HW9 Solutions

May 10, 2001

27.1-6 Skew Symmetry: If $f_1(u,v) = -f_1(v,u)$ and $f_2(u,v) = -f_2(v,u)$, then $f_1(u,v) + f_2(u,v) = -f_1(v,u) - f_2(v,u)$, thus $(f_1 + f_2)(u,v) = -(f_1 + f_2)(v,u)$.

Flow conservation: Since the identities of the source and sink are identical in f_1 and f_2, the set $V - s, t$ is the same in f_1 and f_2. If $\sum_{x \in V} f_1(u,x) = 0$ and $\sum_{x \in V} f_2(u,x) = 0$, then $\sum_{x \in V} f_1(u,x) + f_2(u,x) = 0$, and $\sum_{x \in V} (f_1 + f_2)(u,x) = 0$.

Capacity constraint: $f_1(u,v) \leq c(u,v)$ and $f_2(u,v) \leq c(u,v)$ do **not** imply that $(f_1 + f_2)(u,v) \leq c(u,v)$. Simple counterexample: $f_1(u,v) = f_2(u,v) = c(u,v)$, but $(f_1 + f_2)(u,v) > c(u,v)$.

In short, flow conservation and skew symmetry hold, but the capacity constraint does not.

27.2-2 See separate .jpg file.

27.2-3 The minimum cut divides V into $\{s, v_1, v_2, v_4\}$ and $\{v_4, t\}$. It is easy to verify your answer because the max-flow min-cut theorem requires the capacity across the minimum cut to be equal to 23! The augmenting paths in (c) and (d) cancel flow. The augmenting path in (c) cancels 7 units of flow from v_2 to v_1. The augmenting path in (d) cancels 4 units of flow from v_3 to v_2.

27.2-7 To show that f_p is a flow in G_f, we need to check that it satisfies the three properties of flows:

Capacity constraint:

If (u,v) is on p, $f_p(u,v) = c_f(p) \leq c_f(u,v)$
If (v,u) is on p, $f_p(u,v) = -c_f(p) \leq 0 \leq c_f(u,v)$
If neither (u,v) nor (v,u) is on p, $f_p(u,v) = 0 \leq c_f(u,v)$

Skew symmetry:

If (u,v) is on p, $f_p(u,v) = c_f(p), f_p(v,u) = -c_f(p) \Rightarrow f_p(u,v) = -f_p(v,u)$
If (v,u) is on p, $f_p(u,v) = -c_f(p)$, $f_p(v,u) = c_f(p) \Rightarrow f_p(u,v) = -f_p(v,u)$
if neither (u,v) nor (v,u) is on p, then $f_p(u,v) = f_p(v,u) = 0 \Rightarrow f_p(u,v) = -f_p(v,u)$

Flow conservation:

From the definition of augmenting path, p is a simple path from s to t. So, for all $u \in V - \{s,t\}$, if u is on p, then there exist v_1, v_2 such that (v_1, u) and (u, v_2) are on p. Moreover, for all

$v \neq v_1, v_2$, $f_p(u,v) = 0$. Thus, $\sum_{v \in V} f_p(u,v) = f_p(u,v_1) + f_p(u,v_2) = -c_f(p) + c_f(p) = 0$ If u is not on p, then for all $v \in V, f_p(u,v) = 0$. Thus, $\sum_{v \in V} f_p(u,v) = 0$

Since p is a simple path, s has only one successor in p, say, u So (s,u) is on p and $\forall v \neq u, f_p(s,v) = 0$. By definition, $|f_p| = \sum_{v \in V} f_p(s,v) = f_p(s,u) = c_f(p) > 0$ (since $\forall (u,v) \in E_f, c_f(u,v) > 0$).

Grading: 1 point for each flow property, 1 point for proving $|f_p| > 0$

16.1-1 $m[i,j]$ table:

0	150	330	405	1655	2010
	0	360	330	2430	1950
		0	180	930	1770
			0	3000	1860
				0	1500
					0

$s[i,j]$ table:

1	2	2	4	2
	2	2	2	2
		3	4	4
			4	4
				5

Thus, the optimal parenthesization is $(A_1A_2)((A_3A_4)(A_5A_6))$.

Grading: 1 point for the correct parenthesization, 4 points for the correct content of the tables, 0.5 point off for each error in the table.

16-3 $c[i,j]$ records the minimum cost of changing $x[i..m]$ into $y[j..n]$, and $op[i,j]$ records the corresponding operation used to achieve this minimum cost. We compute them with a dynamic programming algorithm; $c[1,1]$ is returned as the edit distance from x to y. We can print out the optimal transformation sequence using the matrix op.

```
Edit_Distance(x, y, c, op)
    m = length(x); n = length(y);
    c[m+1, n+1] = 0;
    for i = m → 1
        c[i, n+1] = min( cost(kill), cost(delete) + c[i+1, n+1] );
        op[i, n+1] = the corresponding operation
    for j = n → 1
        c[m+1, j] = cost(insert) + c[m+1, j+1];
        op[m+1, j] = inxsert;
    for i = m → 1
```

```
    for j = n → 1
        if (x_i = y_j)
            c1 = cost(copy) + c[i+1, j+1];
        else c1 = ∞;
        if (i < m && j < n && x_i = y_{j+1} && x_{i+1} = y_j)
            c2 = cost(twiddle) + c[i+2, j+2];
        else c1 = ∞;
        c3 = cost(replace) + c[i+1, j+1];
        c4 = cost(insert) + c[i, j+1];
        c5 = cost(delete) + c[i+1, j];
        c[i, j] = min(c1, c2, c3, c4, c5);
        op[i, j] = the corresponding operation
return c[1, 1];
```

```
Print_Trans_Seq(op, x, y)
    i = 1; j = 1;
    while ( i ≤ m && j ≤ n)
    switch (op[i,j])
        case copy: PRINT("copy" + x[i]); i++; j++;
        case replace: PRINT("replace" + x[i] + "by" + y[j]); i++; j++;
        case insert: PRINT("insert" + y[j]); j++;
        case delete: PRINT("delete" + x[i]); i++;
        case twiddle: PRINT("twiddle" + x[i] + x[i+1] + "into" + x[i+1] + x[i]); i += 2; j += 2;
        case kill: PRINT("kill"); i = m + 1;
```

Both running time and space requirements are $O(mn)$.

