci przepływowej G
  while istnieje ścieżka powiększająca p w sieci G_f **do**
    Znajdź ścieżkę powiększającą p w sieci G_f
    $c_f(p) = min\{c_f(v_i, v_j) : \text{łuk } (v_i, v_j) \text{ należy do ścieżki } p\}$
    for all łuki $(v_i, v_j) \in p$ **do**
      $f[v_i, v_j] \mathrel{+}= c_f(p)$
      $f[v_j, v_i] = -f[v_i, v_j]$
    end for
    Wykonaj sieć residualną G_f dla sieci przepływowej G
  end while return f
end procedure

Idea wyznaczania maksymalnego przepływu oparta jest o iteracyjne tworzenie sieci residualnej i szukania w niej ścieżki powiększającej. W każdej iteracji:

- tworzona jest sieć residualna G_f na podstawie sieci przepływowej G i jej aktualnego przepływu,
- znajdowana jest ścieżka powiększająca p w sieci residualnej,
- wartością powiększającą $c_f(p)$ jest najmniejsza przepustowość wśród przepustowości łuków w ścieżce p,
- dla każdego łuku (v_i, v_j) ze ścieżki powiększającej p zwiększ przepływ w odpowiadającym łuku w sieci przepływowej G o wartość $c_f(p)$,
- dla każdego łuku sąsiadującego z łukiem (v_i, v_j) utwórz przepływ zwrotny.

W definicji algorytmu nie jest podany sposób wyznaczania ścieżki powiększającej, ponieważ nie ma on wpływu na wynik - przepływ zawsze jest maksymalny, z kolei wybór ma wpływ na czas wykonywania algorytmu. Przykładowe działanie algorytmu znajduje się w dodatku K.

2.2.2 Wykorzystanie przepływu blokującego

Przepływem blokującym f_b nazywamy taki przepływ sieci, dla którego każda ścieżka ze źródła do ujścia posiada co najmniej jeden łuk nasycony. ***Łukiem nasyconym*** nazywamy taki łuk (v_i, v_j), którego przepływu netto nie można już zwiększyć, $c(v_i, v_j) = f(v_i, v_j)$. Wykorzystanie *warstwowej sieci residualnej* 2.1.6 zapewnia, że każda ścieżka powiększająca w tej sieci będzie posiadać co najmniej jeden łuk nasycony. Dzięki temu można łatwo wyznaczyć przepływ blokujący w tej sieci szukając wszystkich możliwych ścieżek powiększających i składając je w jeden przepływ f_b. Zwiększenie przepływów netto w sieci pierwotnej traktujemy dokładnie tak samo jak dla zwykłej ścieżki powiększającej 2.1.3.

Działanie algorytmu poszukującego maksymalnego przepływu w sieci jest analogiczne do 2.2.1, ale zamiast sieci residualnej tworzy się warstwową sieć residualną, a zamiast ścieżki powiększającej - przepływ blokujący.

Algorithm 2 Wyznaczenie maksymalnego przepływu z wykorzystaniem przepływu blokującego

procedure ZNAJDŹ MAKSYMALNY PRZEPŁYW(Sieć przepływowa $G = (V, E)$)
  Wykonaj warstwową sieć residualną G_f^w dla sieci przepływowej G
  while istnieje ścieżka ze źródła do ujścia w sieci G_f^w **do**
    Znajdź przepływ blokujący f_b w sieci G_f^w
    Zwiększ przepływ f w sieci G o wartość f_b
    Wykonaj warstwową sieć residualną G_f^w dla sieci przepływowej G
  end while return f
end procedure

2.2.3 Algorytm Dinica

Ten algorytm wykorzystuje wyznaczanie maksymalnego przepływu za pomocą warstwowej sieci residualnej oraz przepływu blokującego, opisane w 2.2.2. Definiuje sposób wyznaczania przepływu blokującego.

Algorithm 3 Wyznaczenie przepływu blokującego algorytmem Dinica

procedure WYZNACZ PRZEPŁYW BLOKUJĄCY(Warstwowa sieć residualna $G_f^w = (V_f^w, E_f^w)$)
  while istnieje ścieżka powiększająca p w sieci G_f^w **do**
    Wyznacz ścieżkę powiększającą p w sieci G_f^w
    $c_f(p) = min\{c_f(v_i, v_j) : \text{łuk } (v_i, v_j) \text{ należy do ścieżki } p\}$
    for all łuki (v_i, v_j) należące do ścieżki p **do**
      $f_b[v_i, v_j] \mathrel{+}= c_f(p)$
      $c_f[v_i, v_j] \mathrel{-}= c_f(p)$
      if $c_f[v_i, v_j] = 0$ **then**
        Usuń łuk (v_i, v_j) z sieci G_f^w
      end if
    end for
    Usuń wierzchołki $v \in V_f^w$, których stopień wejściowy lub stopień wyjściowy jest równy 0
  end while
end procedure

Idea polega na usuwaniu w każdej iteracji łuków nasyconych z warstwowej sieci residualnej, a następnie wszystkich wierzchołków, które mają zerowy stopień wejściowy lub wyjściowy. Konstrukcja sieci zapewnia, że przy wyznaczaniu kolejnej ścieżki powiększającej w przepływie blokującym na pewno nie będzie ona przechodzić przez łuki nasycone oraz wierzchołki, które nigdzie nie prowadzą lub nie można do nich dojść, więc można je usunąć w celu przyspieszenia obliczeń. Przykładowe działanie algorytmu znajduje się w dodatku L.

2.2.4 Algorytm MKM

Ten algorytm, podobnie jak algorytm Dinica, również wykorzystuje wyznaczanie maksymalnego przepływu przy pomocy przepływu blokującego opisanego w 2.2.2, i jedynie definiuje sposób tworzenia przepływu blokującego. Jednak do tworzenia przepływu blokującego nie używa ścieżek powiększających, ale wykorzystuje pojęcie ***potencjału przepływowego wierzchołka*** $p_f(v)$.

$$p_f(v) = \min\{p_f^{we}(v), p_f^{wy}(v)\}, \quad p_f^{we}(v) = \sum_{u \in V} c(u, v), \quad p_f^{wy}(v) = \sum_{u \in V} c(v, u)$$

Przez $p_f^{we}(v)$ określony jest potencjał wejściowy wierzchołka v (suma przepustowości łuków wchodzących), a przez $p_f^{wy}(v)$ potencjał wyjściowy wierzchołka v (suma przepustowości łuków wychodzących). Potencjał przepływowy jest mniejszą z tych dwóch wartości, czyli określa maksymalną wartość przepływu, jaka może zostać przerzucona przez wierzchołek. Definiuje się ponadto, że $p_f^{we}(s) = \infty$ oraz $p_f^{wy}(t) = \infty$.

Algorithm 4 Wyznaczenie przepływu blokującego algorytmem MKM

procedure WYZNACZ PRZEPŁYW BLOKUJĄCY(Warstwowa sieć residualna $G_f^w = (V_f^w, E_f^w)$)
  Oblicz dla każdego wierzchołka $v \in V_f^w$ potencjały $p_f^{we}(v)$, $p_f^{wy}(v)$, $p_f(v)$
  repeat
    Znajdź wierzchołek $v \in V_f^w$ o najmniejszym potencjale przepływowym $p_f(v)$
    Utwórz ścieżkę $v \to t$, prześlij nią $p_f(v)$ jednostek i uaktualnij f_b
    Utwórz ścieżkę $s \to v$, prześlij nią $p_f(v)$ jednostek i uaktualnij f_b
    Oblicz dla każdego wierzchołka $v \in V_f^w$ potencjały $p_f^{we}(v)$, $p_f^{wy}(v)$, $p_f(v)$
    Usuń z sieci G_f^w wszystkie wierzchołki $v \in V_f^w$ dla których $p_f(v) = 0$
  until $s \notin V_f^w$ lub $t \notin V_f^w$ **return** f_b
end procedure

W każdej iteracji w warstwowej sieci residualnej G_f^w poszukuje się wierzchołka v o najmniejszym potencjale przepływowym. Następnie tworzy się dwie ścieżki: prowadzącą ze źródła s do wierzchołka v oraz

z wierzchołka v do ujścia t, które razem tworzą ścieżkę powiększającą przechodzącą przez wierzchołek v. Następuje zwiększenie przepływu w G_f^w, a następnie usunięcie wszystkich wierzchołków o zerowym potencjale przepływowym i w konsekwencji łuków wchodzących i wychodzących z tych wierzchołków. Algorytm kończy pracę gdy zostanie usunięte źródło s lub ujście t. Przykładowe działanie algorytmu znajduje się w dodatku M.

2.3 Wykorzystane technologie

2.3.1 Język programowania

Aplikacja ma mieć charakter edukacyjny, więc na pierwszym planie nie stała efektywność wykonywania samych algorytmów, ale sposób prezentacji ich działania. Użytkownik korzystający z aplikacji powinien móc odwzorować w niej książkowy przykład sieci przepływowej oraz móc zaobserwować jak każdy z interesujących go algorytmów wykonuje swoją pracę krok po kroku, jakie wprowadza zmiany w jego sieci i dlaczego. W tym celu zdecydowałem się na platformę graficzną ***Qt*** oraz język programowania ***C++***. Qt oferuje możliwość tworzenia prostych kształtów geometrycznych w grafice wektorowej; sieć tworzona przez użytkownika będzie tak samo czytelna w dowolnym przybliżeniu. Framework ten umożliwia również proste i wygodne w użyciu tworzenie okręgów, linii, łączenie ich w większe konstrukcje, zmiany parametrów, obsługę zdarzeń itp. Dzięki niemu da się stworzyć pole edycyjne rysunku podobne do tych z takich aplikacji desktopowych jak *Inkscape* lub *Photoshop* oraz wygodne dla użytkownika tworzenie sieci przepływowych przy pomocy kilku prostych narzędzi. Wybrana przez mnie wersja Qt to *Open Source*.

2.3.2 Środowisko deweloperskie

Decydując się na język C++ i platformę Qt miałem możliwość pracy w dwóch środowiskach IDE: *Visual Studio 2013* z wtyczką do Qt oraz *Qt Creator*. Zdecydowałem się na środowisko ***Visual Studio 2013***, ponieważ w trakcie pracy chciałbym móc korzystać ze wszystkich dobrodziejstw kontroli kodu i debugowania, jakie oferuje produkt firmy Microsoft. Nie mogłem korzystać z najnowszego Visual Studio 2015, ponieważ w tym czasie wtyczka zespalająca VS z Qt, ***Qt Visual Studio Add-in 1.2.4***, nie obsługiwała jeszcze tej wersji. Konsekwentnie moja aplikacja została napisana w ***Qt 5.5*** oraz skompilowana pod kompilatorem ***MSVC v12.0 x64*** na systemy Windows. Przez pewien czas nad rozwojem aplikacji pracowałem z wtyczką ***Resharper C++*** (wersja próbna), która znacznie usprawniała pracę w środowisku Visual Studio. Tworząc okna i okienko dialogowe, z których korzysta aplikacja, korzystałem dodatkowo z ***Qt Designer***a, który dawał możliwość komfortowego projektowania GUI w na tę platformę.

2.3.3 Format plików przy serializacji

Jednym z podstawowych wymagań aplikacji jest możliwość zapisu i odczytu sieci przepływowych z plików zewnętrznych. Zdecydowałem się na zapisywanie ich do formatu ***XML*** bez wykorzystania funkcji haszujących. Format jest bardzo powszechny i posiada wsparcie w większości języków programowania, czy to w postaci bibliotek czy dzięki frameworkowi. Format pliku jest czytelny zarówno dla maszyny, jak i dla człowieka. Zmiany można wprowadzać edytując sam plik, dzięki temu możliwe jest wprowadzanie drobnych zamian zarówno przez użytkownika, który chciałby wprowadzić pewne zmiany bez otwierania aplikacji, jak i dla dewelopera, który chciałby testować i debugować w ten sposób działanie kodu parsującego.

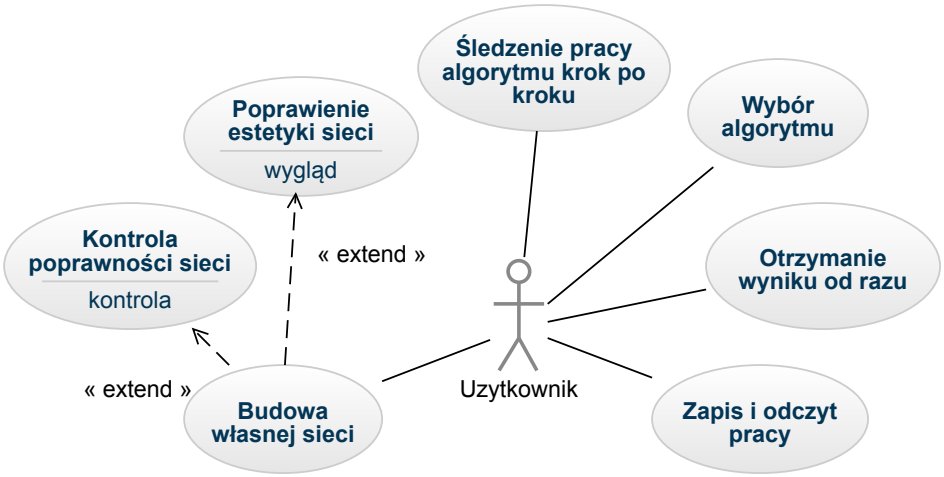
2.3.4 Kontrola wersji

Pracując nad aplikacją chciałem mieć repozytorium kodu, na którym trzymałbym kod źródłowy swojej pracy oraz mógł obserwować i kontrolować postęp pracy, jak również móc wrócić do poprzednich wersji w przypadku popełnienia błędów. Zdecydowałem się na repozytorium w systemie ***Git***, najlepiej mi znanym środowisku kontroli kodu, z wykorzystaniem klienta ***Bitbucket***, z którymi pracuję na co dzień.

