CS 255: Homework 2

Due: March 18, at 9 pm

Instructions

1. You may work in groups of up to 3 students.
2. Each group makes one submission through Canvas.
3. Include the names and SJSU ids for each student in the group.

Problem 1

The Floyd-Warshall algorithm computes shortest paths between all pairs of vertices in a weighted graph. Give pseudocode for an efficient multithreaded implementation of the Floyd-Warshall algorithm. (See section 25.2 of CLRS for more details on the algorithm). What is the work, span, and parallelism of your algorithm?

Solution. Let $n = |V|$ and $w(i,j)$ be the weight of the edge between vertices i and j. We want to compute $dist[1 \ldots n][1 \ldots n]$, for all pairs of vertices i and j. Essentially, we just parallelize the two inner most loops of the standard single processor algorithm. A quick, somewhat lazy implementation is as follows.

```
Floyd-Warshall($V, E$):
    $n \leftarrow |V|$
    parallel for $i \leftarrow 1$ to $n$ do
        parallel for $j \leftarrow i$ to $n$ do
            if $i = j$ then
                $dist[i][j] \leftarrow 0$
            else if $(i, j)$ in $E$ then
                $dist[i][j] \leftarrow w(i, j)$
            else
                $dist[i][j] \leftarrow \infty$
    for $k \leftarrow 1$ to $n$ do
        parallel for $i \leftarrow 1$ to $n$ do
            parallel for $j \leftarrow 1$ to $n$ do
                $dist[i][j] \leftarrow \min(dist[i][j],\ dist[i][k] + dist[k][j])$
```

We have the follow metrics.

- Work is $T_1(n) = \Theta(n^3)$.
- The span of the nested parallel loops is $\Theta(\log n)$. Overall, with the outer for loop, the span is $T_\infty(n) = \Theta(n \log n)$.
- Parallelism is $T_1(n)/T_\infty(n) = \Theta(n^2/\log n)$.

The above is enough for full credit, but it does not have the best performance in practice. For a constant number of processors p, we can distribute the work more evenly between processors by breaking the matrix *dist* into p 'equal' sized blocks. Assuming $\sqrt{p}$ and $n/\sqrt{p}$ are integers, we create p square blocks of size length $s = n/\sqrt{p}$ (each block contains n^2/p entires). We can parallel nested for loops over the blocks, then run the standard single processor algorithm within a block.

```
Floyd-Warshall($V, E$):
    $n \leftarrow |V|$
    $s \leftarrow n/\sqrt{p}$
    initialize $dist$ as above
    for $k \leftarrow 1$ to $n$ do
        parallel for $r \leftarrow 0$ to $\sqrt{p} - 1$ do
            parallel for $c \leftarrow 0$ to $\sqrt{p} - 1$ do
                // standard update within a block
                for $i \leftarrow r * s$ to $(r + 1) * s - 1$ do
                    for $j \leftarrow c * s$ to $(c + 1) * s - 1$ do
                        $dist[i][j] \leftarrow \min(dist[i][j],\ dist[i][k] + dist[k][j])$
```

The nested parallel for loops spawn $\sqrt{p} \times \sqrt{p} = p$ threads, one for each block. The final nested for loops are just the standard algorithm's update within a given block. The two inner most loops run in $O(s \times s) = O(n^2/p)$ time.

- Work is $O(n \times \sqrt{p} \times \sqrt{p} \times n/\sqrt{p} \times n/\sqrt{p}) = O(n^3)$.
- With p processors the time to execute the parallel for loops for a fixed value of k is: $O(2 \log \sqrt{p} + n^2/p) = O(\log p + n^2/p)$. Overall k, runtime is $T_p(n) = O(n \times (\log p + n^2/p)) = O(n^3/p)$, since p is a constant. Therefore, the speedup is $T_1/T_p = O(p)$. That is, the algorithm runs about p times faster if we have p processors.
- To compute the span T_∞, observe that the minimum side length of a block, s, is 1. Since $s = n/\sqrt{p}$, this happens when $p \geq n^2$. Then, we don't really benefit from having more than n^2 processors. Substituting $p = n^2$ into the analysis of $T_p(n)$ yields, $T_\infty(n) = O(n \times (\log \sqrt{p} + n^2/p)) = O(n \log n)$.

Problem 2

Let $G = (V, E)$ be a flow network with integer edge capacities. Suppose f is a maximum flow in G. Describe algorithms for the following operations:

(a) Increment(e): Increase the capacity of edge e by 1 and update the maximum flow.

(b) Decrement(e): Decrease the capacity of edge e by 1 and update the maximum flow.

Clearly, you could just recompute a maximum flow, but we want to do this faster. Your algorithms should run in $O(m + n)$ time, where $m = |E|$, and $n = |V|$.

