Homework 2 Solutions

Question 1

(a) Pseudocode for adding an edge:

```
Add-Edge(u, v, w):
    Node *tmp = new Node(v, w);
    if m_head[u] != NULL:
        tmp->m_next = m_head[u];
    m_head[u] = tmp;
```

(b) For the AL implementation: m_head has n entries and there are a total of $2m$ Node objects. Each Node object stores three values (v, w, m_next). Therefore, the total storage is $n + 6m$.

For the CSR implementation: m_re has $n + 1$ entries. m_nz and m_ci both have $2m$ entries. Therefore, the total storage is $n + 1 + 4m = n + 4m + 1$.

Thus the memory usage for AL is *slightly* higher. Using asymptotic notation, the storage requirement is $O(n + m)$ for both methods. [**Note:** not all sections have covered asymptotic notation.]

(c) Edge insertion is faster for AL. It consists of a small number of constant-time operations. CSR requires looping to find the insertion

point *and* shifting elements of the arrays to the right. Put together, these take *nnz* loop iterations where *nnz* is the number of non-zero elements at the time of the insertion. Thus edge insertion in CSR is linear in *nnz*.

(d) Some reasons why one might prefer CSR: slightly smaller memory requirement, better cache utilization [**Note:** only Sections 01 and 05 covered cache utilization], if we know the number of edges in advance, we can avoid resizing the arrays.

Some reasons one might prefer AL: faster insertion time (as long as order within each row doesn't matter), avoids resizing and shifting required by CSR.

Question 2

(a) Possible implementation of matrix-vector multiplication:

```
void Graph::matVec(int *x, int *y) {
    for (int r = 0; r < m_numVert; r++) {
        y[r] = 0;
        for (int indx = m_re[r]; indx < m_re[r+1]; indx++) {
            y[r] += m_nz[indx] * x[m_ci[indx]];
        }
    }
}
```

(b) It wouldn't change at all.

Question 3

Solutions 1

Let *A* be the input array of length *n*. Create a second array *B* of length *n*–1 and initialize it to all zeros. Loop over the elements of *A*. For each element *x*, if *B*[*x*] > 0, then *x* is the repeated value; otherwise, increment *B*[*x*].

```
Find-Repeat(A):
    n = A.length
    B is a new array of length n-1
    for i = 0 to n-2:
        B[i] = 0
    for i = 0 to n-1:
        x = A[i]
        if B[x] > 0:
            return x
        else:
            B[x] = B[x] + 1
    return Nil
```

Note: *Nil* is a special value that in this case means "no repeat found."

This algorithm requires a single loop of length *n*–1 to initialze *B*, and a second loop of length *n* to find the repeated value. The loop bodies

both consist of constant-time operations. Since the entire algorithm has fewer than 2*n* loop iterations, the running time is linear.

Solution 2

We know the array contains the numbers 0, 1, 2, …, *n*–2 *and* one additional number in the range 0 to *n* - 2; call the additional number *x*. Therefore, the sum of the numbers in the array will be

```
0 + 1 + 2 + ... + n-2 + x = (n-2)(n-1)/2 + x
```

Since we know *n*, we can compute *t* = (*n*–2)(*n*–1)/2 directly in O(1) time. Now, loop over the array, computing the sum *s* of all the entries, which is done in O(*n*) time. Then *x* = *s* - *t*, an O(1) computation, and we have found the repeated value *x*. The total running time is O(1) + O(*n*) + O(1) = O(*n*).

Extra Credit

The first solution most people try is the following:

1. 1 & 2 go to Princeton [2 hours]
2. 1 returns to BWI with the train passes [1 hour]
3. 1 & 3 go to Princeton [5 hours]
4. 1 returns with the train passes [1 hour]
5. 1 & 4 go to Princeton [10 hours]

The total travel time is 19 hours. The key to reducting the total time is to find a way to have 3 & 4 travel to Princeton together:

1. 1 & 2 go to Princeton [2 hours]
2. 1 returns to BWI with the train passes [1 hour]
3. 3 & 4 go to Princeton [10 hours]
4. 2 returns to BWI with the train passes [2 hours]
5. 1 & 2 go to Princeton [2 hours]

The total time for this solution is 17 hours.