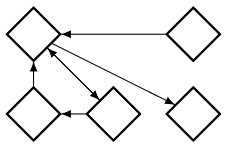
# Graphs

**Last updated:** November 26th 2018, at 12.39pm

## Contents

## 1 Graphs

Here, we shall *not* be talking about *curves* drawn on a pair of axis. By the term *graph*, we shall mean a structure such as:



Thus:

> A graph is a *collection of nodes* (sometimes called *vertices*), connected by *edges* (sometimes referred to as *arcs*).

## 2 Implementing Graphs

### 2.1 Adjacency Tables

One way to represent graphs is using a *two dimensional array*. Each entry in the array is of type boolean (i.e. we explicitly encode the graph's *accessibility* relation).

Since each entry in the table is addressed by *two* co-ordinates (we have two dimensions!), we may view the *boolean value* stored in a particular entry as being an answer to the question: is there an edge joining vertex $i$ to $j$?.

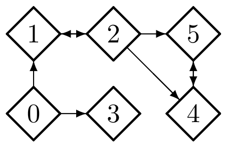
If node $i$ has an edge to node $j$ in our graph then:

$$\texttt{graph}[\texttt{i}][\texttt{j}] = \textbf{true}$$

otherwise:

$$\texttt{graph}[\texttt{i}][\texttt{j}] = \textbf{false}$$

**Example:** The following graph:



can be represented by the following *adjacency table*:

| | **0** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|
| **0** | false | true | false | true | false | false |
| **1** | false | false | true | false | false | false |
| **2** | false | true | false | false | true | true |
| **3** | false | false | false | false | false | false |
| **4** | false | false | false | false | false | true |
| **5** | false | false | false | false | true | false |

However, this is often space inefficient. In the table above there are far more false entries than there are true entries. This is typical for many graphs. Many graphs are *sparsely linked* — given any two nodes from such a graph it is much more likely that they will not be linked than that they will be. We want some way of listing only those nodes that are linked, and not those that aren't.

### 2.2 Adjacency List

Adjacency lists are sometimes also referred to as *associative lists*.

Here we use a one dimensional array.

Each entry in the array is a *set of vertices*.

Each set describes the graph vertices accessible from a given vertex.

If vertex $i$ is *only* joined (via an edge) to the vertices $j$, $k$ and $l$:

$$\begin{tikzcd}
 & i \arrow[dl] \arrow[d] \arrow[dr] & \\
j & k & l
\end{tikzcd}$$

then:

$$\texttt{graph}[\texttt{i}] = \{\texttt{j}, \texttt{k}, \texttt{l}\}$$

The following example illustrates how this works in practice:

**Example:** The graph:

$$\begin{tikzcd}
0 \arrow[r] \arrow[d] & 1 \arrow[r, leftrightarrow] & 2 \arrow[dl] \arrow[d] \\
3 & 4 \arrow[r, leftrightarrow] & 5
\end{tikzcd}$$

can be represented by the following adjacency list:

| | |
|---|---|
| **0** | $\{1, 3\}$ |
| **1** | $\{2\}$ |
| **2** | $\{1, 4, 5\}$ |
| **3** | $\emptyset$ |
| **4** | $\{5\}$ |
| **5** | $\{4\}$ |

## 2.3 Notes

### 2.3.1 Non-Integer Nodes

Notice that, so far, we have made the implicit assumption that vertices of a graph should be labelled using integers.

If we use Hashtables instead of arrays to implement our graphs, then we may label vertices using other objects e.g. strings, doubles, etc.

This is particularly useful when we use graphs to model, e.g.:

- parse trees
- lazy data structures (in functional programming languages)
- relational data structures (in databases and logic programming languages)
- states of machines (e.g. Turing machines and push-down automaton)
- etc.

### 2.3.2 Space Efficiency

For sparse graphs (i.e. many more nodes than edges), adjacency tables *waste* a lot of space.

- need (number of nodes)$^2$ memory cells in order to store the array

- for large graphs (e.g. number of vertices $\geq 2^{10}$), we do not have enough space to store an adjacency table!

From now on, we shall assume that *all* our graphs will be modeled using *adjacency lists*.

