<http://stackoverflow.com/questions/7329427/help-with-java-graph-implementation>

[**Help with Java Graph Implementation**](http://stackoverflow.com/questions/7329427/help-with-java-graph-implementation)

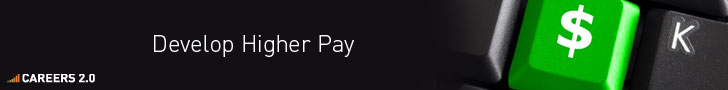
[](http://engine.adzerk.net/redirect/0/5923/7472/8277/8734bafa51824472b7dcc9b66d46f235/43/1178/10188/634950737619088910/?keywords=java,homework,graph)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| up vote1down vote[favorite](http://stackoverflow.com/questions/7329427/help-with-java-graph-implementation) | I have to write up a program that stores verticies into a graph and connect them. The input for the program is given as:  Plot 3 5 to 5 8  Plot 6 1 to 3 5  etc  From this input, I would store a vertix of (3,5) which are the x and y coordinates. I would then have to connect this coordinate to (5,8).  My question is, how would I go in implementing this graph? I was thinking I would need to store the vertixes in an arraylist or a map, and keep the edges in a list too...but since I'm not given an actual max size of the graph, I'm a little bit lost in the overall implementation. Basically, if you guys could give me an idea how to start this, it'd be sweet as.  [java](http://stackoverflow.com/questions/tagged/java) [homework](http://stackoverflow.com/questions/tagged/homework) [graph](http://stackoverflow.com/questions/tagged/graph)   |  |  |  | | --- | --- | --- | | [share](http://stackoverflow.com/q/7329427)|[improve this question](http://stackoverflow.com/posts/7329427/edit) | edited [Sep 7 '11 at 6:10](http://stackoverflow.com/posts/7329427/revisions) | asked Sep 7 '11 at 5:53  [[http://www.gravatar.com/avatar/167a036376cdefb06ff5542e194af3ac?s=32&d=identicon&r=PG](http://stackoverflow.com/users/932082/wnnnnn)](http://stackoverflow.com/users/932082/wnnnnn)  [wnnnnn](http://stackoverflow.com/users/932082/wnnnnn) **6**2 | |
|  | |  |  | | --- | --- | |  | If this is for homework, you should add the [homework](http://stackoverflow.com/questions/tagged/homework) tag – [corsiKa](http://stackoverflow.com/users/330057/corsika) [Sep 7 '11 at 6:08](http://stackoverflow.com/questions/7329427/help-with-java-graph-implementation#comment8838108_7329427) | |  | @glowcoder, yeah np – [wnnnnn](http://stackoverflow.com/users/932082/wnnnnn) [Sep 7 '11 at 6:10](http://stackoverflow.com/questions/7329427/help-with-java-graph-implementation#comment8838132_7329427) |   Was this post useful to you? |

**3 Answers**

[active](http://stackoverflow.com/questions/7329427/help-with-java-graph-implementation?answertab=active#tab-top)[oldest](http://stackoverflow.com/questions/7329427/help-with-java-graph-implementation?answertab=oldest#tab-top)[votes](http://stackoverflow.com/questions/7329427/help-with-java-graph-implementation?answertab=votes#tab-top)

|  |  |  |  |
| --- | --- | --- | --- |
| up vote1down vote | An easy way to think about graphs is to break them down into their Node and Edge components. (Note: what you call a vertex I call a node. Close enough.)  Let us consider the following options:  // Option 1  class Node<V> {  V data;  Set<Node<V>> edges;  // if it were a directed graph, you'd need:  // Set<Node<V>> edgesAway;  }  // Option 2  class Node<V> {  V data;  Set<Edge<V>> edges;  // if it were a directed graph, you'd need:  // Set<Edge<V>> edgesAway;  }  class Edge<V> {  Node<V> source;  Node<V> destination;  }  Now a graph is nothing more than:  // option 1  class Graph<V> {  Set<Node<V>> nodes;  }  // option 2  class Graph<V> {  Set<Node<V>> nodes;  Set<Edge<V>> edges;  }  Option 1 is the simplest and easiest to implement. Option 2 gives you some more flexibility, such as adding weights to the edge values.  Now, you have some data at these nodes, correct? For now, let's just have the data be the string representation of the coordinates.  class SomeObject {  String data;  int x;  int y;  public boolean equals(Object o) {  if(o instanceof SomeObject) {  SomeObject so = (SomeObject)o;  return so.x == x && so.y == y;  }  return false;  }  public int hashCode() {  return x \* 100 + y; // it works... close enough :)  }  }  // somewhere later:  Graph<SomeObject> graph = ...  Now as for what functionality you'll want you'd need a more complete list. But this should get you well on your way to understanding how to implement a graph.   |  |  | | --- | --- | | [share](http://stackoverflow.com/a/7329528)|[improve this answer](http://stackoverflow.com/posts/7329528/edit) | answered Sep 7 '11 at 6:07  [[http://www.gravatar.com/avatar/eba3df62ea97ea9065ca70e9c06b80aa?s=32&d=identicon&r=PG](http://stackoverflow.com/users/330057/corsika)](http://stackoverflow.com/users/330057/corsika)  [corsiKa](http://stackoverflow.com/users/330057/corsika) **30.4k**43278 | |
|  | feedback |

