Problem 1. (a) As stated, if the directed graph of the neighborhood is Eulerian, then there exists an optimal route to solve the street cleaning problem. Let's assume that our given graph is not Eulerian, such that there are some vertices that do not have the same in and out degree. We want to copy in a set of edges to make the in and out degree of each vertex match, while minimizing the total cost of the copied edges (since the eventual solution will traverse these edges, and we want to minimize the total distance). Thus, for each vertex with indegree smaller than outdegree, we can copy the incoming edge with the least cost, multiple times if needed, until the degrees match, and likewise for each vertex with outdegree smaller than indegree, we can copy the outgoing edge with the least cost, multiple times if needed. Once this process is complete, we will have an Eulerian graph, and the route that traverses each edge once will have a minimized total cost, giving us an optimal solution for the augmented graph. To translate this back to the original graph, for each collection of identical edges that have resulted from copying, we simply traverse this edge multiple times in the original graph, therefore solving the street cleaning problem.

(b) We can use the result from part (a), which tells us that the street cleaning problem can be solved by finding the min-cost set of edges to copy. To determine which edges to copy, we can formulate a 2-iteration min-cost max-flow problem as follows:

1. For nodes with larger indegree:
   - Construct an auxiliary graph G' and for each vertex V in the original graph G with indegree larger than outdegree, create a corresponding vertex $V' \in G'$.
   - Create a source S and add an edge (S, V') with cost 0 and capacity indegree(V) − outdegree(V) for all $V' \in G'$.
   - Create another set of vertices V''.
   - For each V', create an edge (V', W'') for each outgoing edge of V to W in the original graph, with cost $d(V, W)$ and capacity ∞.
   - Create a sink T and add edges from all V'' to T with cost 0 and capacity ∞.

   Solve the min-cost max-flow problem and verify that the max-flow $f = \sum u(S, V')$. Then for each edge (V', W'') with non-zero flow, we will copy the edge (V, W) in the original graph a total of $f(V', W'')$ times.
2. For nodes with larger outdegree:

   Repeat the initialization as above for vertices V with outdegree larger than indegree, and set the capacity of (S, V') to be outdegree(V) − indegree(V). Then, follow the rest of the steps except for each V', create an edge (V', W'') for each incoming edge from W to V with cost $d(W, V)$ and capacity ∞. After solving for the min-cost max-flow, for each edge (V', W'') with non-zero flow, we will copy the edge (W, V) in the original graph $f(V', W'')$ times.

Thus, after two iterations of the min-cost max-flow problem, we will have determined the set of edges to copy, and we can use this information to solve the street cleaning problem as in part (a).

Problem 2. (a) Suppose we are given an optimal solution and we change the cost of some edge (i, j) by one. Since before the change, the solution was optimal, we will at most have one negative cost edge in the residual graph after the change, i.e. if we decrease $c(i, j)$ by 1, then the reverse edge (j, i) will have negative cost in the residual, and if we increase $c(i, j)$, then (i, j) will have negative cost. Then, we can do BFS starting from j to find a path to i such that if we augment it to (i, j), the total cost of this cycle is negative (we can do i to j in the other case). We can repeat this until the capacity of (i, j) is exhausted, at which point there are no more negative cost cycles in the graph. We are guaranteed that besides these cycles which involve (i, j), there are no other negative cycles in the graph since we had an optimal solution prior. Thus, after finding negative cycles in the graph, we can run max-flow to determine how much flow to augment through these cycles and obtain a new optimal solution. It takes $O(m)$ time to do the BFS, and $\tilde{O}(mn)$ time for the max-flow algorithm to run, so the total time for the re-optimization step is $\tilde{O}(mn)$.

