Solutions, Class 03/14, Topic 14A: Graph Representations

Efficient Graph Transposition

The transpose of a directed graph G = (V, E) is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u)$ such that $(u, v) \in E\}$. In other words, G^T is the graph G with all of its edges reversed.

Algorithms for computing transpose by *copying* to a new graph are trivial to write under matrix and adjacency list representations. However, we will be working with very large graphs and want to avoid copies. Can we transpose a graph "in place" without making a copy, and minimizing the amount of extra space needed. What is the time and space cost to do so?

(In the following, you are writing algorithms that work "inside" the ADT: you are allowed to access the matrix and list representations directly. Examples of representations are on the next page.)

1. (1pt) Describe an efficient algorithm for the **edge list** representation of G for computing G^T from G in place (don't make a copy). Do this by modifying the list. Analyze the space and time requirements of your algorithm.

> **Solution:** Since an edge-list representation is a sequence of edges ⟨ ... (i, j) ... ⟩, we can access all the edges sequentially in O(E) time. For each edge, we use a temporary variable to swap i and j, producing (j, i). This takes O(1) space. This is the simplest and fastest way to compute a transpose, but edge-lists are not a good representation for many of the other operations we need to do.

2. (2 pts) Describe an efficient algorithm for the **adjacency matrix** representation of G for computing G^T from G in place (don't make a copy). Do this by modifying the matrix. Analyze the space and time requirements of your algorithm.

> **Solution:**
> Flipping the bits (0 to 1 and 1 to 0) will not work: this makes the complement graph.
> The solution is to to swap entries A[i,j] and A[j,i] for every pair of (i,j). We must be careful not to do it twice for each pair (therefore, leaving the matrix unchanged).

To achieve that we just need to iterate for every element below the main diagonal of the matrix and swap each element with the corresponding element above the main diagonal:

```
for i = 2 to n {
     for j = 1 to i-1 {
          temp = A[i,j];
          A[i,j] = A[j,i];
          A[j,i] = temp;
     }
}
```

Note that j is always less than i, so if we have processed this with A[k,l] as the cell assigned to temp we will never inadvertently reverse it with A[l,k] as cell assigned to temp. Also, no cell A[i,i] is processed as that would be pointless. The loops can also be written with i = 1 to n-1 and j = i+1 to n (*What does that correspond to in our high level description of the solution?*).

Runtime is $O((V^2-V)/2) = O(V^2)$ time, and requires O(1) additional space (just one temp variable).

3. (2 pts) Describe an efficient algorithm for computing G^T from G using the **adjacency list** representation of G, using as little extra space as possible. Can you do it "in place"? Analyze the space and time requirements of your algorithm.

Solution: Adjacency lists are space efficient for large sparse graphs, but they achieve this at the cost of requiring sequential access to the vertices adjacent to a given edge. One can't go directly to a data item saying whether vertices i and j are adjacent: one must search the adjacency list for i (and also for j when dealing with undirected graphs). Thus, we cannot efficiently use the swap procedure discussed above for Matrix.

In fact, all of the solutions we can think of require nontrivial additional space. Let's consider some alternatives:

A. Copying the graph while reversing edges.
This requires O(V+E) additional space and O(V+E) time (iterating over the vertices and over the edges, and copying each in reverse during the iteration). For a sparse graph, O(V+E) is close to O(V), while for a dense graph it is closer

to $O(V^2)$.

B. Reversing the edges in the adjacency lists.
The basic idea is to iterate over each adjacency list, deleting edge (i, j) from i's adjacency list when processing that list, and adding the reversed version (j, i) to j's adjacency list. The deletion is accomplished by keeping track of the previous node in the list so it is easy to splice around the current node in O(1) time (this is only necessary because some nodes may be new reversed ones) and writing a special case for processing the first node. The adding is accomplished in O(1) time simply by adding the reversed edge to the front of j's list.