[](http://engine.adzerk.net/redirect/0/10769/2444/8277/1eefa449e7b446579373f5b42736aee2/44/1178/16886/634950737619191310/?keywords=java,homework,graph)

|  |  |  |  |
| --- | --- | --- | --- |
| up vote0down vote | You can use a already made library like [JDSL](http://www.cs.brown.edu/cgc/jdsl/doc/index.html) for that   |  |  | | --- | --- | | [share](http://stackoverflow.com/a/7329512)|[improve this answer](http://stackoverflow.com/posts/7329512/edit) | answered Sep 7 '11 at 6:05  [[http://www.gravatar.com/avatar/7fe217f8b21988d8724de92c80f73a84?s=32&d=identicon&r=PG](http://stackoverflow.com/users/180100/rc)](http://stackoverflow.com/users/180100/rc)  [RC.](http://stackoverflow.com/users/180100/rc) **11.8k**1434 | |
|  | |  |  | | --- | --- | |  | Typically this is for a school assignment to help teach the fundamentals of algorithms and data structures... – [corsiKa](http://stackoverflow.com/users/330057/corsika) [Sep 7 '11 at 6:08](http://stackoverflow.com/questions/7329427/help-with-java-graph-implementation#comment8838115_7329512) | |  | yup your answer is more helpful in that case (+1). OP can also take a look at JDSL sources and doc. – [RC.](http://stackoverflow.com/users/180100/rc)[Sep 7 '11 at 6:09](http://stackoverflow.com/questions/7329427/help-with-java-graph-implementation#comment8838122_7329512) |   feedback |
| up vote0down vote | One way is to implement Node/Edge classes like somebody already mentioned.  Another way is to implement a so-called adjacency matrix, which is basically an n x n matrix of booleans (n = # of nodes) where a 1 at (i, j) indicates the presence of an edge from i to j and a 0 indicates an absence.  The best approach in any given situation really depends in part on what you want to do with the graph, in part on the assumptions you can make about the graph itself (e.g., sparse vs. dense), etc. The adjacency matrix, for example, supports directedness, but it wastes space if your edges aren't directed, because (i, j) = (j, i). You can of course modify the approach to improve space performance for undirected edges but I'm just highlighting the fact that the representation is better or worse depending on context.   |  |  | | --- | --- | | [share](http://stackoverflow.com/a/7329762)|[improve this answer](http://stackoverflow.com/posts/7329762/edit) | answered Sep 7 '11 at 6:33  [[http://www.gravatar.com/avatar/b0a72ae61c5c74a51fe46cf66599b6c5?s=32&d=identicon&r=PG](http://stackoverflow.com/users/41871/willie-wheeler)](http://stackoverflow.com/users/41871/willie-wheeler)  [Willie Wheeler](http://stackoverflow.com/users/41871/willie-wheeler) **4,963**1536 | |
|  | |  |  | | --- | --- | |  | since my max size is not defined, doesn't the adjacency matrix get rendered useless as I cannot define the size of the array. My plan is to use this graph to build an A\*star type algorithm – [wnnnnn](http://stackoverflow.com/users/932082/wnnnnn) [Sep 7 '11 at 6:56](http://stackoverflow.com/questions/7329427/help-with-java-graph-implementation#comment8838602_7329762) | |  | Oh, sorry, missed that part of your question. :-) Yes, I don't think an adjacency matrix would be good for A\* for exactly the reason you state... – [Willie Wheeler](http://stackoverflow.com/users/41871/willie-wheeler) [Sep 7 '11 at 7:36](http://stackoverflow.com/questions/7329427/help-with-java-graph-implementation#comment8839071_7329762) | |

<http://stackoverflow.com/questions/10255479/graphs-implementation-in-java>

[**Graphs implementation in java**](http://stackoverflow.com/questions/10255479/graphs-implementation-in-java)

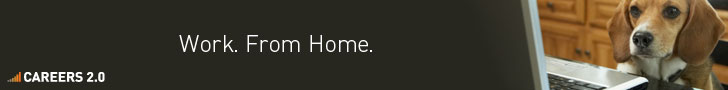
[](http://engine.adzerk.net/redirect/0/5094/2444/8277/999354f877de4146a9a9c1262f64b2c2/43/1178/10139/634950737629717240/?keywords=java,graph,implementation)

|  |  |  |  |
| --- | --- | --- | --- |
| up vote2down vote[favorite](http://stackoverflow.com/questions/10255479/graphs-implementation-in-java) | I am trying to create a Graph class that uses another class, the Vertex class to represent all the vertices of the graph. I am not sure if I need an Edge class that will represent the possible connections between two vertices, because every vertex can keep track of the other nodes it is connected to. But I am not sure if this is correct. What do you think?  Thank you.  [java](http://stackoverflow.com/questions/tagged/java) [graph](http://stackoverflow.com/questions/tagged/graph) [implementation](http://stackoverflow.com/questions/tagged/implementation)   |  |  | | --- | --- | | [share](http://stackoverflow.com/q/10255479)|[improve this question](http://stackoverflow.com/posts/10255479/edit) | asked Apr 21 '12 at 1:42  [[http://www.gravatar.com/avatar/6aa478a20133fae6f279cb7dfaf09815?s=32&d=identicon&r=PG](http://stackoverflow.com/users/1181847/franxh)](http://stackoverflow.com/users/1181847/franxh)  [FranXh](http://stackoverflow.com/users/1181847/franxh) **446**19 | |
|  | |  |  | | --- | --- | |  | Show us some code :) – [paulsm4](http://stackoverflow.com/users/421195/paulsm4) [Apr 21 '12 at 1:49](http://stackoverflow.com/questions/10255479/graphs-implementation-in-java#comment13184117_10255479) | |  | Q: Is this "homework"? If so, please flag it as such. In any case, take a look at this:[en.literateprograms.org/Dijkstra%27s\_algorithm\_%28Java%29](http://en.literateprograms.org/Dijkstra%27s_algorithm_%28Java%29) – [paulsm4](http://stackoverflow.com/users/421195/paulsm4) [Apr 21 '12 at 1:51](http://stackoverflow.com/questions/10255479/graphs-implementation-in-java#comment13184128_10255479) | |  | It is not homework. I know how to implement the class with and without using the edge class. I just want to know if it is correct without the Edge class. Anw thanks for the link. – [FranXh](http://stackoverflow.com/users/1181847/franxh) [Apr 21 '12 at 2:03](http://stackoverflow.com/questions/10255479/graphs-implementation-in-java#comment13184220_10255479) |   feedback |