Rozdział 3

Wymagania

Aplikacja ma na celu pomóc użytkownikowi zrozumieć i zobrazować działanie algorytmów na sieciach przepływowych. Student, który przygotowuje się do kolokwium z Algorytmów powinien móc utworzyć własną sieć przepływową, prześledzić działanie wybranych algorytmów krok po kroku oraz móc zapisać postęp swojej pracy. Aplikacja ma mieć charakter edukacyjny, umożliwić jak najlepsze przyswojenie omawianych w pracy algorytmów. Wymagania zostały przedstawione w postaci przypadków użycia.



Rysunek 4: Funkcjonalności wymagane przez potencjalnego użytkownika

Ponadto aplikacja powinna zapewnić niezawodność wykonywanych algorytmów. Jeżeli sieci zbudowane przez użytkownika są niepoprawne i nie spełniają założeń, nie powinno dać się wykonywać algorytmów. Użytkownik powinien zostać poinformowany jakie dokładnie błędy popełnił w budowie i co dokładnie należy zrobić by je wyeliminować. Jeżeli siec jest poprawna, algorytm powinien móc dać się wykonać, a jego proces być łatwo kontrolowany przez użytkownika, najlepiej przy pomocy kilku przycisków. Wszystkie zmiany, jakie wprowadził algorytm w danym kroku powinny zostać opisane oraz podświetlone na grafie podglądowym, jeżeli jest to możliwe.

- Algorytm Forda-Fulkersona operuje na sieci przepływowej i sieci residualnej, więc wymagane są dwa poglądy zmian,
- algorytmy Dinica i MKM ponadto operują na przepływie blokującym, więc wymagane jest utworzenie trzech podglądów zmian.

Czas oczekiwania na wykonanie pojedynczego kroku algorytmu powinien być krótki, nieprzekraczający kilku sekund, aby nie nadużywać cierpliwości użytkownika. Aplikacja powinna umożliwiać śledzenie pracy algorytmu zarówno przez małe kroki, jak i skok do końca działania procesu i otrzymania wyniku natychmiastowo, np. w celu porównania go z obliczeniami na kartce.

Rozdział 4

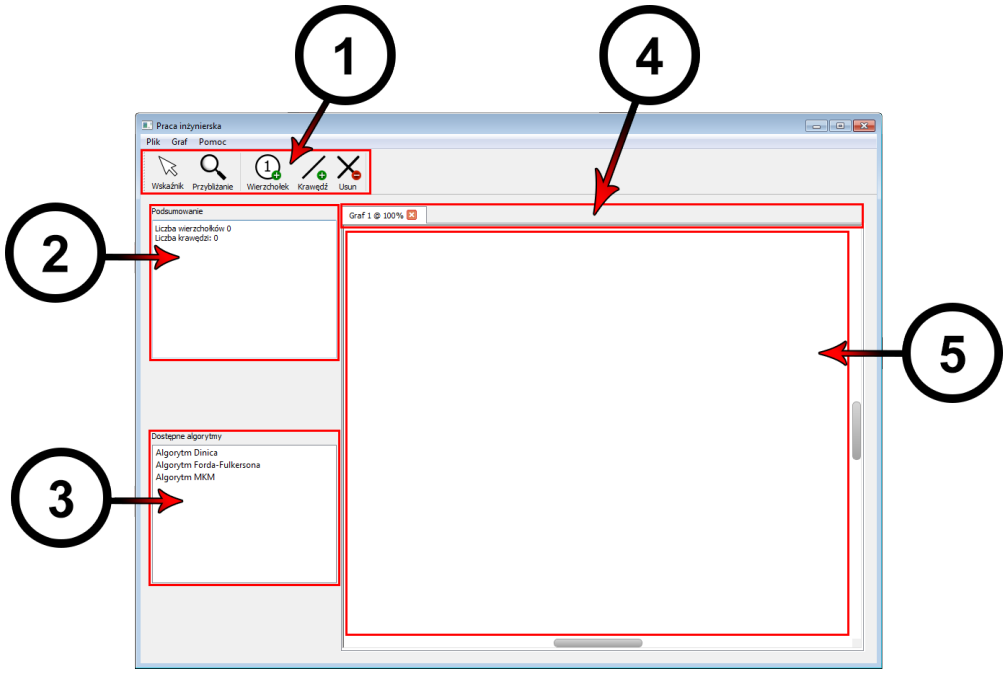
Specyfikacja zewnętrzna

4.1 Wymagania sprzętowe

Aplikacja działa pod systemami Windows 7 i nowszymi, na architekturze 64-bitowej. Instalacja nie jest wymagana, wystarczy otworzyć plik aplikacji .exe by móc z niej korzystać. Wszystkie biblioteki i pliki konfiguracyjne nie mogą zostać zmienione.

4.2 Korzystanie z aplikacji

Po uruchomieniu programu pojawia się okno z interfejsem graficznym złożonym z kilku podstawowych sekcji.

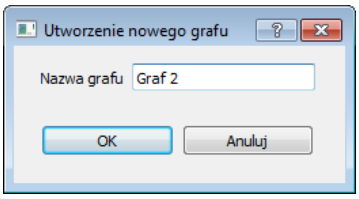


Rysunek 5: Interfejs graficzny

1. *Pasek narzędziowy* obejmuje pięć narzędzi do pracy z siecią przepływową: wskaźnik, lupę, dodanie grafu, dodanie krawędzi i usunięcie elementu.
2. *Podsumowanie* zawiera informacje na temat aktywnej w danej chwili sieci.
3. *Dostępne algorytmy* zawierają listę operacji, jakie można na sieci wykonać.
4. *Pasek kart* zawiera listę istniejących sieci w instancji aplikacji między którymi można się przełączać.
5. *Pole edycji grafu* to miejsce robocze, gdzie można umieszczać elementy grafu i pracować nad nim.

4.2.1 Tworzenie nowej sieci

Aby utworzyć nową sieć należy z paska menu wybrać *Plik* → *Nowy* lub skorzystać ze skrótu *CTRL* + *N*. Następnie pojawi się okienko w które należy wpisać roboczą nazwę sieci (można też pozostawić wygenerowaną domyślną nazwę). Po zatwierdzeniu zostanie dodana nowa karta z pustym polem edycyjnym.



Rysunek 6: Okienko z wyborem nazwy

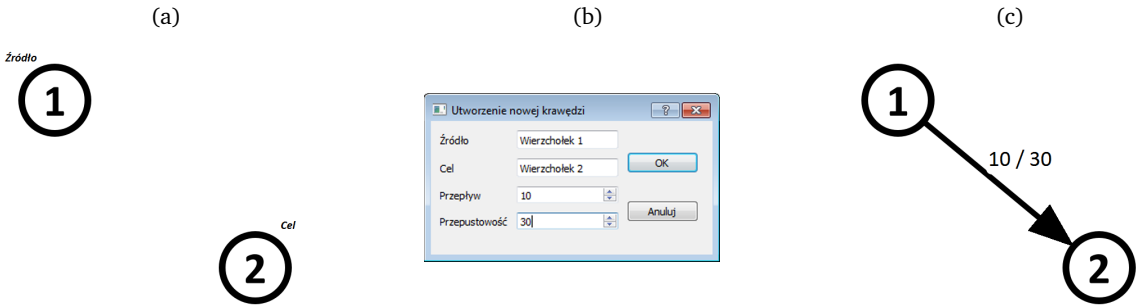
4.2.2 Rysowanie grafu

Dodanie wierzchołka

Aby dodać nowy wierzchołek należy wybrać narzędzie dodawania wierzchołków, a następnie kliknąć lewym przyciskiem myszy na puste miejsce w polu edycyjnym. Pojawi się nowy obiekt, miejsce kliknięcia będzie środkiem wierzchołka. Jeżeli klikniemy istniejący już obiekt, zostanie on zaznaczony.

Dodanie łuku

Aby móc dodać łuk na polu roboczym muszą istnieć co najmniej dwa wierzchołki. Należy wybrać narzędzie dodawania krawędzi i w pierwszej kolejności kliknąć na wierzchołek który będzie źródłem. Pojawi się nad nim etykieta "*Źródło*". Potem należy kliknąć drugi wierzchołek, po najechaniu myszką pojawi się nad nim etykieta "*Ujście*". Kliknięcie lewym przyciskiem myszy spowoduje potwierdzenie operacji i pojawienie okienka, gdzie należy wpisać przepływ i przepustowość tego łuku, domyślnie zero i jeden. Jeżeli między wierzchołkami już istnieje dane połączenie, operacja zostanie zignorowana. Łuk jest trwało przymocowany do wierzchołków i nie można zmienić jego pozycji. Etykieta zawiera informację w formacie "*przepływ / przepustowość*".



Rysunek 7: Etapy dodawania łuku

Usuwanie elementów

W każdej chwili można usunąć wierzchołek lub łuk w sieci. W tym celu należy wybrać narzędzie usuwania oraz kliknąć wybrany element. Usunięcie łuku powoduje pozbycie się z sieci tylko niego, z kolei usunięcie wierzchołka powoduje zlikwidowanie wszystkich łuków, które wchodzą lub wychodzą z niego. Można usuwać wiele elementów naraz, narzędzie usuwa wszystko co jest zaznaczone.

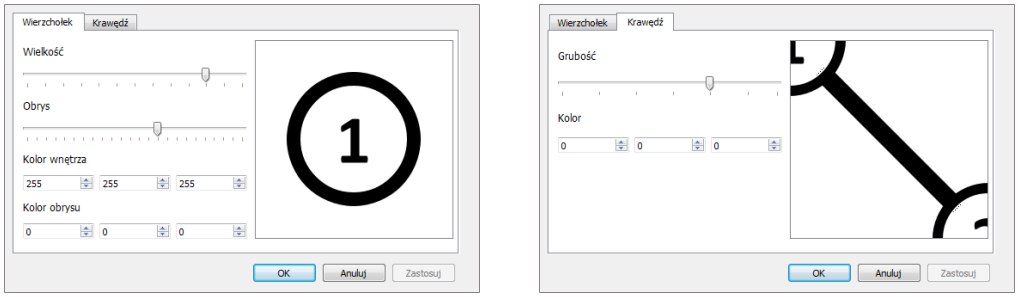
Poprawa estetyki sieci

Aby poprawić wygląd i czytelność sieci istnieje możliwość wyrównania pozycji wierzchołków. W tym celu należy zaznaczać dwa lub więcej wierzchołków, a następnie kliknąć *SHIFT* + jedną ze strzałek kierunkowych (↑, ↓, →, ←). Wszystkie wierzchołki w sieci posiadają te same rozmiary i komendy należy interpretować następująco:

- *SHIFT* + ↑ lub ↓ - ustawia pozycję pionową wszystkich wierzchołków równą pozycji wierzchołka najbardziej wysuniętego do góry lub na dół.
- *SHIFT* + → lub ← - ustawia pozycję poziomą wszystkich wierzchołków równą pozycji wierzchołka najbardziej wysuniętego na prawo lub na lewo.

Konfiguracja wyglądu grafu

Rozmiar i kolory wierzchołków oraz łuków można modyfikować w ustawieniach. W tym celu należy wybrać z paska menu *Graf* → *Kształt* lub skorzystać ze skrótu klawiszowego *CTRL+SHIFT+K*. Pojawi się okienko z konfiguracją lokalnego grafu.



(a) Konfiguracja wierzchołków (b) Konfiguracja łuków

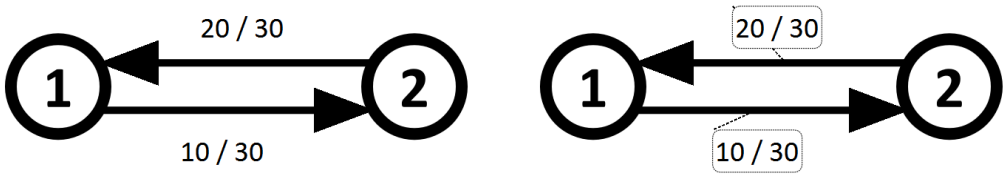
Rysunek 8: Możliwe zmiany w wyglądzie sieci

Oba okienka umożliwiają zmianę koloru i rozmiaru, w przypadku wierzchołka możliwa jest ponadto kontrola nad grubością obrysu oraz jego koloru. Kliknięcie przycisku *OK* lub *Zastosuj* skutkuje zapisaniem konfiguracji i aktualizacją wyglądu. Każda sieć posiada swoją lokalną konfigurację, w przypadku zapisywania jej do pliku, ona również jest zachowywana.

Operowanie na sieci

Wybierając narzędzie wskaźnika możliwe jest przesuwanie wierzchołków po polu edycyjnym oraz ich zaznaczanie. Można zaznaczyć wiele wierzchołków i łuk przytrzymując klawisz *CTRL* lub skorzystać z myszki - wystarczy kliknąć lewym przyciskiem na pustym polu, przytrzymać i przeciągnąć kursor by pojawił się prostokąt zaznaczający elementy sieci. Jeżeli jest wiele zaznaczonych elementów przesunięcie jednego z nich spowoduje przesunięcie wszystkich.

