System kolorujący krawędzie w grafie w zależności od wag

Paweł Suder
Mirosław Sajdak

01.06.2011

Celem oprogramowania jest wykonanie kolorowania krawędzi grafu skierowanego w zależności od wagi krawędzi oraz zakresu wag krawędzi w całości grafu. W porównaniu do poprzednich wersji, oprogramowanie pozwala na rozproszenie informacji o grafie, jak i wykonywania na nich operacji taki jak przeszukiwanie w celu znalezienia największej i najmniejszej krawędzi oraz przeliczenia wartości koloru krawędzi. Kolor krawędzi grafu będzie tym ciemniejszy, im większa jest waga krawędzi, w stosunku do innych krawędzi.

Model grafu składa się z zbioru wierzchołków, z których to każdy wierzchołek posiada zbiór krawędzi skierowanych, wychodzących z danego wierzchołka. Każdy wierzchołek, każda krawędź posiada referencję na graf, w przypadku krawędzi posiadają one również referencję na wierzchołki początkowy i końcowy krawędzi skierowanej. Rozproszenie polega na tym, że serwer, nie dysponuje bezpośrednio informacją o grafie, ale deleguje to zachowanie na poszczególne węzły zerejestrowane na serwerze. Każdy z węzłów posiada informacje o fragmencie grafu, tj. przechowuje część zbioru wierzchołków, które przechowują wszystkie swoje krawędzie skierowane.

W przypadku wykonania operacji przeszukiwania grafu lub przeliczenia wartości koloru, serwer deleguje tę operację na wszystkie węzły, gdzie dla każdego wierzchołka zostanie uruchomiony wątek, wew. którego nastąpi wykonanie operacji na krawędziach.

Lista wymagań funkcjonalnych:

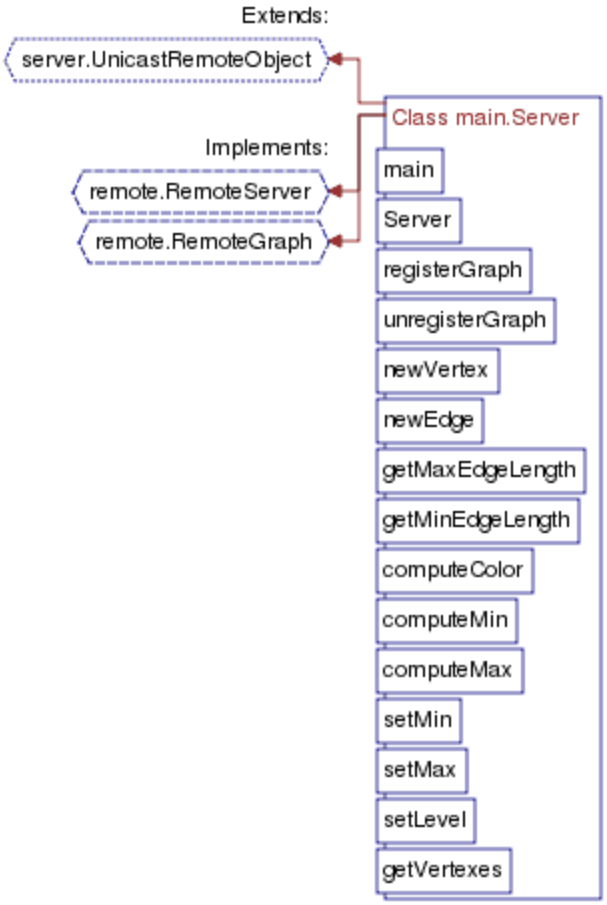
- dodawanie wierzchołka
- dodawanie krawędzi skierowanej, ważonej
- zmiana wagi krawędzi

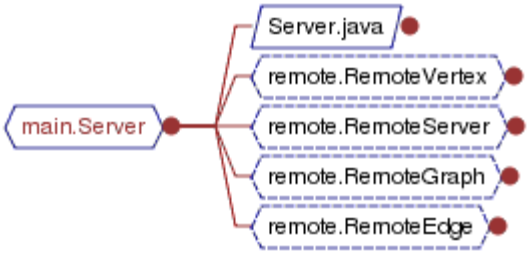
Lista wymagań niefunkcjonalnych:

