

Библиотека сетевого анализа QGIS: описание и примеры

[Обсудить в форуме](#) Комментариев — 4

Эта страница опубликована в основном списке статей сайта по адресу <http://gis-lab.info/qa/qgis-network-analysis-lib.html>

В статье описаны основные сведения и приемы работы с QGIS network-analysis library — библиотекой сетевого анализа ГИС Quantum GIS. Статья дополнена готовыми скриптами-примерами, которые можно использовать при разработке своих расширений.

QGIS network-analysis library — библиотека входящая в состав свободной ГИС Quantum GIS, которая:

- может создавать математический граф из географических данных (линейных векторных слоев), пригодный для анализа методами теории графов
- реализует базовые методы теории графов (в настоящее время только метод Дейкстры)

Содержание

- [1 История](#)
- [2 Применение](#)
 - [2.1 Получение графа](#)
 - [2.2 Анализ графа](#)
 - [2.2.1 Нахождение кратчайших путей](#)
 - [2.2.2 Нахождение областей доступности](#)
- [3 Актуальная документация](#)

История

Библиотека QGIS network-analysis появилась путем экспорта базовых функций из плагина RoadGraph в отдельную библиотеку.

Начиная с [ee19294562](#), появилась возможность использовать функционал библиотеки в своих расширениях, а также из Консоли Python QGIS.

Применение

Алгоритм применения библиотеки network-analysis можно записать в трех шагах:

1. Получить граф из географических данных
2. Выполнить анализ графа
3. Использовать результат анализа в своих целях, например, визуализировать

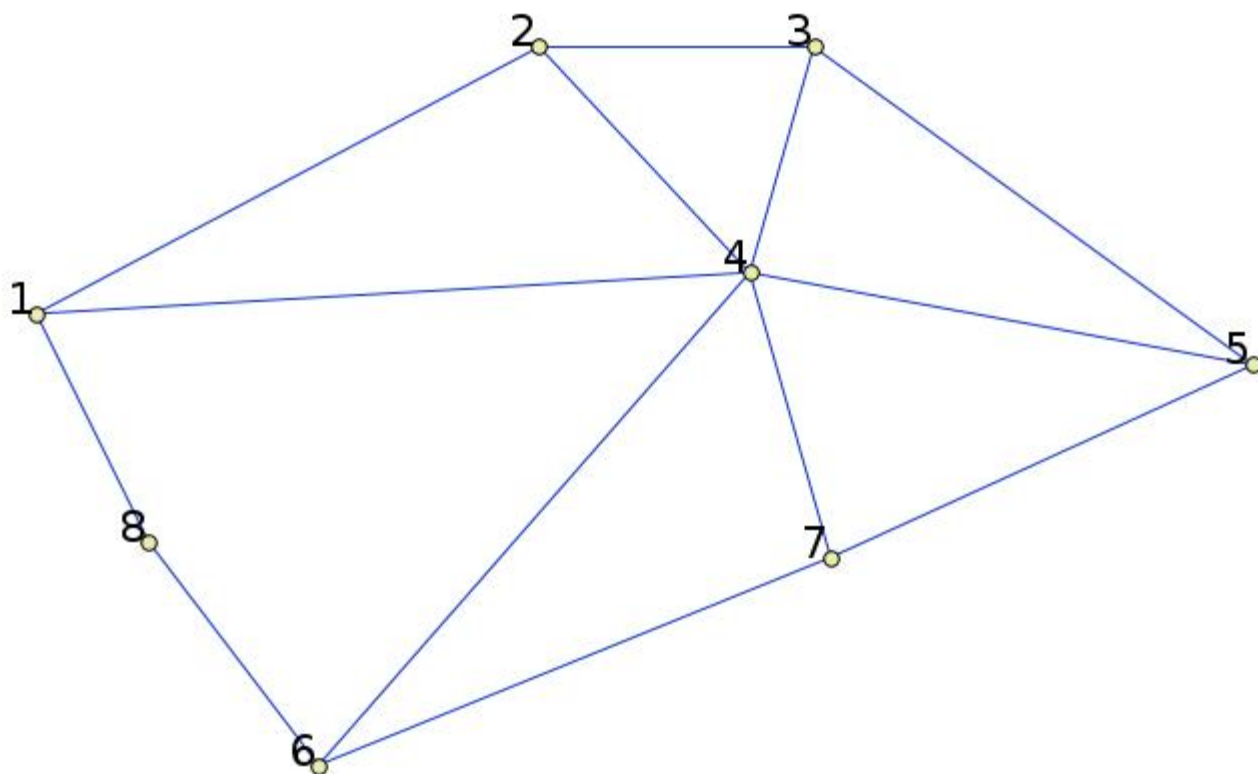
Получение графа

Первое, что нужно сделать — это подготовить исходные данные, т.е. преобразовать векторный слой в граф. Все дальнейшие действия будут выполняться именно с этим графом.

