

Writeup for first project of
CMSC 420: “Data Structures”
Section 0101, Spring 2019

Theme: Graph representations

Handout date: **TODO**

On-time deadline: **TODO**

Late deadline (30% penalty): **TODO**

Contents

1	Overview	2
2	Graph representations	2
2.1	Adjacency Matrix	2
2.2	Adjacency List	3
2.3	Sparse Adjacency Matrix	4
3	Getting started	5
4	Hints / Tips / FAQs	6
5	Submission / Grading	9

1 Overview

In this project you will implement three different graph representations for computer memory: an *adjacency matrix*, *adjacency list*, as well as a twist on the adjacency matrix representation known as a *sparse adjacency matrix*. The graphs that you will be building will be **directed** and **weighted**. The weights will be **non-negative integers**. You will have to implement several methods for insertion of nodes and edges, queries for the existence of edges or for the weight of an edge, as well as **shortest path computation**. There are also methods for dynamically transforming every type of **Graph** into a different one.

The entire point of the project is to have you think about the **computational complexity**, both in **space** and in **time**, of various graph implementations. For example, for graphs with a big number of nodes and a small number of edges (sparse graphs), the representation of an adjacency matrix **wastes a lot of space**, but it makes searching for edges (or their weights) a **constant-time** operation. An adjacency list **saves space** by only storing neighbors of nodes, but searching for a particular neighbor becomes a **linear-time** operation (on the number of nodes in the graph).

2 Graph representations

Consider the graph of Figure 1, which is the graph on which your Release Tests are based.

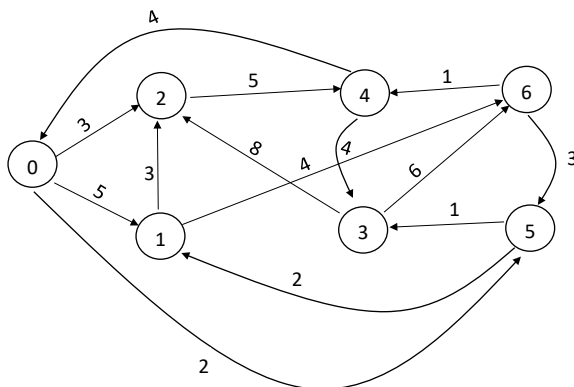


Figure 1: A weighted, directed graph that you will be tested on.

Figure 1 is just a conceptual figure that helps humans visualize the graph. In computer memory, we have to use appropriate, simple **data structures** to store the graph. Our choice of data structure has straightforward implications about the memory footprint of our graph as well as the computational complexity of the operations that we expect our graph to allow. You will be implementing graphs in three different days: An *adjacency matrix* (section 2.1), an *adjacency list* (section 2.2), as well as a *sparse adjacency matrix* (section 2.3).

During the explanation of all the different representations that you will have to implement, we will be using V to denote the number of **vertices** in the graph and E to denote the number of its **edges**.

2.1 Adjacency Matrix

The adjacency matrix is a $V \times V$ matrix M defined as follows:

$$M(i, j) = \begin{cases} w_{ij}, & \text{if there is an edge from vertex } i \text{ to vertex } j \text{ with weight } w_{ij} > 0, \\ 0, & \text{otherwise} \end{cases}$$

For example, consider Figure 2, which shows what the **adjacency matrix** representation of the graph in Figure 1 would look like:

	0	1	2	3	4	5	6
0	0	5	3	0	0	2	0
1	0	0	3	0	0	0	4
2	0	0	0	0	5	0	0
3	0	0	8	0	0	0	6
4	4	0	0	4	0	0	0
5	0	2	0	1	0	0	0
6	0	0	0	0	1	3	0

Figure 2: The adjacency matrix representation of the graph of Figure 1.

We can quickly see that the adjacency matrix representation allows for constant-time edge operations: A new edge between existing nodes is added in $\mathcal{O}(1)$, an existing one is removed in constant time as well, similarly for the altering of its weight. However, finding the neighbors of a given node is an $\mathcal{O}(V)$ operation (linear time on the number of nodes). Additionally, adding a new node is a *quadratic* operation $\mathcal{O}(V^2)$, since we have to allocate new space for a $(V + 1) \times (V + 1)$ matrix, and then copy over the old values into the new matrix. However, with some simple engineering you can drop this to $\mathcal{O}(V)$!