- prezentacja danych tabelarycznych o punktach i krawędziach

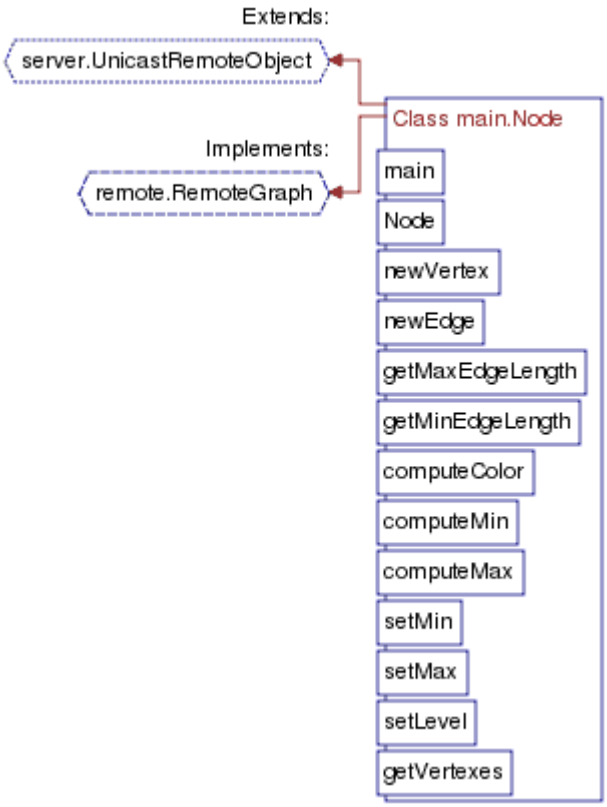
Klasy:

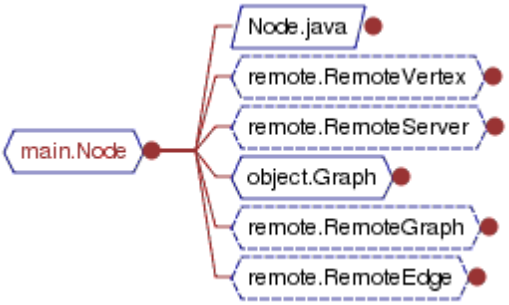
- Server - klasa, która implementuje interfejsy *RemoteGraph*, *RemoteServer*. Klasa ta nie przechowuje informacji o grafie bezpośrednio, ale przechowuje informację o częściach tego grafu, rozproszonego na poszczególnych węzłach, które się rejestrują na serwer. Wołając metodą na tym obiekcie powodujemy wykonanie jej na wszystkich grafach lokalnych rozproszonych po węzłach. Server, jako graf, przechowuje informację o maksymalnej i minimalnej wadze krawędzie, które to są wspólne dla wszystkich części grafu.





- Node - klasa, która implementuje tylko interfejs *RemoteGraph*. Klasa ta jest adapterem do obiektów klasy Graph, czyli jedynie przechowuje referencję na obiekt grafu, delegując większość operacji na ten obiekt, w przypadku operacji na krawędzi minimalnej czy maksymalnej, to wówczas woła metody obiektu serwera.

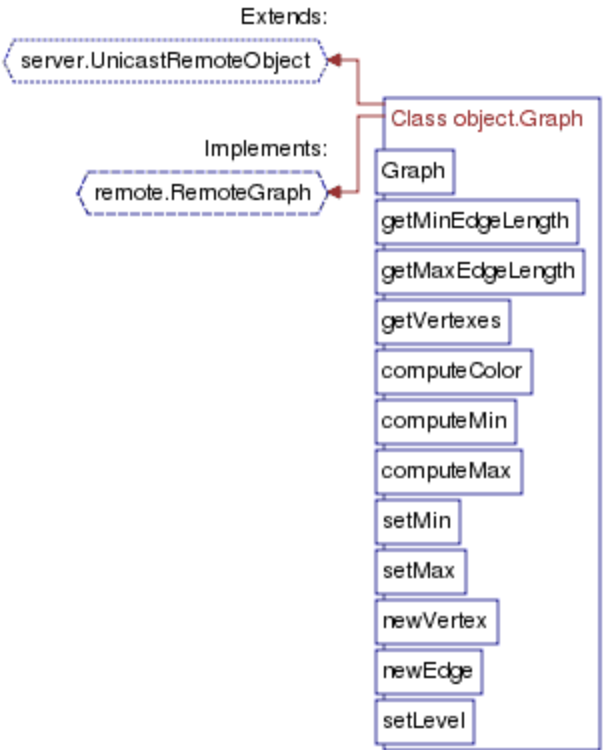




- Client - nie stanowi klasy, jedynie pozwala na uruchomienie aplikacji w wersji okienkowej, czy tworzy obiekt klasy MainFrame, przekazując do niego referencję na graf klasy Server, czyli ten ogólny.
- Auto - podobnie, jak Client, nie stanowi klasy, pozwala na zasymulowanie dodawania wierzchołków, dodawania krawędzi oraz zmiany wagi krawędzi, w celu określenia poprawności oprogramowania.

