AmbulanceServices - specyfikacja implementacyjna

Aleksandra Kowalczyk, Kacper Seredyn, Kacper Achramowicz

Grudzień 2020

Spis treści

1 Opis pakietów

1.1 Pakiet `reading`

Pakiet `reading` jest odpowiedzialny za odczytanie danych z plików, obsługę wyjątków i wyświetlenie odpowiednich komunikatów. Powiązany z pakietami `gui` i `main`.

1.2 Pakiet `geometry`

Pakiet zawiera klasę, która implementuje i obsługuje punkty na mapie. Zawiera metodę odpowiedzialną za wyznaczenie strony wektora, po której znajduje się punkt. Zawiera również implementację wzoru, który wyznacza przecięcia dróg, jego współrzędne i stosunek otrzymanych odcinków. Pakiet powiązany jest z pakietem main (pobranie danych), i pakietami graph, graham (przekazanie niezbędnych danych, a także informacji o współrzędnych punktów i ich położeniu).

1.3 Pakiet `graham`

Pakiet zawiera implementacje algorytmu Grahama, odpowiedzialnego za wyznaczenie otoczki wypukłej(granic kraju). Powiązany jest z pakietem `geometry` (pobranie danych, metody odpwiedzialne za wyznaczenie położenia punktów względem danych wektorów, informacje o współrzędnych punktów).

1.4 Pakiet `graph`

Pakiet `graph` zawiera implemntacje algorytmu konstrukcji grafu, który pozwala stworzyć graf szpitali i dróg oraz algorytmu Dijkstry, który wyznacza nakrótsze drogi między punktami. Korzysta z pakietu `geometry` (pobranie informacji o drogach, o współrzędnych punktów, przecinania linii,pobranie wyznaczonych granic).

1.5 Pakiet `gui`

Pakiet gui odpowiedzialy jest za obsługę graficzną i zmiany wyglądu ekranu działania programu. Powiązany jest ze wszystkimi pakietami.

1.6 Pakiet `main`

Pakiet main zawiera reprezentacje obiektu szpital, pacjent i obiekt(pomnik), a także stanu pacjenta. Pakiet main zawiera klasę odpowiedzialną za uruchomienie programu i wywoływanie kolejnych działań. Powiązany z pakietem reading(pobranie danych z plików), gui i pakietem geometry.

2 Opis algorytmów

2.1 Alogrytm wyznaczenia, po której stronie wektora znajduje się punkt

Algorytm ten korzysta z utworzonych w algorytmie Grahama wektorów i determinuje, czy dany punkt znajduje się po lewej, czy po prawej stronie każdego z nich. Jeżeli punkt znajduje się po lewej stronie każdego wektora to znaczy, że znajduje się w środku otoczki wypukłej.
Używając dwóch punktów leżących na wektorze (A, B) sprawdzamy, czy punkt C leży po lewej stronie wektora przy użyciu poniższego wzoru:

$$((bX - aX) * (cY - aY) - (bY - aY) * (cX - aX)) > 0$$

Jeżeli nierówność okaże się prawdziwa, to znaczy, że punkt leży po lewej stronie wektora.

2.2 Algorytm przecięcia linii (line-line intersection)

Algorytm ten określa współrzędne puntku przecięcia linii oraz stosunek w jakim każdy z tych odcinków został podzielony. Korzystamy z lekko zmodyfikowanej wersji oryginalnego algorytmu, ponieważ szukamy punktu przecięcia odcinków, a nie prostych. Z tego względu najpierw zapisujemy linie L1 i L2 w postaci parametrów Béziera pierwszego stopnia, czyli:
$L_1 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + t\begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix}, \qquad L_2 = \begin{pmatrix} x_3 \\ y_3 \end{pmatrix} + u\begin{pmatrix} x_4 - x_3 \\ y_4 - y_3 \end{pmatrix},$

gdzie t i u są liczbami rzeczywistymi:

$$t = \frac{\begin{vmatrix} x_1 - x_3 & x_3 - x_4 \\ y_1 - y_3 & y_3 - y_4 \end{vmatrix}}{\begin{vmatrix} x_1 - x_2 & x_3 - x_4 \\ y_1 - y_2 & y_3 - y_4 \end{vmatrix}} = \frac{(x_1 - x_3)(y_3 - y_4) - (y_1 - y_3)(x_3 - x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}$$

$$u = -\frac{\begin{vmatrix} x_1 - x_2 & x_1 - x_3 \\ y_1 - y_2 & y_1 - y_3 \end{vmatrix}}{\begin{vmatrix} x_1 - x_2 & x_3 - x_4 \\ y_1 - y_2 & y_3 - y_4 \end{vmatrix}} = -\frac{(x_1 - x_2)(y_1 - y_3) - (y_1 - y_2)(x_1 - x_3)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)},$$

Wtedy współrzędne punktu przecięcia znajdujemy według jednego ze wzorów:
$(P_x, P_y) = (x_1 + t(x_2 - x_1),\ y_1 + t(y_2 - y_1))$
$(P_x, P_y) = (x_3 + u(x_4 - x_3),\ y_3 + u(y_4 - y_3))$
Jeżeli t >= 0 i t <= 1 to wtedy punkt przecięcia przypada na pierwszy odcinek, natomiast gdy u >= 0 i u <= 1.