Perhaps the biggest, and unfortunately unavoidable, problem with the adjacency matrix is that it **wastes a lot of space, particularly for sparse graphs**. Consider, for example, a graph with 100 different nodes, only 2 of which are connected. The adjacency matrix representation of that graph would consist of 9,999 cells of zeroes, and 1 cell with the weight of the single edge that exists in the graph. While this is without a doubt an extreme example, the problem is real. The numerical programming language **MATLAB** has a built-in keyword called **sparse** that turns sparse matrices like the aforementioned into a **sparsified representation** (section 2.3) to avoid this rather large memory footprint. As we will soon see, this can have **vast** spatial benefits, but it also makes several operations linear - time!

2.2 Adjacency List

Another common representation of graphs is the so-called *adjacency list*. This representation consists of a V -sized one-dimensional array of lists, where $V(i)$ is a pointer to a list that has all the neighbors of node i , along with the weights of the edges that connect i to the respective neighbors.¹ Figure 3 shows what the **adjacency list** representation of the graph in Figure 1 would look like:

¹If i doesn't have neighbors, the pointer could be **null** or the list pointed to could be empty; that's an implementation detail.

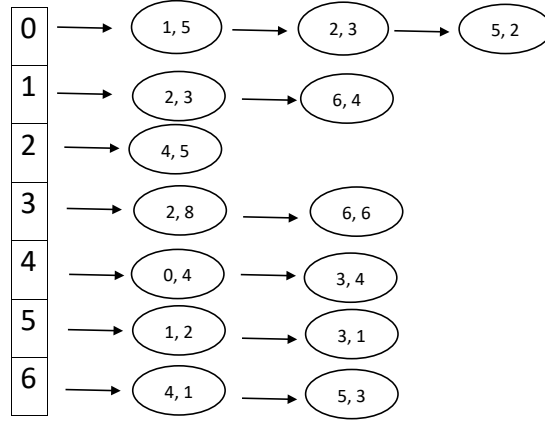


Figure 3: The adjacency list representation of the graph of Figure 1. Note that the order of the neighbors of each node is entirely arbitrary (but it doesn't have to be this way).

Compared to the adjacency matrix representation, this adjacency list can potentially save a lot of space. This is even more true for undirected graphs, where essentially half of the adjacency matrix is superfluous information that could be thrown away. Unfortunately, the trade-off is that some of the constant-time operations of the adjacency matrix are now **linear-time** (in V). Specifically, deleting an edge and querying about whether a certain edge exists implies scanning through the entirety of the i -th list, which might imply scanning through $\mathcal{O}(V)$ nodes. Interestingly, adding a new edge is **also** linear-time since we need to check if the edge already exists and, if so, update its weight!

On the other hand, retrieving the neighbors of node i is a **constant**-time operation, since those are already linked to i in the representation, via a simple reference. This is very important for algorithms that query nodes for their neighbors often, such as Dijkstra's Shortest Path algorithm!

2.3 Sparse Adjacency Matrix

The adjacency matrix and adjacency list representations are the most standard ways that graphs are implemented in computer memory, and can be found in several Algorithms textbooks and software libraries. There also exist several variations of those representations. For example, in undirected graphs, there are ways for us to only store the **upper triangular** part of the matrix (since for undirected graphs the adjacency matrix is symmetric). When it comes to the adjacency list, some libraries implement it as a **list - of - lists** instead of an **array - of - lists**. The lists themselves can also be of variable nature: linked lists, arraylists, sorted by some metric, unsorted, etc.

In this project, we expand our available representations with another one: the *sparse* adjacency matrix. A sparse matrix is a *linearization* of a matrix, in either row or column-major order, where all of the non-null (or non-zero) elements of the matrix are gathered into a list. Every node of the list needs to contain the data point of the cell as well as its (i, j) indices, so that we can re-construct the dense matrix if necessary.

Figure 4 details the sparse adjacency matrix representation of the graph of Figure 1.

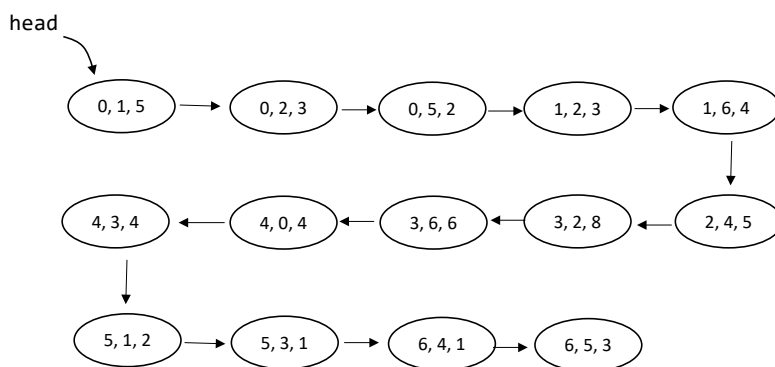


Figure 4: The sparse adjacency matrix representation of the graph of Figure 1. The matrix has been linearized in **row-major** order.