Pakiety:

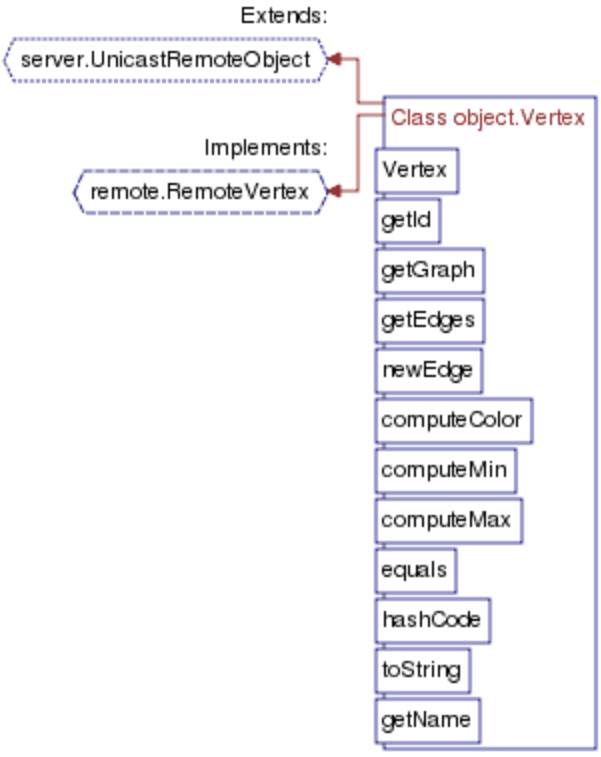
- *main* - zawiera powyższe klasy
- *main.gui* - zawiera klasy odpowiedzialne za wersję GUI aplikacji, czyli
  - *MainFrame* - główne okno, zawierające dwie tabele: tabela wierzchołków i tabela krawędzie
  - *VertexFrame* - okno wierzchołka, gdzie jest tylko jedna tabela, zawierająca informacje o krawędziach skierowanych, wychodzących z tego wierzchołka
  - *NewEdgeFrame* - okno tworzenia nowej krawędzi, z podaniem końca tejże krawędzi, gdyż początek wynika z kontekstu uruchomienia tego okna.
- *main.gui.model* - modele danych tabelarycznych oraz list rozwijanych, wymaganych na potrzeby ujednolicenia wyświetlania danych w tabeli
- *main.object* - zawiera implementację trzonu projektu, czyli fragmentu grafu, który jest umieszczony w węźle. Klasami sią *Graph*, *Vertex* i *Edge*.
- *main.remote* - zestaw interfejsów współdzielonych przez serwera, węzeł oraz klienta. Interfejsami są *RemoteServer*, *RemoteGraph*, *RemoteVertex*, *RemoteEdge*.

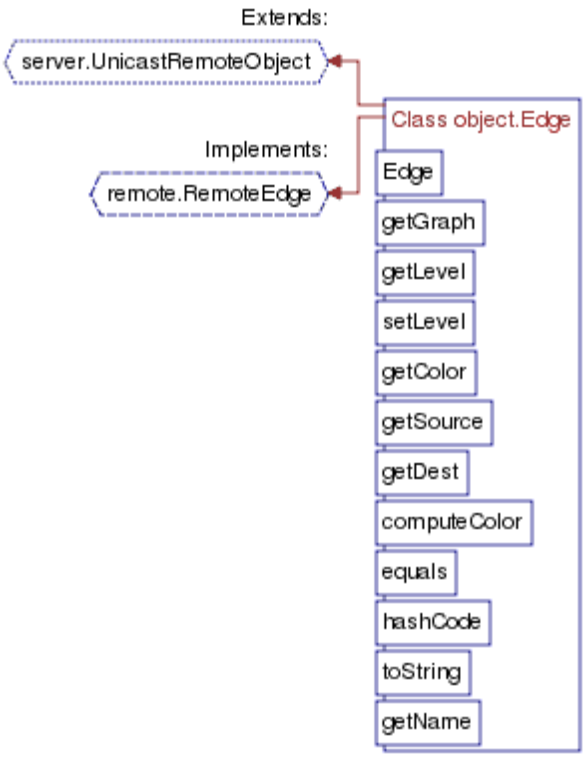


Model:

- *Graph* - stanowi klasę obiektów, które faktycznie są grafami lub fragmentem całości grafu. Przechowują kolekcje wierzchołków.

- *Vertex* - stanowi klasę obiektów, które faktycznie są wierzchołkami. Nie mogą być dzielone. Przechowują pełną kolekcję krawędzie skierowanych, których początek jest w tym wierzchołku.





- *Edge* - stanowi klasę obiektów, które faktycznie są krawędziami. Posiadają referencje na wierzchołki końców krawędzi, wartości wagi oraz przypisaną wartość koloru.

W celu umożliwienia rozpraszania wykonywania operacji, zostało zastosowane Java RMI, które pozwala na wykonanie zdalne procedur, a także przekazywanie obiektów i wykonywania operacji na nich. To pozwoliło nam na wykonanie grafu, a raczej jego modelu, który dla klienta jest jednolity, zaś w rzeczywistości jest on podzielony na kilka zbiorów

wierzchołków i krawędzie skierowanych. Z wykorzystaniem interfejsów zdalnych, możliwe jest przekazywanie między węzłami informacji o istnieniu obiektu, a nie całego obiektu. To czyni model rozproszonym i pozwala na wykonywanie operacji, jak i obliczeń w sposób równoległy na różnych maszynach i w różnych miejscach.