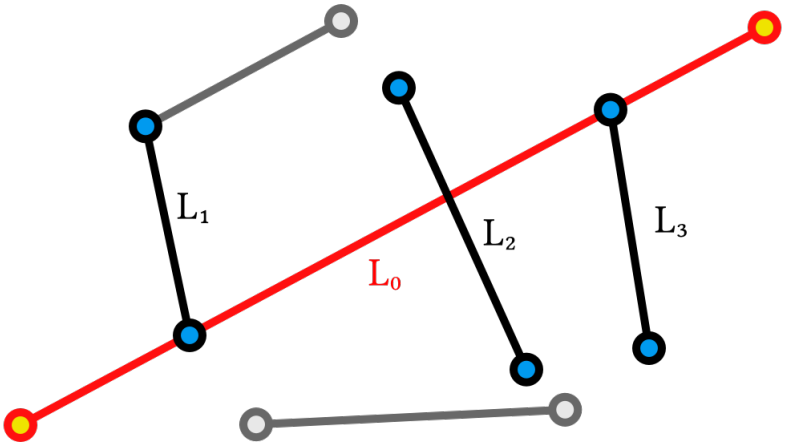
2.3 Algorytm konstrukcji grafu

Aby utworzyć graf szpitali oraz dróg je łączących, należy najpierw odpowiednio dokonać obróbki surowych danych połączeń - przecięcia odcinków dróg powinny skutkować utworzeniem skrzyżowań.
Algorytm konstrukcji grafu działa przy poniższych założeniach:

- Linie połączeń dodawane są do algorytmu pojedynczo,
- Algorytm sprawdza interakcje nowych linii z liniami już odpowiednio przeciętymi.

Dzięki powyższym obostrzeniom, algorytm może przyjąć postać iteracyjną, wykonywaną dla każdego odcinka drogi w systemie:

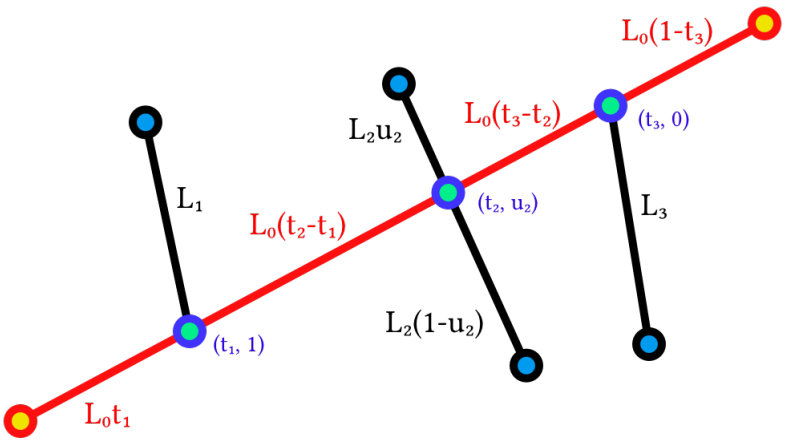
1. *Obliczenie przecięć z liniami już przetworzonymi:*



Przy użyciu algorytmu z sekcji 2.2, znajdywane są linie, które się przecinają z drogą tnącą (zaznaczoną na czerwono), wraz z pozycjami (t, u) punktu przecięcia:

- t - pozycja punktu przecięcia na drodze tnącej,
- u - pozycja punktu przecięcia na drodze ciętej.

2. *Tworzenie nowych dróg:*



Na podstawie wartości (t, u) tworzone są nowe drogi o odpowiednio obliczonych długościach (jak na powyższym diagramie). W przypadku, gdy wartość u nie jest równa 0 lub 1, tworzony jest dodatkowy punkt skrzyżowania.

2.4 Algorytm Dijkstry

Do wyznaczenia najkrótszych dróg między szpitalami, program używa algorytmu grafowego Dijsktry:

1. *Przygotowanie algorytmu*:
   Przed rozpoczęciem iteracji po węzłach grafu, konieczne jest przygotowanie kilku zmiennych i dokonanie pewnych operacji:
   - V[n] - tablica wszystkich n węzłów grafu,
   - S - węzeł źródłowy,
   - D[n] - tablica odległości n węzłów od węzła S.
     Wszystkie wartości tablicy D są zainicjalizowane na dodatnią nieskończoność, poza D[j] = 0, dla takiego j, gdzie V[j] = S.
   - P[n] - tablica poprzedników.
     Wszystkie wartości tablicy P są zainicjalizowane na `null`.
   - Wszystkie węzły grafu zostają odznaczone (`Graph.setAllMarks`).
2. *Iteracja po V[n]*:
   (a) Pobierany jest *nieoznaczony* węzeł V[i] = N o najmniejszej wartości D[i]. Jest on następnie oznaczany (`Graph.setMark`).