Zaznaczenie etykiety łuku, informującej o wartościach przepływu i przepustowości, spowoduje pojawienie się linii przerywanej prowadzącej do środka łuku. Ułatwia to pracę z siecią, gdy łuków i etykiet pojawia się coraz więcej.



(a) Niezaznaczone etykiety (b) Zaznaczone etykiety

Rysunek 9: Przedstawienie linii łączącej etykietę z łukiem

Zmiana parametrów łuku

Nad każdym łukiem znajduje się etykieta, która zawiera informację o wartościach kolejno przepływu i przepustowości oddzielone ukośnikiem. Jeżeli w etykiecie znajduje się jedna liczba, oznacza ona przepustowość. Wówczas wartość przepływu jest zerowa i dla czytelności nie jest wyświetlany. Te dwie wartości można w każdej chwili zmienić klikając dwukrotnie lewym przyciskiem myszy na etykietę. Pojawi się pole edycji, gdzie można wprowadzić nowe wartości. Akceptowane są wyłącznie formaty $< liczba >$

oraz $< liczba >$ / $< liczba >$. Zmianę zatwierdza się kliknięciem myszy. Nowy łańcuch w poprawnym formacie spowoduje aktualizację wartości i etykiety. Wprowadzenie czegokolwiek innego powoduje przywrócenie pierwotnych wartości.

4.2.3 Wykonanie algorytmów

W momencie utworzenia karty z siecią, nowym lub wczytanym z pliku, w lewym dolnym rogu pojawia się lista dostępnych algorytmów, jakie można wykonać w ramach programu: Forda-Fulkersona, Dinica oraz MKM.

Warunki poprawności sieci

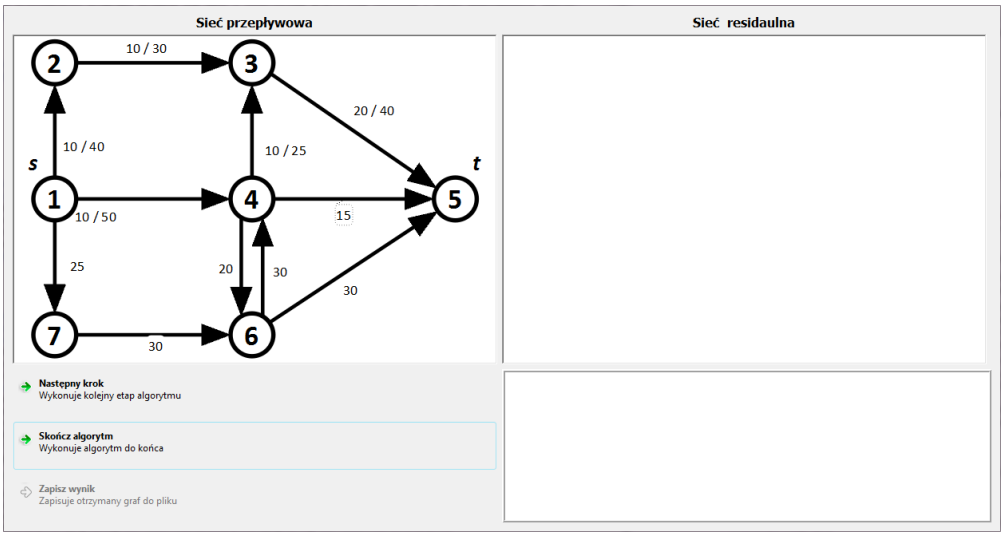
Aby uruchomić algorytm sieć musi być poprawnie zbudowana, tzn. spełniać następujące założenia:

- wierzchołek źródłowy oraz wierzchołek ujściowy muszą być oznaczone, by móc wyznać maksymalny przepływ. W tym celu należy kliknąć prawym przyciskiem myszki na wierzchołkach i wybrać z menu kontekstowego odpowiednio *oznacz jako źródło* oraz *oznacz jako ujście*. Nad wierzchołkiem źródłowym pojawi się etykieta ***s***, a nad wierzchołkiem ujściowym etykieta ***t***.
- Nie może istnieć żaden wierzchołek, inny niż źródło i ujście, przez który nie istnieje droga ze źródła do ujścia. Ma to na celu zapewnić, że w zbudowanym grafie istnieje co najmniej jedna ścieżka ze źródła do ujścia. Wystarczy, że dla każdego wierzchołka będą istniały co najmniej dwa łuki, wchodzący i wychodzący z niego.
- Zasada zachowania przepływu nie może zostać złamana. Program nie dopuszcza, żeby łuk mógł mieć większą wartość przepływu niż przepustowość, ale podczas budowania sieci nie można cały czas kontrolować tej zasady. Jest ona sprawdzana dopiero przy próbie uruchomienia algorytmu.

Jeżeli któryś z tych warunków nie zostanie spełniony, pojawi się okienko informujące dokładnie o rodzaju błędu oraz w którym miejscu grafu on wystąpił.

Okno wykonania

Jeżeli graf został poprawnie zbudowany, po kliknięciu w jeden z algorytmów pojawi się nowe okno w którym zostanie on wykonany.

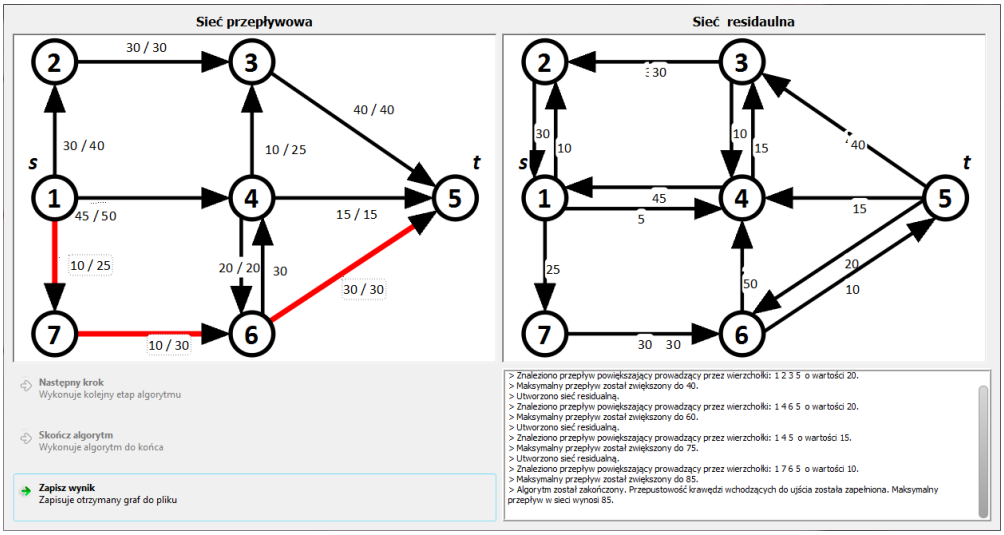


Rysunek 10: Okno ilustrujące przebieg algorytmu Forda-Fulkersona

Okno zostało podzielone na 4 obszary:

- W górnym lewym rogu znajduje się pierwotna sieć przepływowa, której przepływ zostaje zwiększany wraz z kolejnymi krokami.

- W górnym prawym rogu jest podgląd tworzonej sieci residualnej oraz ścieżki powiększającej, jaka została w niej znaleziona. W oknie dla algorytmów Dinica i MKM znajduje się jeszcze dodatkowy, trzeci podgląd, obrazujący zmiany w warstwowej sieci residualnej.
- W dolnym lewym rogu znajdują się trzy przyciski:
  - *Następny krok*, jego naciśnięcie powoduje wykonanie istotnej instrukcji algorytmu i przedstawienie zaistniałych zmian w podglądach. *Skończ algorytm*, realizuje całość zadania, aż maksymalny przepływ zostanie znaleziony. Aplikacja wykonuje wówczas instrukcję *Następny krok* w odstępach równych 0.5 sekundy, aż do zakończenia obliczeń.
  - *Zapisz wynik*, przycisk staje się uaktywniony po zakończeniu zadania, umożliwia zapisanie sieci ze znalezionym maksymalnym przepływem do pliku XML.
- W dolnym prawym rogu znajduje się konsola w której pojawiają się informacje o przebiegu algorytmu. Za każdym razem, gdy instrukcja *Następny krok* zostanie wykonana, w tym polu pojawiają się szczegółowe informacje o utworzonych strukturach, znalezionych przepływach i innych zmianach w sieci.



Rysunek 11: Okno po zakończeniu wykonywania algorytmu

Przykładowe przedstawienie krok po kroku wyszukiwania maksymalnego przepływu znajduje się w dodatku K.

4.2.4 Otwieranie i zapisywanie

Aby móc zapisać sieć przepływową do pliku w sesji programu musi być otwarta co najmniej jedna karta. Klikając *CTRL+S* lub wybierając *Plik* → *Zapisz jako* można zapisać stan pracy do pliku XML. Program poprosi o wskazanie folderu oraz wpisanie nazwy pliku. Potwierdzenie utworzy nowy plik. Aby otworzyć plik należy wybrać z paska menu *Plik* → *Otwórz* lub skorzystać ze skrótu klawiszowego *CTRL+O*. Program otworzy okno dialogowe i poprosi o wskazanie pliku XML jaki ma otworzyć. Jeżeli ma on poprawny format, sieć przepływowa zostanie wczytana i pokazana w nowej karcie.

Rozdział 5

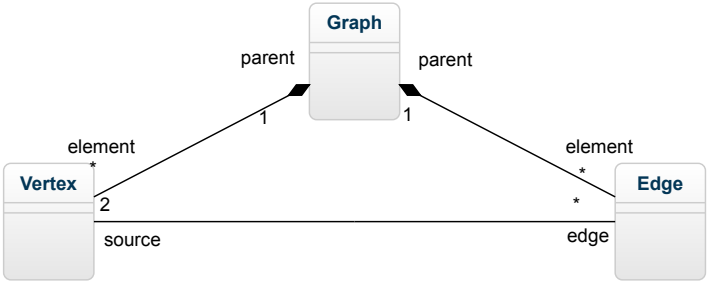
Specyfikacja wewnętrzna

5.1 Struktura danych

Reprezentację grafów w pamięci komputera można wykonać na dwa sposoby:

- *niskopoziomowy*: grafem jest macierz $n \times n$ liczb całkowitych, gdzie n to liczba wierzchołków.[Zbi10a]
- *wysokopoziomowy*: wierzchołki i łuki grafu są zakapsułkowane do osobnych klas.[Agn12]

W swojej pracy wybrałem drugie podejście. Poprawne zdefiniowanie klas i zależności między nimi pozwala mi w intuicyjny sposób implementować algorytmy z pseudokodu: operacje rodzaju "*wykonaj daną instrukcję na wszystkich krawędziach*" można łatwo odwzorować pobierając kolekcję krawędzi i dodając im odpowiednią metodę. Nie ma potrzeby adaptować operacji na zupełnie inną strukturę.



Rysunek 12: Diagram klas struktury sieci

Zgodnie z definicją w 2.1.1, graf (*Graph*) składa się z dwóch zbiorów, wierzchołków (*Vertex*) oraz łuków (*Edge*). Graf jest kompozytem, wierzchołki i łuki stanowią jego ciało, a usunięcie obiektu grafu powoduje usunięcie jego wszystkich podrzędnych elementów. Wierzchołek jest niezależnym elementem, z kolei łuk posiada referencje do dwóch obiektów klasy wierzchołka - łuk nie może istnieć bez pary wierzchołków, które go tworzą, a usunięcie wierzchołka powoduje usunięcie wszystkich łuków wychodzących i wchodzących do niego.

5.2 Konfiguracja wyglądu sieci

Każdy obiekt sieci przepływowej posiada swoją konfigurację, w której znajdują się informacje dotyczące rysowania sieci. Każda sieć musi posiadać istniejącą konfigurację, jest to wymuszone poprzez jej konstruktor.

```
explicit FlowNetwork(GraphConfig * config);
```

Wierzchołki i łuki sieci przepływowej podczas wydarzenia rysowania pobierają swoje ustawienia z rodzica. Klasa `GraphConfig` zawiera w sobie pięć składowych: nazwę sieci oraz cztery konteksty - sposób rysowania zwykłego elementu oraz zaznaczonego.

```
class GraphConfig
{
protected:
    QString _name;
    VertexContext * _normalVertexContext;
    EdgeContext * _normalEdgeContext;
    VertexContext * _selectedVertexContext;
    EdgeContext * _selectedEdgeContext;
}
```

Informacje o sposobie rysowania wierzchołka są zawarte w klasie `VertexContext`, a łuki w klasie `EdgeContext`. Idea oparta jest o wzorzec ***Pyłku***. Każdy obiekt wierzchołka i łuku posiada swój niezmienny stan wewnętrzny, jak identyfikator oraz pozycję, a także stan zewnętrzny, jakim jest kolor i kształt[Eri10]. Kontekst oznacza właśnie stan zewnętrzny, którymi grafy mogą się różnić. Każdy wierzchołek i łuk posiada swój wskaźnik na aktualny kontekst, a w momencie, gdy następuje kliknięcie w elementu graf i mechanizm *Qt* wykrywa zmianę zaznaczenia, wskaźnik wskazuje na przeciwny kontekst.

```
QVariant VertexImage::itemChange(GraphicsItemChange change, const QVariant &value)
{
    // ...
    else if (change == ItemSelectedChange)
    {
        if (value.toBool() == true)
        {
            _context = getParent()->getConfig()->SelectedVertexContext();
        }
        else
        {
            _context = getParent()->getConfig()->NormalVertexContext();
        }
    }
}
```

5.3 Serializacja sieci

Zapis oraz odczyt plików z grafami odbywa się w klasie `GraphSerializer`. Posiada ona dwie publiczne metody:

- `void serialize(GraphImage const & graph, std::string const & fileName);`
  Metoda *serialize* przyjmuje jako parametr referencję do graficznej reprezentacji grafu, który ma zapisać oraz nazwę pliku wyjściowego wraz ze ścieżką.
- `GraphImage * deserialize(std::string const & filePath);`
  Metoda *deserialize* przyjmuje jako parametr ścieżkę do pliku, który ma odczytać. Zwraca wskaźnik do graficznej reprezentacji grafu odczytanego z pliku.