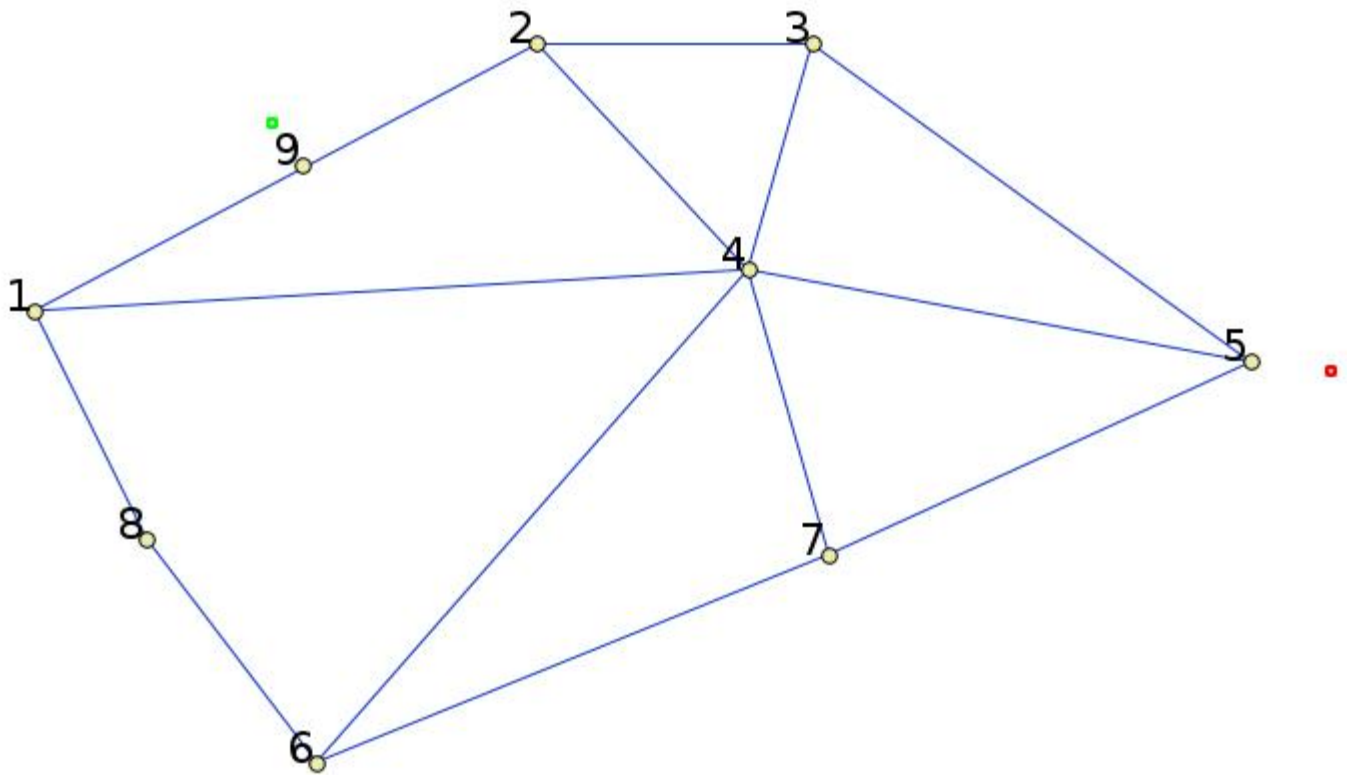
В качестве источника графа может выступать любой линейный векторный слой. Узлы линий образуют


множество вершин графа. В качестве ребер графа выступают отрезки линий векторного слоя. Узлы, имеющие одинаковые координаты, считаются одной и той же вершиной графа. Таким образом, две линии, имеющие общий узел, оказываются связанными между собой.

В дополнение к этому, при построении графа можно «привязать» к векторному слою любое количество дополнительных точек. Для каждой дополнительной точки будет найдено соответствие — либо ближайшая вершина графа, либо ближайшее ребро. В последнем случае ребро будет разбито на две части и будет добавлена новая общая вершина.



Граф без дополнительных точек




 Граф с двумя дополнительными («привязанными») точками. Красной точке соответствует вершина №5. Зеленой точке соответствует новая добавленная вершина №9

В качестве свойств ребер графа могут быть использованы атрибуты векторного слоя и протяженность (длина) ребра.