(b) Dla każdego nieoznaczonego sąsiada V[j] = P węzła N obliczana jest nowa odległość: $L = D[i] + len(N, P)$, gdzie len(A, B) - długość krawędzi łączącej A i B.
Jeśli $L < D[j]$, to $D[j] = L$ i $P[j] = N$.

(c) Jeśli pozostały w grafie nieoznaczone węzły, powrót to kroku pierwszego iteracji.

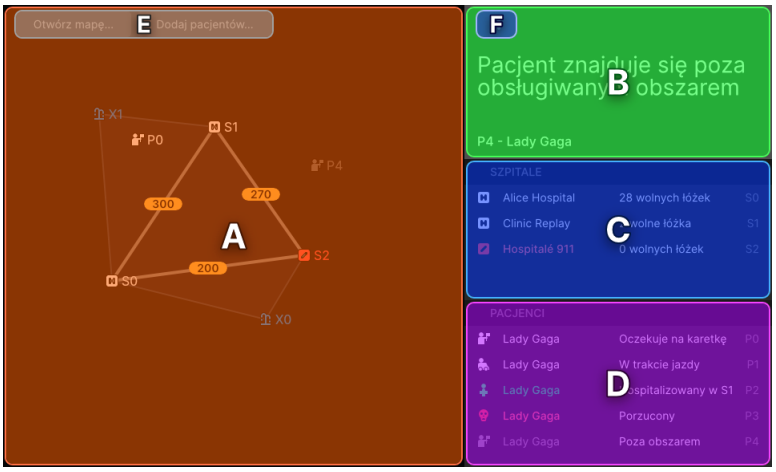
2.5 Algorytm Grahama

Algorytm Grahama służy do znajdowania otoczki wypukłej. Na początku znajduje punkt najbardziej wysunięty na lewo. Jeżeli jest kilka takich punktów wybiera punkt znajdujący się najbardziej na dole. Jest to tzw. punkt bazowy. Kolejne punkty są sortowane względem kątów nachylenia ich wektorów względem osi, którą w płaszczyźnie x tworzy punkt startowy. Algorytm przechodząc od punktu startowego do kolejnych punktów z posortowanej listy,umieszcza je na stosie i sprawdza kierunek, w którym nastąpiło to przejście:

- jeżeli odchylenie nastąpiło w prawą stronę, zdejmowany jest wierzchołek ze stosu
- jeżeli odchylenie nastąpiło w stronę lewą, wierzchołek pozostaje na stosie

Finalnie otrzymujemy stos, który zawiera jedynie te punkty, które tworzą otoczkę wypukłą.

3 Opis graficznego interfejsu użytkownika



- A - mapa szpitali, obiektów i położeń pacjentów,
- B - sekcja informacyjna,
- C - lista szpitali,
- D - lista pacjentów,
- E - przyciski ładowania plików,
- F - przycisk kontroli symulacji ("Uruchom"/"Wstrzymaj")

3.1 Przyciski *ładowania plików*

Słuchaczem przycisku "Otwórz mapę..." i "Dodaj pacjentów..." jest klasa reading.Reader która pobiera i przetwarza dane z pliku mapy i pliku pacjenci. Wynikiem naciśnięcia przycisku są pola "Mapa szpitali, obiektów i pacjentów", "Lista szpitali", "Lista pacjentów".

3.2 Przyciski *kontroli symulacji*

Przyciski "Uruchom"/"Wstrzymaj" służą do rozpoczęcia/ wstrzymania działania przeprowadzania symulacji. Słuchaczami przycisków jest klasa odpowiedzialna na rysowanie na ekranie kolejnych stanów obsługi pacjenta.

3.3 Pole *Mapa szpitali, obiektów i położeń pacjentów*

W tym polu wyświetlana jest mapa, zawierająca wyznaczoną otoczkę wypukłą, obiekty, szpitale, drogi między nimi wraz z ich odległościami, skrzyżowania i

oznaczenia pacjentów wraz z ich aktualnym stanem. Słuchaczami tego pola są klasy z pakietu graph, geometry, graham.

3.4 Pole *Sekcja informacyjna*

Pole wyświetla infomacje o aktualnym stanie obsługi danego pacjenta. Słuchaczem pola jest klasa main.Patient oraz klasa main.State.

3.5 Pole *Lista szpitali*

Słuchaczem pola jest klasa main.Hospital. W tym polu wyświetlane są nazwy szpitali, liczba dostępnych wolnych łóżek,która na bieżąco jest aktualizowana i skrót, którymi oznaczone są na mapie.

3.6 Pole *Lista pacjentów*

Słuchaczem pola jest klasa Patient. W tym polu wyświetlane są id pacjentów, stan przyjęcia, który jest na bieżąco aktualizowany i skrót, którymi oznaczeni są na mapie.

4 Opis klas

Poniżej znajduje się lista pól i metod klas projektu. Diagram UML klas znajduje się na końcu rozdziału.

4.1 Klasa `reading.Reader`

Wczytuje dane z pliku do programu.

4.1.1 Pole `Reader.Parser`

`private final Parser parser`
Obiekt `Parser`.

4.1.2 Metoda `Reader.load`

`public State load(String fileName)`
Wczytuje dane z pliku, którego nazwa została podana jako argument wywołania.