Sieci przepływowe są zapisywane do formatu XML. W celu obsługi tego języka, klasa `GraphSerializer` korzysta z darmowej biblioteki do parsowania plików XML, *RapidXML*, autorstwa Marcina Kalicińskiego[1]

5.3.1 Format pliku XML

Plik zaczyna się od korzenia o nazwie `<Graph>`. Format pliku można podzielić na dwie części, konfiguracyjną (gałąź `<Config>`) i modelową (gałąź `<Model>`). W pierwszej znajdują się globalne ustawienia wyglądu, jak rozmiary i kolory wierzchołków i łuków. Druga część zawiera model sieci przepływowej - zarówno pozycje i numery wierzchołków, jak i ich połączenia poprzez krawędzie, ich kierunek, przepustowość, przepływ oraz położenie napisu z informacją. Przykładowa sieć przepływowa oraz jej serializacja do pliku znajduje się w dodatku A.

[1] Strona domowa *RapidXML*: `http://rapidxml.sourceforge.net/`

5.4 Algorytmy

Wszystkie trzy algorytmy wykonane w tej pracy zostały zakapsułkowane do osobnych klas. Ich klasą bazową bazową jest `FlowNetworkAlgorithm`, która definiuje interfejs algorytmów oraz zawiera wspólną dla wszystkich funkcjonalność.

5.4.1 Tworzenie sieci residualnej

Algorytm służący do wygenerowania sieci residualnej jest niezmienny dla wszystkich klas i jest zawarty w metodzie klasy `FlowNetworkAlgorithm`.

```
int makeResidualNetwork(FlowNetwork * network, FlowNetwork *& outResidaulNetwork)
```

Funkcja przyjmuje dwa parametry:

- *network* jest siecią przepływową na podstawie której jest utworzona sieć przepływowa
- *outResidaulNetwork* jest referencją na wskaźnik do obiektu sieci, która będzie siecią residualną. W pierwszym kroku jest to głęboka kopia sieci przepływowej.

Wartością zwracaną jest najkrótsza odległość między źródłem, a ujściem, jeżeli tworzona sieć jest *warstwową siecią residualną*. Jeżeli nie jest, funkcja zwraca zero. W pierwszym kroku, algorytm usuwa wszystkie łuki z wyjściowej sieci residualnej zostawiając same wierzchołki. Następnie przechodzi przez wszystkie łuki sieci przepływowej i wykonuje poniższy algorytm. Utworzenie tablicy odwiedzonych łuków ma za-

Algorithm 5 Tworzenie nowego łuku w sieci residualnej

procedure UTWÓRZ NOWY ŁUK(Łuk w sieci przepływowej)
  Oblicz przepustowość residualną
  if Istnieje łuk sąsiedni oraz łuk sąsiedni nie został odwiedzony **then**
    Oblicz przepływy między wierzchołkami zgodnie z zachowaniem przepływu netto w 2.1.3
    Dodaj łuk oraz łuk sąsiedni do odwiedzonych
  end if
  if Przepływ w łuku $\neq 0$ **then**
    Utwórz nowy łuk w sieci residualnej w tym samym kierunku o przepustowości równej przepływowi
  end if
  if Przepustowość residualna $\neq 0$ **then**
    Utwórz nowy łuk w sieci residualnej w przeciwnym kierunku o przepustowości równej przepustowości residualnej
  end if
end procedure

pewnić, że w sieci residualnej nie zostaną utworzone nadmiarowe łuki gdy pętla dojdzie do sąsiada. Pełny kod algorytmu znajduje się dodatku B.

5.4.2 Szukanie ścieżki powiększającej

Podczas realizacji algorytmów wymagane jest znalezienie pewnej ścieżki pomiędzy źródłem, a ujściem, jednak w literaturze ([Agn12],[Tho09]) nie jest podany żaden konkretny sposób. Udowodniono, że wybór algorytmu do szukania ścieżki nie ma znaczenia na wynik działania omawianych algorytmów - uzyskana wartość przepływu zawsze jest maksymalna. W mojej pracy zastosowałem algorytm poszukiwania ścieżek własnego pomysłu, który zaimplementowałem w uogólnionej metodzie poszukującej drogi pomiędzy dwoma dowolnymi wierzchołkami sieci przepływowej.

```
QList<EdgeImage*> FlowNetworkAlgorithm::findPathBetween(FlowNetwork * network,
↪  VertexImage * from, VertexImage * to)
```

Funkcja przyjmuje trzy parametry:

- *network* jest siecią przepływową, która jest przeszukiwana.
- *from* jest wierzchołkami startowym, z którego rozpoczynane jest szukanie ścieżki.
- *to* jest wierzchołkiem docelowym, do którego dąży algorytm.

Wartością zwracaną jest lista wskaźników na łuki w sieci, która stanowi reprezentację znalezionej ścieżki. Jeżeli lista jest pusta, ścieżka między wierzchołkami nie istnieje.

Algorithm 6 Poszukiwanie ścieżki między wierzchołkami

```
procedure ZNAJDŹ ŚCIEŻKĘ(Sieć, wierzchołek startowy, wierzchołek docelowy)
    Aktualnym wierzchołkiem jest startowy
    while Szukanie się nie zakończyło do
        for all łuk wychodzący z aktualnego wierzchołka do
            if jeżeli łuk nie prowadzi do źródła, odrzuconego lub odwiedzonego wierzchołka then
                Dodaj łuk do listy możliwych łuków
            end if
        end for
        if lista możliwych łuków jest pusta then
            if ostatni łuk jest nullem then return ścieżka
            else
                Dodaj aktualny wierzchołek do odrzuconych
                Usuń łuk z listy możliwych
                if ścieżka nie jest pusta then
                    Cofnij się do poprzedniego wierzchołka
                else
                    Aktualny łuk = null                    ▷ Aktualnym wierzchołkiem jest startowy
                end if
            end if
        end if
        Wylosuj jeden z możliwych łuków
        Wierzchołek docelowy tego łuku jest aktualnym
        Dodaj aktualny wierzchołek do odwiedzonych
        Dodaj łuk do ścieżki
        Ostatnim łukiem jest aktualny
        if aktualny wierzchołek jest docelowym then
            zakończ algorytm
        end if
    end while return ścieżka
end procedure
```

Algorytm jest bardzo prosty. Zaczynając od pierwszego wierzchołka, losowany jest jeden z jego łuków. Jeżeli łuk prowadzi do wierzchołka docelowego, koniec. Jeżeli nie, przechodzimy do tego wierzchołka i losujemy kolejny łuk, tak długo aż cel zostanie znaleziony. W przypadku gdy dalsza droga nie istnieje, następuje cofnięcie się do poprzedniego wierzchołka, aktualny wierzchołek oraz droga do niego zostają oznaczone jako odrzucone. Następnie losowany jest kolejny łuk z dostępnych, jeżeli już ich nie ma, następuje kolejne cofnięcie o wierzchołek. Jeżeli algorytm powróci do wierzchołka startowego, to ścieżka między wierzchołkami nie istnieje i zostaje zwrócona pusta lista. Aby algorytm nie odrzucił pierwszej pętli, sprawdzane jest dodatkowo czy ostatni wybrany łuk nie jest wartością null. Pełny kod algorytmu znajduje się dodatku C.

Dzięki swojej losowości, za każdym razem droga między wierzchołkami, wyznaczona przez program, będzie inna. Dzięki temu ścieżki powiększające i przepływy blokujące, w trakcie poszukiwania maksymalnego przepływu, również mogą się różnić przy rożnych wykonaniach algorytmu dla tej samej sieci. Nie ma tutaj jedynej słusznej drogi, jest jedynie jeden poprawny wynik.

5.4.3 Zwiększenie przepływu w sieci

Algorytm zwiększania przepływu jest wspólny dla wszystkich algorytmów znajdowania maksymalnego przepływu.

```
void FlowNetworkAlgorithm::increaseFlow(FlowNetwork *& network,
↪  QList<EdgeImage*> const & path, int increase)
```

Funkcja przyjmuje trzy parametry:

- *network* jest siecią przepływową, gdzie przepływ zostanie zwiększony
- *path* to zbiór łuków w których przepływ zostanie zwiększony
- *increase* jest wartością o jaką przepływy zostaną zwiększone

Algorithm 7 Zwiększenie przepływu w sieci

```
procedure ZWIĘKSZ PRZEPŁYW(Sieć, ścieżka, wartość powiększająca)
    for all łuk w ścieżce do
        if nie istnieje odpowiadający łuk w sieci then
            Utwórz przepływ zwrotny w sąsiednim łuku
        else if istnieje odpowiadający łuk w sieci, ale posiada łuk sąsiedni then
            Wykonaj optymalizację przepływu netto zgodnie z 2.1.3
        else                                          ▷ łuk istnieje i nie ma sąsiada
            Zwiększ przepływ w odpowiadającym łuku w sieci o wartość powiększającą
        end if
    end for
end procedure
```

Algorytm musi uwzględniać trzy przypadki. Najprostszy występuje wtedy, gdy sieć posiada odpowiadający łuk, a ten nie posiada łuku sąsiedniego. Wówczas wystarczy jedynie zwiększyć przepływ. Jeżeli łuk posiada sąsiada, należy zwiększyć przepływ i dokonać wyrównania wartości. Może też zdarzyć się przypadek, że sieć residualna będzie posiadać łuki, które w sieci przepływowej nie istnieją, zgodnie z 2.1.5. Wówczas należy dokonać przepływu zwrotnego, czyli zmniejszyć wartość przepływu w łuku, którzy dokonuje przepływu między wierzchołkami, ale w przeciwnym kierunku (który zgodnie z założeniami musi istnieć). Pełny kod algorytmu znajduje się dodatku D.

5.4.4 Warunki stopu

Poszukiwanie maksymalnego przepływu zostaje zakończone gdy zostanie spełniony jeden z warunków:

1. Suma przepływów w łukach wychodzących ze źródła jest równa sumie ich przepustowości,
2. Suma przepływów w łukach wchodzących do ujścia jest równa sumie ich przepustowości,
3. Algorytm znajdowania ścieżki powiększającej [5.4.2] zwrócił pustą listę łuków.

Pełna treść funkcji, które je sprawdzają znajdują się dodatku E.

5.4.5 Algorytm Forda-Fulkersona

Algorytm został zakapsułkowany do klasy *FordFulkersonAlgorithm*, która dziedziczy po *FlowNetworkAlgorithm*. Pseudokod tego algorytmu znajduje się w rozdziale 2.2.1. Jest to najprostsza z metod znajdowania maksymalnego przepływu i korzysta wyłącznie z metod odziedziczonych po klasie *FlowNetworkAlgorithm*:

- tworzenia sieci residualnej 5.4.1,
- znajdowania ścieżki powiększającej 5.4.2,
- zwiększania przepływu 5.4.3.

W pętli wykonywane są powyższe trzy instrukcje w kolejności w jakiej zostały wymienione. Efektem działania algorytmu jest sieć z maksymalnym przepływem. Pełny kod klasy tego algorytmu znajduje się dodatku F.

5.4.6 Przepływ blokujący

W klasie *BlockingFlowAlgoritm* została zakapsułkowana cała logika związana z tworzeniem przepływu blokującego. Znajdują się tutaj wszystkie funkcje, które są wymagane do wykonania przepływu. Zgodnie z teorią zawartą w 2.2.2 do wykonania przepływu blokującego potrzebna jest informacja o odległościach pomiędzy wierzchołkiem źródłowym, a wszystkimi pozostałymi. Informacja ta jest przechowywana w macierzy liczb zmiennoprzecinkowych $n \times n$, gdzie n to liczba wierzchołków.

```
typedef std::vector<std::vector<float>> FloatMatrix;
class BlockingFlowAlgoritm
{
protected:
    FloatMatrix _pathMatrix;
}
```

Z kolei do wyznaczenia odległości między wierzchołkami został wykorzystany algorytm Floyda-Warshalla [Zbi10b], który wyszukuje odległości między wszystkimi parami wierzchołków. W sieci przepływowej odległość między wierzchołkami połączonymi przez łuk jest równa jeden. Implementacja funkcji znajduje się w dodatku G.

5.4.7 Algorytm Dinica

Algorytm został zwarty w klasie *DinicAlgorithm* dziedziczącej po klasach *FlowNetworkAlgorithm* oraz *BlockingFlowAlgoritm*. Implementacja tego algorytmu różni od implementacji algorytmu Forda-Fulkersona 5.4.5 tworzeniem sieci residualnej, która jest warstwowa. W efekcie klasa tworzy sieć residualną metodą odziedziczoną po *FlowNetworkAlgorithm*, a następnie usuwa zbędne elementy w metodzie odziedziczonej po *BlockingFlowAlgoritm*.

```
int DinicAlgorithm::makeResidualNetwork(FlowNetwork * network, FlowNetwork *&
 ↪ residualNewtork)
{
    FlowNetworkAlgorithm::makeResidualNetwork(network, residualNewtork);
    return removeRedundantElements(residualNewtork);
}
```

Funkcja *removeRedundantElements* usuwa z przekazanej sieci residualnej nadmiarowe elementy. W tym celu pobiera z tablicy `_pathMatrix` (5.4.6) potrzebne wartości i porównuje je zgodnie z warunkami opisanymi w 2.1.6. Pełna treść tej funkcji znajduje się w dodatku H.