Реализация построения графа из векторного слоя использует шаблон программирования [строитель](#). За построение графа дорог отвечает так называемый Director. В настоящее время библиотека располагает только одним директором: [QgsLineVectorLayerDirector](#). Директор задает основные настройки, которые будут использоваться при построении графа по линейному векторному слою, и «руками» строителя [QgsGraphBuilder](#) выполняет создание графа типа [QgsGraph](#). В настоящее время, как и в случае с директором, реализован только один строитель: [QgsGraphBuilder](#), создающий граф [QgsGraph](#). При желании можно реализовать строителя, который будет строить граф, совместимый с такими библиотеками как [BGL](#) или [networkX](#).

Для вычисления свойств ребер используется шаблон проектирования [стратегия](#). Пока в библиотеке реализована только одна стратегия, учитывающая длину маршрута: [QgsDistanceArcProperter](#). При необходимости, можно создать свою стратегию, которая будет учитывать необходимые параметры. Например, в модуле Road graph используется стратегия, вычисляющая время движения по ребру графа на основании длины ребра и поля скорости.

Рассмотрим процесс создание графа более подробно.

Чтобы получить доступ к функциям библиотеки сетевого анализа необходимо импортировать модуль networkanalysis

```
from qgis.networkanalysis import *
```

Теперь нужно создать директора

```
# не использовать информацию о направлении движения из атрибутов слоя, все дороги
```

трактуются как двусторонние

```
director = QgsLineVectorLayerDirector( vLayer, -1, '', '', '', 3 )
```

информация о направлении движения находится в поле с индексом 5. Односторонние дороги с прямым направлением

движения имеют значение атрибута "yes", односторонние дороги с обратным направлением — "1", и соответственно

двусторонние дороги — "no". По умолчанию дороги считаются двусторонними. Такая схема подходит для использования

с данными OpenStreetMap

```
director = QgsLineVectorLayerDirector( vLayer, 5, 'yes', '1', 'no', 3 )
```

В конструктор директора передается линейный векторный слой, по которому будет строиться граф, а также информация о характере движения по каждому сегменту дороги (разрешенное направление, одностороннее или двустороннее движение). Рассмотрим эти параметры:

- `vl` — векторный слой, по которому будет строиться граф.
- `directionFieldId` — индекс поля атрибутивной таблицы, которое содержит информацию о направлении движения. -1 не использовать эту информацию
- `directDirectionValue` — значение поля, соответствующее прямому направлению движения (т.е. движению в порядке создания точек линии, от первой к последней)
- `reverseDirectionValue` — значение поля, соответствующее обратному направлению движения (от последней точки к первой)
- `bothDirectionValue` — значение поля, соответствующее двустороннему движению (т.е. допускается движение как от первой точки к последней, так и в обратном направлении)
- `defaultDirection` — направление движения по умолчанию. Будет использоваться для тех участков дорог, у которых значение поля `directionFieldId` не задано или не совпадает ни с одним из вышеперечисленных.

Следующим шагом необходимо создать стратегию назначения свойств ребрам графа

```
properter = QgsDistanceArcProperter()
```

Сообщаем директору об используемой стратегии. Один директор может использовать несколько стратегий

```
director.addProperter( properter )
```

Теперь создаем строителя, который собственно и будет строить граф заданного типа.

Конструктор `QgsGraphBuilder` принимает следующие параметры:

- `crs` — используемая система координат. Обязательный параметр.
- `otfEnabled` — указывает на использование перепроецирования «на лету». По умолчанию `true`.
- `topologyTolerance` — топологическая толерантность. Значение по умолчанию 0.
- `ellipsoidID` — используемый эллипсоид. По умолчанию "WGS84".

задана только используемая СК, все остальные параметры по умолчанию

```
builder = QgsGraphBuilder( myCRS )
```

Также можно задать одну или несколько точек, которые будут использоваться при анализе. Например так:

```
startPoint = QgsPoint( 82.7112, 55.1672 )
```

```
endPoint = QgsPoint( 83.1879, 54.7079 )
```

Затем строим граф и «привязываем» к нему точки

```
tiedPoints = director.makeGraph( builder, [ startPoint, endPoint ] )
```

Построение графа может занять некоторое время (зависит от количества объектов в слое и размера самого слоя). В `tiedPoints` записываются координаты «привязанных» точек. После построения мы получим граф, пригодный для анализа

```
graph = builder.graph()
```

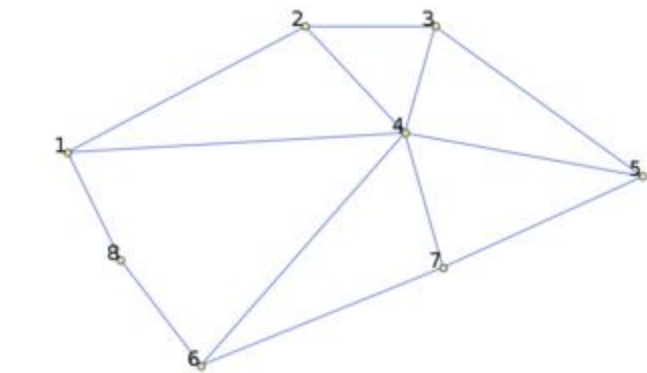
Теперь можно получить индексы наших точек

```
startId = graph.findVertex( tiedPoints[ 0 ] )  
endId = graph.findVertex( tiedPoints[ 1 ] )
```