4.2 Klasa `reading.Parser`

Zajmuje się obróbką danych przekazanych do programu.

4.2.1 Metoda `Parser.parseHospital`

`public void parseHospital(State state, String[] buffer)`
Tworzy obiekt `Hospital` na podstawie argumentu `buffer` i dodaje go do listy szpitali w `state`.

4.2.2 Metoda `Parser.parseLandmark`

`public void parseLandmark(State state, String[] buffer)`
Tworzy obiekt `Landmark` na podstawie argumentu `buffer` i dodaje go do listy obiektów w `state` .

4.2.3 Metoda `Parser.parsePatient`

`public void parsePatient(State state, String[] buffer)`
Tworzy obiekt `Patient` na podstawie argumentu `buffer` i dodaje go do listy pacjentów w `state`.

4.2.4 Metoda `Parser.parseConnection`

`public void parseConnection(State state, String[] buffer)`
Tworzy obiekt `Connection` na podstawie argumentu `buffer`. i dodaje go do listy połączeń w `state`.

4.3 Klasa `geometry.LineIntersection`

Przechowuje metody pozwalające na wyznaczenie punktu przecięcia linii i stosunku, w jakim zostały te linie podzielone.

4.3.1 Metoda `LineIntersection.intersect`

`public double[] intersect`
Metoda jest implementacją algorytmu przecięcia linii.

4.4 Klasa `geometry.Point`

Klasa będąca bazą dla klas Hospital, Patient i Landmark. Zawiera metody pozwalające pobrać informacje punktach na mapie.

4.4.1 Pole `Point.x`

`private double x`
Przechowuje wartość współrzędnej X punktu.

4.4.2 Pole `Point.y`

`private double y`
Przechowuje wartość współrzędnej Y punktu.

4.4.3 Konstruktor `Point`

`public Point(double x, double y)`
Tworzy obiekt `Point` o podanych współrzędnych.

4.4.4 Konstruktor `Point`

`public Point(double x, Point start, Point end)`
Tworzy obiekt `Point` w miejscu wyznaczonym przez proporcję `x` na odcinku utworzonym przez dwa `punkty`.

4.4.5 Metoda `Point.getRelativeDirection`

`public double getRelativeDirection()`
Zwraca wartość potrzebną do wyznaczenia, po której stronie wektora znajduje się punkt (algorytm 2.1).

4.4.6 Metoda `Point.getX`

`public double getX()`
Zwraca wartość współrzędnej X.

4.4.7 Metoda `Point.getY`

`public double getY()`
Zwraca wartość współrzędnej Y.

4.4.8 Metoda `Point.isLeft`

`public boolean isLeft()`
Implementuje algorytm wyznaczenia, po której stronie wektora znajduje się punkt (algorytm 2.1).

4.5 Klasa `graham.ConvexHull`

Reprezentuje otoczkę wypukłą.

4.5.1 Konstruktor `ConvexHull`

`public ConvexHull(List<Point> points)`
Tworzy obiekt `ConvexHull` na podstawie danej listy punktów otoczki.

4.5.2 Pole `ConvexHull.points`

`private List<Points> points`
Lista punktów otoczki wypukłej.

4.5.3 Metoda `ConvexHull.chooseStartPoint`

`public Point chooseStartPoint(Point point)`
Zwraca punkt `Point StartPoint`, którego współrzędna x jest najmniesza. W przypadku gdy jest kilka punktów o tej samej współrzędnej x wybiera punkt o najmniejszej wartości y.

4.5.4 Metoda `ConvexHull.calculateAngles`

`public double[] calculateAngles(Point startPoint, List<Point> points)`
Oblicza wartości kątowe między punktem startowym (osią x wyznaczoną przez punkt startowy) a pozostałymi punktami. Zwraca tablice zawierającą wartość kątową dla każdego punktu.

4.5.5 Metoda `ConvexHull.sortByAngles`

`public List<Point> sortByAngles(List<Point> points, double[] angles)`
Zwraca posortowaną listę puntów `List<Point> sortedPoints` od najmniejszej do największej wartości kątowej.

4.5.6 Metoda `ConvexHull.isPointInHull`

`public boolean isPointInHull(Point point)`
Zwraca `true` jeśli punkt `point` znajduje się wewnątrz otoczki wypukłej, w przeciwnym wypadku zwraca `false`.

4.6 Klasa `graph.Graph<T>`

Przechowuje graf nieskierowany.

4.6.1 Konstruktor `Graph`

`public Graph()`
Tworzy obiekt `Graph`.

4.6.2 Pole `Graph.edges`

`private Double[][] edges`
Macierz połączeń węzłów grafu.
Jeśli wartość $edges[i][j]$ jest równa `null`, między wierzchołkami i i j nie ma połączenia. W przeciwnym razie, liczba w $edges[i][j]$ jest długością połączenia między i i j.

4.6.3 Pole `Graph.marks`

`private List<boolean> marks`
Lista oznaczeń węzłów grafu. Oznaczenia używane są przez algorytm Dijkstry (2.4).

4.6.4 Pole `Graph.nodes`

`private List<T> nodes`
Lista węzłów grafu.