W momencie, gdy klient podłącza się do serwera, uzyskuje dostęp do obiektu grafu globalnego, tego jaki implementuje klasa Server. Wołając metody, z wykorzystaniem Java RMI, wołane są one na serwerze, skąd kolejno wołane są na wszystkich zarejestrowanych węzłach, przechowujących fragmenty tego grafu. W przypadku dodawania nowego wierzchołka, w sposób rotacyjny, zapisywany jest do kolejnego w cyklu węzła, zaś krawędzie zapisywane są w wierzchołkach, gdzie mają swój początek.

Operacje związane z przeliczaniem koloru na wszystkich krawędziach czy przeszukiwania grafu w celu znalezienia najmniejszej lub największej, zaczynają się w grafie głównym, skąd następnie, na wszystkich węzłach wołana jest podobna metoda, gdzie to dla każdego wierzchołka tworzony jest wątek, wew. którego dla każdej krawędzi skierowanej, wychodzącej z tego wierzchołka, wykonuje się żądaną operację, tj. obliczenia koloru w zależności od wagi lub ustalenie czy to minimalna czy maksymalna krawędź.

Problemem modelu jest synchronizacja dostępu do krawędzi minimalnej i maksymalnej, a także w przypadku podłączenia się klienta do serwera, uzyskanie danych o grafie, których przesłanie ze wszystkich węzłów do serwera, w przypadku dużego grafu, jest bardzo czasochłonne i powoduje opóźnienia. Samo przeliczanie wartości kolorów w zależności od wag, odbywa się bardzo szybko. Każdy fragment grafu, ulokowany w węźle, wykonuje to wielowątkowo (wątek na wierzchołek), gdzie dla każdej krawędzi wołana jest metoda *computeColor*. Opóźnienia jakie mogą wynikać, są związane z synchronizacją dostępu do wartości maksymalnej i minmalnej wagi krawędzi przez każdy z krawędzi, w trakcie wykonywania tej operacji.

Podobnie się to ma w przypadku przeszukiwania grafu w celu znalezienia maksymalnej i minimalej krawędzi. Każda z krawędzi odpytuje serwer, w celu ustalenia czy jest minimalną, a w przypadku wielu węzłów i wielu wierzchołków (wątków na węzłach), każda z krawędzi odpytuje wówczas serwer. Podobnie opóźnienia generowane są przez potrzebę synchronizacji dostępu do pól *minEdge* i *maxEdge* w serwerze.