**2 Answers**

[active](http://stackoverflow.com/questions/10255479/graphs-implementation-in-java?answertab=active#tab-top)[oldest](http://stackoverflow.com/questions/10255479/graphs-implementation-in-java?answertab=oldest#tab-top)[voes](http://stackoverflow.com/questions/10255479/graphs-implementation-in-java?answertab=votes#tab-top)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| epted | You don't have to use an Edge class. You can use adjacency lists and still represent an **unweighted** graph correctly. For a weighted graph you need a way to represent edge costs, and thus using anEdge class would be appropriate.  class Graph<E> {  private List<Vertex<E>> vertices;  private static class Vertex<E> {  E elem;  List<Vertex<E>> neighbors;  }  }   |  |  |  | | --- | --- | --- | | [share](http://stackoverflow.com/a/10255593)|[improve this answer](http://stackoverflow.com/posts/10255593/edit) | edited [Apr 21 '12 at 2:20](http://stackoverflow.com/posts/10255593/revisions) | answered Apr 21 '12 at 2:10  [[http://www.gravatar.com/avatar/a983ccf67fa7942199386a01feeb43f7?s=32&d=identicon&r=PG](http://stackoverflow.com/users/626318/blackcompe)](http://stackoverflow.com/users/626318/blackcompe)  [blackcompe](http://stackoverflow.com/users/626318/blackcompe) **1,737**28 | |
|  | |  |  | | --- | --- | |  | Yes this is what I was thinking. To keep track of the other vertices without using an Edge class at all. Thank you!!!! – [FranXh](http://stackoverflow.com/users/1181847/franxh) [Apr 21 '12 at 2:13](http://stackoverflow.com/questions/10255479/graphs-implementation-in-java#comment13184283_10255593) | | |  |  | | --- | --- | | **1** |  | | All well and good until you need a weighted directed graph. Then you'll need metadata or better yet an Edge class. – [Andrew Finnell](http://stackoverflow.com/users/553308/andrew-finnell) [Apr 21 '12 at 2:16](http://stackoverflow.com/questions/10255479/graphs-implementation-in-java#comment13184309_10255593) | |  | @AndrewFinnell: Thanks. Updated. – [blackcompe](http://stackoverflow.com/users/626318/blackcompe) [Apr 21 '12 at 2:21](http://stackoverflow.com/questions/10255479/graphs-implementation-in-java#comment13184335_10255593) | |  | +1 Short and to the point. The best kind of code. – [Andrew Finnell](http://stackoverflow.com/users/553308/andrew-finnell) [Apr 21 '12 at 2:22](http://stackoverflow.com/questions/10255479/graphs-implementation-in-java#comment13184343_10255593) |   feedback |

[](http://engine.adzerk.net/redirect/0/10768/7472/8277/9e75b11d9ab4411eb93b0a6151185b63/44/1178/16874/634950737629837180/?keywords=java,graph,implementation)

|  |  |  |  |
| --- | --- | --- | --- |
| up vote2down vote | Typically, a [*representation*](http://en.wikipedia.org/wiki/Graph_%28abstract_data_type%29#Representations) is chosen based in its suitability to the intended use. In this simple example,[GraphPanel](https://sites.google.com/site/drjohnbmatthews/graphpanel) uses nothing more than a List<Edge> as its model.   |  |  | | --- | --- | | [share](http://stackoverflow.com/a/10255573)|[improve this answer](http://stackoverflow.com/posts/10255573/edit) | edited [Apr 21 '12 at 2:17](http://stackoverflow.com/posts/10255573/revisions) | |

<http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html>

## Dijkstra's shortest path algorithm in Java

### Lars Vogel

Version 1.1

Copyright © 2009, 2010, 2011, 2011 Lars Vogel

02.11.2009

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision History** | | | |
| Revision 0.1 | 30.10.2009 | Lars Vogel | Created |
| Revision 0.2 - 1.1 | 01.11.2009 | Lars Vogel | updates and bugfixes |

**Dijkstra's Shortest Path Algorithm in Java**

Dijkstra's Algorithms describes how to find the shortest path from one node to another node in a directed weighted graph. This article presents a Java implementation of this algorithm.