4.6.5 Metoda `Graph.addNode`

`public int addNode(T node)`
Dodaje węzeł do grafu i zwraca ilość węzłów po dodaniu.

4.6.6 Metoda `Graph.connectNodes`

`public void connectNodes(T n1, T n2, double length)`
Łączy dwa węzły grafu połączeniem o długości `length`.

4.6.7 Metoda `Graph.finalizeNodes`

`public void finalizeNodes()`
Tworzy tablicę `Double[][] edges` o wymiarach $[n][n]$ (n - liczba węzłów) zainicjalizowaną wartościami `null`.

4.6.8 Metoda `Graph.getLength`

`public Double getLength(T n1, T n2)`
Zwraca długość połączenia między węzłami `n1` i `n2` lub `null` jeśli połączenie nie istnieje.

4.6.9 Metoda `Graph.getMark`

`public boolean getMark(T node)`
Zwraca wartość oznaczenia węzła `node`.

4.6.10 Metoda `Graph.getNeighbors`

`public List<T> getNeighbors(T node)`
Zwraca sąsiadów węzła `node` (węzłów połączonych do węzła `node`).

4.6.11 Metoda `Graph.getNodes`

`public List<T> getNodes(Boolean mark = null)`
Zwraca węzły w grafie. Jeśli wartość `mark` nie jest `null`, zwracane są jedynie węzły o oznaczeniu `mark`.

4.6.12 Metoda `Graph.getPathLength`

`public double getPathLength(List<T> path)`
Zwraca długość ścieżki na podstawie listy węzłów w ścieżce.

4.6.13 Metoda `Graph.setAllMarks`

`public void setAllMarks(boolean mark)`
Ustawia oznaczenia wszystkich węzłów na `mark`.

4.6.14 Metoda `Graph.setMark`

`public void setMark(T node, boolean mark)`
Ustawia oznaczenia wszystkich węzła `node` na `mark`.

4.7 Klasa `graph.DijkstraAlgorithm<T>`

Pozwala na wykonanie algorytmu Dijkstry na grafie.

4.7.1 Konstruktor `DijkstraAlgorithm`

`public DijkstraAlgorithm(Graph<T> graph)`
Tworzy obiekt `DijkstraAlgorithm` dla danego grafu.

4.7.2 Pole `DijkstraAlgorithm.distances`

`private Double[] distances`
Tablica odległości w grafie (algorytm 2.4).

4.7.3 Pole `DijkstraAlgorithm.graph`

`private Graph graph`
Graf, na którym działa algorytm.

4.7.4 Pole `DijkstraAlgorithm.previousNodes`

`private T[] previousNodes`
Tablica poprzedników węzłów grafu (algorytm 2.4).

4.7.5 Metoda `DijkstraAlgorithm.getNextNode`

`private T getNextNode()`
Zwraca kolejny węzeł do przetwarzania w algorytmie (algorytm 2.4).

4.7.6 Metoda `DijkstraAlgorithm.iterateOverNeighbors`

`private void iterateOverNeighbors(T node)`
Dokonuje iteracji algorytmu po danym węźle (algorytm 2.4).

4.7.7 Metoda `DijkstraAlgorithm.execute`

`private void execute(T source)`
Uruchamia algorytm dla węzła startowego `source` (algorytm 2.4).

4.7.8 Metoda `DijkstraAlgorithm.getPath`

`private List<T> getPath(T target)`
Zwraca ścieżkę z węzła zdefiniowanego przy wywołaniu `execute()` a `target` (alogrytm 2.4). Jeśli ścieżka nie istnieje, zwraca `null`.

4.8 Klasa `graph.GraphConstructorLine`

Odcinek drogi między dwoma punktami na mapie.

4.8.1 Konstruktor `GraphConstructorLine`

`public GraphConstructorLine(Point start, Point end, double length)`
Tworzy obiekt `GraphConstructorLine` dla danej drogi.

4.8.2 Pole `GraphConstructorLine.end`

`private Point end`
Punkt końcowy drogi.

4.8.3 Pole `GraphConstructorLine.length`

`private double length`
Długość drogi (niekoniecznie na podstawie odległości euklidesowej między punktami).

4.8.4 Pole `GraphConstructorLine.start`

`private Point start`
Punkt początkowy drogi.

4.8.5 Metoda `GraphConstructorLine.getEnd`

`public Point getEnd()`
Zwraca punkt końcowy drogi.

4.8.6 Metoda `GraphConstructorLine.getLength`

`public double getLength()`
Zwraca długość drogi.

4.8.7 Metoda `GraphConstructorLine.getStart`

`public Point getStart()`
Zwraca punkt początkowy drogi.

4.9 Klasa `graph.GraphConstructorCut`

Przechowuje informacje o przecięciu drogi.

4.9.1 Konstruktor `GraphConstructorCut`

`public GraphConstructorCut(GraphConstructorLine line, double linePosition, double cutterPosition)`
Tworzy obiekt `GraphConstructorCut` na podstawie informacji o przecięciu.

4.9.2 Pole `GraphConstructorCut.cutterPosition`

`private double cutterPosition`
Pozycja punktu przecięcia na przecinającej linii w zakresie $[0 - 1]$.