We can immediately see the large spatial benefit that this representation could have when compared with the adjacency matrix representation. Unfortunately, by saving space in this way we have to “pay” with *quadratic* ($\mathcal{O}(V^2)$) complexity for all edge operations, where those were of *constant* ($\mathcal{O}(1)$) complexity in the adjacency matrix representation and *linear* ($\mathcal{O}(V)$) complexity in the adjacency list. Interestingly, adding a new node is very efficient in this representation: we just add a new triplet (`new_node_id`, `new_node_id`, 0) at the head of the list and make sure we internally increase V by 1, so that we can re-construct the dense matrix appropriately if required. That is a **constant-time** operation.

A practical aspect that you will have to pay attention to when linearizing an adjacency matrix is whether to scan the adjacency matrix in **row** or **column**-major order (along rows or columns, respectively). This requires knowledge of how your programming language manages array storage. In the Java programming language (as well as C/C++), memory is organized in a **row - major** order, so for efficiency considerations you should always linearize your matrices in that order.²

3 Getting started

Everything you need to get started is available in our [common Git repository](#). You will need to fill in the implementation of the following 4 classes:

- `AdjacencyMatrixGraph`
- `AdjacencyListGraph`
- `SparseAdjacencyMatrixGraph`
- `StudentTests` (you should add more `jUnit` tests to this file)

All of those classes extend an **abstract** class called `Graph`. You **should study the Javadocs for both `Graph` and the three classes that you have to implement in order to understand how they work and how corner cases should be handled**. For example, you should study the documentation of `Graph::addEdge(int source, int target, int weight)` in order to understand what should happen when you provide a weight of 0 (zero) or a negative weight (the expected behaviors are different, and we test for them!).

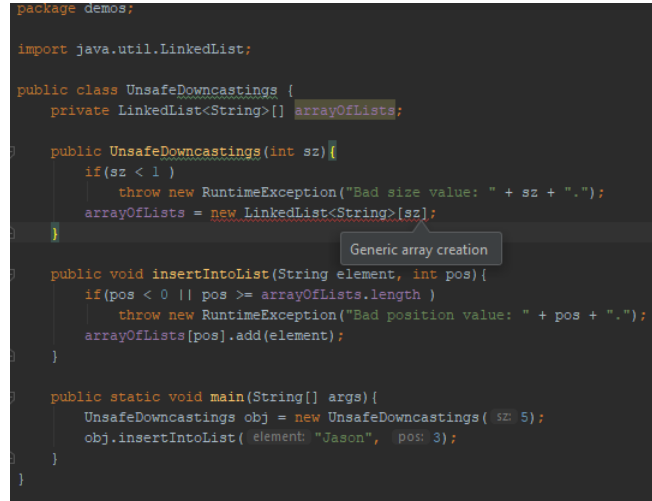
²Octave, MATLAB and LISP follow **column-major** order.

The classes come with **documentation** that you will be able to find under the directory `doc`, so that you can have a full view of the functionality exposed by the class' **public** interface. The three `Graph` subtypes that you have to implement, as well as the method `Graph::shortestPath(int source, int target)` which you will implement inside `Graph.java` throw a `RuntimeException` by default to warn you that you need to implement the relevant method; you should replace that exception throwing with your implementation when you are ready.