**Table of Contents**

[**1. The shortest path problem**](http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html#dijkstra)

[**1.1. Shortest path**](http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html#shortestpath_problem)

[**1.2. Graph**](http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html#shortestpath_graph)

[**1.3. Typical graph problems**](http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html#shortestpath_graphproblems)

[**2. Dijkstra's algorithm**](http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html#dijkstra)

[**2.1. High level description**](http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html#dijkstra_overview)

[**2.2. Algorithms Description**](http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html#dijkstra_algorithms)

[**3. Model**](http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html#model)

[**4. Algorithmus**](http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html#algorithmus)

[**4.1. Implementation**](http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html#algorithmus_implementation)

[**4.2. Test**](http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html#algorithmus_test)

[**5. Thank you**](http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html#thankyou)

[**6. Questions and Discussion**](http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html#questions)

[**7. Links and Literature**](http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html#resources)

[**7.1. Links**](http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html#links)

## 1. The shortest path problem

### 1.1. Shortest path

Finding the shortest path in a network is a commonly encountered problem. For example you want to reach a target in the real world via the shortest path or in a computer network a network package should be efficiently routed through the network.

This tutorial describes the problem modeled as graph and the Dijkstra algorithm to solve the problem.

### 1.2. Graph

A graph is made of out nodes and directed edges which defines a connection from one node to another node.

A node (or vertex) is a discrete position in a graph. Edges can be directed an undirected. Edges have an associated distance (also called costs or weight). The distance between two nodes a and b is labeled as [a,b].

The mathematical description for graphs is G= {V,E}, meaning that a graph is defined by a set of vertexes (V) and a collection of edges.

The order of a graph is the number of nodes. The size of a graph is the number of edges.

### 1.3. Typical graph problems

Typical graph problems are described in the following list.

* finding the shortest path from a specific node to another node
* finding the maximum possible flow through a network where each edges has a pre-defined maximum capacity (maximum-flow minimum-cut problem)

The following will focus on finding the shortest path from one node to another node in a graph.

## 2. Dijkstra's algorithm

### 2.1. High level description

The Dijkstra Algorithm finds the shortest path from a source to all destination in a directed graph (single source shortest path problem). During this process it will also determine a spanning tree for the graph.

### 2.2. Algorithms Description

The idea of Dijkstra is simple.

Dijkstra partitions all nodes into two distinct sets. Unsettled and settled. Initially all nodes are in the unsettled sets, e.g. they must be still evaluated. A node is moved to the settled set if a shortest path from the source to this node has been found.

Initially the distance of each node to the source is set to a very high value.

First only the source is in the set of unsettledNodes.

The algorithms runs until the unsettledNodes are empty. In earch iteration it selects the node with the lowest distance from the source out the unsettled nodes. If reads all edges which are outgoing from the source and evaluates for each destination node in these edges which is not yet settled if the known distance from the source to this node can be reduced if the selected edge is used. If this can be done then the distance is updated and the node is added to the nodes which need evaluation.

In pseudocode the algorithm can be described as follows. Please note that Dijkstra also determines the pre-successor of each node on its way to the source. I leave that out of the pseudo code to simplify it.

Foreach node set distance[node] = HIGH

SettledNodes = empty

UnSettledNodes = empty

Add sourceNode to UnSettledNodes

distance[sourceNode]= 0

**while** (UnSettledNodes is not empty) {

evaluationNode = getNodeWithLowestDistance(UnSettledNodes)

remove evaluationNode from UnSettledNodes

add evaluationNode to SettledNodes

evaluatedNeighbors(evaluationNode)

}

getNodeWithLowestDistance(UnSettledNodes){

find the node with the lowest distance in UnSettledNodes and **return** it

}

evaluatedNeighbors(evaluationNode){

Foreach destinationNode which can be reached via an edge from evaluationNode AND which is not in SettledNodes {

edgeDistance = getDistance(edge(evaluationNode, destinationNode))

newDistance = distance[evaluationNode] + edgeDistance

**if** (distance[destinationNode] > newDistance) {

distance[destinationNode] = newDistance

add destinationNode to UnSettledNodes

}

}

}

## 3. Model

A graph consists of vertices and edges. These are represented by the following model.

**package** de.vogella.algorithms.dijkstra.model;

**public** **class** Vertex {

**final** **private** String id;

**final** **private** String name;

**public** Vertex(String id, String name) {

**this**.id = id;

**this**.name = name;

}

**public** String getId() {

**return** id;

}

**public** String getName() {

**return** name;

}

*@Override*

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + ((id == null) ? 0 : id.hashCode());

**return** result;

}

*@Override*

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** true;

**if** (obj == null)

**return** false;

**if** (getClass() != obj.getClass())

**return** false;

Vertex other = (Vertex) obj;

**if** (id == null) {

**if** (other.id != null)

**return** false;

} **else** **if** (!id.equals(other.id))

**return** false;

**return** true;

}

*@Override*

**public** String toString() {

**return** name;

}

}

A edge has a source and a destination.

**package** de.vogella.algorithms.dijkstra.model;

**public** **class** Edge {

**private** **final** String id;

**private** **final** Vertex source;

**private** **final** Vertex destination;

**private** **final** **int** weight;

**public** Edge(String id, Vertex source, Vertex destination, **int** weight) {

**this**.id = id;

**this**.source = source;

**this**.destination = destination;

**this**.weight = weight;

}

**public** String getId() {

**return** id;

}

**public** Vertex getDestination() {

**return** destination;

}

**public** Vertex getSource() {

**return** source;

}

**public** **int** getWeight() {

**return** weight;

}

*@Override*

**public** String toString() {

**return** source + " " + destination;

}

}

Both represent a graph.

