# A Directed Acyclic Graph Map in Java

# CITS2200 (Data Structures and Algorithms)

## By

## Caleb Fetzer (21384976) Reece Notargiacomo (21108155)

**A report describing:**

Implementation Choice (20%)

* Details of implementation
* Reasoning behind choice

The Complexity of Methods (40%)

* Describe methods
* Analysis of performance (Big-O)
* Justification for choice
* What could be done to improve choices

Test results section (20%)

* Instructions for running the tests
* The rationale behind the tests
* Evidence that the practical complexity of the methods agrees with the theoretical growth rate

# Abstract

We implemented a Directed Acyclic Graph Map, which invovles the functionality of a directed graph with no cycles, and the Key-Value pairing of a map. The implementation was used with an Adjacency List data structure as an excercise for recursive data structures, and Java API's Util LinkedHashSet functions for the storing of Keys and Values. The time complexity of the standard lookups are reflected by the general time complexities (in Big O) of the respective implementation – adjacency list. We found the theoretical growth rates of method functionality matched practicality, using JUnit testing as a basis for a time vs operations comparison.The main methods, getWidth, getLongestPath and getMax for finding the width of the DAG, the longest path of the DAG and the Max-flow of the DAG respectively, were near the theoretical rates.

# Implementation choice

We decided to implement an Adjacency List for the DAGMap, as opposed to an Adjacency Matrix. The Adjacency Lists also have lists for predecessor and successor nodes. To store the key value pairs, a set of keys is created. A rootNode value is also created, which keeps track of the first parent node (a node with children).

There were points identified, to justify the decision, such as the pro’s and con’s of each data type. The time complexity for the standard operations these data types implemented, were also addressed and compared.

­Although the Adjacency List on the surface appears to be a less appealable data type, just via inspection of the general type of methods; it uses less space depending on the size of the graph. An adjacency matrix creates space for a graph with vertices that may not exist in the graph, thus occupying more space. On very sparse graphs, an adjacency matrix uses a lot of space that is not needed, and thus was our main argument for an adjacency list over an adjacency matrix. From a non-scientific point of view, anecdotal evidence from Stack Overflow, would suggest that in most usual cases an adjacency list is preferable to an adjacency matrix implementation1

The learning curve, for us, was trying to implement a recursive data type – to gain a better understanding of recursion and it’s role in data structures and algorithms.

|  |  |  |
| --- | --- | --- |
| **Operation** | **Adj Matrix Time Complexity** | **Adj List Time Complexity** |
| Adding/Deleting edge | O(1) | O(log (n)) |
| Does edge exist between vi & vj? | O(1) | O(log (n)) |
| Successors of vertex | O(n) | O(k) |
| Finding path between two vertices | O(n2) | O(n + m) , m <= n |

(2014, Algorithm of the Week). ‘*k’ refers to the length of lists containing the successors of a vertex, m refers to the number of edges.*

# The method complexities and implementation details

To note, methods that run in constant time, such as isEmpty, won’t have their time complexity detailed.

**public void put (Key newKey, Value newValue)**

* The method calls the methods containsKey(from DAGMap), add(from java.util.Set). This method accepts two parameters, newKey and newValue. It checks to make sure newKey does not already exist in the set of keys in the DAGMap, and if not adds it to the set.

**public object get (Key k)**

* This method checks for the value of a given key. It makes sure that the key exists, else an illegal argument exception is thrown. This runs in O(n) time, as it peforms a single look-up operation, using the contains() method from the java.util.set.

**public void remove (Key k)**

* This method removes the given key from the keys in the DAGMap, as well as removing it from the list of predecessors and list of successors. Therefore the associated dependencies/requirements of this key are removed. It runs in o(n^2) time as it performs a lookup of all the keys in the predecessors that listed ‘k’ as a dependent, all the keys in the successors that listed ‘k’ as a requirement, and then removes ‘k’ correspondingly.

**public set<Key> getPredecessors(Key k)**

* This method returns the set of keys of predecessors (i.e. parents) of a given Key ‘k’. A single look up is performed with a boolean check, and the list of predecessors for k are returned. If the key is not defined an IllegalArgumentException is thrown. This method runs in O(n) time.

**public set<Key> getSuccessors(Key k)**

* This method returns the set of keys of successors (i.e child(ren)) of a given Key ‘k’. A single look up is performed with a boolean check, and the list of successors for k are returned. If the key is not defined an IllegalArgumentException is thrown. This method runs in O(n) time.

**public void addDependency (Key kReq, Key kDep)**

* This method adds an edge between a requirement key (kReq) and a dependent key (kDep), making kDep dependent on kReq. An ‘isDependent’ checkup is performed, making sure that kDep is not already dependent on kReq. It then adds kDep to the successor list for, and kReq to the predecessor list. It also removes the kDep key from the list of orphan keys (Keys with no predecessors). Runs in .. time

**public void removeDependency (Key kReq, Key kDep)**

* This method removes an existing edge between a requirement key (kReq) and a dependent key (kDep), making kDep no longer dependent on kReq. An isDependent check is performed and following, kReq is removed from the set of successors and kDep is removed from the set of predecessor. If no edge exists between kReq and kDep, an IllegalArgumentException is thrown.

**public Boolean containsKey (Key k)**

* This method checks the set of keys in the DAGMap for the given key k, if k exists, return true. It checks for the set of keys not equal to null, and then iterates through each key in the set of keys in the DAGMap. If the key is found, true is returned.

**public Boolean containsValue (Value value)**

* This method checks all of the keys in the DAGMap for a given value, for any key that the value is paired with. If the value doesn’t exist in the DAGMap, false is returned. As a standard lookup function, this runs in O(n) time, as it is simply performing an iteration.

**public Boolean isDependent (Key haystackKey, Key needleKey)**

* This refers to the finding a needle in a haystack metaphor. The method checks to make sure haystackKey and needleKey are not null, and if haystackKey equals needleKey it returns true, as this means that the haystackKey is dependent on itself. If not the case, haystackKey is checked against the list of successors, by performing a getSuccessors check, and then recursively checking for children keys that are dependent on the haystackKey through the needleKey. This way it is checking not just immediate dependencies, but rather all dependencies. If haystackKey or needleKey do not exist, an IllegalArgumentException is thrown.

**public Set<K> getKeySet()**

* This method performs a simple lookup and returns all of the keys in the DAGMap as a set. Because it is retrieving keys from a given set, it performs in O(n) time.

Bibliography

A list of references

Popov, Stoimen, Accessed on May 20th 2014, Algorithm of the Week: Graphs and their Representation, <http://java.dzone.com/articles/algorithm-week-graphs-and?utm_source=twitterfeed&utm_medium=twitter&utm_campaign=Feed%3A+javalobby%2Ffrontpage+%28Javalobby+%2F+Java+Zone%29>

GitHub