Solutions, 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 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 the 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.

Solutions: 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 x and y are adjacent: one must search for y on the adjacency list for x.

All of the solutions we are aware of require nontrivial additional space, but we can minimize this space. Let's consider some alternatives, starting with the naive solution:

A. Copying the graph while reversing edges.
This solution is the default solution we are trying to avoid: making an entirely new graph instance. Make a new vertex array, and then iterate over the vertex array of the original graph and over the adjacency list for each vertex to copy the edges in reverse. Specifically if a linked list cell referencing vertex y is found on the adjacency list for vertex x, we make a linked list cell referencing vertex x and place it at the front of the adjacency list for vertex y. This is conceptually simple, but assuming we also keep the original graph requires O(V+E) additional space and O(V+E) time. For a sparse graph, O(V+E) is close to O(V), while for a dense graph it is closer to $O(V^2)$.

If we did not want to keep the original graph, we might 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. However, garbage collection takes time that is difficult to analyze without details of the programming language implementation. Yet this gives us the idea of managing the reuse of memory ourselves:

B. Copying while reusing linked list cells.
Use plan A, copy the graph, but as we copy each edge to the new graph, delete the linked list cell for y (found on x's list) from the original graph and re-use that cell when creating a linked list cell for x (to be put on y's list). Then we will never need any more than O(V) additional storage for the second vertex array. 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 array in order to modify the original graph in place. Time cost is the same as copying.

Other solutions have been proposed to avoid making the second O(V) vertex array:

C. Using existing adjacency lists with marked list cells.
The basic idea is the same as B in which we reuse the O(E) list cells, but instead of constructing new lists off a second O(V) array, we construct those lists within the original linked lists using the original O(V) array. However, here we run into a problem similar to the one we saw in problem 2 with matrix representations: We need a way to keep track of which linked list cells represent original edges and which represent edges that have been reversed, so that if we encounter a linked list node for vertex y on x's adjacency list and put a linked list node for x on y's adjacency list, we don't later move it back when processing y's adjacency list. (Note that unlike with the matrix this error would not necessarily restore the graph to its original state: y may or may not actually be processed after x.)

One way to do this is to "mark" linked list cells. We could add an extra bit to each linked list cell, set to 0 in the original and 1 when reversed. Then, when traversing an adjacency list we only reverse (remove and reuse on a new list) those that are set to 0, and then set the bit to 1. Students have also proposed using a hash table to keep track of reversed edges (what would be the key?) But both of these solutions require O(E) extra space. O(E) can range from O(V) in sparse graphs to O(V^2) in dense graphs, so any list cell marking approach is potentially worse than plan B above, depending on the graph.

D. Tail pointers to separate existing and new adjacency lists.
There is another approach to keeping track of reversed edges that requires only O(V) additional space: a tail pointer for each adjacency list. This solution was first proposed to us by students. This solution assumes that you have a tail pointer to the end of each

original linked list. (It would cost O(E) time to set them up if they are not already present.) The tail pointers will mark the boundaries between the old and new lists, serving simultaneously as the sentinel for the end of the old list and the head of the new list. Similarly to Plan C, for each reversal, remove the node from the list you found it in (preserving the remaining list), and add the node for the reversed edge on the appropriate list. Different from Plan C, add the nde *after the node the tail pointer points to*. When processing a linked list to reverse its target nodes, *stop when you reach the same cell as the tail pointer*: the remaining linked list will be the reversed edges. (At this point, the tail pointer is the head pointer for the new list, and we no longer have a tail pointer for the new list unless we also maintained that during the above process.)

Of all the solutions presented here, Plan B (a second O(V) vertex array) and Plan D (O(V) tail pointers) are the most space efficient solutions for adjacency lists. Plan D requires additional code for careful pointer manipulation, while Plan B uses standard list manipulation methods. I would recommend Plan B as the code will be easier to write and maintain, but appreciate the creativity of students in thinking of Plan D.

4. (Challenge Problem). 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 for the entire graph!!! Analyze space and time requirements. *(Hint: take advantage of the ADT abstraction layer.)*

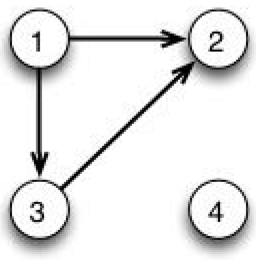
Solution: The one extra bit will indicate whether the graph is transposed or not.

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 they check the bit and if it is set to 1, they access 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.)

$$
\begin{array}{ccccc}
 & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} \\
\mathbf{1} & 0 & 1 & 1 & 0 \\
\mathbf{2} & 0 & 0 & 0 & 0 \\
\mathbf{3} & 0 & 1 & 0 & 0 \\
\mathbf{4} & 0 & 0 & 0 & 0
\end{array}
$$

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. The order of the lists is *not* significant: they are representing sets.

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

Fastest to get a 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)
```

Edges are listed in no particular order, and are accessed sequentially. 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.