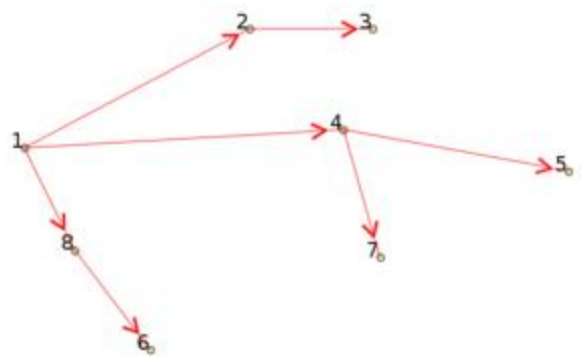
Анализ графа

В основе сетевого анализа лежат задача связности вершин графа и задача поиска кратчайших путей. Для решения этих задач в библиотеке network-analysis реализован [алгоритм Дейкстры](#).

Алгоритм Дейкстры находит оптимальный маршрут от одной из вершин графа до всех остальных и значение оптимизируемого параметра. Хорошим способом представления результата выполнения алгоритма Дейкстры является [дерево кратчайших путей](#).



Исходный граф



Дерево кратчайших путей с корнем в вершине №1.

Дерево кратчайших путей — это ориентированный взвешенный граф (точнее дерево) обладающий следующими свойствами:

- только одна вершина не имеет входящих в нее ребер — корень дерева
- все остальные вершины имеют только одно входящее в них ребро
- Если вершина B достижима из вершины A, то путь, соединяющий их, единственный и он же кратчайший (оптимальный) на исходном графе.

Дерево кратчайших путей можно получить вызывая методы `shortestTree` и `dijkstra` класса [QgsGraphAnalyzer](#). Рекомендуется пользоваться именно методом `dijkstra`. Он работает быстрее и, в общем случае, эффективнее расходует память. Метод `shortestTree` может быть полезен в тех случаях когда необходимо совершить обход дерева кратчайших путей.

Метод `shortestTree` создает новый объект (всегда `QgsGraph`) и принимает три аргумента:

- `source` — исходный граф
- `startVertexIdx` — индекс точки на графе (корень дерева)
- `criterionNum` — порядковый номер свойства ребра (отсчет ведется от 0).

```
tree = QgsGraphAnalyzer.shortestTree( graph, startId, 0 )
```

Метод `dijkstra` имеет аналогичные параметры, но возвращает не граф, а кортеж из двух массивов. В первом массиве *i*-ый элемент содержит индекс дуги, входящей в *i*-ю вершину, в противном случае — -1. Во втором массиве *i*-ый элемент содержит расстояние от корня дерева до *i*-ой вершины, если вершина достижима из корня или максимально большое число которое может хранить тип C++ `double` (эквивалент плюс бесконечности), если вершина не достижима.

```
(tree, cost) = QgsGraphAnalyzer.dijkstra( graph, startId, 0 )
```

Вот так выглядит простейший способ отобразить дерево кратчайших путей с использованием графа, полученного в результате вызова метода `shortestTree` (только замените координаты начальной точки на свои, а также выделите слой дорог в списке слоёв карты). **ОСТОРОЖНО:** код создает огромное количество объектов [QgsRubberBand](#), используйте его только для очень маленьких слоёв.

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector( vl, -1, '', '', '', 3 )
properter = QgsDistanceArcProperter()
director.addProperter( properter )
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder( crs )

pStart = QgsPoint( -0.743804, 0.22954 )
tiedPoint = director.makeGraph( builder, [ pStart ] )
pStart = tiedPoint[ 0 ]

graph = builder.graph()

idStart = graph.findVertex( pStart )

tree = QgsGraphAnalyzer.shortestTree( graph, idStart, 0 )