(b) We can scale each edge cost down to 0 and proceed to scale up by doubling each edge cost and shifting in a new bit. Thus, there will be a total of $O(\log C)$ iterations. At each iteration, we can use the algorithm from part (a) to re-optimize the solution after shifting in a new bit, since each edge after doubling will change by at most one unit depending on the newest bit. Repeating this process for all m edges on each iteration, we have a total runtime of $O(m \log C)$. The correctness of the algorithm comes from the fact that doubling each edge does not alter the min-flow, only the magnitude, and therefore if we start with an optimal solution on the initial graph with all 0 edges, and re-optimize the solution on the shifted bits at each iteration, after each iteration the solution will still be optimal for that set of edge costs. Thus, upon the final shifting in, the algorithm will have scaled up to mirror the original problem exactly and produce the correct solution.

Problem 3. (a) In order to minimize cx, we want to maximize x_1 subject to the constraints $x_1, x_2, x_3 \geq 0$, $x_1 + x_2 \geq 1$, and $x_1 + 2x_2 \leq 3$. Rearranging the latter two inequalities, we get $1 - x_2 \leq x_1 \leq 3 - 2x_2$. We can maximize the right hand side by setting $x_2 = 0$, producing $1 \leq x_1 \leq 3$. Thus we can also set $x_1 = 3$, and since this is maximum possible value for x_1 under the constraints, it follows that the optimum value is $cx = -3$ and the set of optimal solutions takes the form $(3, 0, x_3)$ where x_3 can take any non-negative value (since the value of x_3 does not affect the other constraints nor does it impact the objective value).

(b) We want to minimize x_2 subject to the constraints $x_1, x_2, x_3 \geq 0$, $x_1 + x_2 \geq 1$, and $x_1 + 2x_2 \leq 3$. Since we want to minimize x_2, we can try setting it to the minimum possible value $x_2 = 0$, in which case the latter two inequalities become $x_1 \geq 1$ and $x_1 \leq 3$. Thus, the optimum value is $cx = 0$, and the set of optimal solutions takes the form $(x_1, 0, x_3)$, where $1 \leq x_1 \leq 3$ and $x_3 \geq 0$.

(c) We want to maximize x_3 subject to the constraints. There is only one constraint involving x_3, namely $x_3 \geq 0$. Since x_3 can be infinitely large, the optimum value is $cx = -\infty$, and the set of optimal solutions takes the form (x_1, x_2, ∞) where x_1 and x_2 are still subject to $x_1 + x_2 \geq 1$ and $x_1 + 2x_2 \leq 3$.

Problem 4. (a) We can formulate a linear program as follows: let x_i be the units of currency that we choose to trade with client i. Then for all i, $x_i \leq u_i$. In addition, for any given currency C, we can obtain (through borrowing and trading) at most $\sum_{b_i=C} r_i x_i$ units of C, which is therefore the upper bound to the amount of C that we use to exchange for other currencies; in other words, $\sum_{a_i=C} x_i \leq \sum_{b_i=C} r_i x_i$ (with the exception of $C_d = dollars$, since we begin with D dollars, which we add to the right hand of the constraint). Lastly, we want to maximize the amount of Yen we are left with at the end of the day, which is given by $\sum_{b_i=C_y} r_i x_i - \sum_{a_i=C_y} x_i$ (i.e. the amount we trade for minus the amount we must trade away).

Thus, our linear program is:

$$\begin{aligned}
\max \quad & \sum_{b_i=C_y} r_i x_i - \sum_{a_i=C_y} x_i & & \\
\text{subject to} \quad & x_i \geq 0 & & \forall i \\
& x_i \leq u_i & & \forall i \\
& \sum_{a_i=C} x_i \leq \sum_{b_i=C} r_i x_i & & \forall C \neq C_d \\
& \sum_{a_i=C_d} x_i \leq \sum_{b_i=C_d} r_i x_i + D & &
\end{aligned}$$

(b) It is possible (and optimal) to never borrow currency when carrying out trades to solve the linear program, because borrowing currency implies that we must pay it back by the end of the day, which is only possible if a) we never spend it, in which case there is no point in borrowing, or b) if we execute a series of trades $C_i \to \cdots \to C_k \to \cdots \to C_i$. As stated in the problem, any such cycle of trades generates a net profit $\prod r_i < 1$, which means that trading cyclically results strictly in a loss, in which case we should not engage in the cyclic trade at all. Thus, the optimal solution does not ever borrow currency in order to avoid losing currency in cyclic trades, which are necessary for paying back the borrowed currencies at the end of the day.