Algorytm Dinica wykorzystuje wyszukiwanie ścieżki powiększającej do zwiększania przepływów w warstwowej sieci residualnej. Po utworzeniu przepływu blokującego, który na początku każdej iteracji jest głęboką kopią warstwowej sieci residualnej, wyszukuje się w nim wszystkich możliwych ścieżek powiększających tym samym fragmentem kodu opisanym w 5.4.2. Uzyskane ścieżki zapisuje się do wektora ścieżek powiększających, a na koniec realizuje je wszystkie zwiększając przepływ w taki sam sposób jaki opisano w 5.4.3, z tą tylko różnicą, że wykonuje się je w pętli. Kod tego procesu został przedstawiony w dodatku I.

5.4.8 Algorytm MKM

Ten algorytm wykorzystuje pojęcie potencjału przepływowego 2.2.4, więc jego klasa wymaga utworzenia dodatkowej składowej. Do każdego wierzchołka należy dodać informację o jego trzech potencjałach (wejściowym, wyjściowym i ogólnym), więc wykorzystałem strukturę `std::tuple`. Wartościami są liczby zmiennoprzecinkowe, które umożliwiają stosowanie nieskończoności (`std::numeric_limits`).

```
typedef QMap<int, std::tuple<float, float, float>> PotentialMap;
class MkmAlgorithm : public FlowNetworkAlgorithm, public BlockingFlowAlgoritm
{
    PotentialMap _potentialMap;
}
```

Mapa potencjałów, gdzie kluczem jest id wierzchołka, jest aktualizowana w każdej iteracji, natychmiast po utworzeniu warstwowej sieci residualnej. Ta klasa wymagała gruntowanej zmiany zwiększania przepływu w przepływie blokującym: w każdej iteracji najpierw musi wyszukać wierzchołek o najmniejszym potencjale, z niego poprowadzić dwie ścieżki, do źródła oraz do ujścia, jak również sprawdzić nowy warunek stopu. W aplikacji jest to zrealizowane w następujący sposób:

Algorithm 8 Tworzenie przepływu blokującego algorytmem MKM

```
procedure ZNAJDŹ ŚCIEŻKĘ POWIĘKSZAJĄCĄ(Sieć G_f^w, referencja na przepustowość residualną c)
    repeat
        Wyczyść obie ścieżki
        Znajdź nieodrzucony wierzchołek v o najmniejszym, niezerowym potencjale w sieci G_f^w
        if jeżeli taki wierzchołek nie istnieje then
            Wyjdź z pętli
        end if
        Znajdź ścieżkę z wierzchołka v do ujścia t
        Znajdź ścieżkę ze źródła s do wierzchołka v
        if jedna ze ścieżek jest pusta then
            Odrzuć wierzchołek
        else
            c = potencjał wierzchołka v
            Zwiększ maksymalny przepływ o c
        end if
    until ścieżka ze źródła = 0 lub ścieżka do ujścia = 0
    return suma ścieżki ze źródła i ścieżki do ujścia
end procedure
```

Dwie ścieżki, ze źródła do wierzchołka i z wierzchołka do źródła, jakie zostały opisane w definicji algorytmu (2.2.4), tak naprawdę tworzą razem pojedynczą ścieżkę powiększającą. W implementacji zostało to wykorzystane i w każdej iteracji zwracana jest suma obu zbiorów - tworzona jest pojedyncza ścieżka o jednej przepustowości residualnej. Dzięki temu nie było potrzeby ani zmiany interfejsu ani skorzystania ze wzorca Adaptera.

Algorytm wykonywany jest tak długo aż zostanie znaleziona ścieżka przepływowa, czyli jest droga ze źródła do ujścia prowadzącą przez wierzchołek v. Algorytm kończy swe poszukiwanie również wtedy jeżeli nie istnieje wierzchołek, który miałby niezerowy potencjał. Po każdej iteracji uzyskana ścieżka dodawana do jest listy ścieżek powiększających wraz ze swoją przepustowością residualną, tak jak w implementacji algorytmu Dinica 5.4.7.

Wierzchołkami odrzuconymi są te znalezione wierzchołki, przez które nie prowadzi żadna ścieżka powiększająca. Mogą takie się znaleźć w warstwowej sieci residualnej i są dodawane do listy odrzuconych, by nie przechodzić przez nie w kolejnych iteracjach. Pełna treść tej procedury znajduje się w dodatku J.

Rozdział 6

Testowanie i uruchamianie

Po zbudowaniu GUI, umożliwieniu budowania sieci przepływowych oraz nakreśleniu interfejsów do wykonywania algorytmów trwała praca nad implementacją esencji pracy inżynierskiej - algorytmów wyznaczania maksymalnego przepływu.

Sposób testowania w ramach pracy (organizacja eksperymentów, przypadki testowe, wyniki, zakres testowania – pełny/niepełny)

Rozdział 7

Uwagi o przebiegu i wynikach prac

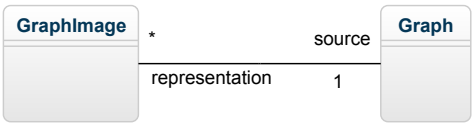
W końcowej wersji aplikacji udało się spełnić wszystkie wymagania opisane w 3: tworzenie sieci przepływowych, ich serializację oraz algorytmy wyszukiwania maksymalnego przepływu. Wszystkie funkcjonalności zostało wielokrotnie przetestowane, chociaż bez testów automatycznych.

7.1 Zachowanie zasad SOLID

Aplikacja była pisana z myślą o potencjalnych zmianach i rozszerzeniach, jakie mogą w niej zajść. W trakcie pracy nad rozwojem aplikacji starałem się przestrzegać pięciu zasad określonych przez koncept SOLID, ponieważ największymi problemami napotkanymi w czasie pracy były te związane z zaprojektowaniem aplikacji. Jaką strukturę powinna mieć sieć przepływowa, jak rozdzielić logikę od reprezentacji, jakie wzorce projektowe zastosować itp.

7.1.1 Zasada *Single responsibility*

Pracując nad podstawową strukturą sieci przepływowej, jaka będzie wykorzystywana w całej aplikacji, spróbowałem wprowadzić rozdzielenie grafu wyłącznie logicznego (*Graph*) oraz jego reprezentacji w polu edycyjnym Qt (*GraphImage* : *QGraphicsItem*). Klasa reprezentacji zawiera wskaźnik do swojej struktury logicznej na podstawie której jest budowany rysunek w Qt.



Rysunek 13: Rozdzielenie rysunku i logiki

Jednakże wierzchołki i łuki również potrzebowały dodatkowych informacji o ich sposobie rysowania, m.in. pozycji, wielkości i kolorów, więc dla reprezentacji ich graficznej również powstały dodatkowe klase (*VertexImage*, *EdgeImage*). Wadą tego rozwiązania było powielenie informacji, de facto w programie istniały dwie reprezentacje połączeń wierzchołków i krawędzi. Klasy z sufiksem *Image* dziedziczą po *QGraphicsItem* z platformy Qt, były więc wymagane do poprawnego wyświetlenia struktur graficznych. W tym przypadku klas pierwotnych, *Graph*, *Vertex* oraz *Edge*, mogłoby nie być, a struktura grafu zawarta w *GraphImage* być jedyną strukturą tworzącą graf. Jednak na tym etapie rozwoju koszt refaktoryzacji był większy niż płynące z niej korzyści, więc zależność między logiką, a reprezentacją pozostała niezmieniona.

7.1.2 Zasada *Open / closed*

W pracy zostały zrealizowane jedynie sieci przepływowe, które są tylko jednym z wielu rodzajów grafów. Aplikacja była pisana z myślą, że w przyszłości może zostać rozszerzona do prezentacji innych algorytmów, na innych strukturach grafowych. Dlatego została utworzona klasa nadrzędna *GraphImage* reprezentująca każdy typ grafu. Elastyczną rozszerzalność starałem się zapewnić wykorzystując wzorce projektowe. O tym, czy graf jest ważony (posiada funkcję odwzorowująca jego zbiór krawędzi w zbiór liczb), decyduje odpowiedni obiekt klasy ***strategii***, który jest wstrzykiwany do grafu. Sieć przepływowa zawsze posiada

strategię grafu ważonego. Zbiór narzędzi do edycji grafu, który przedstawia pasek narzędziowy, został oparty o wzorzec ***stanu***. Z punktu widzenia klas zewnętrznych narzędzie jest tylko jedno, pod wpływem naciśnięcia przycisku zmienia ono tylko swój stan wewnętrzny. Dzięki temu, aby dodać kolejne narzędzie, nie trzeba modyfikować istniejącego kodu, wystarczy jedynie utworzyć nową klasę - funkcjonalność jest odporna na zmiany, otwarta na rozszerzenia. Każda klasa narzędzia jest ***singletonem*** - w całej aplikacji potrzebne są wyłącznie pojedyncze egzemplarze. Lista algorytmów, jaka wyświetlana jest w okienku "*Dostępne algorytmy*", oparta jest o ***fabrykę obiektów***. Klasy algorytmów do wyszukiwania maksymalnego przepływu same się w niej rejestrują przy uruchomieniu programu ([And11]), a przy wybraniu jednego z nich jest pobierany wskaźnik do metody fabrykującej, która tworzy użytkowany obiekt algorytmu. W konsekwencji tego dodanie kolejnego algorytmu wymaga jedynie utworzenia nowej klasy. Fabryka również jest singletonem, potrzeba tylko jednej utworzonej na początku pracy aplikacji.

7.1.3 Zasada *Liskov substitution*

Hierarchia dziedziczenia w aplikacji nie jest duża, ale zdarzały się momenty, kiedy i ta zasada była łamana. Konstruktor reprezentacji grafu otrzymuje jako parametr wskaźnik na konfigurację grafu.

```
explicit GraphImage(GraphConfig * config);
```

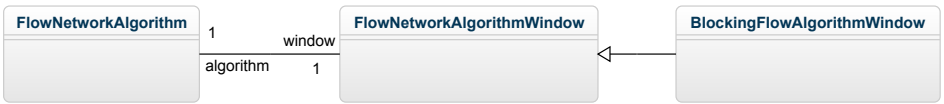
Tworząc konstruktor sieci przepływowej (klasa *FlowNetwork*), dziedziczącej po grafie, w zamierzeniu miał on otrzymywać jeszcze informację, który wierzchołek ma być źródłem, a który ujściem.

```
explicit FlowNetwork(GraphConfig * config, int source, int target);
```

Jednak takie podejście uniemożliwiało utworzenia obiektu sieci przepływowej w ten sam sposób, co grafu, i należało w każdej metodzie, która sieć tworzyła, przekazywać owe dwa dodatkowe parametry. Co więcej, z czasem informacja o źródle i ujściu okazywała się zbędna. Każdy graf tworzony od nowa jest pusty i dopiero użytkownik aplikacji, w czasie budowy sieci, decyduje, który wierzchołek ma być źródłem, a który ujściem. Były potrzebne do tego dwie nowe metody, a nie parametry konstruktora.