i = 0;
while ( i < tree.arcCount() ):
    rb = QgsRubberBand( qgis.utils.iface.mapCanvas() )
    rb.setColor ( Qt.red )
    rb.addPoint ( tree.vertex( tree.arc( i ).inVertex() ).point() )
    rb.addPoint ( tree.vertex( tree.arc( i ).outVertex() ).point() )
    i = i + 1
```

То же самое, но с использованием метода `dijkstra`:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector( vl, -1, '', '', '', 3 )
properter = QgsDistanceArcProperter()
director.addProperter( properter )
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder( crs )

pStart = QgsPoint( -1.37144, 0.543836 )
tiedPoint = director.makeGraph( builder, [ pStart ] )
pStart = tiedPoint[ 0 ]

graph = builder.graph()

idStart = graph.findVertex( pStart )

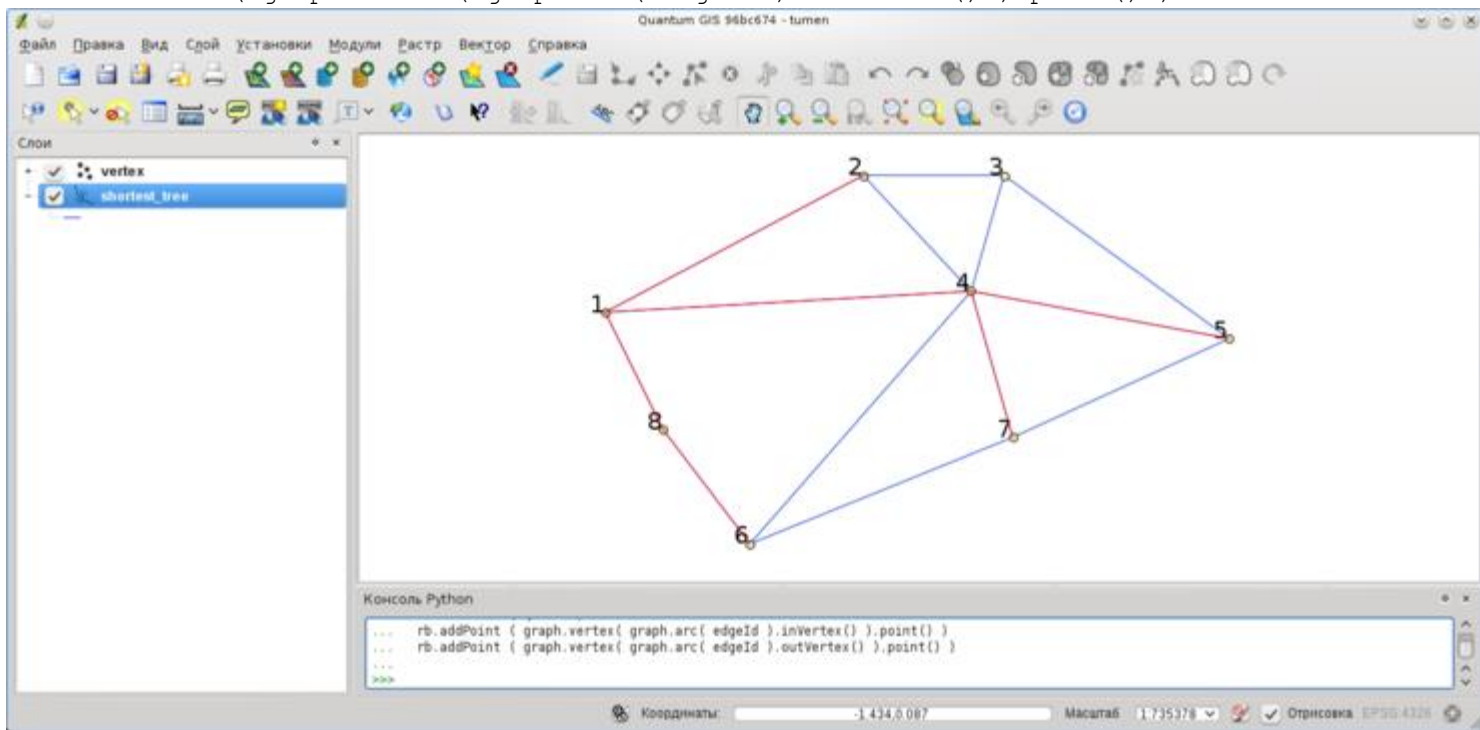
( tree, costs ) = QgsGraphAnalyzer.dijkstra( graph, idStart, 0 )

for edgeId in tree:
```

```

if edgeId == -1:
    continue
rb = QgsRubberBand( qgis.utils.iface.mapCanvas() )
rb.setColor ( Qt.red )
rb.addPoint ( graph.vertex( graph.arc( edgeId ).inVertex() ).point() )
rb.addPoint ( graph.vertex( graph.arc( edgeId ).outVertex() ).point() )

```



Результат выполнения скрипта — дерево кратчайших путей с корнем в вершине №1.