**package** de.vogella.algorithms.dijkstra.model;

**import** java.util.List;

**public** **class** Graph {

**private** **final** List<Vertex> vertexes;

**private** **final** List<Edge> edges;

**public** Graph(List<Vertex> vertexes, List<Edge> edges) {

**this**.vertexes = vertexes;

**this**.edges = edges;

}

**public** List<Vertex> getVertexes() {

**return** vertexes;

}

**public** List<Edge> getEdges() {

**return** edges;

}

}

## 4. Algorithmus

### 4.1. Implementation

The following is a simple implementation of Dijkstra's algorithm. It does not use any performance optimization (e.g. by using a PriorityQueue for the UnSettledNodes of does not cache the result of the target evaluation of the edges) to make the algorihms as simple as possible.

**package** de.vogella.algorithms.dijkstra.engine;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.HashMap;

**import** java.util.HashSet;

**import** java.util.LinkedList;

**import** java.util.List;

**import** java.util.Map;

**import** java.util.Set;

**import** de.vogella.algorithms.dijkstra.model.Edge;

**import** de.vogella.algorithms.dijkstra.model.Graph;

**import** de.vogella.algorithms.dijkstra.model.Vertex;

**public** **class** DijkstraAlgorithm {

**private** **final** List<Vertex> nodes;

**private** **final** List<Edge> edges;

**private** Set<Vertex> settledNodes;

**private** Set<Vertex> unSettledNodes;

**private** Map<Vertex, Vertex> predecessors;

**private** Map<Vertex, Integer> distance;

**public** DijkstraAlgorithm(Graph graph) {

// Create a copy of the array so that we can operate on this array

**this**.nodes = **new** ArrayList<Vertex>(graph.getVertexes());

**this**.edges = **new** ArrayList<Edge>(graph.getEdges());

}

**public** **void** execute(Vertex source) {

settledNodes = **new** HashSet<Vertex>();

unSettledNodes = **new** HashSet<Vertex>();

distance = **new** HashMap<Vertex, Integer>();

predecessors = **new** HashMap<Vertex, Vertex>();

distance.put(source, 0);

unSettledNodes.add(source);

**while** (unSettledNodes.size() > 0) {

Vertex node = getMinimum(unSettledNodes);

settledNodes.add(node);

unSettledNodes.remove(node);

findMinimalDistances(node);

}

}

**private** **void** findMinimalDistances(Vertex node) {

List<Vertex> adjacentNodes = getNeighbors(node);

**for** (Vertex target : adjacentNodes) {

**if** (getShortestDistance(target) > getShortestDistance(node)

+ getDistance(node, target)) {

distance.put(target, getShortestDistance(node)

+ getDistance(node, target));

predecessors.put(target, node);

unSettledNodes.add(target);

}

}

}

**private** **int** getDistance(Vertex node, Vertex target) {

**for** (Edge edge : edges) {

**if** (edge.getSource().equals(node)

&& edge.getDestination().equals(target)) {

**return** edge.getWeight();

}

}

**throw** **new** RuntimeException("Should not happen");

}

**private** List<Vertex> getNeighbors(Vertex node) {

List<Vertex> neighbors = **new** ArrayList<Vertex>();

**for** (Edge edge : edges) {

**if** (edge.getSource().equals(node)

&& !isSettled(edge.getDestination())) {

neighbors.add(edge.getDestination());

}

}

**return** neighbors;

}

**private** Vertex getMinimum(Set<Vertex> vertexes) {

Vertex minimum = null;

**for** (Vertex vertex : vertexes) {

**if** (minimum == null) {

minimum = vertex;

} **else** {

**if** (getShortestDistance(vertex) < getShortestDistance(minimum)) {

minimum = vertex;

}

}

}

**return** minimum;

}

**private** **boolean** isSettled(Vertex vertex) {

**return** settledNodes.contains(vertex);

}

**private** **int** getShortestDistance(Vertex destination) {

Integer d = distance.get(destination);

**if** (d == null) {

**return** Integer.MAX\_VALUE;

} **else** {

**return** d;

}

}

/\*

\* This method returns the path from the source to the selected target and

\* NULL if no path exists

\*/

**public** LinkedList<Vertex> getPath(Vertex target) {

LinkedList<Vertex> path = **new** LinkedList<Vertex>();

Vertex step = target;

// Check if a path exists

**if** (predecessors.get(step) == null) {

**return** null;

}

path.add(step);

**while** (predecessors.get(step) != null) {

step = predecessors.get(step);

path.add(step);

}

// Put it into the correct order

Collections.reverse(path);

**return** path;

}

}

### 4.2. Test

The following is a small JUnit Test to validate the correctness of the algorithm.

**package** de.vogella.algorithms.dijkstra.test;

**import** java.util.ArrayList;

**import** java.util.LinkedList;

**import** java.util.List;

**import** org.junit.Test;

**import** de.vogella.algorithms.dijkstra.engine.DijkstraAlgorithm;

**import** de.vogella.algorithms.dijkstra.model.Edge;

**import** de.vogella.algorithms.dijkstra.model.Graph;

**import** de.vogella.algorithms.dijkstra.model.Vertex;

**import** **static** org.junit.Assert.assertNotNull;

**import** **static** org.junit.Assert.assertTrue;

**public** **class** TestDijkstraAlgorithm {

**private** List<Vertex> nodes;

**private** List<Edge> edges;

*@Test*

**public** **void** testExcute() {

nodes = **new** ArrayList<Vertex>();

edges = **new** ArrayList<Edge>();