## 3 Traversing Graphs

### 3.1 Methods

Many graph algorithms require us to visit *all* nodes of the graph e.g. in determining what nodes are *path-accessible* from a given node or *searching* for an appropriate node

There are *two* main ways in which we may traverse graphs:

- depth-first traversal
- breadth-first traversal

#### 3.1.1 Depth-First Traversal

The idea of a *depth-first traversal* is to follow a chain of edges as far as we can into the graph (i.e. we go as deep as the rabbit hole allows).

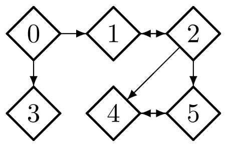
When we can follow the edges no more (i.e. we can go no deeper), we backtrack to find previously visited nodes out of which other edge chains (i.e. paths) may lead.

To avoid following the same path more than once, it is imperative that we *mark nodes* as visited when we reach them.

Thus, each node needs a *flag* associated with it[1].

We will illustrate how the *depth-first traversal* strategy works using the following example:

**Example:** Consider the graph:



which may be represented by the following adjacency list:

| | | |
|---|---|---|
| **0** | false | $\{1,3\}$ |
| **1** | false | $\{2\}$ |
| **2** | false | $\{1,4,5\}$ |
| **3** | false | $\emptyset$ |
| **4** | false | $\{5\}$ |
| **5** | false | $\{4\}$ |

[1]In the following example we can check a node's status by checking the "traversal" list, but this need not generally be so.

Notice how we have included a column to record when each node has been visited.

In addition to this data structure, we shall also maintain a traversal list and a backtrack list.

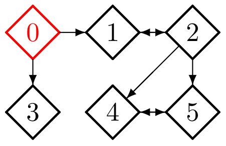
The traversal list is initially empty.

However, when our algorithm ends, the traversal list will contain the *actual* traversal (i.e. the list of nodes) that was followed with our depth-first traversal.

The backtrack list is initially empty.

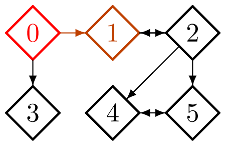
It is used during the algorithm to determine where we have to backtrack to.

- Assume that we start at node 0 — so mark node 0 as visited:



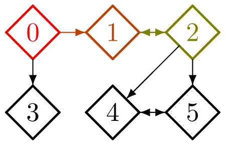
| | | |
|---|---|---|
| **0** | *true* | $\{1,3\}$ |
| **1** | false | $\{2\}$ |
| **2** | false | $\{1,4,5\}$ |
| **3** | false | $\emptyset$ |
| **4** | false | $\{5\}$ |
| **5** | false | $\{4\}$ |

- Traversal list: [0]
- Backtrack list: [0] (If you implement the traversal recursively you get the backtrack list “for free”.)
- Node 1 is accessible from node 0 and has not yet been visited:



| | | |
|---|---|---|
| **0** | *true* | $\{1,3\}$ |
| **1** | *true* | $\{2\}$ |
| **2** | false | $\{1,4,5\}$ |
| **3** | false | $\emptyset$ |
| **4** | false | $\{5\}$ |
| **5** | false | $\{4\}$ |

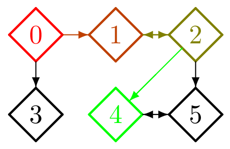
- Traversal list: $[0,1]$
- Backtrack list: $[0,1]$
- Node 2 is accessible from node 1 and has not yet been visited:



| | | |
|---|---|---|
| **0** | *true* | $\{1,3\}$ |
| **1** | *true* | $\{2\}$ |
| **2** | *true* | $\{1,4,5\}$ |
| **3** | false | $\emptyset$ |
| **4** | false | $\{5\}$ |
| **5** | false | $\{4\}$ |

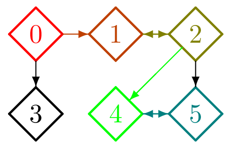
- Traversal list: $[0,1,2]$

- Backtrack list: $[0, 1, 2]$
- The next node to be visited is node 4 — node 1 is accessible but it has already been visited



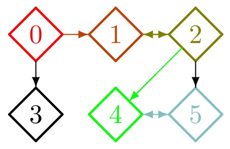
| | | |
|---|---|---|
| **0** | *true* | $\{1, 3\}$ |
| **1** | *true* | $\{2\}$ |
| **2** | *true* | $\{1, 4, 5\}$ |
| **3** | false | $\emptyset$ |
| **4** | *true* | $\{5\}$ |
| **5** | false | $\{4\}$ |