Нахождение кратчайших путей

Для получения оптимального маршрута между двумя произвольными точками используется следующий подход. Обе точки (начальная A и конечная B) «привязываются» к графу на этапе построения, затем при помощи метода `shortestTree` или `dijkstra` находится дерево кратчайших маршрутов с корнем в начальной точке A. В этом же дереве находим конечную точку B и начинаем спуск по дереву от точки B к точке A. В общем виде алгоритм можно записать так:

1. присвоим $T = B$
2. пока $T \neq A$ цикл
 1. добавляем в маршрут точку T
 2. берем ребро, которое входит в точку T
 3. находим точку TT, из которой это ребро выходит
 4. присваиваем $T = TT$
3. добавляем в маршрут точку A

На этом построение маршрута закончено. Мы получили инвертированный список вершин (т.е. вершины идут в обратном порядке, от конечной точки к начальной), которые будут посещены при движении по кратчайшему маршруту.

Посмотрите еще раз на [дерево кратчайших путей](#) и представьте, что вы можете двигаться только против направления стрелочек. При движении из точки №7 мы рано или поздно попадем в точку №1 (корень дерева) и не сможем двигаться дальше.

Вот работающий пример поиска кратчайшего маршрута для Консоли Python QGIS (только замените координаты начальной и конечной точки на свои, а также выделите слой дорог в списке слоёв карты) с использованием метода `shortestTree`

```

from PyQt4.QtCore import *
from PyQt4.QtGui import *

```



```

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector( vl, -1, '', '', '', 3 )
properter = QgsDistanceArcProperter()
director.addProperter( properter )
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder( crs )

pStart = QgsPoint( -0.835953, 0.15679 )
pStop = QgsPoint( -1.1027, 0.699986 )

tiedPoints = director.makeGraph( builder, [ pStart, pStop ] )
graph = builder.graph()

tStart = tiedPoints[ 0 ]
tStop = tiedPoints[ 1 ]

idStart = graph.findVertex( tStart )
tree = QgsGraphAnalyzer.shortestTree( graph, idStart, 0 )

idStart = tree.findVertex( tStart )
idStop = tree.findVertex( tStop )

if idStop == -1:
    print "Path not found"
else:
    p = []
    while ( idStart != idStop ):
        l = tree.vertex( idStop ).inArc()
        if len( l ) == 0:
            break
        e = tree.arc( l[ 0 ] )
        p.insert( 0, tree.vertex( e.inVertex() ).point() )
        idStop = e.outVertex()

    p.insert( 0, tStart )
    rb = QgsRubberBand( qgis.utils.iface.mapCanvas() )
    rb.setColor( Qt.red )

    for pnt in p:
        rb.addPoint(pnt)

```

А вот пример с использованием метода dikstra

```

from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector( vl, -1, '', '', '', 3 )
properter = QgsDistanceArcProperter()
director.addProperter( properter )
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder( crs )

pStart = QgsPoint( -0.835953, 0.15679 )
pStop = QgsPoint( -1.1027, 0.699986 )

tiedPoints = director.makeGraph( builder, [ pStart, pStop ] )