Kod, dla serwera, który imituje całość grafu w oparciu o informacje z poszczególnych węzłów, w wersji okrojonej do fragmentów najważniejszych, ma następująca postać:

```
/**
 * Obiekty tej klas intefejsem przeypominają graf, w rzeczywistości stanowią proxy dla całości
 * grafu, który jest rozproszony po węzła, i dopiero tam są rzeczywiste informacje o grafie.
 */
public class Server extends UnicastRemoteObject implements Serializable, RemoteServer, RemoteGraph {

	private List<RemoteGraph> graphs;
	private AtomicInteger currentGraph, next;
	private RemoteEdge minEdge, maxEdge;

	@Override
	public RemoteVertex newVertex(Integer id) throws RemoteException {
		RemoteVertex vertex;
		RemoteGraph graph;

		synchronized(graphs) {
			synchronized(currentGraph) {
				currentGraph.set((currentGraph.get() +1) % graphs.size());
				graph = graphs.get(currentGraph.get());
			}
		}

		synchronized(next) {
			vertex = graph.newVertex(next.incrementAndGet());
		}

		return vertex;
	}

	@Override
	public RemoteEdge newEdge(RemoteVertex v1, RemoteVertex v2, Integer level) throws RemoteException {
		RemoteEdge edge = v1.getGraph().newEdge(v1, v2, level);
```

```
        boolean t = false;

        if(minEdge == null && maxEdge == null) {
                minEdge = edge;
                maxEdge = edge;
        } else {
                synchronized(minEdge) {
                synchronized(maxEdge) {

                        /* Dodana zostanie krawędź, która stanie się nową krawędzią minimalną. */
                        if(level < minEdge.getLevel()) {
                                minEdge = edge;
                                t = true;

                        /* Dodana zostanie krawędź, która stanie się nową krawędzią maksymalną. */
                        } else if(level > maxEdge.getLevel()) {
                                maxEdge = edge;
                                t = true;

                        /* Dodana zostanie krawędź, której wartość wagi jest w zakresie minmax'a */
                        } else {
                                t = false;

                        }

                }
                }
        }

        if(t)
                this.computeColor();
        else
                edge.computeColor();

        return edge;
}

@Override
public Integer getMaxEdgeLength() throws RemoteException {
        synchronized(maxEdge) {
                return maxEdge.getLevel();
        }
}

@Override
public Integer getMinEdgeLength() throws RemoteException {
        synchronized(minEdge) {
                return minEdge.getLevel();
        }
}

@Override
public void computeColor() throws RemoteException {
        synchronized(graphs) {
                for(RemoteGraph graph : graphs)
                        graph.computeColor();
        }
}

@Override
public void computeMin() throws RemoteException {
        synchronized(graphs) {
                for(RemoteGraph graph : graphs)
                        graph.computeMin();
        }
}

@Override
public void computeMax() throws RemoteException {
        synchronized(graphs) {
                for(RemoteGraph graph : graphs)
                        graph.computeMax();
        }
}
```

```
@Override
public void setMin(RemoteEdge edge) throws RemoteException {
        synchronized(minEdge) {
                if(edge.getLevel() < minEdge.getLevel()) {
                        minEdge = edge;
                }
        }
}

@Override
public void setMax(RemoteEdge edge) throws RemoteException {
        synchronized(maxEdge) {
                if(edge.getLevel() > maxEdge.getLevel()) {
                        maxEdge = edge;
                }
        }
}

@Override
public void setLevel(RemoteEdge edge, Integer level) throws RemoteException {
        if(edge.getLevel().equals(level))
                return;

        int min = minEdge.getLevel(), max = maxEdge.getLevel(), current = edge.getLevel();
        boolean tmin = false, tmax = false, tcolor = false;

        synchronized(minEdge) {
        synchronized(maxEdge) {

                /* Krawędź jest jedną z granicznych i zmiana jej wagi spowoduje powiększenie zakresu wag,
                 * czyli krawędź nadal będzie jedną z granicznych. */
                if((minEdge.equals(edge) && level < current) || (maxEdge.equals(edge) && level > current)) {
                        edge.getGraph().setLevel(edge, level);
                        tcolor = true;

                /* Krawędź jest minimalną krawędzią i nowa wartość powoduje zawężenie zakresu wag. */
                } else if(minEdge.equals(edge) && level > current){
                        edge.getGraph().setLevel(edge, level);
                        tmin = true;
                        tcolor = true;

                /* Krawędź jest maksymalną krawędzią i nowa wartość powoduje zawężenie zakresu wag. */
                } else if(maxEdge.equals(edge) && level < current) {
                        edge.getGraph().setLevel(edge, level);
                        tmax = true;
                        tcolor = true;

                /* Krawędź nie jest krawędzią skrajną, ale nowa wartość podowuje powiększenie zakresu wag. */
                } else if(level > max) {
                        edge.getGraph().setLevel(edge, level);
                        maxEdge = edge;
                        tcolor = true;

                /* Krawędź nie jest krawędzią skrajną, ale nowa wartość podowuje powiększenie zakresu wag. */
                } else if(level < min) {
                        edge.getGraph().setLevel(edge, level);
                        minEdge = edge;
                        tcolor = true;

                /* Nie powoduje żadnych zmian, po prostu ustalamy wartość wagi krawędzi. */
                } else {
                        edge.getGraph().setLevel(edge, level);

                }
        }
        }

        if(tmin)
                this.computeMin();
        if(tmax)
                this.computeMax();
        if(tcolor)
                this.computeColor();
        else
```

```
                edge.computeColor();
        }

        @Override
        public List<RemoteVertex> getVertexes() throws RemoteException {
                List<RemoteVertex> vertexes = new LinkedList<RemoteVertex>();

                synchronized(graphs) {
                        for(RemoteGraph graph : graphs)
                                vertexes.addAll(graph.getVertexes());
                }

                return vertexes;
        }
}
```