**for** (**int** i = 0; i < 11; i++) {

Vertex location = **new** Vertex("Node\_" + i, "Node\_" + i);

nodes.add(location);

}

addLane("Edge\_0", 0, 1, 85);

addLane("Edge\_1", 0, 2, 217);

addLane("Edge\_2", 0, 4, 173);

addLane("Edge\_3", 2, 6, 186);

addLane("Edge\_4", 2, 7, 103);

addLane("Edge\_5", 3, 7, 183);

addLane("Edge\_6", 5, 8, 250);

addLane("Edge\_7", 8, 9, 84);

addLane("Edge\_8", 7, 9, 167);

addLane("Edge\_9", 4, 9, 502);

addLane("Edge\_10", 9, 10, 40);

addLane("Edge\_11", 1, 10, 600);

// Lets check from location Loc\_1 to Loc\_10

Graph graph = **new** Graph(nodes, edges);

DijkstraAlgorithm dijkstra = **new** DijkstraAlgorithm(graph);

dijkstra.execute(nodes.get(0));

LinkedList<Vertex> path = dijkstra.getPath(nodes.get(10));

assertNotNull(path);

assertTrue(path.size() > 0);

**for** (Vertex vertex : path) {

System.out.println(vertex);

}

}

**private** **void** addLane(String laneId, **int** sourceLocNo, **int** destLocNo,

**int** duration) {

Edge lane = **new** Edge(laneId,nodes.get(sourceLocNo), nodes.get(destLocNo), duration);

edges.add(lane);

}

}

## 5. Thank you

<http://learnedstuffs.wordpress.com/2012/03/18/my-graph-implementation-in-java/>

# My Graph Implementation in Java

March 18, 2012

8 Votes

These three classes will be used as my reference for all the succeeding Graph tutorials.

**Node.java**

|  |  |  |
| --- | --- | --- |
| 01 | public class Node<T> implements Comparable<Node<T>> { | |
| 02 |  |

|  |  |
| --- | --- |
| 03 | protected T data; |
| 04 | protected boolean visited; | |

|  |  |
| --- | --- |
| 05 | public Integer index = null; |
| 06 | public Integer lowlink = null; | |

|  |  |  |
| --- | --- | --- |
| 07 | public double distance = Double.POSITIVE\_INFINITY; | |
| 08 | public Node<T> predecessor = null; |

|  |  |
| --- | --- |
| 09 |  |
| 10 | public Node(T data) { | |

|  |  |  |
| --- | --- | --- |
| 11 | this.data = data; | |
| 12 | } |

|  |  |
| --- | --- |
| 13 |  |
| 14 | public Node() { | |

|  |  |  |
| --- | --- | --- |
| 15 |  | |
| 16 | } |

|  |  |
| --- | --- |
| 17 |  |
| 18 | public boolean isVisited() { | |

|  |  |  |
| --- | --- | --- |
| 19 | return visited; | |
| 20 | } |

|  |  |
| --- | --- |
| 21 |  |
| 22 | public void visit() { | |

|  |  |  |
| --- | --- | --- |
| 23 | visited = true; | |
| 24 | } |

|  |  |
| --- | --- |
| 25 |  |
| 26 | public void unvisit() { | |

|  |  |  |
| --- | --- | --- |
| 27 | visited = false; | |
| 28 | } |

|  |  |
| --- | --- |
| 29 |  |
| 30 | public int compareTo(Node<T> ob) { | |

|  |  |  |
| --- | --- | --- |
| 31 | String tempA = this.toString(); | |
| 32 | String tempB = ob.toString(); |

|  |  |
| --- | --- |
| 33 |  |
| 34 | return tempA.compareTo(tempB); | |

|  |  |  |
| --- | --- | --- |
| 35 | } | |
| 36 |  |

|  |  |
| --- | --- |
| 37 | public String toString() { |
| 38 | return data.toString(); | |

|  |  |  |
| --- | --- | --- |
| 39 | } | |
| 40 |  |

|  |  |
| --- | --- |
| 41 | } |

This Node class is a generic class. You can specify the data type when you want to use it.

**Edge.java**

|  |  |  |
| --- | --- | --- |
| 01 | public class Edge { | |
| 02 |  |

|  |  |
| --- | --- |
| 03 | protected Node a, b; |
| 04 | protected double weight; | |

|  |  |
| --- | --- |
| 05 |  |
| 06 | public Edge(Node a, Node b) { | |

|  |  |  |
| --- | --- | --- |
| 07 | this(a, b, Double.POSITIVE\_INFINITY); | |
| 08 | } |

|  |  |
| --- | --- |
| 09 |  |
| 10 | public Edge(Node a, Node b, double weight) { | |

|  |  |
| --- | --- |
| 11 | this.a = a; |
| 12 | this.b = b; |

|  |  |  |
| --- | --- | --- |
| 13 | this.weight = weight; | |
| 14 | } |

|  |  |
| --- | --- |
| 15 |  |
| 16 | public double getWeight() { | |

|  |  |  |
| --- | --- | --- |
| 17 | return weight; | |
| 18 | } |

|  |  |
| --- | --- |
| 19 |  |
| 20 | public String toString() { | |

|  |  |  |
| --- | --- | --- |
| 21 | return a + " ==> " + b; | |
| 22 | } |

|  |  |
| --- | --- |
| 23 |  |
| 24 | } | |

**Graph.java**

|  |  |  |
| --- | --- | --- |
| 001 | import java.util.\*; | |
| 002 |  |

|  |  |  |
| --- | --- | --- |
| 003 | public class Graph { | |
| 004 |  |

|  |  |
| --- | --- |
| 005 | protected Vector<Node> nodes = new Vector<Node>(); |
| 006 | protected Vector<Edge> edges = new Vector<Edge>(); |