Solution:
(a) Intuitively, increasing the capacity of a single edge e should not increase the maximum flow value by more than one. The max-flow min-cut theorem provides a simple way to prove this. The

value of maximum flow equals the capacity of the minimum cut. By incrementing (or decrementing) the capacity of a single edge by one, the capacity of minimum cut increases (or decreases) by no more than one.

Suppose we increase the capacity of edge e by 1. Can we increase the maximum flow value? Running a single iteration of Ford-Fulkerson provides the answer. If there is an s-t path in the residual graph G_f, we augment along the path to increase the maximum flow by one. Otherwise, the current flow is optimal. One iteration of Ford-Fulkerson takes $O(m+n)$ time (it is basically just BFS or DFS in G_f starting from s).

(b.) By the argument of part (a), the maximum flow value either stays the same or decreases by one. There are two cases to consider, we reduce the capacity of a saturated edge $f_e = c_e$, or an unsaturated edge $f_e < c_e$.

It is easy to verify that decreasing the capacity of any unsaturated edge does not require any changes to the existing flow. Further, it cannot add any new s-t paths in G_f. Therefore, f remains a maximum flow.

If the edge e is saturated $f_e = c_e$, then we need to find an s-t path (or a cycle) with positive flow that uses e, and remove one unit of flow from it. We can find such a path or cycle as follows. Let edge $e = u \to v$, for some vertices u and v. First, find a path from s to u using BFS or DFS on edges that have positive flow. Next, traverse the edge $u \to v$. Finally, starting from v take any edge with positive flow until we reach t (forming a s to t path) or vertex w that is on the path from s to u (forming a cycle). It is easy to verify that removing one unit of flow from each edge on the path (or cyce) gives a feasible flow. That is, capacity and flow conservation constraints still hold. After removing one unit of flow from this path (cycle) we can reduce the capacity of edge e by 1. This step runs in $O(n+m)$ time. Are we done? Not quite. Removing flow from this path changes the residual graph G_f. We still need to check if there are s-t paths in G_f and augment along any such path. This step is easy. Just run one iteration of Ford-Fulkerson. After at most one iteration there will be no s-t paths in G_f and we will have a maximum flow. This entire process runs in $O(n+m)$ time.

Problem 3

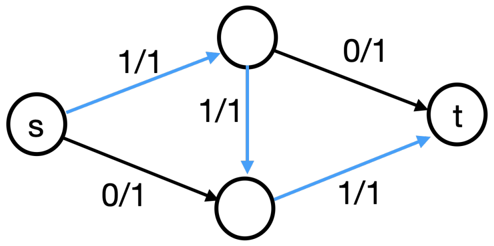
Your friend suggests a greedy modification to the Ford-Fulkerson algorithm. Instead of maintaining the entire residual graph, just use the forward edges showing the residual capacity $c_e - f_e$ of each edge in the graph G for the current flow f. Notice that once we saturate an edge we can just remove it, no reverse edges!

(a) Show that your friend's algorithm is not guaranteed to compute a maximum flow.

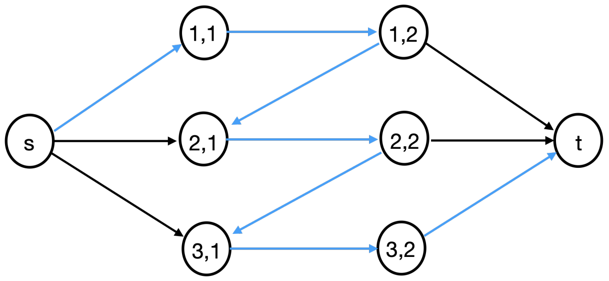
(b) Your friend sees the issue, but says not to worry; their greedy algorithm always produces a flow with a value within a constant fraction of the maximum flow value! Unfortunately, this is also wrong. Show that for any constant $\alpha > 1$ there is a flow network G such that the value of the maximum flow is more than α times the value of a flow computed by your friend's greedy algorithm.
[Hint: there are no restrictions on the augmenting paths you can choose, so pick the worst possible one.]

Solution:
(a.) The algorithm only using forward edges in the residual graph G_f is the same the naive greedy

approach discussed in lecture 3. We can use a similar graph (or the same one) as in lecture. In the figure below, edges are labeled (flow/capacity). The maximum flow value is 2 using the top and bottom s-t paths. The flow shown in blue has value 1, and blocks the greedy algorithm's progress.



(b.) Essentially, we want to extend the idea from part (a) to a graph with $\alpha > 1$, s-t paths. That is, there should be α distinct 'straight-line' s-t paths that can be blocked by single bad choice of flow. One way to achieve this is to have each 'straight-line' s-t path consist of 2 intermediate points. Call the left and right points of the ith path $(i, 1)$ and $(i, 2)$ respectively; refer to the figure below which shows 3 distinct paths: top, middle, and bottom. In the figure, all edges have unit capacity (omitted to remove clutter). Creating a directed edge from node $(i, 2)$ to node $(i + 1, 1)$ gives a way for flow on the ith path to block flow on path $i + 1$. By chaining this together of over $\alpha > 1$ 'straight-line' s-t paths, we ensure a single flow (shown in blue) blocks all other s-t paths. Thus, the maximum flow value is α, but the greedy algorithm can obtain a flow with value 1 with the wrong choice of augmenting path.