4 Hints / Tips / FAQs

- (a) Familiarize yourselves with the most elementary `git` commands (`clone`, `pull`, `add`, `commit`, `push`, `alias`). Our ELMS page has startup resources for you under a page called “More Resources”. Your elementary `git` knowledge will also be tested on your **first homework**. We **highly** suggest that you **pull** the code (this is literally a single command) **instead of downloading the zip** from GitHub. This is because, if we ever post a correction on the docs or the source code, you will only need to type **one single command** to pull **only the changes themselves**, without affecting your existing solutions! If you always download the ZIP whenever a correction is made, you will have to nitpick the files that you copy-and-paste and **possibly overwrite your work by mistake**. In fact, this is **guaranteed to happen** if you download the ZIP the first time, work on the project for a while, and when a correction is posted you decide to try out `git` and do a `git clone`. This will **immediately overwrite everything in your current directory and replace it with the file state of the latest upstream commit**. **IN THE NOT SO DISTANT PAST, THE INSTRUCTOR LOST TWO WEEKS' WORTH OF CODE BECAUSE OF THIS MISTAKE.**
- (b) **TEST FIRST, IMPLEMENT SECOND. WE CANNOT STRESS HOW EASIER THIS WILL MAKE YOUR DEVELOPMENT.** We provide some starting tests in a file called `StudentTests.java` which you should extend with your own tests before submitting your code. Further suggestion: find somebody from the class, **write their unit tests for them** and have **them** write **your** unit tests. Both of you should view the other one as your **adversary** in the class! Think in the following manner: *“Based on the provided interface as it is described in the JavaDocs, how can I write tests that will only be passed by a tip-top implementation? How can I try my best so the other person's code DOES NOT GET 100% credit?”*
- (c) You are **free** to add more **private** data elements or methods in any one of the three classes that you have to implement. You should **not** touch the **private** elements that we have already given you, as well as the **public** methods that you should implement. If you change the signature of the methods, the submit server tests will not run properly against your code! If you touch the **private** elements, you are changing the inner representation of the classes and you are missing the entire point of the project; we will be forced to manually deduct points (section 5)!
- (d) You might notice that the private field of `AdjacencyListGraph` is an array over a custom type called `NeighborList`. `NeighborList` itself is a simple linked list over instances of a

custom type called `Neighbor`.³ You might wonder why we choose to not simply give you a `java.util.List` over the type `Neighbor`, instead implementing and giving you an entirely new list from scratch. We do this because *creating a raw array over generic types* in Java is a *pain*. As you can see in figure 5, in Java it is **not** possible to create a raw array over generics.

A screenshot of a code editor showing Java code. The code defines a class `UnsafeDowncastings` with a private field `LinkedList<String>[] arrayOfLists`. In the constructor, it attempts to create an array of `LinkedList<String>` objects using `new LinkedList<String>[sz]`. A tooltip points to this line with the text "Generic array creation". The `insertIntoList` method also uses `arrayOfLists[pos].add(element)`. The `main` method creates an instance of `UnsafeDowncastings` and calls `insertIntoList`. The code is syntactically correct but will fail to compile due to the generic array creation.

```
package demos;

import java.util.LinkedList;

public class UnsafeDowncastings {
    private LinkedList<String>[] arrayOfLists;

    public UnsafeDowncastings(int sz){
        if(sz < 1 )
            throw new RuntimeException("Bad size value: " + sz + ".");
        arrayOfLists = new LinkedList<String>[sz];
    }

    public void insertIntoList(String element, int pos){
        if(pos < 0 || pos >= arrayOfLists.length )
            throw new RuntimeException("Bad position value: " + pos + ".");
        arrayOfLists[pos].add(element);
    }

    public static void main(String[] args){
        UnsafeDowncastings obj = new UnsafeDowncastings( 5);
        obj.insertIntoList( element: "Jason", pos: 3);
    }
}
```

Figure 5: Creating an array of generic types leads to a **compile-time** error.

Instead, one would need to declare a raw array over `Objects` and then hope that the down-castings involved will be safe. **Unfortunately**, `java.util.LinkedList` is a type that implements the interface `Iterable` (and so are *all* `java.util.Lists`), an interface that is **not** implemented by `java.lang.Object`. This in turn means that the downcasting of *any* `Object` reference to a `LinkedList` reference is **inherently unsafe**, since the line of code that is making the downcasting might request access to the `iterator()` method available to a `LinkedList` instance, but **unavailable** to an `Object` instance. This is further demonstrated by Figure 6:

³Both of those types have been implemented under the sub-package `utils` and they come with sufficient Javadoc for you to study.

```

package demos;

import java.util.LinkedList;

public class UnsafeDowncastings {
    private Object[] arrayOfLists;

    public UnsafeDowncastings(int sz) {
        if (sz < 1)
            throw new RuntimeException("Bad size value: " + sz + ".");
        arrayOfLists = (LinkedList<String>[]) new Object[sz];
    }

    public void insertIntoList(String element, int pos) {
        if (pos < 0 || pos >= arrayOfLists.length)
            throw new RuntimeException("Bad position value: " + pos + ".");
        ((LinkedList<String>) arrayOfLists[pos]).add(element);
    }

    public static void main(String[] args) {
        UnsafeDowncastings obj = new UnsafeDowncastings(5);
        obj.insertIntoList("Jason", 3);
    }
}

```

Unchecked cast: 'java.lang.Object[]' to 'java.util.LinkedList<java.lang.String>[]' more... (Ctrl+F1)

Figure 6: An example of an unchecked and unsafe downcast. This small snippet of code has been included for you under the package `demos` on our Git repo, for your experimentation. The code, as shown, throws an instance of `java.lang.ClassCastException`.