4.9.3 Pole `GraphConstructorCut.line`

`private GraphConstructorLine line`
Przecinana linia grafu.

4.9.4 Pole `GraphConstructorCut.linePosition`

`private double linePosition`
Pozycja punktu przecięcia na przecinanej linii w zakresie $[0 - 1]$.

4.9.5 Metoda `GraphConstructorCut.getCutterPosition`

`public double getCutterPosition()`
Zwraca pozycję punktu przecięcia na przecinającej linii w zakresie $[0 - 1]$.

4.9.6 Metoda `GraphConstructorCut.getLine`

`public GraphConstructorLine getLine()`
Zwraca przecinaną linię grafu.

4.9.7 Metoda `GraphConstructorCut.getLinePosition`

`public double getLinePosition()`
Zwraca pozycję punktu przecięcia na przecinanej linii w zakresie $[0 - 1]$.

4.10 Klasa `graph.GraphConstructor`

Pozwala na utworzenie grafu na podstawie informacji o przecinających się drogach.

4.10.1 Pole `GraphConstructor.lines`

`private List<GraphConstructorLine> lines`
Lista już przetworzonych linii (algorytm 2.3).

4.10.2 Pole `GraphConstructor.points`

`private Set<Point> points`
Zbiór punktów grafu.

4.10.3 Metoda `GraphConstructor.cutLines`

`private void cutLines(GraphConstructorLine cutter, List<GraphConstructorCut> lines)`
Tworzy skrzyżowania dróg i oblicza nowe długości odcinków (algorytm 2.3).

4.10.4 Metoda `GraphConstructor.findIntersections`

`private List<GraphConstructorCut> findIntersections(GraphConstructorLine cutter)`
Znajduje kolizje dróg i tworzy listę przecinanych dróg (algorytm 2.3 oraz 2.2).

4.10.5 Metoda `GraphConstructor.addLine`

`public void addLine(Point start, Point end, double length)`
Dodaje linię do algorytmu.

4.10.6 Metoda `GraphConstructor.constructGraph`

`public Graph<Point> constructGraph()`
Generuje ostateczny graf.

4.11 Klasa `main.Hospital`

Reprezentuje obiekt szpitala.
Klasa `Hospital` dziedziczy po klasie `Point`.

4.11.1 Konstruktor `Hospital`

`public Hospital(int id, double x, double y, String name, int vacantBeds)`
Tworzy obiekt `Hospital`.

4.11.2 Pole `Hospital.id`

`private int id`
Identyfikator szpitala.

4.11.3 Pole `Hospital.name`

`private String name`
Nazwa szpitala.

4.11.4 Pole `Hospital.vacantBeds`

`private int vacantBeds`
Liczba wolnych łóżek.

4.11.5 Metoda `Hospital.getId`

`public int getId()`
Zwraca identyfikator szpitala.

4.11.6 Metoda `Hospital.getName`

`public String getName()`
Zwraca nazwę szpitala.

4.11.7 Metoda `Hospital.getVacantBeds`

`public int getVacantBeds()`
Zwraca liczbę wolnych łóżek w szpitalu.

4.12 Klasa `main.Landmark`

Reprezentuje inny obiekt mapy (np. pomnik), używany do wyznaczenia otoczki wypukłej.
Klasa `Landmark` dziedziczy po klasie `Point`.

4.12.1 Konstruktor `Landmark`

`public Landmark(int id, double x, double y, String name, int vacantBeds)`
Tworzy obiekt `Landmark`.

4.12.2 Pole `Landmark.id`

`private int id`
Identyfikator obiektu.

4.12.3 Pole `Landmark.name`

`private String name`
Nazwa obiektu.

4.12.4 Metoda `Landmark.getId`

`public int getId()`
Zwraca identyfikator obiektu.

4.12.5 Metoda `Landmark.getName`

`public String getName()`
Zwraca nazwę obiektu.

4.13 Klasa `main.Patient`

Reprezentuje obiekt pacjenta.
Klasa `Patient` dziedziczy po klasie `Point`.

4.13.1 Konstruktor `Patient`

`public Patient(int id, double x, double y)`
Tworzy obiekt `Patient`.

4.13.2 Pole `Patient.id`

`private int id`
Identyfikator pacjenta.

4.13.3 Pole `Patient.name`

`private String name`
Imię i nazwisko pacjenta wygenerowane metodą `generateName()`

4.13.4 Metoda `Patient.generateName`

`private String generateName()`
Generuje losowe imię i nazwisko dla pacjenta.

4.13.5 Metoda `Patient.getId`

`public int getId()`
Zwraca identyfikator pacjenta.

4.13.6 Metoda `Patient.getName`

`public String getName()`
Zwraca imię i nazwisko pacjenta.

4.13.7 Metoda `Patient.getState`

`public PatientState getState()`
Zwraca aktualny stan pacjenta.

4.13.8 Metoda `Patient.setState`

`public void setState(PatientState state)`
Ustawia stan pacjenta.

4.14 Enumeracja `main.PatientState`

Deklaruje stany pacjenta.