Problem 4

Suppose we are given an array $A[1 \ldots m][1 \ldots n]$ of non-negative real numbers. We want to *round* A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of the entries in any row or column of A. For example

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \to \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

Assume that the sum of entries in each row is an integer, and the sum of entries in each column is an integer. Describe and analyze an efficient algorithm to round a matrix A in this fashion using max-flow as a subroutine.

Solution. Let $a_{ij} = A[i][j]$ for $i \in 1 \ldots m$ and $j \in 1 \ldots n$. Without loss of generality, we can assume that $0 \leq a_{ij} < 1$. To see this, let B be the matrix given by $b_{ij} = a_{ij} - \lfloor a_{ij} \rfloor$. It is easy to check that if $\tilde{B}$ is a valid rounding of B, then $\tilde{A}$ given by $\tilde{a}_{ij} = \lfloor a_{ij} \rfloor + \tilde{b}_{ij}$ is a valid rounding of A.

We construct the following flow network. For each row $i \in 1 \ldots m$, we create a vertex R_i. For each column $j \in 1 \ldots n$, we create a vertex C_j. For each non-zero entry a_{ij}, we create an edge $R_i \to C_j$ with capacity 1. We add a source s and create edges $s \to R_i$ for each row i with capacity $c(s, R_i) = \sum_{j=1}^{n} a_{ij}$. We also add a sink t and create edges $C_j \to t$ for each column j with capacity $c(C_j, t) = \sum_{i=1}^{m} a_{ij}$. Note that $c(s, R_i)$ is the sum of the entries in row i, and $c(C_j, t)$ is the sum of entries in column j. Since the sum of entries in every row (and column) is an integer, all capacities are integers. Thus, there exists an integer valued max-flow.

We can interpret a flow in the network as follows. Initially, all entries of A are rounded down. The capacity $c(s, R_i)$ is the number of entries in row i that need to be increased by 1 (rounded up) to maintain the original row sum. Similarly, $c(C_j, t)$ is the number of entries column j that need to be rounded up. Any s to t path has the form $s \to R_i \to C_j \to t$. Since the edge $R_i \to C_j$ has a capacity of 1, at most one s to t path uses this edge. Further, in an integer valued max-flow, the flow on each edge $R_i \to C_j$ is either 0 or 1. Then, a positive flow on edge $R_i \to C_j$ corresponds to rounding up the entry a_{ij}.

It remains to verify that a valid rounding exists. The value of any flow is upper bounded by:

$$\text{value of } f \leq \sum_{i=1}^{m} c(s, R_i) = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij},$$

where the flow saturates all the edges leaving s. It is easy to check that a flow achieving this bound exists (although not necessarily integer valued). For each entry a_{ij}, create a path $s \to R_i \to C_j \to t$ with flow value a_{ij}. Let f be the sum of all such paths. Since $0 \leq a_{ij} < 1$, $c(s, R_i) =$ the sum of entries in row i, and $c(C_j, t) =$ the sum of entries in column j, then f satisfies capacity constraints. Further, since f is the sum of s to t paths, it satisfies flow conservation constraints. Thus, the value of the max-flow is $\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$, where the flow saturates all edges out of s. Note that such a flow must also saturate all edges into t.

Given an integer valued max-flow, if flow on the edge $R_i \to C_j > 0$, then we set $\lceil a_{ij} \rceil$; otherwise, set $\lfloor a_{ij} \rfloor$. This gives a valid rounding. For each row i, the max-flow saturates the edge $s \to R_i$. From flow conservation:

$$\sum_{j=1}^{n} a_{ij} = c(s, R_i) = \text{ flow into } R_i = \text{ flow out of } R_i = \sum_{j=1}^{n} f_{ij}.$$

Similarly, for each column j, the max-flow saturates the edge $C_j \to t$, so that:

$$\sum_{i=1}^{m} a_{ij} = c(C_j, t) = \text{ flow out of } C_j = \text{ flow into } C_j = \sum_{i=1}^{m} f_{ij}.$$

The number of vertices in the network is $|V| = n + m + 2 = O(\max(m, n))$, and the number of edges is $|E| = m + n + mn = O(mn)$. Since the max-flow value is equal to the sum of all the entries in A, and each entry is $0 \leq a_{ij} < 1$, then $f = O(mn)$. Ford-Fulkerson runs in $O(E \cdot f) = O(m^2 n^2)$ time. Finding all edges $R_i \to C_j$ takes $O(mn)$ time.