|  |  |
| --- | --- |
| 007 | protected boolean directed = false; |
| 008 | protected boolean sortedNeighbors = false; | |

|  |  |
| --- | --- |
| 009 |  |
| 010 | public double[][] getAdjacencyMatrix() { | |

|  |  |  |
| --- | --- | --- |
| 011 | double[][] adjMatrix = new double[nodes.size()][nodes.size()]; | |
| 012 |  |

|  |  |
| --- | --- |
| 013 | for(int i = 0; i < nodes.size(); i++) |
| 014 | for(int j = 0; j < nodes.size(); j++) | |

|  |  |
| --- | --- |
| 015 | if(i == j) |
| 016 | adjMatrix[i][j] = 0; | |

|  |  |
| --- | --- |
| 017 | else |
| 018 | adjMatrix[i][j] = Double.POSITIVE\_INFINITY; | |

|  |  |
| --- | --- |
| 019 |  |
| 020 | for(int i = 0; i < nodes.size(); i++) { | |

|  |  |
| --- | --- |
| 021 | Node node = nodes.elementAt(i); |
| 022 | //System.out.println("Current node: " + node); | |

|  |  |
| --- | --- |
| 023 |  |
| 024 | for(int j = 0; j < edges.size(); j++) { | |

|  |  |  |
| --- | --- | --- |
| 025 | Edge edge = edges.elementAt(j); | |
| 026 |  |

|  |  |
| --- | --- |
| 027 | if(edge.a == node) { |
| 028 | int indexOfNeighbor = nodes.indexOf(edge.b); | |

|  |  |
| --- | --- |
| 029 |  |
| 030 | adjMatrix[i][indexOfNeighbor] = edge.weight; | |

|  |  |  |
| --- | --- | --- |
| 031 | } | |
| 032 | } |

|  |  |  |
| --- | --- | --- |
| 033 | } | |
| 034 |  |

|  |  |  |
| --- | --- | --- |
| 035 | return adjMatrix; | |
| 036 | } |

|  |  |
| --- | --- |
| 037 |  |
| 038 | public void setDirected() { | |

|  |  |  |
| --- | --- | --- |
| 039 | directed = true; | |
| 040 | } |

|  |  |
| --- | --- |
| 041 |  |
| 042 | public void setUndirected() { | |

|  |  |  |
| --- | --- | --- |
| 043 | directed = false; | |
| 044 | } |

|  |  |
| --- | --- |
| 045 |  |
| 046 | public boolean isDirected() { | |

|  |  |  |
| --- | --- | --- |
| 047 | return directed; | |
| 048 | } |

|  |  |
| --- | --- |
| 049 |  |
| 050 | public boolean isSortedNeighbors() { | |

|  |  |  |
| --- | --- | --- |
| 051 | return sortedNeighbors; | |
| 052 | } |

|  |  |
| --- | --- |
| 053 |  |
| 054 | public void setSortedNeighbors(boolean flag) { | |

|  |  |  |
| --- | --- | --- |
| 055 | sortedNeighbors = flag; | |
| 056 | } |

|  |  |
| --- | --- |
| 057 |  |
| 058 | public int indexOf(Node a) { | |

|  |  |
| --- | --- |
| 059 | for(int i = 0; i < nodes.size(); i++) |
| 060 | if(nodes.elementAt(i).data.equals(a.data)) | |

|  |  |  |
| --- | --- | --- |
| 061 | return i; | |
| 062 |  |

|  |  |  |
| --- | --- | --- |
| 063 | return -1; | |
| 064 | } |

|  |  |
| --- | --- |
| 065 |  |
| 066 | public Vector<Node> getNodes() { | |

|  |  |  |
| --- | --- | --- |
| 067 | return nodes; | |
| 068 | } |

|  |  |
| --- | --- |
| 069 |  |
| 070 | public Vector<Edge> getEdges() { | |

|  |  |  |
| --- | --- | --- |
| 071 | return edges; | |
| 072 | } |

|  |  |
| --- | --- |
| 073 |  |
| 074 | public Node getNodeAt(int i) { | |

|  |  |  |
| --- | --- | --- |
| 075 | return nodes.elementAt(i); | |
| 076 | } |

|  |  |
| --- | --- |
| 077 |  |
| 078 | public void unvisitAllNodes() { | |

|  |  |  |
| --- | --- | --- |
| 079 | for(int i = 0; i < nodes.size(); i++) | |
| 080 | nodes.elementAt(i).unvisit(); |

|  |  |  |
| --- | --- | --- |
| 081 | } | |
| 082 |  |

|  |  |
| --- | --- |
| 083 | public Vector<Node> getNeighbors(Node a) { |
| 084 | Vector<Node> neighbors = new Vector<Node>(); | |

|  |  |
| --- | --- |
| 085 |  |
| 086 | for(int i = 0; i < edges.size(); i++) { | |

|  |  |  |
| --- | --- | --- |
| 087 | Edge edge = edges.elementAt(i); | |
| 088 |  |

|  |  |
| --- | --- |
| 089 | if(edge.a == a) |
| 090 | neighbors.add(edge.b); | |