```



```

graph = builder.graph()

tStart = tiedPoints[ 0 ]
tStop = tiedPoints[ 1 ]

idStart = graph.findVertex( tStart )
idStop = graph.findVertex( tStop )

( tree, cost ) = QgsGraphAnalyzer.dijkstra( graph, idStart, 0 )

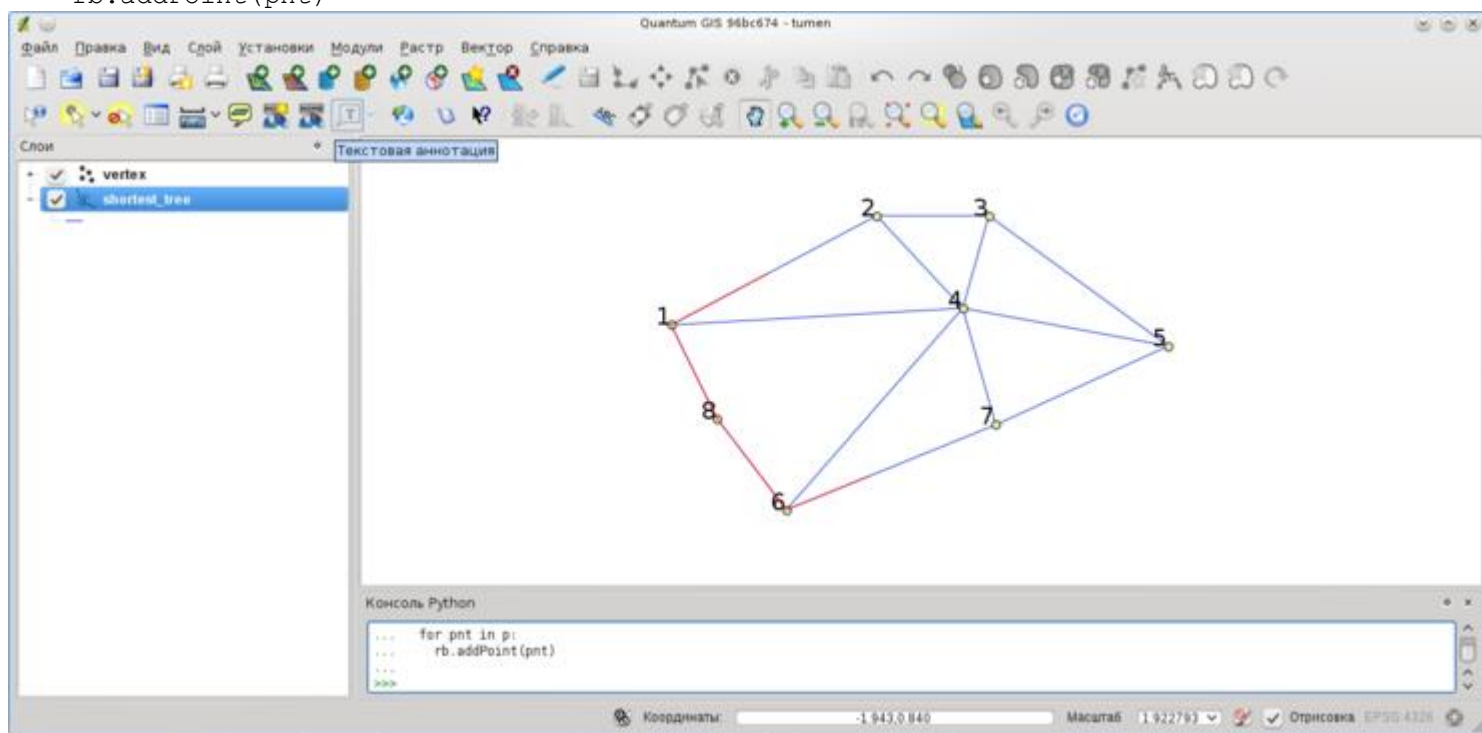
if tree[ idStop ] == -1:
    print "Path not found"
else:
    p = []
    curPos = idStop
    while curPos != idStart:
        p.append( graph.vertex( graph.arc( tree[ curPos ] ).inVertex() ).point() )
        curPos = graph.arc( tree[ curPos ] ).outVertex();

    p.append( tStart )

    rb = QgsRubberBand( qgis.utils.iface.mapCanvas() )
    rb.setColor( Qt.red )

    for pnt in p:
        rb.addPoint( pnt)

```



Результат выполнения скрипта — кратчайший путь

Нахождение областей доступности

Назовем областью доступности вершины графа A такое подмножество вершин графа, доступных из вершины A , что стоимость оптимального пути от A до элементов этого множества не превосходит некоторого заданного значения.

Более наглядно это определение можно объяснить на следующем примере: «Есть пожарное депо. В какую часть города сможет попасть пожарная машина в за 5 минут, 10 минут, 15 минут?». Ответом на этот вопрос и являются области доступности пожарного депо.

Поиск областей доступности легко реализовать при помощи метода `dijkstra` класса `QgsGraphAnalyzer`. Достаточно сравнить элементы возвращаемого значения с заданным параметром. Если величина `cost` i

меньше заданного параметра или равна ему, тогда i-я вершина графа принадлежит множеству доступности, в противном случае — не принадлежит.

Не столь очевидным является нахождение границ доступности. Нижняя граница доступности — множество вершин которые **еще** можно достигнуть, а верхняя граница — множество вершин которых **уже** нельзя достигнуть. На самом деле все просто: граница доступности проходит по таким ребрам дерева кратчайших путей, для которых вершина-источник ребра доступна, а вершина-цель недоступна.

Вот пример

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector( vl, -1, '', '', '', 3 )
properter = QgsDistanceArcProperter()
director.addProperter( properter )
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder( crs )

pStart = QgsPoint( 65.5462, 57.1509 )
delta = qgis.utils.iface.mapCanvas().getCoordinateTransform().mapUnitsPerPixel() * 1

rb = QgsRubberBand( qgis.utils.iface.mapCanvas(), True )
rb.setColor( Qt.green )
rb.addPoint( QgsPoint( pStart.x() - delta, pStart.y() - delta ) )
rb.addPoint( QgsPoint( pStart.x() + delta, pStart.y() - delta ) )
rb.addPoint( QgsPoint( pStart.x() + delta, pStart.y() + delta ) )
rb.addPoint( QgsPoint( pStart.x() - delta, pStart.y() + delta ) )