- Traversal list: $[0, 1, 2, 4]$
- Backtrack list: $[0, 1, 2, 4]$
- Node 5 is accessible from node 4 and has not yet been visited:



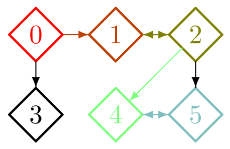
| | | |
|---|---|---|
| **0** | *true* | $\{1, 3\}$ |
| **1** | *true* | $\{2\}$ |
| **2** | *true* | $\{1, 4, 5\}$ |
| **3** | false | $\emptyset$ |
| **4** | *true* | $\{5\}$ |
| **5** | *true* | $\{4\}$ |

- Traversal list: $[0, 1, 2, 4, 5]$
- Backtrack list: $[0, 1, 2, 4, 5]$
- Backtrack to node 4 since there are no more nodes to visit from node 5



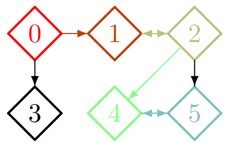
| | | |
|---|---|---|
| **0** | *true* | $\{1, 3\}$ |
| **1** | *true* | $\{2\}$ |
| **2** | *true* | $\{1, 4, 5\}$ |
| **3** | false | $\emptyset$ |
| **4** | *true* | $\{5\}$ |
| **5** | *true* | $\{4\}$ |

- Traversal list: $[0, 1, 2, 4, 5, 4]$
- Backtrack list: $[0, 1, 2, 4]$
- Here we also need to backtrack. Backtrack to node 2.



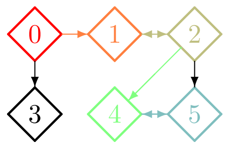
| | | |
|---|---|---|
| **0** | *true* | $\{1, 3\}$ |
| **1** | *true* | $\{2\}$ |
| **2** | *true* | $\{1, 4, 5\}$ |
| **3** | false | $\emptyset$ |
| **4** | *true* | $\{5\}$ |
| **5** | *true* | $\{4\}$ |

- Traversal list: $[0, 1, 2, 4, 5, 4, 2]$
- Backtrack list: $[0, 1, 2]$
- Nodes 1, 4 and 5 all visited so backtrack to node 1



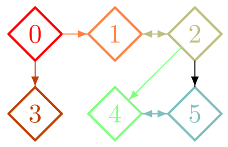
| | | |
|---|---|---|
| **0** | *true* | $\{1,3\}$ |
| **1** | *true* | $\{2\}$ |
| **2** | *true* | $\{1,4,5\}$ |
| **3** | false | $\emptyset$ |
| **4** | *true* | $\{5\}$ |
| **5** | *true* | $\{4\}$ |

- Traversal list: $[0, 1, 2, 4, 5, 4, 2, 1]$
- Backtrack list: $[0, 1]$
- Again no new nodes to be visited so backtrack to node 0



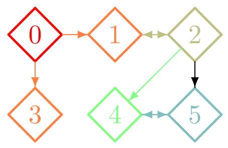
| | | |
|---|---|---|
| **0** | *true* | $\{1,3\}$ |
| **1** | *true* | $\{2\}$ |
| **2** | *true* | $\{1,4,5\}$ |
| **3** | false | $\emptyset$ |
| **4** | *true* | $\{5\}$ |
| **5** | *true* | $\{4\}$ |

- Traversal list: $[0, 1, 2, 4, 5, 4, 2, 1, 0]$
- Backtrack list: $[0]$
- Node 3 is accessible from node 0 and has not yet been visited:



| | | |
|---|---|---|
| **0** | *true* | $\{1,3\}$ |
| **1** | *true* | $\{2\}$ |
| **2** | *true* | $\{1,4,5\}$ |
| **3** | *true* | $\emptyset$ |
| **4** | *true* | $\{5\}$ |
| **5** | *true* | $\{4\}$ |

- Traversal list: $[0, 1, 2, 4, 5, 4, 2, 1, 0, 3]$
- Backtrack list: $[0, 3]$
- No nodes are accessible from node 3, so we backtrack to node 0:



| | | |
|---|---|---|
| **0** | *true* | $\{1,3\}$ |
| **1** | *true* | $\{2\}$ |
| **2** | *true* | $\{1,4,5\}$ |
| **3** | *true* | $\emptyset$ |
| **4** | *true* | $\{5\}$ |
| **5** | *true* | $\{4\}$ |

- Traversal list: $[0, 1, 2, 4, 5, 4, 2, 1, 0, 3, 0]$
- Backtrack list: $[0]$
- Nodes 1 and 3 are accessible from node 0, but they have both been visited.