|  |  |
| --- | --- |
| 091 |  |
| 092 | if(!directed && edge.b == a) | |

|  |  |  |
| --- | --- | --- |
| 093 | neighbors.add(edge.a); | |
| 094 | } |

|  |  |
| --- | --- |
| 095 |  |
| 096 | if(sortedNeighbors) | |

|  |  |  |
| --- | --- | --- |
| 097 | Collections.sort(neighbors); | |
| 098 |  |

|  |  |  |
| --- | --- | --- |
| 099 | return neighbors; | |
| 100 | } |

|  |  |
| --- | --- |
| 101 |  |
| 102 | public int addNode(Node a) { | |

|  |  |  |
| --- | --- | --- |
| 103 | nodes.add(a); | |
| 104 |  |

|  |  |  |
| --- | --- | --- |
| 105 | return nodes.size() - 1; | |
| 106 | } |

|  |  |
| --- | --- |
| 107 |  |
| 108 | public void addEdge(Edge a) { | |

|  |  |  |
| --- | --- | --- |
| 109 | edges.add(a); | |
| 110 |  |

|  |  |
| --- | --- |
| 111 | if(!directed) |
| 112 | edges.add(new Edge(a.b, a.a, a.weight)); | |

|  |  |  |
| --- | --- | --- |
| 113 | } | |
| 114 |  |

|  |  |
| --- | --- |
| 115 | public void printNodes() { |
| 116 | System.out.println(nodes); | |

|  |  |  |
| --- | --- | --- |
| 117 | } | |
| 118 |  |

|  |  |
| --- | --- |
| 119 | public void printEdges() { |
| 120 | System.out.println(edges); | |

|  |  |  |
| --- | --- | --- |
| 121 | } | |
| 122 |  |

|  |  |
| --- | --- |
| 123 | } |

The graph implemented here can either be directed or undirected. After creating a Graph object, all you have to do is to call the needed methods to set its properties.

By default, the graph is undirected. Also, the arrangement of the neighbors does not matter. If the arrangement matters, set the **sortedNeighbors** flag to true.

<http://www.params.me/2011/10/graph-data-structure-java.html>

## Monday, October 31, 2011

### Graph Data Structure Java Implementation

This post illustrates the simple implementation of Graph data structure in Java:  
  
Graph Node Implementation:

package com.param.datastructures;  
  
/\*\*  
 \* Building block for graph data structure  
 \*   
 \* @author parampreetsethi  
 \*   
 \*/  
public class GraphNode {  
  
    public int data;  
    public State state;  
  
    public GraphNode(int d) {  
        this.data = d;  
        this.state = State.UnVisited;  
    }  
}  
  
enum State {  
    UnVisited, Visited, Processed;  
}

Graph Structure Implementation:  
  
package com.param.datastructures;  
  
import java.util.ArrayList;  
import java.util.LinkedList;  
import java.util.List;  
  
/\*\*  
 \* Implementation of Graph data structure  
 \*   
 \* @author parampreetsethi  
 \*   
 \*/  
public class Graph {  
    public GraphNode root;  
    ArrayList<GraphNode> nodes;  
    int[][] adjMatrix; // Adjacency Matrix  
    boolean isDirected; // Graph directed?  
    int size;  
  
    public static final int DEFAULT\_SIZE = 5; // default size  
  
    public Graph(int data) {  
        this(DEFAULT\_SIZE, data);  
    }  
  
    /\*\*  
     \* Assuming there are Max five nodes in the graph  
     \*   
     \* This restriction can be removed by using adjList which grows dynamically  
     \*   
     \* @param root  
     \*/  
    public Graph(int size, int data) {  
        this.root = new GraphNode(data);  
        nodes = new ArrayList<GraphNode>();  
        nodes.add(root);  
        this.size = size;  
        adjMatrix = new int[size][size];  
    }  
  
    public void addNode(int data) {  
        GraphNode node = new GraphNode(data);  
        nodes.add(node);  
    }  
  
    public void connectNode(GraphNode g1, GraphNode g2) {  
        int g1Idx = nodes.indexOf(g1);  
        int g2Idx = nodes.indexOf(g2);  
        if (g1Idx < 0 || g2Idx < 0)  
            throw new NullPointerException("node not found");  
        adjMatrix[g1Idx][g2Idx] = 1;  
        if (!isDirected) {  
            adjMatrix[g2Idx][g1Idx] = 1;  
        }  
    }  
  
    /\*\*  
     \* get all nodes connected to the given node  
     \*   
     \* @param g  
     \* @return  
     \*/  
    public List<GraphNode> getAdjacents(GraphNode g) {  
        int gIdx = nodes.indexOf(g);  
  
        if (gIdx < 0)  
            throw new NullPointerException("node not found in the graph");  
        LinkedList<GraphNode> adjNodes = new LinkedList<GraphNode>();  
        for (int j = 0; j < adjMatrix.length; j++) {  
            if (adjMatrix[gIdx][j] == 1) {  
                adjNodes.add(nodes.get(j));  
            }  
        }  
  
        return adjNodes;  
    }  
  
    // Similarly implement remove edge  
    public void removeEdge(GraphNode g1, GraphNode g2) {  
        int g1Idx = nodes.indexOf(g1);  
        int g2Idx = nodes.indexOf(g2);  
          
        if(g1Idx<0 || g2Idx <0){  
            throw new NullPointerException("Node not found");  
        }  
          
        adjMatrix[g1Idx][g2Idx] = 0;  
        if(!isDirected){  
            adjMatrix[g2Idx][g1Idx] = 0;  
        }  
    }  
      
}