4.14.1 Wartość waiting

`waiting = 0`
Deklaruję stan - pacjent oczekuje na przyjazd karetki i przypisuje mu wartość 0.

4.14.2 Wartość outOfBounds

`outOfBounds = 1`
Deklaruję stan - pacjent znajduje się poza granicami kraju i przypisuje mu wartość 1.

4.14.3 Wartość riding

`riding = 2`
Deklaruję stan - pacjent jedzie do danego szpitala i przypisuje mu wartość 2.

4.14.4 Wartość rejected

`rejected = 3`
Deklaruję stan - pacjent został odrzucony w danym szpitalu i przypisuje mu wartość 3.

4.14.5 Wartość accepted

`accepted = 4`
Deklaruję stan - pacjent został przyjęty do dango szpitala i przypisuje mu wartość 4.

4.14.6 Wartość abandoned

`abandoned = 5`
Deklaruję stan - pacjent został porzucony(nigdzie nie może zostać przyjęty) i przypisuje mu wartość 5.

4.15 Klasa `main.State`

Reprezentuje obiekt stanu pacjenta.

4.15.1 Konstruktor `State`

`public State()`
Tworzy obiekt `State`.

4.15.2 Pole `graphConstructor`

`private GraphConstructor graphConstructor`
Konstruktor grafu szpitali.

4.15.3 Pole `hospitals`

`private List<Hospital> hospitals`
Lista szpitali.

4.15.4 Pole `hospitalPaths`

`private List<Point>[][] hospitalPaths`
Lista zawierająca współrzędne szpitali.

4.15.5 Pole `hospitalPathLengths`

`private Double[][] hospitalPathLengths`
Lista zawierająca długość dróg między szpitalami.

4.15.6 Pole `landmark`

`private List<Landmark> landmark`
Lista obiektów.

4.15.7 Pole `patients`

`private List<Patient> patients`
Lista pacjentów.

4.15.8 Metoda `State.addConnection`

`public void addConnection(Hospital h1, Hospital h2, double length)`
Dodaje do mapy połączenia (drogi) między szpitalami.

4.15.9 Metoda `State.addLandmark`

`public void addLandmark(Landmark landmark)`
Dodaje obiekt (pomnik) do mapy.

4.15.10 Metoda `State.addHospital`

`public void addHospital(Hospital hospital)`
Dodaje szpital do mapy.

4.15.11 Metoda `State.addPatient`

`public void addPatient(Patient patient)`
Nanosi pacjenta na mapę.

4.15.12 Metoda `State.finalizeConnections`

`public void finalizeConnections()`
Finalizuje dodawanie połączeń i tworzy ostateczną sieć dróg.

4.15.13 Metoda `State.getHospitalById`

`public Hospital getHospitalById(int id)`
Zwraca szpital na podstawie podane identyfikatora.

4.15.14 Metoda `State.getHospitalPaths()`

`public List<Point>[][] getHospitalPaths()`
Zwraca macierz dróg między szpitalami.

4.15.15 Metoda `State.getHospitalPathLengths`

`public Double[][] getHospitalPathLengths()`
Zwraca macierz długości dróg między szpitalami.

4.15.16 Metoda `State.getHospitals`

`public List<Hospital> getHospitals()`
Zwraca listę szpitali.

4.15.17 Metoda `State.getLandmarks`

`public List<Landmark> getLandmarks()`
Zwraca listę pomników.

4.15.18 Metoda `State.getNextPatient`

`public Patient getNextPatient()`
Zwraca kolejnego pacjenta z listy pacjentów do przetwarzania (kolejny nieobsłużony pacjent).

4.15.19 Metoda `State.getPatientById`

`public Patient getPatientById(int id)`
Zwraca pacjenta na podstawie podanego identyfikatora.

4.15.20 Metoda `State.getPatients`

`public List<Patient> getPatients()`
Zwraca listę pacjentów.