  They are no other nodes to consider at this level so we attempt to backtrack.

  In doing this, our backtrack list becomes empty.

  We now search through our array looking for a node that has not yet been visited. No such node exists so our algorithm ends.

Thus, we see that a *depth-first traversal* for this graph is:

$$[0, 1, 2, 4, 5, 4, 2, 1, 0, 3, 0]$$

or, ignoring revisits:

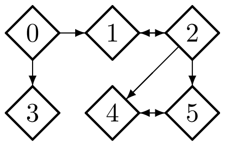
$$[0, 1, 2, 4, 5, 3]$$

### 3.1.2 Breadth-First Traversal

The idea of the *breadth-first traversal* is to first visit the *immediate successors* of a given vertex, and then to successively look at each of their immediate successors (which have not yet been visited)[2].

Again, this algorithm is best illustrated using the following example:

**Example:** Consider the graph:



which may be represented by the following adjacency list:

| | | |
|---|---|---|
| **0** | false | $\{1, 3\}$ |
| **1** | false | $\{2\}$ |
| **2** | false | $\{1, 4, 5\}$ |
| **3** | false | $\emptyset$ |
| **4** | false | $\{5\}$ |
| **5** | false | $\{4\}$ |

Again, notice how we have included a column to record when each node has been visited.

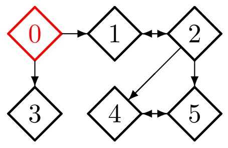
Also, in addition to this data structure, we shall also maintain a traversal list and a to do list.

[2]Again we use a "visited" flag, although that is not strictly necessary if we maintain a "traversal" list.

The traversal list is initially empty. However, when our algorithm ends, the traversal list will contain the actual traversal (i.e. the list of nodes) that was followed with our breadth-first traversal.

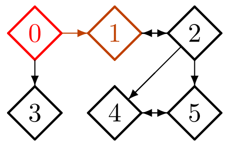
The to do list is initially empty. It is used during the algorithm to determine which nodes we have yet to visit.

- We shall assume that we are to start at node 0, thus it is marked as visited:



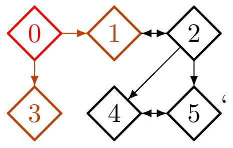
| | | |
|---|---|---|
| **0** | *true* | $\{1, 3\}$ |
| **1** | false | $\{2\}$ |
| **2** | false | $\{1, 4, 5\}$ |
| **3** | false | $\emptyset$ |
| **4** | false | $\{5\}$ |
| **5** | false | $\{4\}$ |

- Traversal list: $[0]$
- To do list: $[1, 3]$ (You have to implement the to do list yourself.)
- Node 1 is at the *current head* of our *to do list* and has not yet been visited



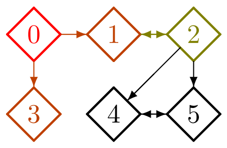
| | | |
|---|---|---|
| **0** | *true* | $\{1, 3\}$ |
| **1** | *true* | $\{2\}$ |
| **2** | false | $\{1, 4, 5\}$ |
| **3** | false | $\emptyset$ |
| **4** | false | $\{5\}$ |
| **5** | false | $\{4\}$ |

- Traversal list: $[0, 1]$
- To do list: $[3, 2]$
- Node 3 is at the *current head* of our *to do list* and has not yet been visited



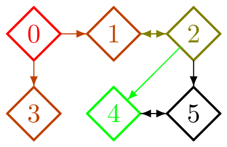
| | | |
|---|---|---|
| **0** | *true* | $\{1, 3\}$ |
| **1** | *true* | $\{2\}$ |
| **2** | false | $\{1, 4, 5\}$ |
| **3** | *true* | $\emptyset$ |
| **4** | false | $\{5\}$ |
| **5** | false | $\{4\}$ |

- Traversal list: $[0, 1, 3]$
- To do list: $[2]$
- Node 2 is at the *current head* of our *to do list* and has not yet been visited