Klasa *Node* stanowi opakowanie obiektu klasy *Graph*, czyli wszystkie zapytania kierowane do węzła, są przenoszone do pola tejże klasy będącej obiektem klasy *Graph*, który już faktycznie przechowuje informacje o grafie pod postacią zbioru wierzchołków.

```
/**
 * Obiekty tej klasy nie są utrzymywane na serwerze, ale w węzłach, jako lokalne grafy, przechowujące
 * fragmenty całego grafu.
 */
public class Graph extends UnicastRemoteObject implements Serializable, RemoteGraph {

        private final RemoteGraph graph;
        private List<RemoteVertex> vertexes;

        public Graph(RemoteGraph graph) throws RemoteException {
                this.graph = graph;
                this.vertexes = Collections.synchronizedList(new LinkedList<RemoteVertex>());
        }

        /**
         * Zwraca minimalną krawędź w całym grafie. "graph" jest referencją na graf na serwerze.
         */
        @Override
        public Integer getMinEdgeLength() throws RemoteException {
                return graph.getMinEdgeLength();
        }

        /**
         * Zwraca maksymalną krawędź w całym grafie. "graph" jest referencją na graf na serwerze.
         */
        @Override
        public Integer getMaxEdgeLength() throws RemoteException {
                return graph.getMaxEdgeLength();
        }

        /**
         * Zwraca lokalny zbiór wszystkich wierzchołków grafu.
         */
        @Override
        public List<RemoteVertex> getVertexes() throws RemoteException {
                return vertexes;
        }

        /**
         * Przelicza wartości koloru, w zależności od wagi krawędzi. Wykonywane jest to wielowątkowo
         * w jednym weźle, w każdym wątku przeliczane są krawędzie skierowane, wychodzące z jednego
         * wierzchołka.
         */
        @Override
        public void computeColor() throws RemoteException {
                new Thread(new Runnable() {

                        @Override
                        public void run() {
                                synchronized(vertexes) {
                                        for(RemoteVertex vertex : vertexes) {
                                                try {
                                                        vertex.computeColor();
```

```
                                        } catch (RemoteException e) {
                                                logger.warn(e.toString());
                                        }
                                }
                        }
                }

        }).start();
}

/**
 * Przeszukiwanie lokalne grafu, w celu znalezienia minimalnej krawędzi. Podobnie jak w przypadku
 * przeliczania koloru, wykonuje się to wielowątkowo w jednym węźle, gdzie jeden wątek odpowiada
 * jednemu wierzchołkowi.
 */
@Override
public void computeMin() throws RemoteException {
        for(final RemoteVertex vertex : vertexes) {
                try {
                        vertex.computeMin();
                } catch (RemoteException e) {
                        logger.warn(e.toString());
                }
        }
}

/**
 * Przeszukiwanie lokalne grafu, w celu znalezienia maksymalnej krawędzi. Podobnie jak w przypadku
 * przeliczania koloru, wykonuje się to wielowątkowo w jednym węźle, gdzie jeden wątek odpowiada
 * jednemu wierzchołkowi.
 */
@Override
public void computeMax() throws RemoteException {
        for(final RemoteVertex vertex : vertexes) {
                try {
                        vertex.computeMax();
                } catch (RemoteException e) {
                        logger.warn(e.toString());
                }
        }
}

/**
 * Ustawienie krawędzi minimalnej, o ile podana ma wagę mniejszą niż obecna minimalna krawędź.
 * "graph" jest referencją na graf na serwerze, informacja o minimalnej krawędzi jest globalna.
 */
@Override
public void setMin(RemoteEdge edge) throws RemoteException {
        graph.setMin(edge);
}

/**
 * Ustawienie krawędzi maksymalnej, o ile podana ma wagę większą niż obecna maksymalna krawędź.
 * "graph" jest referencją na graf na serwerze, informacj o maksymalnej krawędzi jest globalna.
 */
@Override
public void setMax(RemoteEdge edge) throws RemoteException {
        graph.setMax(edge);
}

/**
 * Dodanie nowej wierzchołka do lokalnego grafu.
 */
@Override
public RemoteVertex newVertex(Integer id) throws RemoteException {
        RemoteVertex v = new Vertex(this, id);
        synchronized(vertexes) {
                vertexes.add(v);
        }
        return v;
}

/**
 * Dodanie nowej krawędzi do grafu, wraz z ustaleniem wagi.
 */
```

```
	@Override
	public RemoteEdge newEdge(RemoteVertex v1, RemoteVertex v2, Integer level) throws RemoteException {
		return v1.newEdge(v2, level);
	}

	/**
	 * Ustawienie wartości wagi dla danej krawędzi. Metoda wołana przez serwer,
	 * nie powinno się jej wołać ręcznie.
	 */
	@Override
	public void setLevel(RemoteEdge edge, Integer level) throws RemoteException {
		edge.setLevel(level);
	}

}
```

Kolejno każdy wierzchołek grafu posiada swój własny zbiór krawędzie skierowanych, wychodzących od danej krawędzi.

```
public class Vertex extends UnicastRemoteObject implements Serializable, RemoteVertex {

	private final int id;
	private final RemoteGraph graph;
	private List<RemoteEdge> edges;

	public Vertex(RemoteGraph graph, int id) throws RemoteException {
		this.graph = graph;
		this.id = id;
		this.edges = Collections.synchronizedList(new LinkedList<RemoteEdge>());
	}

	/**
	 * Pobranie unikatowego w skali całego grafu identyfikatora wierzchołka.
	 */
	@Override
	public Integer getId() {
		return id;
	}

	/**
	 * Uzyskanie referencji na graf lokalny, gdyż w momencie tworzenia wierzchołka, jako
	 * parametr konstruktora, przekazywany jest this Graph'u - lokalnego grafu.
	 */
	@Override
	public RemoteGraph getGraph() {
		return graph;
	}

	/**
	 * Uzyskanie listy krawędzi skierowanych wychodzących z tego wierzchołka.
	 */
	@Override
	public List<RemoteEdge> getEdges() {
		return edges;
	}

	/**
	 * Dodanie nowej krawędzi skierowanej do zbioru krawędzi skierowanych, wychodzących
	 * z tego wierzchołka.
	 */
	@Override
	public RemoteEdge newEdge(RemoteVertex vertex, Integer level) throws RemoteException {
		RemoteEdge edge = new Edge(graph, this, vertex, level);
		synchronized(edges) {
			edges.add(edge);
		}
		return edge;
	}

	/**
	 * Przeliczanie wartości koloru każdej krawędzi skierowanej wychodzącej z tego wierzchołka.
	 */
	@Override
```