(c) Let Y be the optimum amount of Yen from part (a). Envisioning each yen as coming from some sequence of trades $C_d \to \cdots \to C_k \to \cdots \to C_y$, we can work backwards and calculate exactly what is required in order to fulfill the next trade in the sequence and only trade that amount, such that we eventually end up with the same amount of yen as the optimal solution (in other words, we only trade what is necessary, so no intermediate currencies between dollar and yen are left). We know that each sequence is acyclic from part (b) — or rather, we know that if there was some cycle of trades, it would be suboptimal — and thus each unit of yen in the optimal solution must have come from such a direct, acyclic sequence of trades that we can calculate exactly so as to not leave any intermediary currencies remaining in our portfolio. Thus, we will be able to end the day with the optimum amount of Yen, and have no currencies other than perhaps some unused dollars.

Problem 5. (a) The constraint $\sum_{j \in N(i)} x_{ij} = 1$ constrains n vertices on the left side, and the constraint $\sum_{i \in N(j)} x_{ij} = 1$ also constrains n vertices on the right side. Thus, there are a total of $2n$ constraints. However, since $\sum_i \sum_{j \in N(i)} x_{ij} = \sum_j \sum_{i \in N(j)} x_{ij}$ (due to the fact that the outgoing flow from the left side must equal the incoming flow of the right side), it follows that these $2n$ constraints are linearly dependent, and there can be at most $2n - 1$ linearly independent constraints. Since a vertex M consists of m entries representing the edges, and each of the linear dependencies corresponds to a 0 element, it follows that M will have at most $m - 2n + 1$ entries equal to 0.

(b) From part (a), we know that there are at most $2n-1$ non-zero entries in the $n \times n$ matrix M. Since M is double stochastic, each row and column sum must equal 1, and it follows that for each of the n rows of M, that row must contain at least one non-zero entry. If we distribute the remaining $n - 1$ non-zero entries among the n rows, there will be at least one row that did not get an additional non-zero entry, in which case there is only one non-zero entry in that row. However, the constraint is that the sum of each row must equal 1, and therefore that entry must be equal to 1. A similar reasoning applies to the columns, and we conclude that for any M representing a vertex, M must have at least one row and one column with all zero entries except for a single 1.

(c) We can show via backwards induction that M is a convex combination of several perfect matchings M_i as follows: from M, take one non-zero element from each row and one non-zero element from each column, and create an integer double stochastic matrix M_1 with 1's in the corresponding positions (corresponding to a perfect matching). Then subtract $M - \lambda_1 M_1$, where λ_1 is the smallest element in M. Since $\lambda_1 M_1$ is still a doubly stochastic matrix, it has constant row and column sums, so subtracting it from M preserves the invariant of constant row and column sums for $M - \lambda_1 M_1$ as well. Repeating this process of forming an integral double stochastic matrix M_i from the remaining elements of M and multiplying it by the smallest remaining element λ_i, we see that we essentially get rid of one element in each iteration, and therefore we are left with a decomposition of M into a convex combination of perfect matchings $\lambda_1 M_1 + \lambda_2 M_2 + \cdots + \lambda_k M_k$, where $\sum \lambda_i = 1$ since we know that the row-wise sums of M must be equal to 1 and likewise for the columns.

To show that the rate λ is sufficient for the internet switch to deliver all the traffic, we recall that at each time step, the switch can only transfer a set of packets that form a matching. Using the result above, we can indeed see that a combination of matchings $\lambda_1 M_1 + \lambda_2 M_2 + \cdots + \lambda_m M_m$ can form the complete graph M of the packets. This shows that the rate λ is sufficient for delivering all the packets, since there are a total of k matchings for k time steps, and each requires λ, which is satisfied.