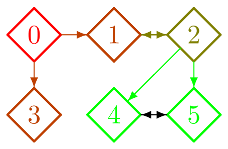
| | | |
|---|---|---|
| **0** | *true* | $\{1,3\}$ |
| **1** | *true* | $\{2\}$ |
| **2** | *true* | $\{1,4,5\}$ |
| **3** | *true* | $\emptyset$ |
| **4** | false | $\{5\}$ |
| **5** | false | $\{4\}$ |

- Traversal list: $[0, 1, 3, 2]$
- To do list: $[4, 5]$ (do not add 1)
- Node 4 is at the *current head* of our *to do list* and has yet to be visited. Node 5 is a neighbour of node 4 and has not yet been visited, but is *due* for a visit (is in the "to do" list), so we do not add it to the "to do" list again.



| | | |
|---|---|---|
| **0** | *true* | $\{1,3\}$ |
| **1** | *true* | $\{2\}$ |
| **2** | *true* | $\{1,4,5\}$ |
| **3** | *true* | $\emptyset$ |
| **4** | *true* | $\{5\}$ |
| **5** | false | $\{4\}$ |

- Traversal list: $[0, 1, 3, 2, 4]$
- To do list: $[5]$
- Node 5 is at the *current head* of our *to do list* and has yet to be visited



| | | |
|---|---|---|
| **0** | *true* | $\{1,3\}$ |
| **1** | *true* | $\{2\}$ |
| **2** | *true* | $\{1,4,5\}$ |
| **3** | *true* | $\emptyset$ |
| **4** | *true* | $\{5\}$ |
| **5** | *true* | $\{4\}$ |

- Traversal list: $[0, 1, 3, 2, 4, 5]$
- To do list: $[]$
- Our *to do list* is empty.

  We now search through our nodes looking for one that has yet to be visited.

  No such nodes exist, so our algorithm ends.

Thus, we see that a *breadth-first traversal* for this graph is:

$$[0, 1, 3, 2, 4, 5]$$

## 3.2 Implementation

### 3.2.1 Breadth First

**Visit to a Single Node** First we define a method that encapsulates the actions required to visit a single node. This method will

- check that the node has not already been visited
- note that this node has been visited
- add any unvisited children of this node to the "to do" list

**Has the node been visited?** Maintain a set of visited nodes

```
// if we have already visited or are due to visit this node
// there is nothing to do
if (isOnVisitList(node)) return;
```

**Visit node**

```
// add this node to the traversal
traversal.add(node);
```

**Add unvisited successors to the "to do" list**

```
// add successors to the "to do" list
for (T next:  getNeighbours(node)) {
   if (!isOnVisitList(next)) {// unless they have been visited
      toDo.add(next);
   }
}
```

**Full Version**

```
private void visitNode(T node) {
   if (isOnVisitList(node)) return;
   traversal.add(node);
   for (T next:  getNeighbours(node)) {
      if (!isOnVisitList(next)) {
         toDo.add(next);
      }
   }
}
```

Note: this visits a single node. We need to use it to visit all the nodes in the "to do" list.

**Visit all the "To Do" Nodes** While there are nodes in the "to do" list we need to:

- get the next node from the "to do" list;
- visit that node.

**Visit all the "to do" nodes**

```
private void traverseToDoList() {
   while (toDo.size() > 0) {
      T node = toDo.remove();
      visitNode(node);
   }
}
```

We need to start this method off by "seeding" the "to do" list with an unvisited node. The method above will then visit all the nodes in the graph if all nodes are reachable from the first node selected. Otherwise we need to repeatedly select a new unvisited node until none are left.

**The Complete Traversal**

```
private Queue<T> toDo = new ArrayDeque<T>();
private List<T> traversal = new ArrayList<T>();

public List<T> traverse() {
   T next = getUnvisitedNode();
   while (next != null) {
      toDo.add(next);
      traverseToDoList();
      next = getUnvisitedNode();
   }
   return traversal;
}
```

**Finding the next unvisited node**

```
protected T getUnvisitedNode() {
   for (T node:  getNodes()) {
      if (!isOnVisitList(node)) {
         return node;
      }
   }
   return null;
}
```

### 3.2.2 Depth First

This one's even easier. A visit to a node should:

- check whether the node has already been visited
- visit the node
- visit all the node's descendants

Implementation: left as an exercise

End of graphs lecture