An alternative would be to drop the raw array and use a `java.util.ArrayList`. Unfortunately, this would mean that you are no longer in control of the re-sizings of the adjacency list, which we want you to do explicitly so that you have an understanding of the complexity of the various operations under different graph representations! So we stick to the raw array. One consequence of this necessary modeling is that your implementation of `getNeighbors()` for `AdjacencyList` will probably be **linear-time**. **That is fine**. As long as you understand that *in theory*, without any programming language intricacies, retrieving the list of neighbors in an adjacency list *should* be a constant-time operation, you will be fine for any relevant exam / homework questions.

- (e) While implementing the project you will notice that `NeighborList` implements the `Iterable` interface. We do this because, in your implementation of `AdjacencyListGraph`, you are very likely to need to iterate over the elements of `NeighborList` using a “for-each” loop. Read the JavaDoc of `NeighborList` and its `iterator()` method for more information on how this is achieved.
- (f) For computing the shortest path between a given source and sink node, you are **not** allowed to use ready-made implementations of Dijkstra’s, Bellman-Ford or Johnson’s algorithm; you **have** to implement the shortest path computation yourselves. However, if you would like to use ready - made priority queues, Fibonacci Heaps, hashes, lists, or any other data structure that you might find helpful, you are absolutely allowed to do so (but you are also responsible for the code you use being **correct**). This requirement is repeated in the documentation of the method `Graph::shortestPath`, which is where you will be implementing shortest-path functionality.

Note: If you want to use Java’s `PriorityQueue` class, the documentation reads:

[...] If multiple elements are tied for least value, the head is one of those elements
 – ties are broken arbitrarily [...]

which is **not** the behavior we are expecting from a priority queue. In a priority queue, when there are ties in the priority of two elements, the tie should be broken in **FIFO** order; the element first inserted is the element that is closer to the “head” of the queue.. So you will have to do some work to make sure that you break ties appropriately.

- (g) Consider switching from Eclipse to an IDE such as [NetBeans](#) or [IntelliJ](#). Over the past 2 years, students have found IntelliJ, in particular, to be a great tool to work with (and we use it as well). You can find IntelliJ resources posted on ELMS.

5 Submission / Grading

Credit in this project is defined by the number of [submit server](#) unit tests that you pass. However, **we will also be inspecting your source code to make sure that your work adheres to certain standards.** Briefly:

- `AdjacencyMatrixGraph` should be implemented as a 2D row array of non-negative integers.
- `AdjacencyListGraph` should be implemented as a 1D row array over instances of a `java.util.List`.
- `SparseAdjacencyMatrixGraph` should be implemented as an instance of `java.util.List` over a simple data type with the triplet of elements (**start, destination, weight**) that we have defined for you in the class.

These modeling guidelines are outlined for you in the `private` fields of the three classes. You should **not change** these fields! **In order to get any credit for the unit tests that correspond to a specific class, your implementation should adhere to the documentation’s guidelines.** In particular, using built-in Java primitives or third-party libraries that implement graphs is an easy way to ensure **zero credit**!

Projects in this class are different from your typical 131/2 projects in that **we do not maintain an Eclipse - accessed CVS repository** for you or us. This means that you can no longer use the Eclipse Course Management Plugin to submit your project on the submit server. This turns out to be a good thing, since it frees you up from the need to use Eclipse if you don’t want to.

To submit your project, run the script `src/Archiver.java` as a **Java application** from your IDE (tested with Eclipse and IntelliJ). This will create a .zip file of your entire project directory at the same directory level of your entire project directory, without including the hidden `git` directory `.git` that can sometimes be very large and cause problems with uploads on `submit.cs`. For example, if your project directory is under `/home/users/me/mycode/project1/`, this script will create the .zip archive `/home/users/me/mycode/project1.zip`, which will contain `src` and any other directories that you may have, but will **not** contain the directory `.git` and `doc`.⁴ After you have done this, upload the archive on the submit server as seen on figure 7.

⁴The script excludes `doc` only because it makes our grading on the submit server easier