Grading: The problem ask you to find the edit distance **AND** print an optimal transformation sequence. 2.5 points off for not including the print-out part. 1 point for time and space analysis.

17.2-1 Assume the n items are labeled $1..n$ in nonincreasing order of v_i/w_i. A solution containing t_i pounds of item i is denoted as $\{v_1..v_n\}$. Suppose with $n-1$ items we can get optimal solution according to greedy algorithm, then with n items, with greedy strategy, we should take $v = min(w_1, W)$ pounds of item 1. Suppose there's an optimal solution with $v_1 < v$ which has a total value $m = \sum_i v_i * t_i/w_i$. Then $\exists j, \sum_{i=2}^{j} v_i \geq v - v_1$, and we can exchange $v - v_1$ pounds of item $2..j$ with item 1, without decreasing m. The resulting solution contains $min(w_1, W)$ of item 1, and will still be an optimal solution.

17.2-2

```
Knapsack(v, w, W)
file://calculating c[i, j]
    n = length(v);
    for j = 0 to W
        c[0, j] = 0;
    for i = 1 to n
        c[i, 0] = 0;
        for j = 1 to W
            if(w_i <= j)
                v = c[i-1, j - w_i] + v_i;
                if (v > c[i-1, j])
                    c[i, j] = v;
                else c[i, j] = c[i-1, j];
file://printint out the items to take
    j = W; i = n;
    while(j > 0)
        while(c[i, j] = c[i-1, j])
            i = i - 1;
        PRINT(i);
        j = j - w_i;
        i = i - 1;
```

Grading: 2 points off for only calculating the matrix c[i, j] without giving the algorithm to decide what items should be taken.

17.2-4 This can be solved with greedy strategy: Only stop at the most distant gas station that can be reached with the available gas. This yields an optimal solution. Suppose the gas stations along the way are numbered as $1...n$ and the greedy strategy chooses to stop at $i_1...i_p$. Suppose there's an optimal solution that stops at $j_1..j_q$. We can prove by induction that $i_k \geq j_k$.

The base case is obvious from the greedy strategy. Now suppose $i_k \geq j_k$. Since j_{k+1} is reachable from j_k, then it's certainly reachable from i_k, and since the greedy strategy chooses the most distant reachable gas station, we have $i_{k+1} \geq j_{k+1}$. Now if $p > q$, since we have $i_q \geq j_q$ and the destination is reachable from j_q, it's also reachable from i_q, we wouldn't have stopped at $i_{q+1}...i_p$, contradiction. So $p \leq q$, and we have an optimal solution using a greedy strategy.

36.1-3 Here is a very simple encoding for adjacency matrices: Create a string of $|V|^2$ bits. If there is an edge from i to j, set bit $i|V| + j$ to 1, otherwise set it to 0. $O(|V|^2)$ time is needed to create this representation. To get back to a graph from this representation, read the string once to determine its length. Take the square root of the length, and call it n. Read through the string again, and for each 1 at location i, "place an edge" from $\lfloor \frac{i}{n} \rfloor$ to $i \bmod \mathrm{n}$. Presuming that placing an edge takes $O(1)$, which is only reasonable, this works in $O(|V|^2)$, polynomial.

There were a few very ingenious and correct encodings given for adjacency lists. The biggest problem in designing an encoding is ensuring that the encoding is one-to-one, that is, no encoding can represent multiple graphs. This was also the biggest cause for loss of credit. This solution is a variation of one given by a student.

Let 00 represent 0, let 01 represent 1, and let 10 represent NEXT, and let 11 represent STOP. Let e be a function taking a binary number and converting it to this representation.

Number the vertices, and move through the in numerical order. For each vertex u: For each vertex v adjacent to u, append "$e(v)$ NEXT" to the string. When all vertices adjacent to u have been appended, remove the last NEXT and append STOP.

So a graph with vertices $\{0, 1, 2\}$ and edges $\{(0, 1), (0, 2), (2, 1)\}$, the representation (grouped by two bits for ease of reading) would be 00 01 10 01 00 11 11 00 01 11. (Undoing function e, this would be 01 NEXT 10 STOP STOP 01 STOP.)

To get the graph back, first read through, two by two, and count the number of STOPs found. This is the number of vertices. Start a counter i at 0. Read again from the beginning - now when you encounter a NEXT, decode the string since the last special code, and add an edge from i to the decoded vertex number. When you encounter STOP, increment the counter and continue. This requires three readings of each code, and there will be $|V|$ STOPs, $|E|$ NEXTs, and $|E| \lg |\mathrm{V}|$ other codes. $O(1)$ work is done decoding each code, and $O(1)$ work is done adding an edge, so this is clearly polynomial in terms of $|V|$ and $|E|$; creating this representation has similar time requirements.

Since one can get compose either representation or get a graph back from either representation in polynomial time, the representations are polynomially related. I.e., to get the matrix from the list, decode the list and re-encode as a matrix.

Credit was given this way: 1 point for matrix representation, 3 points for list representation, and 2 for proving their mutual convertibility in polynomial time.