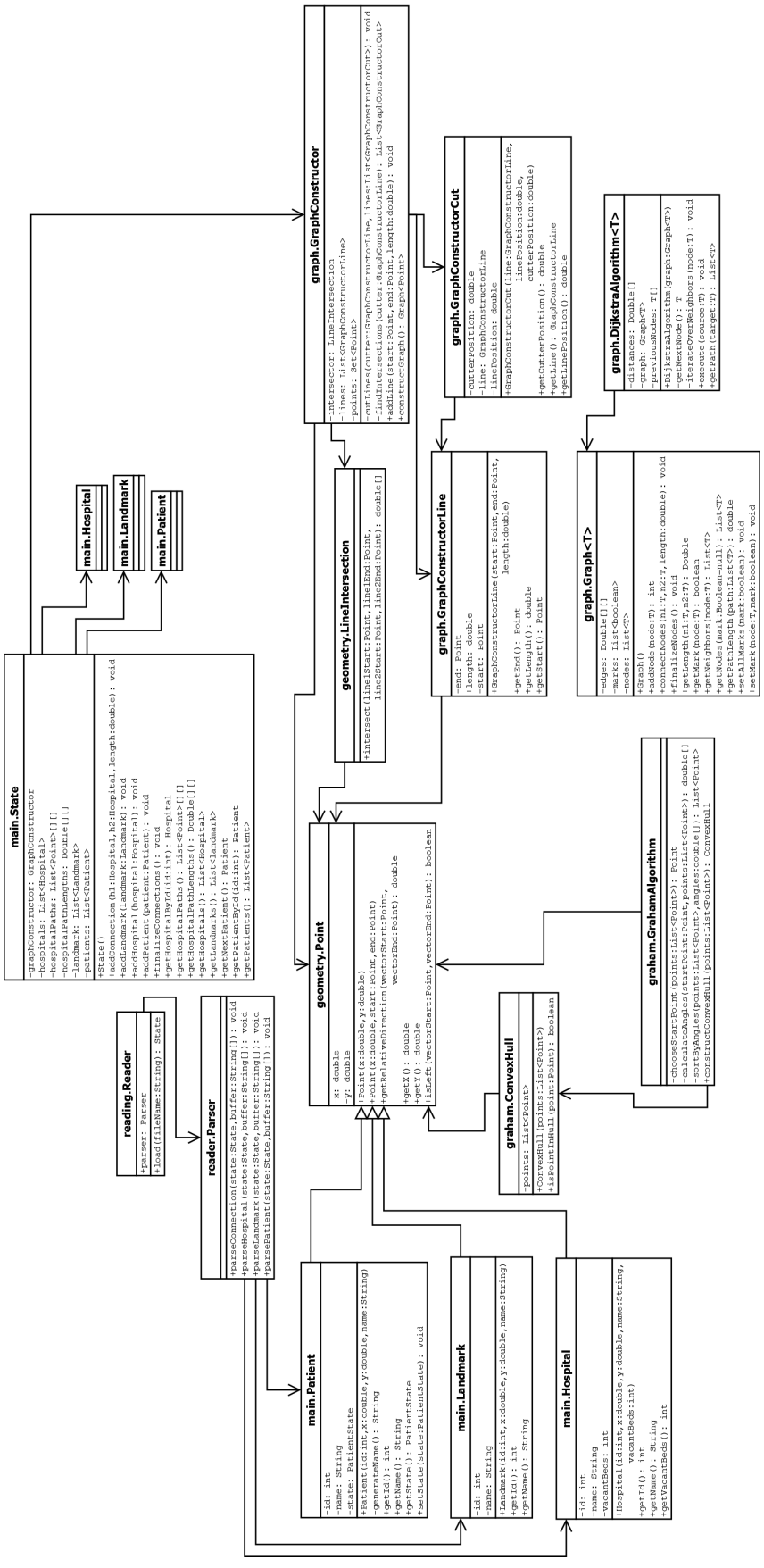


Figure 1: Diagram UML klas projektu.

5 Testowanie

Do testów jednostkowych wykorzystamy narzędzie JUnit 5. Program jest złożony, wykorzystuje kilka oddzielnych algorytmów, dlatego staramy się jak najdokładniej przetestować każdy z nich. Za działanie każdego algorytmu odpowiedzialnych jest kilka/kilkanaście metod, które chcemy szczegółowo przetestować. Graficzny interfejs użytkownika testować będziemy podczas pisania programu. Po każdej operacji związanej ze zmianą wyglądu gui będziemy sprawdzać, czy zmiana nastąpiła prawidłowo, a także czy wszystkie dane są widoczne i czytelne.

5.1 Warunki brzegowe

Podczas testowania musimy zwrócić szczególną uwagę na następujące przypadki:

- czy klasa `reading.Parser` prawidłowo rozpatruje wszystkie możliwe błędy, czy wyświetla się stosowny komunikat
- w klasie `graham.ConvexHull` należy sprawdzić, czy metody dobrze obliczają kąty i prawidłowo je sortują, a także czy położenie wszystkich obiektów jest brane pod uwagę przy wyznaczaniu otoczki wypukłej
- czy metoda `isLeft()` z klasy geometry.Point za każdym razem prawidłowo określa położenie punktu względem wektora
- czy proporcje odcinków po przecięciu są prawidłowo obliczane w klasie graph.GraphConstructorLine
- czy klasa `graph.DijkstraAlgorithm<T>` bierze pod uwagę wszytskie możliwe połączenia(nie pomija żadnych dróg)
- czy w klasie `graph.DijkstraAlgorithm<T>` prawidłowo jest określana długość dróg i czy w każdym przypadku wybierana jest najkrótsza z nich
- czy klasa `graph.GraphConstructor` korzysta ze wszystkich dostępnych skrzyżowań (przecięć dróg między szpitalami)

5.2 Warunki brzegowe dla graficznego interfejsu użytkownika

- czy lista szpitali/pacjentów jest na bieżąco aktualizowana
- czy sekcja informacyjna jest aktualizowana, czy wyświetla prawidłowe informacje
- czy podczas manualnego dodawania pacjenta współrzędne jego położenia są prawidłowo określane i wczytywane
- czy mapa jest poprawnie załadowana, widoczna w całości (odpowiednie wyskalowanie), punkty na mapie są podpisane i czytalne, a oznaczenia symbloliczne i kolorystyczne poprawne