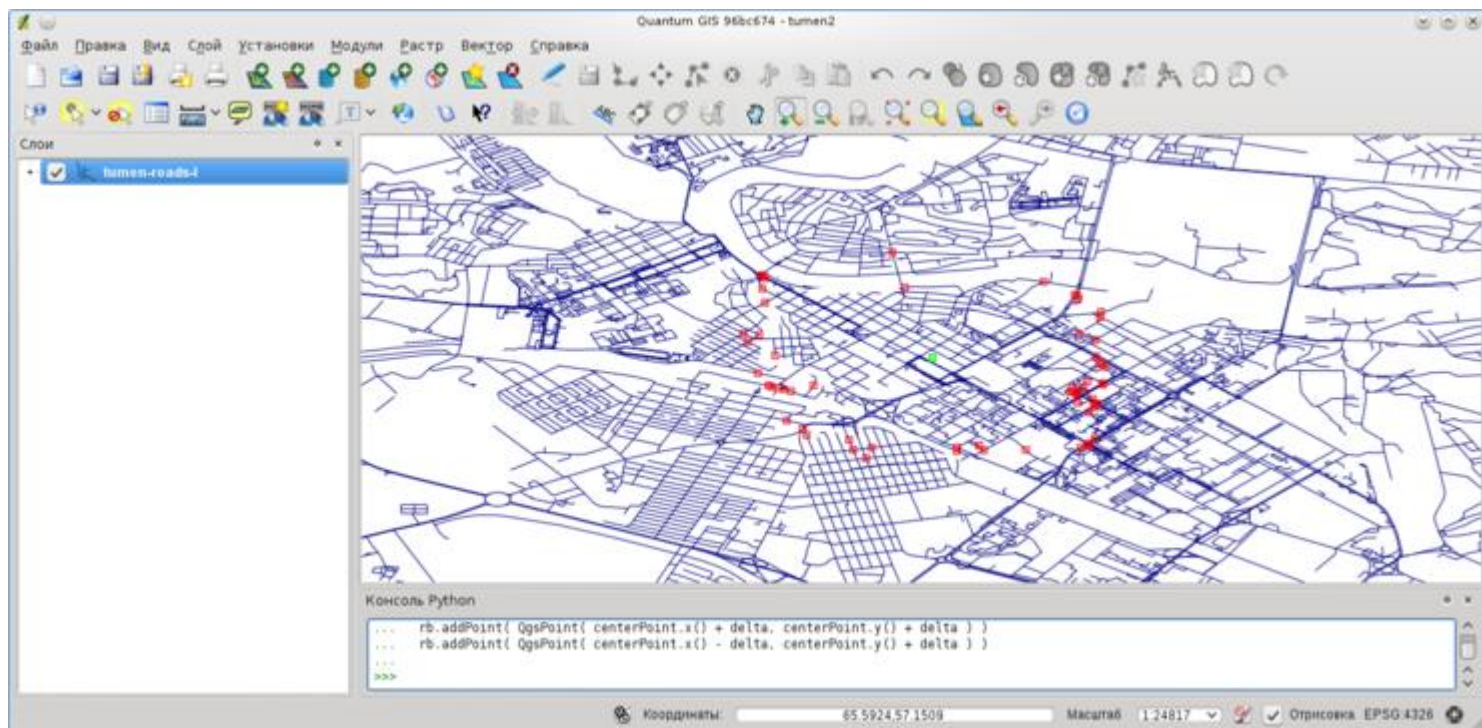
tiedPoints = director.makeGraph( builder, [ pStart ] )
graph = builder.graph()
tStart = tiedPoints[ 0 ]

idStart = graph.findVertex( tStart )

( tree, cost ) = QgsGraphAnalyzer.dijkstra( graph, idStart, 0 )

upperBound = []
r = 2000.0
i = 0
while i < len(cost):
    if cost[ i ] > r and tree[ i ] != -1:
        outVertexId = graph.arc( tree [ i ] ).outVertex()
        if cost[ outVertexId ] < r:
            upperBound.append( i )
        i = i + 1

for i in upperBound:
    centerPoint = graph.vertex( i ).point()
    rb = QgsRubberBand( qgis.utils.iface.mapCanvas(), True )
    rb.setColor( Qt.red )
    rb.addPoint( QgsPoint( centerPoint.x() - delta, centerPoint.y() - delta ) )
    rb.addPoint( QgsPoint( centerPoint.x() + delta, centerPoint.y() - delta ) )
    rb.addPoint( QgsPoint( centerPoint.x() + delta, centerPoint.y() + delta ) )
    rb.addPoint( QgsPoint( centerPoint.x() - delta, centerPoint.y() + delta ) )
```



Результат выполнения скрипта. Зеленый квадратик — центр области, красные квадратiki — верхняя граница доступности, вершины **уже** не входящие в область.

Актуальная документация

Актуальную документацию всегда можно получить в разделе [QGIS network analysis library](#) описания [QGIS API](#).

[Обсудить в форуме](#) Комментариев — 4

Последнее обновление: 2014-05-14 23:48

Дата создания: 05.01.2012

Автор(ы): [Сергей Якушев \(stopa85\)](#)