CIS 575. Introduction to Algorithm Analysis
Material for February 2, 2024

Graph Representations

The topic of this note is covered in *Cormen*'s Section 20.1.

1 Graph Operations

We shall now discuss how to **represent** a graph. A representation should allow a number of operations, in particular GET and PUT and DELETE:

- a call GET(i, j) will check if there is an edge from node i to node j, and if so, return any extra data. Such a call will thus return a boolean for a graph where edges do not carry data; if edges do carry data, it would be natural to return a value of `option` type (as may be known from say OCaml as seen in CIS505/705).

- a call PUT(i, j, x) will add an edge from node i to node j (for an undirected graph that will also add an edge from j to i), together with extra data x (which may be null).

  We shall assume that PUT does not check if there already is an edge, and simply overwrites any edge that may exist; if the user wants some other behavior (such as keeping the old edge or reporting an error) this can be accommodated by taking appropriate action after first calling GET.

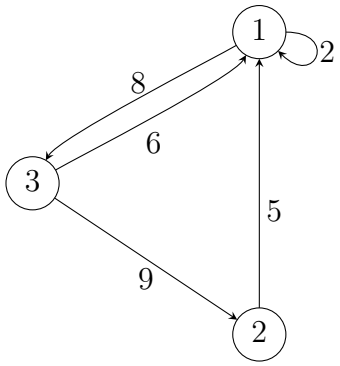
- a call DELETE(i, j) will remove the edge that may exist from i to j but takes no action if no such edge exists.

It is also very convenient to have an operator ALLFROM such that

- a call ALLFROM(i) returns a *list* of the edges with source i (for an undirected graph, this would be the edges with i as one of their two end points).

Similarly, we could provide an ALLTO operator but shall seldom do so.

It turns out that there are two ways to represent a graph, to be presented in the following. Both shall be illustrated by the example directed graph



2 Adjacency Matrix Representation

When a graph is represented by an **adjacency matrix**, the matrix entry at row i and column j specifies whether there is an edge from i to j, and if so also contains any extra data. For our example graph, we would have the matrix (where missing entries could be represented as null values, depending on the language used)

	1	2	3
1	2		8
2	5		
3	6	9	

Running Times Obviously, the operators GET, PUT and DELETE all run in constant time, that is, $\Theta(1)$.

But ALLFROM will always run in time $\mathbf{\Theta(n)}$ since *all* entries in the given row have to be examined. This will be the case even when the list eventually returned is very short.

3 Adjacency Lists Representation

Alternatively, a graph may be represented by **adjacency lists** which to each node i associates a list of the nodes j (with extra data) such that there is an edge from i to j.

For our example graph, we get:

1. the adjacency list for node 1 contains two elements: node 1 itself (with data 2) and node 3 (with data 8)
2. the adjacency list for node 2 contains only one element: node 1 (with data 5)
3. the adjacency list for node 3 contains two elements: node 1 (with data 6) and node 2 (with data 9).

Running Times Some operations run in constant time:

- PUT runs in $\Theta(1)$ since a new node (plus data) can just be inserted in front of the current list (recall we don't check if the edge already exists)

- AllFrom runs in $\Theta(1)$ since it just returns a pointer to the adjacency list (but to *process* that list may take substantial time).

On the other hand, Get and Delete may need to examine all elements of the adjacency list, and in the worst case they thus run in time $\Theta(n)$ (strictly speaking, if duplicates have been inserted the length of the adjacency list may be longer than n, but let us ignore this).

4 Assessment

We have seen that each of the two representations have their advantages and their disadvantages.

The main **drawback of adjacency lists** comes when a graph is **dense** since then Get (and Delete) will take linear time.

The main **drawback of an adjacency matrix** comes when a graph is **sparse** since then AllFrom takes linear time even though it may only take constant time to process the resulting list (and also, the matrix then occupies much more space than necessary).