7.1.4 Zasada *Interface segregation*

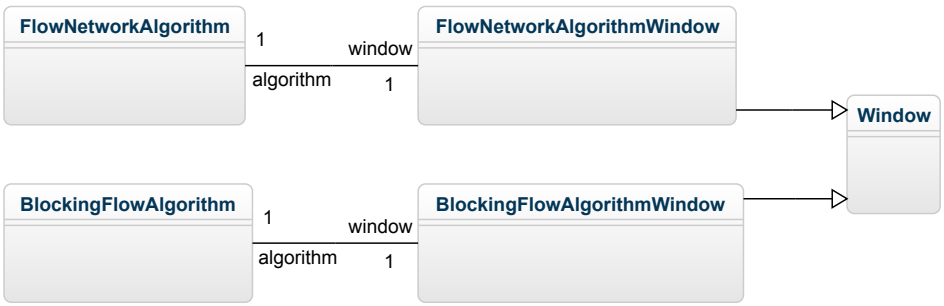
Innym problemem było wyznaczenie poprawnych interfejsów do obsługi algorytmów wyszukiwania maksymalnego przepływu. Po przeczytaniu analizy algorytmów wydawało się, że dla każdego algorytmu wystarczą trzy abstrakcyjne funkcje: utworzenie sieci residualnej, znalezienie przepływu w tej sieci i zwiększenie przepływu w sieci pierwotnej, które będą wykonywane w pętli, aż do znalezienia maksymalnego przepływu. Algorytm Forda-Fulkersona idealnie wpisywał się w ten schemat, ale algorytmy Dinica i Mkm wymagały pewnych zmian. Po pierwsze, potrzebowały warstwowej sieci residualnej, ale tu wystarczyło tylko zmienić treść funkcji, a algorytm tworzenia warstwowej sieci wydzielić do wspólnej klasy. Po drugie, wymagały utworzenia przepływu blokującego zamiast ścieżki powiększającej i tu również, z punktu widzenia logiki, wystarczyłoby przeciążenie funkcji do znajdowania przepływu w sieci residualnej. Jednak tworzenie przepływu blokującego jest procesem o wiele bardziej skomplikowanym niż znalezienie ścieżki powiększającej i aby aplikacja mogła spełnić swój walor edukacyjny, on również powinien zostać zilustrowany. Zrodziła się potrzeba, aby algorytmy Dinica i MKM posiadały dwie dodatkowe funkcje: tworzenie przepływu blokującego i zwiększenie przepływu w nim. Działanie algorytm było przedstawiane w dedykowanym oknie i zależność wygląda następująco:



Rysunek 14: Zależność między algorytmem, a oknem

Okno z przepływem blokującym jedynie rozszerza funkcjonalność domyślnego okna dodając do niego podgląd zmian w owym przepływie. Jednak istniejąca do tej pory wersja okna posiadała obiekt algorytmu z

interfejsem zawierającym trzy funkcje. Okno z przepływem blokującym również widziało tylko taki wskaźnik do algorytmu. Jeżeli okno potrzebowałoby dostępu do nowych funkcji, wymagane byłoby rzutowanie. Aby tego uniknąć model mógłby wyglądać następująco:

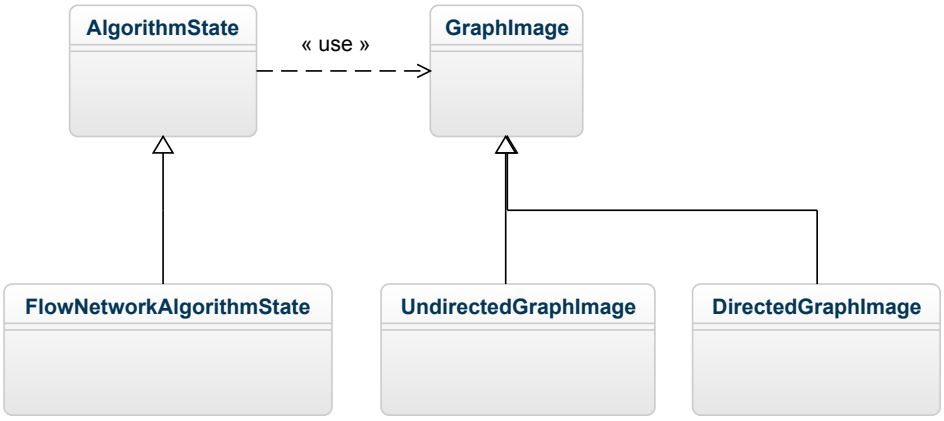


Rysunek 15: Zależność między algorytmami, a oknami

Ale możliwe było podejście z wykorzystaniem wcześniejszego modelu (rys. 14). Tworzenie przepływu blokującego to wielokrotne szukanie ścieżek powiększających w warstwowej sieci residualnej, a zwiększenie przepływu w sieci pierwotnej polega na ich odwzorowaniu. Korzystając z pierwszego modelu można było zrealizować te dwie dodatkowe funkcje realizując już istniejące, wystarczyło jedynie zmieniać argumenty metod lub wywoływać je wielokrotnie. Odpowiadała za to klasa z oknem do przepływu blokującego. Dzięki temu kod stał się trochę mniej elastyczny i rozszerzalny, ale ilość pracy potrzebnej do zrealizowania tych dwóch algorytmów - znacznie mniejsza.

7.1.5 Zasada *Dependency inversion*

Ta zasada nie byłą trudna do zachowania, chociaż nie była przestrzegana w całej aplikacji. Wszędzie, gdzie to było możliwe, klasy posiadały składowe, a metody argumenty, które były wskaźnikami lub referencjami na jak najogólniejsze struktury. Okno algorytmu posiada wskaźnik na abstrakcyjną klasę algorytmu, klasa związana ze stanem algorytmu zwraca wskaźnik na potrzebne okno, bez jawnego przekazywania informacji, czy to okno z przepływem blokującym lub nie, itp. Jeżeli zdarzały się klasy pochodnych jako składowe lub argumenty, to dlatego, że były naprawdę wymagane, np. sieć przepływowa (klasa *FlowNetwork*) dla algorytmów wyszukiwania maksymalnego przepływu lub tworzenia sieci residualnej. Hierarchia klas w aplikacji posiada kilka poziomów i na późnym etapie rozwoju aplikacji nie można było zrezygnować z dziedziczenia na rzecz pełnego *DI* tak, by było to opłacalne.



Rysunek 16: Przykład zastosowania zasady *Dependency Inversion*

Rozdział 8

Podsumowanie

Bibliografia

[Agn12] Agnieszka Debudaj-Grabysz, Sebastian Deorowicz, Jacek Widuch. *Algorytmy i struktury danych. Wybór zaawansowanych metod*. Wydawnictwo Politechniki Śląskiej, Gliwice 2012.

[And11] Andrei Alexandrescu. "Nowoczesne projektowanie w C++. Uogólnione implementacje wzorców projektowych". W: Wydawnictwo HELION, 2011. Rozd. Wytwórnie obiektów, s. 225–235.

[Eri10] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. "Wzorce projektowe. Elementy oprogramowania wielokrotnego użytku". W: Wydawnictwo HELION, 2010. Rozd. PYŁEK (FLYWEIGHT).

[Tho09] Thomas H. Corman, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. Third. The MIT Press, 2009.

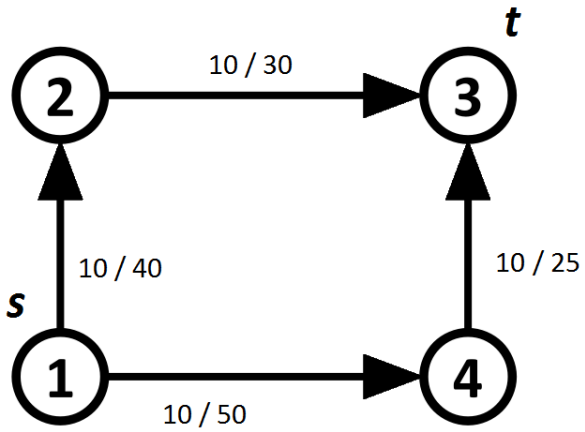
[Zbi10a] Zbigniew J. Czech, Sebastian Deorowicz, Piotr Fabian. "Algorytmy i struktury danych. Wybrane zagadnienia". W: Wydawnictwo Politechniki Śląskiej, Gliwice 2010, s. 133.

[Zbi10b] Zbigniew J. Czech, Sebastian Deorowicz, Piotr Fabian. "Algorytmy i struktury danych. Wybrane zagadnienia". W: Wydawnictwo Politechniki Śląskiej, Gliwice 2010. Rozd. Algorytm Floyda-Warshalla, s. 135–138.

Załączniki

Dodatek A

Przykład serializacji




```
<?xml version="1.0" encoding="UTF-8"?>
<Graph type="flow_network" source="1" target="3" weighted="true">
   <Config>
      <Name>Graf 2</Name>
      <VertexContext type="normal" size="59" strokeSize="12">
         <Color type="color" r="255" g="255" b="255"/>
         <Color type="strokeColor" r="0" g="0" b="0"/>
         <Font bold="true" size="60" family="Calibri"/>
      </VertexContext>
      <VertexContext type="selected" size="59" strokeSize="12">
         <Color type="color" r="0" g="0" b="255"/>
         <Color type="strokeColor" r="0" g="0" b="80"/>
         <Font bold="true" size="60" family="Calibri"/>
      </VertexContext>
      <EdgeContext type="normal" size="10">
         <Color type="color" r="0" g="0" b="0"/>
      </EdgeContext>
      <EdgeContext type="selected" size="15">
         <Color type="color" r="255" g="0" b="0"/>
      </EdgeContext>
   </Config>
```

```
22    <Model>
23       <Vertex id="1">
24          <Position x="-244.531250" y="72.376556"/>
25          <Point id="1" x="-185" y="72"/>
26          <Point id="2" x="-244" y="13"/>
27          <Point id="6" x="-244" y="131"/>
28       </Vertex>
29       <Vertex id="2">
30          <Position x="-244.531250" y="-302.734344"/>
31          <Point id="2" x="-244" y="-243"/>
32          <Point id="3" x="-185" y="-302"/>
33          <Point id="4" x="-185" y="-302"/>
34       </Vertex>
35       <Vertex id="3">
36          <Position x="298.898376" y="-302.734344"/>
37          <Point id="3" x="298" y="-243"/>
38          <Point id="4" x="239" y="-302"/>
39       </Vertex>
40       <Vertex id="4">
41          <Position x="298.898376" y="72.376556"/>
42          <Point id="1" x="239" y="72"/>
43          <Point id="3" x="298" y="13"/>
44          <Point id="4" x="298" y="13"/>
45          <Point id="8" x="298" y="131"/>
46          <Point id="10" x="293" y="13"/>
47       </Vertex>
48       <Edge type="straight" id="2" vertexFrom="1" vertexTo="2" capacity=
            "40" flow="10" offsetType="false" offsetValue="0.000000">
49          <Position x="-244.531250" y="13.376556"/>
50          <EdgeTextItem>
51             <Position x="20.013672" y="-115.109367"/>
52          </EdgeTextItem>
53       </Edge>
54       <Edge type="straight" id="1" vertexFrom="1" vertexTo="4" capacity=
            "50" flow="10" offsetType="false" offsetValue="0.000000">
55          <Position x="-185.531250" y="72.376556"/>
56          <EdgeTextItem>
57             <Position x="72.968758" y="20.110939"/>
58          </EdgeTextItem>
59       </Edge>
60       <Edge type="straight" id="4" vertexFrom="2" vertexTo="3" capacity=
            "30" flow="10" offsetType="false" offsetValue="0.000000">
61          <Position x="-185.531250" y="-302.734344"/>
62          <EdgeTextItem>
63             <Position x="134.121063" y="-70.312492"/>
64          </EdgeTextItem>
65       </Edge>
66       <Edge type="straight" id="3" vertexFrom="4" vertexTo="3" capacity=
            "25" flow="10" offsetType="false" offsetValue="0.000000">
67          <Position x="298.898376" y="13.376556"/>
68          <EdgeTextItem>
69             <Position x="33.179668" y="-123.274200"/>
70          </EdgeTextItem>
71       </Edge>
72    </Model>
73 </Graph>
```

Dodatek B

Tworzenie sieci residualnej

```
int FlowNetworkAlgorithm::makeResidualNetwork(FlowNetwork * network, FlowNetwork *&
    outResidaulNetwork)
{
    // usunięcie starych krawędzi
    auto oldEdges = outResidaulNetwork->getEdges();
    for (auto it = oldEdges.begin(); it != oldEdges.end(); ++it)
    {
        outResidaulNetwork->removeEdge(*it);
    }
    // analiza krawędzi i utworzenie sieci residualnej
    auto edges = network->getEdges();
    QList<EdgeImage*> visitedNeighbours;
    EdgeImage *neighbor;
    for (EdgeImage * edge : edges)
    {
        int capacity = edge->getCapacity();
        int flow = edge->getFlow();
        int vertexFromId = edge->VertexFrom()->getId();
        int vertexToId = edge->VertexTo()->getId();
        int residualCapacity = capacity - flow;
        if (edge->hasNeighbor() && !visitedNeighbours.contains(neighbor =
    network->edgeAt(vertexToId, vertexFromId)))
        {
            visitedNeighbours.push_back(neighbor);;
            visitedNeighbours.push_back(edge);
            int neighborFlow = neighbor->getFlow();
            int neighborCapacity = neighbor->getCapacity();
            residualCapacity = capacity - flow + neighborFlow;
            int neighborResidualCapacity = neighborCapacity - neighborFlow + flow;
            if (residualCapacity != 0)
                outResidaulNetwork->addEdge(vertexFromId, vertexToId, residualCapacity);
            if (neighborResidualCapacity != 0)
                outResidaulNetwork->addEdge(vertexToId, vertexFromId, neighborResidualCapacity);
        }
        else
        {
            if (flow != 0)
                outResidaulNetwork->addEdge(vertexToId, vertexFromId, flow);
            if (residualCapacity != 0)
                outResidaulNetwork->addEdge(vertexFromId, vertexToId, residualCapacity);
        }
    }
    return 0;
}
```

Dodatek C

Szukanie ścieżki w sieci

```
QList<EdgeImage*> FlowNetworkAlgorithm::findPathBetween(FlowNetwork * network, VertexImage *
 ↪  from, VertexImage * to)
{
    QList<EdgeImage*> path;
    VertexImage * source = network->getSource();
    EdgeImageMap edges = network->getEdges();
    bool finished = false;
    VertexImage * currentVertex = from;
    QList<VertexImage*> visitedVertices;
    QList<VertexImage*> rejectedVertices;
    EdgeImage * lastEdge = nullptr;
    srand(time(NULL));
    while (!finished)
    {
        QList<EdgeImage*> possibleEdges;
        for (auto edge : edges)
        {
            addEdgeToPath(possibleEdges, edge, currentVertex, source, visitedVertices,
 ↪  rejectedVertices);
        }
        if (possibleEdges.empty())
        {
            if (lastEdge == nullptr)
            {
                finished = true;
                return path;
            }
            else
            {
                rejectedVertices.push_back(currentVertex);
                path.pop_back();
                if (!path.empty())
                {
                    lastEdge = path.last();
                    currentVertex = lastEdge->VertexTo();
                }
                else
                {
                    lastEdge = nullptr;
                    currentVertex = from;
                }
                continue;
            }
        }
        EdgeImage * chosenEdge = possibleEdges.at(rand() % possibleEdges.size());
```

```
44          VertexImage * nextVertex = chosenEdge->VertexTo();
45          visitedVertices.push_back(nextVertex);
46          path.push_back(chosenEdge);
47          currentVertex = nextVertex;
48          lastEdge = chosenEdge;
49          if (currentVertex == to)
50              finished = true;
51      }
52      return path;
53  }
54
55  void FordFulkersonAlgorithm::addEdgeToPath(QList<EdgeImage*> & possibleEdges, EdgeImage * edge,
     ↪  VertexImage * currentVertex, VertexImage * source, QList<VertexImage*> const &
     ↪  visitedVertices, QList<VertexImage*> const & rejectedVertices)
56  {
57      if (edge->VertexFrom() == currentVertex &&
58          edge->VertexTo() != source &&
59          !visitedVertices.contains(edge->VertexTo()) &&
60          !rejectedVertices.contains(edge->VertexTo()))
61      {
62          possibleEdges.push_back(edge);
63      }
64  }
```