We need a way to mark edges as to whether they have been reversed, so that if we encounter a linked list node for edge (i, j) on i's adjacency list and put a linked list node (j, i) on j's adjacency list we don't then reverse it back when processing j's adjacency list. Two ways to do this are:

- Using the annotations facility on edges, annotate each edge once it is processed under some unique key (e.g., the system time stamp at the time the algorithm started). Then, don't reverse edges that are annotated with this key. We should pay the cost of clearing out these annotations so as not to leave memory "garbage" behind.
- Using a hash table, put each reversed edge in the hash table (that is, the new edges we create, not the old ones). Before reversing an edge on j's list, check whether it in the hash table. If it is, don't reverse it. When the algorithm is done, the hash table can be dereferenced for garbage collection.

Both of these approaches require O(V+E) time (to check each vertex's adjacency list and to process the edges found) and O(E) additional memory (either to annotate the edges or to store references to them in hash tables). Since O(E) can range from O(V) in sparse graphs to $O(V^2)$ in dense graphs, this has only a constant memory advantage over copying, but the code is more complex.

C. A hybrid solution:
Use plan A, copy the graph, but reuse the list cells as we copy each edge to the new graph. Alternatively, delete the old edges from the original graph as they are copied to the new one so that they can be garbage collected as we proceed: we create O(E) new edge objects, but we also free up edge objects at the same rate, so their memory can be reclaimed if needed. Then we will never need any more

than O(V) additional storage. When done, we can either return a new graph, or make each vertex in the original graph point to the new adjacency lists and dispense with the second vertex list in order to modify the original graph in place. Time cost is the same as copying.

The code for plan B is much more complex than a simple copy, and there are no real space advantages, so plan C may be the best choice.

4. (2 pts extra credit). Now describe a way to compute G^T from G under the **adjacency matrix** representation without modifying the matrix, and by using only one bit of extra storage! Analyze space and time requirements. *(Hint: take advantage of the ADT abstraction layer.)*

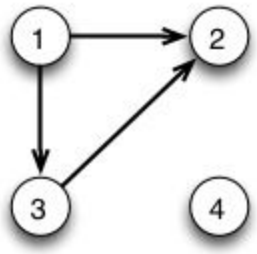
Solution: The one extra bit will indicate whether the graph is transposed or not.

Initially the bit is set to 0. The transposeGraph() operation sets it to the complement of the current value (i.e. 1 if it was 0, and 0 if it was 1). Then we need to modify all the other Graph methods so that it checks the bit and if it is set to 1, it accesses the graph as if it had been transposed, by reversing the source and target as appropriate. For example, a call to getArc(i,j) would actually get arc (j,i); inAdjacentVertices(v) would return out-adjacent vertices and outAdjacentVertices(v) would return in-adjacent vertices; Origin and Destination would be flipped, and similarly for the mutators, taking care to add or remove arcs in the reverse direction. When transposeGraph() is called again, the bit is set to 0 and the normal versions of the methods are used.

This requires one bit of extra storage for the graph which is O(1). It is O(1) to transpose and the run times of the other methods are not affected beyond constant factor overhead to check the bit value.

Given this solution, why would one want to modify the graph as in question 2? The code becomes more complex to maintain, so it is worth it only if transpose is a common operation. Why not use the same approach with adjacency lists? Without random access to (i, j) the "flipped" operations would become much more expensive.

Examples:



Formal Specification

G = (V, E)
V = {1, 2, 3, 4}
E = {(1, 2), (1, 3), (3, 2)}

Matrix Representation

(Rows and columns are indexed by vertex number.)

	1	2	3	4
1	0	1	1	0
2	0	0	0	0
3	0	1	0	0
4	0	0	0	0

(Fastest to test whether (i, j) in E, but slower to get all adjacent vertices. Not space efficient for sparse graphs.)

Adjacency List Representation

(Left hand column is an array indexed by vertices and containing pointers to linked lists.)

```
1 → v2 → v3 → null
2 → null
3 → v2 → null
4 → null
```

(Fastest to get list of adjacent vertices; time to test whether (i, j) in E is proportional to average degree, usually OK in sparse graphs. Space efficient.)

Edge List Representation

```
(1, 2)
(1, 3)
(3, 2)
```

(Fastest to iterate over all edges but very costly to find all the edges out of a vertex. No way to represent unconnected vertices or any vertex attributes: vertices are not first class objects. Space efficient.)