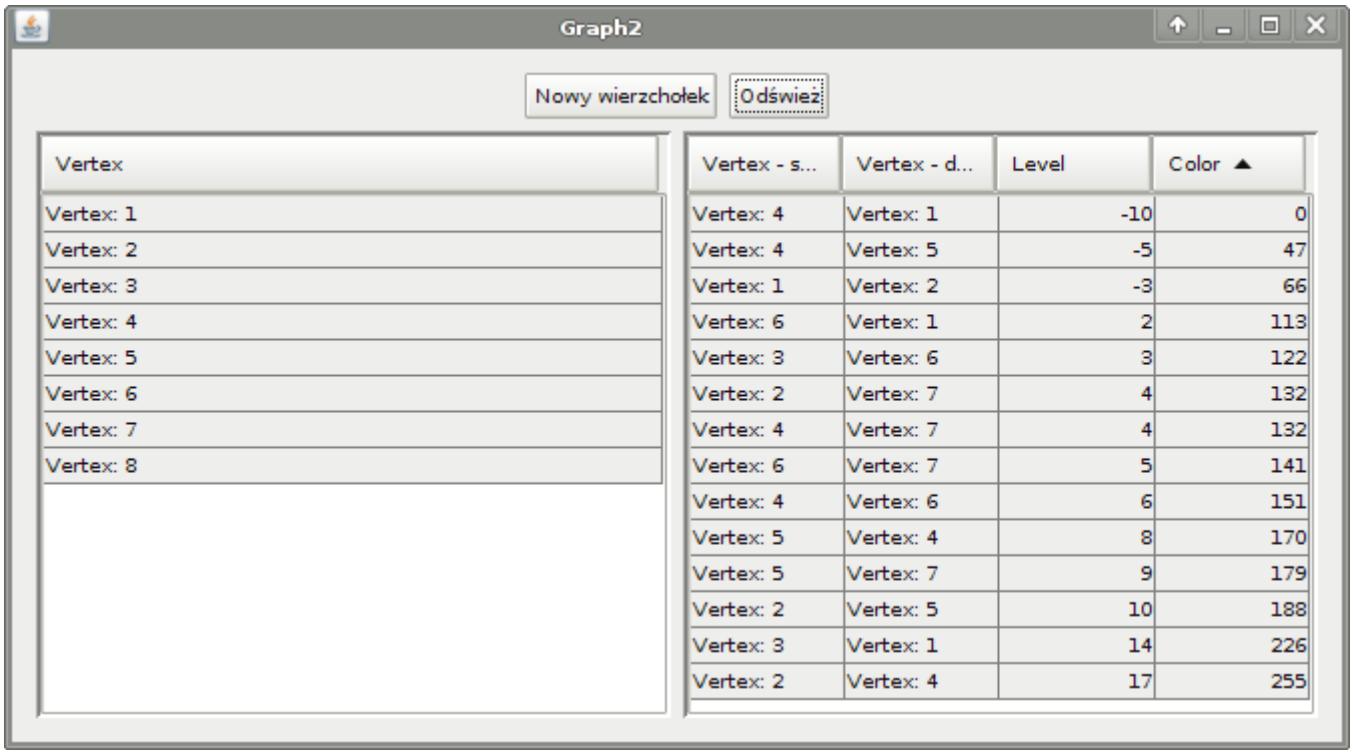
```
	public void computeColor() throws RemoteException {
		new Thread(new Runnable() {

			@Override
			public void run() {
				synchronized(edges) {
					for(RemoteEdge edge : edges) {
						try {
							edge.computeColor();
						} catch (RemoteException e) {
							logger.warn(e.toString());
						}
					}
				}
			}

		}).start();
	}

	/**
	 * Przeszukiwanie zbioru krawędzie skierowanych, wychodzących z tego wierzchołka,
	 * w celu ustalenia minimalnej.
	 */
	@Override
	public void computeMin() throws RemoteException {
		for(RemoteEdge edge : edges) {
			try {
				graph.setMin(edge);
			} catch (RemoteException e) {
				logger.warn(e.toString());
			}
		}
	}

	/**
	 * Przeszkuwanie zbioru krawędzie skierowanych, wychodzących z tego wierzchołka,
	 * w celu ustalenia maksymalnej.
	 */
	@Override
	public void computeMax() throws RemoteException {
		for(RemoteEdge edge : edges) {
			try {
				graph.setMax(edge);
			} catch (RemoteException e) {
				logger.warn(e.toString());
			}
		}
	}
}
```

Oraz ostatnia klasa modelu, czyli klasa *Edge*, tj. krawędzi.