Dodatek D

Zwiększenie przepływu w sieci

```
1  void FlowNetworkAlgorithm::increaseFlow(FlowNetwork *& network, QList<EdgeImage*> const & path,
    ↪  int increase)
2  {
3      int oldFlow;
4      EdgeImage * networkEdge;
5      for (EdgeImage * edge : path)
6      {
7          int vertexFromId = edge->VertexFrom()->getId();
8          int vertexToId = edge->VertexTo()->getId();
9          networkEdge = network->edgeAt(vertexFromId, vertexToId);
10         // jeżeli krawędź nie istnieje w prawdziwej sieci, należy utworzyć przepływ zwrotny
11         if (networkEdge == nullptr)
12         {
13             networkEdge = network->edgeAt(vertexToId, vertexFromId);
14             oldFlow = networkEdge->getFlow();
15             networkEdge->setFlow(oldFlow - increase);
16         }
17         // jeżeli krawędź istnieje, ale posiada sąsiada, należy zmniejszyć jego przepływ
18         else if (networkEdge != nullptr && networkEdge->hasNeighbor())
19         {
20             oldFlow = networkEdge->getFlow();
21             EdgeImage * neighbourEdge = network->edgeAt(edge->VertexTo()->getId(),
    ↪  edge->VertexFrom()->getId());
22             int currentNeighborFlow = neighbourEdge->getFlow();
23             networkEdge->setFlow(std::max(increase - currentNeighborFlow, 0));
24             neighbourEdge->setFlow(std::max(currentNeighborFlow - increase, 0));
25         }
26         // istnieje, ale nie posiada sąsiada, zwykłe zwiększenie
27         else
28         {
29             oldFlow = networkEdge->getFlow();
30             networkEdge->setFlow(oldFlow + increase);
31         }
32         edge->setSelected(false);
33         networkEdge->setSelected(true);
34     }
35 }
```

Dodatek E

Warunki stopu

```
bool FlowNetworkAlgorithm::checkExistingPathsFromSource(FlowNetwork * network)
{
    VertexImage * source = network->getSource();
    bool pathExists = false;
    for (auto networkEdge : network->getEdges())
    {
        if (networkEdge->VertexFrom() == source)
            if (networkEdge->getFlow() == networkEdge->getCapacity())
                continue;
            else
            {
                pathExists = true;
                break;
            }
    }
    return pathExists;
}
```

```
bool FlowNetworkAlgorithm::checkExistingPathsToTarget(FlowNetwork * network)
{
    VertexImage * target = network->getTarget();
    bool pathExists = false;
    for (auto networkEdge : network->getEdges())
    {
        if (networkEdge->VertexTo() == target)
        {
            if (networkEdge->getFlow() == networkEdge->getCapacity())
                continue;
            else
            {
                pathExists = true;
                break;
            }
        }
    }
    return pathExists;
}
```

```
QList<EdgeImage*> FlowNetworkAlgorithm::findAugumentingPath(FlowNetwork * network, int &
    capacity)
{
    VertexImage * source = network->getSource();
    VertexImage * target = network->getTarget();
    QList<EdgeImage*> augumentingPath = augumentingPath = findPathBetween(network, source,
    target);
    if (augumentingPath.size() == 0)
        capacity = 0;
    else
    {
        auto it = std::min_element(augumentingPath.begin(), augumentingPath.end(), [&](EdgeImage
    * edge1, EdgeImage * edge2)
        {
            return edge1->getCapacity() < edge2->getCapacity();
        });
        _currentMaxFlow += (capacity = (*it)->getCapacity());
    }
    return augumentingPath;
}
```

Dodatek F

Klasa algorytmu Forda-Fulkersona

```
class FordFulkersonAlgorithm : public FlowNetworkAlgorithm
{
public:
    int makeResidualNetwork(FlowNetwork * network, FlowNetwork *& outResidaulNetwork) override;
    QList<EdgeImage*> findAugumentingPath(FlowNetwork * residualNetwork, int & capacity)
     ↪  override;
    void increaseFlow(FlowNetwork *& network, QList<EdgeImage*> const & path, int increase)
     ↪  override;
};

int FordFulkersonAlgorithm::makeResidualNetwork(FlowNetwork * network, FlowNetwork *&
     ↪  residualNewtork)
{
    return FlowNetworkAlgorithm::makeResidualNetwork(network, residualNewtork);
}

QList<EdgeImage*> FordFulkersonAlgorithm::findAugumentingPath(FlowNetwork * residualNetwork, int
     ↪  & capacity)
{
    return FlowNetworkAlgorithm::findAugumentingPath(residualNetwork, capacity);
}

void FordFulkersonAlgorithm::increaseFlow(FlowNetwork *& network, QList<EdgeImage*> const &
     ↪  path, int increase)
{
    FlowNetworkAlgorithm::increaseFlow(network, path, increase);
}
```

Dodatek G

Realizacja algorytmu Floyda-Warshalla

```
/// <summary>
/// Tworzy macierz najkrótszych dróg z każdego wierzchołka do innych.
/// Długości dróg obliczane są algorytmem Floyda-Warshalla.
/// Ponieważ krawędzi w sieci przepływowej posiadają przepustowość i przepływ,
/// zakłada się, że każda droga między sąsiednimi wierzchołkami jest równa 1.
/// </summary>
/// <param name="newtork">Siec przepływowa.</param>
void BlockingFlowAlgoritm::createShortestPathsMatrix(FlowNetwork *& newtork)
{
    const int n = newtork->getHighestVertexId();
    _pathMatrix = FloatMatrix(n);
    for (int i = 0; i < n; ++i)
    {
        _pathMatrix[i].resize(n);
    }
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
        {
            if (i != j)
                _pathMatrix[i][j] = std::numeric_limits<float>::infinity();
        }
    }
    for (auto coord : newtork->getEdges().keys())
    {
        _pathMatrix[coord.first - 1][coord.second - 1] = 1.0f;
    }
    for (int k = 0; k < n; ++k)
    {
        for (int i = 0; i < n; ++i)
        {
            if (_pathMatrix[i][k] != std::numeric_limits<float>::infinity())
            {
                for (int j = 0; j < n; ++j)
                {
                    float first = _pathMatrix[i][j];
                    float second = _pathMatrix[i][k] + _pathMatrix[k][j];
                    _pathMatrix[i][j] = std::min(first, second);
                }
            }
        }
    }
}
```

Dodatek H

Usuwanie elementów nadmiarowych z sieci residualnej

```
/// <summary>
/// Usuwa wszystkie nadmiarowe elementy grafu, które nie pojawią się w przepływie blokującym.
/// </summary>
/// <param name="residualNewtork">Sieć przepływowa.</param>
/// <returns></returns>
int BlockingFlowAlgoritm::removeRedundantElements(FlowNetwork *& residualNewtork)
{
    _sourceId = residualNewtork->getSourceId() - 1;
    _targetId = residualNewtork->getTargetId() - 1;
    createShortestPathsMatrix(residualNewtork);
    hideRedundantVertices(residualNewtork);
    removeRedundantEdges(residualNewtork);
    float sourceTargetDistance = _pathMatrix[_sourceId][_targetId];
    return sourceTargetDistance == std::numeric_limits<float>::infinity() ? 0 :
 ↪  sourceTargetDistance;
}
```

```
/// <summary>
/// Ukrywa nadmiarowe wierzchołki, które nie pojawią się w przepływie blokującym.
/// Przeszukiwanie przepływu odbywa się wyłącznie poprzez krawędzie, więc wierzchołki
/// wystarczy ukryć, nie trzeba ich usuwać.
/// Schowane zostają wierzchołki, które spełniają warunek v ∈ V\{t} : d_min(s,t) ≤ d_min(s,v)
/// </summary>
/// <param name="residualNewtork">Sieć przepływowa.</param>
void BlockingFlowAlgoritm::hideRedundantVertices(FlowNetwork *& residualNewtork)
{
    for (auto vertex : residualNewtork->getVertices())
    {
        int vertexId = vertex->getId() - 1;
        if (_pathMatrix[_sourceId][_targetId] <= _pathMatrix[_sourceId][vertexId] && vertexId !=
 ↪  _targetId)
        {
            _currentHiddenVertices.push_back(vertex);
            vertex->hide();
        }
    }
}
```

```
/// <summary>
/// Usuwa nadmiarowe krawędzie, które nie biorą udziału w tworzeniu przepływu blokującego w warstwowej sieci residualnej.
/// Usuwane są krawędzie, które spelniają warunek $(v_j, v_i) \in E : d_{min}(s, v_i) \leq d_{min}(s, v_j)$
/// </summary>
/// <param name="residualNewtork">Sieć przepływowa.</param>
void BlockingFlowAlgoritm::removeRedundantEdges(FlowNetwork *& residualNewtork)
{
    QList<EdgeImage*> edgesToRemove;
    for (auto edge : residualNewtork->getEdges())
    {
        int from = edge->VertexFrom()->getId() - 1;
        int to = edge->VertexTo()->getId() - 1;
        int first = _pathMatrix[_sourceId][to];
        int second = _pathMatrix[_sourceId][from];
        if (first <= second)
            edgesToRemove.push_back(edge);
    }
    for (auto edge : edgesToRemove)
    {
        residualNewtork->removeEdge(edge);
    }
    for (auto vertex : _currentHiddenVertices)
    {
        for (auto edge : residualNewtork->getEdges())
        {
            if (edge->VertexFrom() == vertex || edge->VertexTo() == vertex)
            {
                residualNewtork->removeEdge(edge);
            }
        }
    }
}
```

Dodatek I

Zapełnianie przepływu blokującego

```
/// <summary>
/// Przeszukanie przepływu blokującego w celu znalezienia ścieżki powiększającej.
/// Jeżeli isnieje, dodaje ją i przepływ do kontenerów.
/// </summary>
void BlockingFlowAlgorithmWindow::findAugumentingPathInBlockingFlow()
{
    _currentCapacity = 0;
    _currentBlockingPath = _algorithm->findAugumentingPath(_blockingFlow, _currentCapacity);
    if (_currentBlockingPath.size() != 0)
    {
        pushBlockingSet(_currentBlockingPath, _currentCapacity);
    }
}
```

```
/// <summary>
/// Dodanie nowej ścieżki powiększającej do zbioru ścieżek.
/// </summary>
/// <param name="path">Ścieżka.</param>
/// <param name="capacity">Maksymalna wartość przeputowości residualnej.</param>
void BlockingFlowAlgorithmWindow::pushBlockingSet(QList<EdgeImage*> const & path, int capacity)
{
    QList<EdgeImage*> residualPath;
    for (auto edge : path)
    {
        int from = edge->VertexFrom()->getId();
        int to = edge->VertexTo()->getId();
        residualPath.push_back(_residualNetwork->edgeAt(from, to));
    }
    _paths.push_back(residualPath);
    _capacities.push_back(capacity);
}
```

```
/// <summary>
/// Zwiększa przepływ w sieci przepływającej o dodatkową wartość w znalezionej ścieżce.
/// </summary>
void FlowNetworkAlgorithmWindow::increaseFlow()
{
    for (int i = 0; i < _capacities.size(); ++i)
    {
        _algorithm->increaseFlow(_network, _paths[i], _capacities[i]);
    }
    clearSets();
}
```

Dodatek J

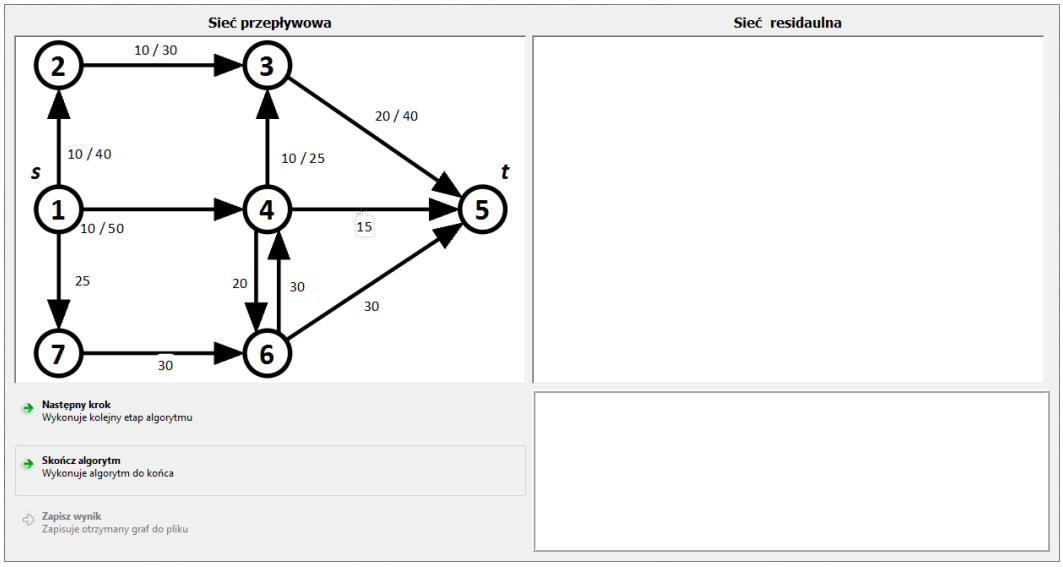
Algorytm MKM

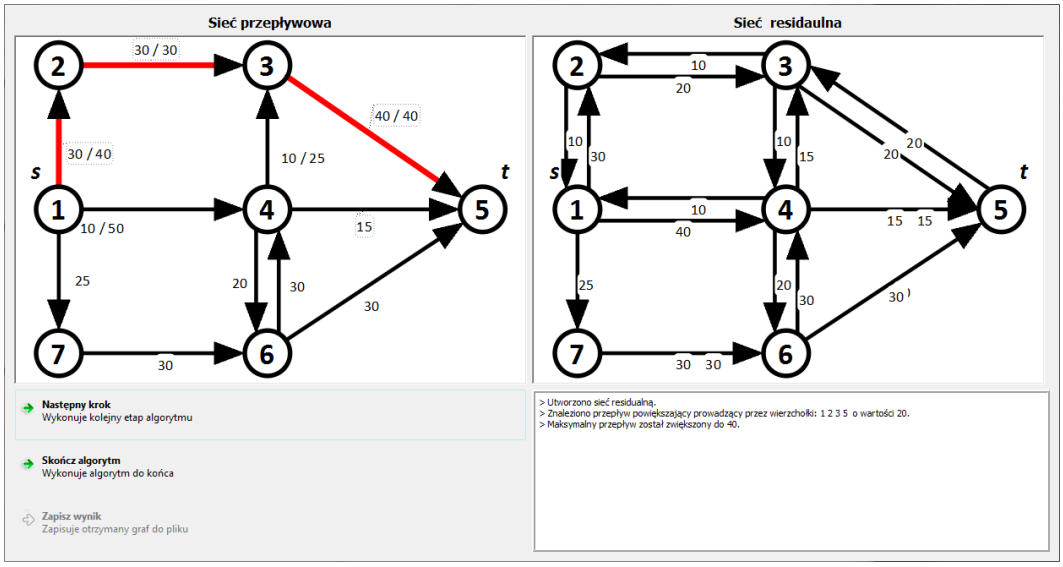
```
QList<EdgeImage*> MkmAlgorithm::findAugumentingPath(FlowNetwork * network, int & capacity)
{
    QList<EdgeImage*> pathToTarget, pathToSource;
    do
    {
        pathToTarget.clear();
        pathToSource.clear();
        VertexImage * vertex = findVertexWithMinimalPotential(network);
        if (vertex == nullptr)
            break;
        pathToTarget = sendUnitsToTarget(network, vertex->getId());
        pathToSource = sendUnitsToSource(network, vertex->getId());
        bool succeeded = pathToSource.size() != 0 && pathToTarget.size() != 0;
        if (!succeeded)
            _rejectedVertices.push_back(vertex);
        else
        {
            capacity = std::get<2>(_potentialMap[vertex->getId()]);
            _currentMaxFlow += capacity;
            vertex->setSelected(true);
            network->vertexAt(vertex->getId())->setSelected(true);
        }
    } while (pathToSource.size() == 0 || pathToTarget.size() == 0);
    return pathToSource + pathToTarget;
}
```

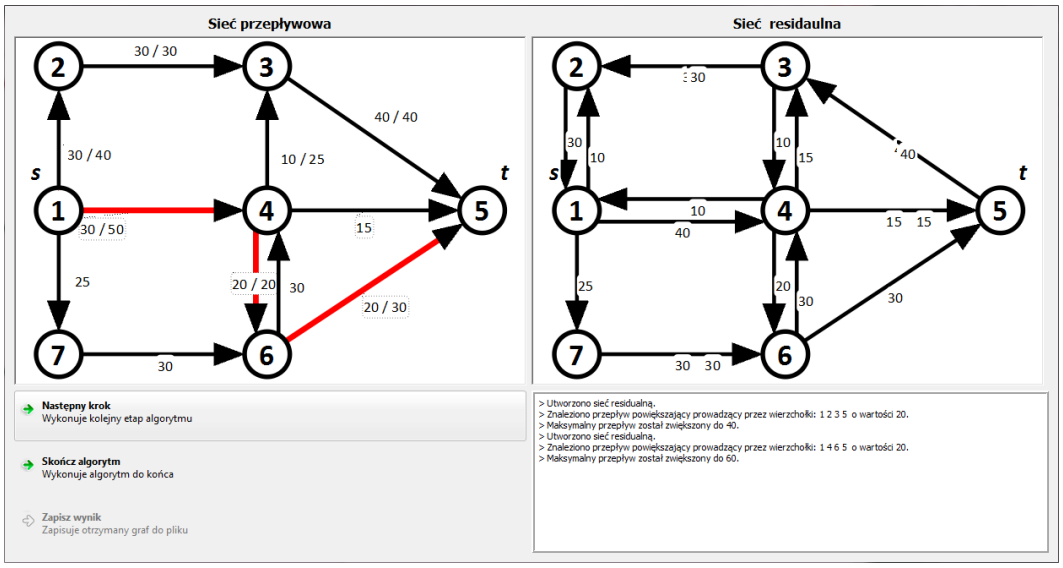
```
VertexImage * MkmAlgorithm::findVertexWithMinimalPotential(FlowNetwork * network)
{
    float minimalPotential = std::numeric_limits<float>::infinity();
    VertexImage * chosenVertex = nullptr;
    for (PotentialMap::const_iterator it = _potentialMap.begin(); it != _potentialMap.end();
    ↪  ++it)
    {
        float currentPotential = std::get<2>(*it);
        VertexImage * vertex = network->vertexAt(it.key());
        if (currentPotential != 0 && vertex->isVisible()
            && !_rejectedVertices.contains(vertex) && currentPotential < minimalPotential)
        {
            chosenVertex = vertex;
            minimalPotential = currentPotential;
        }
    }
    return chosenVertex;
}
```

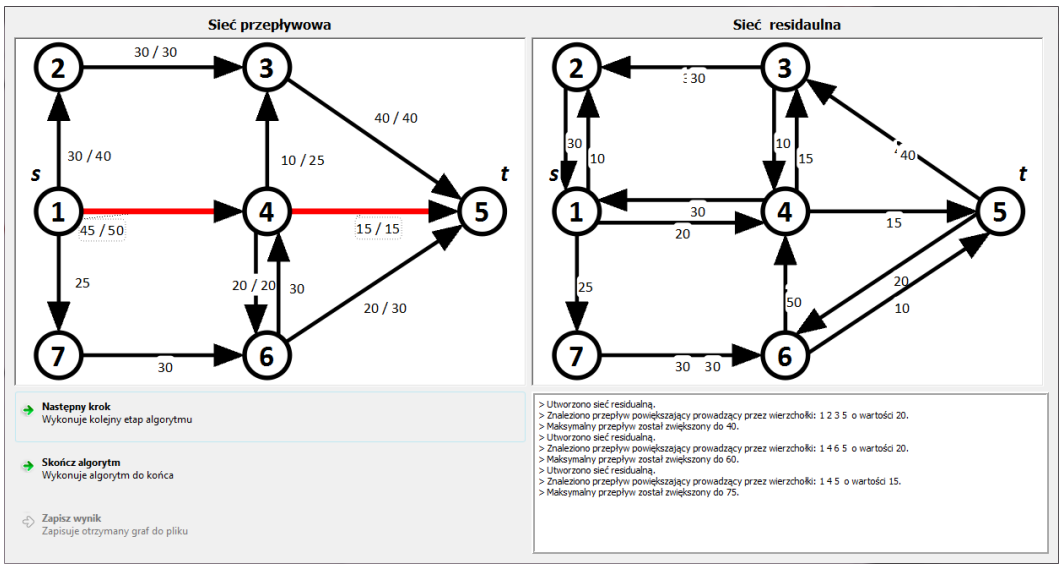
Dodatek K

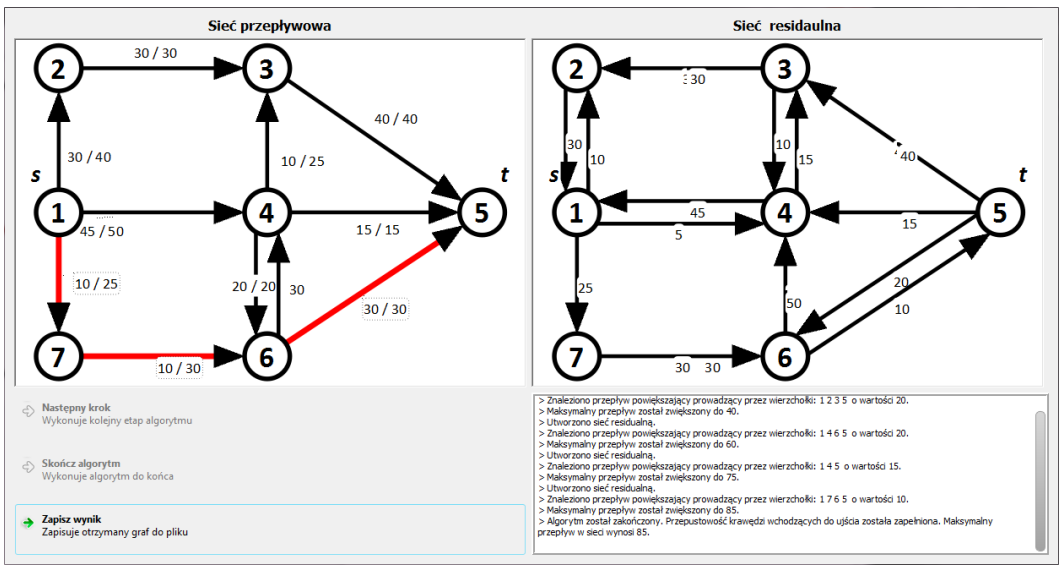
Realizacja algorytmu Forda-Fulkersona






Sieć przepływowa
Sieć residualna
30 / 30
40 / 40
30 / 40
10 / 25
30 / 50
15
25
20 / 20
30
20 / 30
30
Następny krok
Wykonuje kolejny etap algorytmu
Skończ algorytm
Wykonuje algorytm do końca
Zapisz wynik
Zapisuje otrzymany graf do pliku
> Utworzono sieć residualną.
> Znaleziono przepływ powiększający prowadzący przez wierzchołki: 1 2 3 5 o wartości 20.
> Maksymalny przepływ został zwiększony do 40.
> Utworzono sieć residualną.
> Znaleziono przepływ powiększający prowadzący przez wierzchołki: 1 4 6 5 o wartości 20.
> Maksymalny przepływ został zwiększony do 60.


Sieć przepływowa
Sieć residualna
30 / 30
40 / 40
30 / 40
10 / 25
45 / 50
15 / 15
25
20 / 20
30
20 / 30
30
Następny krok
Wykonuje kolejny etap algorytmu
Skończ algorytm
Wykonuje algorytm do końca
Zapisz wynik
Zapisuje otrzymany graf do pliku
> Utworzono sieć residualną.
> Znaleziono przepływ powiększający prowadzący przez wierzchołki: 1 2 3 5 o wartości 20.
> Maksymalny przepływ został zwiększony do 40.
> Utworzono sieć residualną.
> Znaleziono przepływ powiększający prowadzący przez wierzchołki: 1 4 6 5 o wartości 20.
> Maksymalny przepływ został zwiększony do 60.
> Utworzono sieć residualną.
> Znaleziono przepływ powiększający prowadzący przez wierzchołki: 1 4 5 o wartości 15.
> Maksymalny przepływ został zwiększony do 75.


Sieć przepływowa
Sieć residualna
30 / 30
40 / 40
30 / 40
10 / 25
45 / 50
15 / 15
10 / 25
20 / 20
30
30 / 30
10 / 30
Następny krok
Wykonuje kolejny etap algorytmu
Skończ algorytm
Wykonuje algorytm do końca
Zapisz wynik
Zapisuje otrzymany graf do pliku
> Znaleziono przepływ powiększający prowadzący przez wierzchołki: 1 2 3 5 o wartości 20.
> Maksymalny przepływ został zwiększony do 40.
> Utworzono sieć residualną.
> Znaleziono przepływ powiększający prowadzący przez wierzchołki: 1 4 6 5 o wartości 20.
> Maksymalny przepływ został zwiększony do 60.
> Utworzono sieć residualną.
> Znaleziono przepływ powiększający prowadzący przez wierzchołki: 1 4 5 o wartości 15.
> Maksymalny przepływ został zwiększony do 75.
> Utworzono sieć residualną.
> Znaleziono przepływ powiększający prowadzący przez wierzchołki: 1 7 6 5 o wartości 10.
> Maksymalny przepływ został zwiększony do 85.
> Algorytm został zakończony. Przepustowość krawędzi wchodzących do ujścia została zapełniona. Maksymalny przepływ w sieci wynosi 85.

Dodatek L

Realizacja algorytmu Dinica

Dodatek M

Realizacja algorytmu MKM