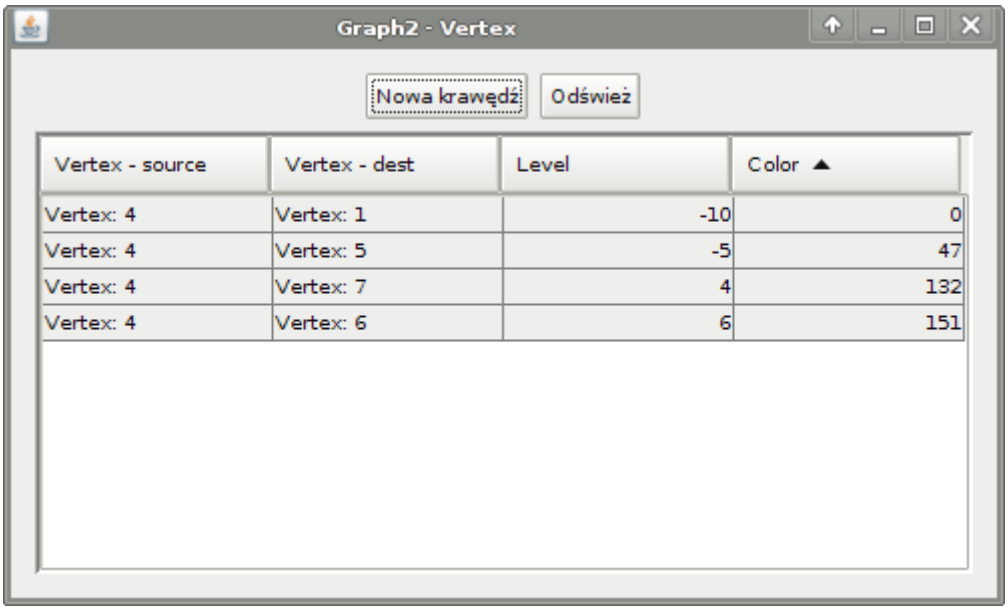
```
public class Edge extends UnicastRemoteObject implements Serializable, RemoteEdge {

	private final RemoteGraph graph;
	private final RemoteVertex v1, v2;
	private AtomicInteger level, color;

	public Edge(RemoteGraph graph, RemoteVertex v1, RemoteVertex v2, Integer level) throws RemoteException {
		this.graph = graph;
		this.v1 = v1;
		this.v2 = v2;
		this.level = new AtomicInteger(level);
		this.color = new AtomicInteger(0);
	}

	/**
	 * Referencja na lokalny graf, gdyż w momencie tworzenia krawędzi, jako
	 * parametr konstruktora, przekazywana jest referencja jaką dostaje
	 * wierzchołek w trakcie tworzenia obiektu.
	 */
	@Override
	public RemoteGraph getGraph() {
		return graph;
	}
```

```
	/**
	 * Pobranie wartości wagi krawędzi.
	 */
	@Override
	public Integer getLevel() {
		synchronized(level) {
			return level.get();
		}
	}

	/**
	 * Ustawienie wartości wagi krawędzi.
	 */
	@Override
	public void setLevel(Integer level) {
		synchronized(level) {
			this.level.set(level);
		}
	}

	/**
	 * Uzyskanie koloru krawędzi.
	 */
	@Override
	public Integer getColor() {
		synchronized(color) {
			return color.get();
		}
	}

	/**
	 * Uzyskanie referencji na początek krawędzi skierowanej.
	 */
	@Override
	public RemoteVertex getSource() throws RemoteException {
		return v1;
	}

	/**
	 * Uzyskanie referencji na koniec krawędzi skierowanej.
	 */
	@Override
	public RemoteVertex getDest() throws RemoteException {
		return v2;
	}

	/**
	 * Przeliczenie koloru.
	 */
	@Override
	public void computeColor() throws RemoteException {
		int level;
		synchronized(this.level) {
			level = this.level.get();
		}
		synchronized(color) {
			color.set((int)(255.0*((double)(level - graph.getMinEdgeLength())
				/(double)(graph.getMaxEdgeLength() - graph.getMinEdgeLength()))));
		}
	}
}
```

Interfejs użytkownika aplikacji klienckiej, która łączy się z serwerem, ma następującą postać:



Widok główny, zawierający informacje o wszystkich wierzchołkach, jak i krawędziach, jakie są obecnie w grafie.



Widok szczegółowy dla danej krawędzi, z możliwością dodania nowej krawędzi skierowanej, wychodzącej z obecnie wybranego wierzchołka.