[QUANT] Manage your Stock Inventory

locked

Problem Submissions Leaderboard Discussions

On a daily basis you are broker facilitating buy and sell of stocks for different hedge funds to support their trading strategies. As a result you are required to manage the stock movements across different accounts.

Primary function is to deliver stocks to the clients based on their needs, any excess stocks that is present can be pledged in "triparty" accounts to be able to raise cash which can be used for client requirements.

Each movement of stock between accounts has a transaction cost associated with it. Lesser the number of movements lesser is the transaction cost. Additionaly, each account belongs to a parent account. Aggregated transaction cost of the movement within the accounts of same parent account is always less than the transaction cost of the movement between accounts of different parent account, and hence should be always prefered. Another way to put this is cost of any number of transactions within the accounts of single parent account is less than cost of single transaction between 2 accounts that belong to different parent entity

Given the daily trading activity, device an alogirthm that will be able to effectively use the stocks to make sure the client demands are met on priority and maximizing the cash value of the asset, with minimum transaction cost possible. (Each row in the output is a transaction/movement.)

Input Format

First line is the number of stocks(n) and the next (n) lines are of format (*stock Id,price*) where *stock Id* is the stock identifier(string) and *price* is the stock price in $USD.

2

P1,2.5

P2,1.25

Post above lines, you are given the number of accounts between which stocks can be transferred(n) and the next (n) lines are of format (*account Id,account type,parent account*) where *account Id* is the account identifier(string), *account type* is the reference data indicating if it is a custody account(CUSTODY) or a triparty account(TRIPARTY) which will give cash in exchange of excess stocks, the third number *parent account* will represent the parent account representing which parent account does the account belong to.

3

Loc1,CUSTODY,1

Loc2,CUSTODY,2

Loc3,TRIPARTY,2

Post above lines, you are given number of lines(n) to read for eligible accounts per stock(indicating in which account can a particular stock be held), followed by (n) lines of format (*stock Id, account Id*), a stock will have multiple eligible accounts therefore multiple rows for a stock.

6

P1,Loc1

P1,Loc2

P1,Loc3

P2,Loc1

P2,Loc2

P2,Loc3

Post above lines, you are given number of lines(n) to read for eligible flows(directed) per stock(indicating the movement between the accounts that is allowed for a stock), followed by (n) lines of format (*stock Id, source account Id, destination account Id*), which indicates the stock(*stock Id*) can move from *source account Id* to *destination account Id*, a stock will have multiple eligible flows therefore multiple rows for a stock.

7

P1,Loc1,Loc2

P1,Loc1,Loc3

P1,Loc2,Loc1

P1,Loc2,Loc3

P2,Loc1,Loc2

P2,Loc2,Loc3

P2,Loc3,Loc1

Post above lines, you are given number of lines(n) to read for balances of each stock per account followed by (n) lines of format (*stock Id, account Id, quantity*), representing the stock *stock Id* currently has *quantity* number of shares in account *account Id*, if *quantity* positive number, it indicates that you have the stock excess and negative number is the demand of that stock that needs to be fulfilled.

3

P1,Loc1,10

P1,Loc2,-5

P2,Loc1,5

Constraints

all recommended quantity should be positive numbers only

Output Format

Output should be in the format (*stock Id,source acccount Id,destination account Id,quantity*) representing we need to move *quantity* shares of *stock Id* from account *source account Id* to *destination account Id*. The output rows should be sorted in ascending order of *Stock Id*, then *source account Id*, and *destination account Id* at last where each Id is a String.

P1,Loc1,Loc2,5

P1,Loc1,Loc3,5

P2,Loc1,Loc2,5

P2,Loc2,Loc3,5

Sample Input 0

```
2
P1,2.5
P2,1.25
```

```
3
Loc1,CUSTODY,1
Loc2,CUSTODY,2
Loc3,TRIPARTY,2
6
P1,Loc1
P1,Loc2
P1,Loc3
P2,Loc1
P2,Loc2
P2,Loc3
7
P1,Loc1,Loc2
P1,Loc1,Loc3
P1,Loc2,Loc1
P1,Loc2,Loc3
P2,Loc1,Loc2
P2,Loc2,Loc3
P2,Loc3,Loc1
3
P1,Loc1,10
P1,Loc2,-5
P2,Loc1,5
```

Sample Output 0

```
P1,Loc1,Loc2,5
P1,Loc1,Loc3,5
P2,Loc1,Loc2,5
P2,Loc2,Loc3,5
```

Submissions: 258
Max Score: 1000

Difficulty: Medium

Rate This Challenge:

More

Python 3

```python
import sys
import heapq
from collections import defaultdict
def read_input():

    # Read stock data
    n_stocks = int(input().strip())
    stocks = {}
    for _ in range(n_stocks):
        stock_id, price = input().strip().split(',')
        stocks[stock_id] = {'price': float(price)}

    # Read number of accounts
    n_accounts = int(input().strip())
    accounts = {}
    for _ in range(n_accounts):
        account_id, account_type, parent_account = input().strip().split(',')
        accounts[account_id] = {
            'type': account_type,
            'parent': parent_account
        }

    # Read eligible accounts per stock
    n_eligible_accounts = int(input().strip())
    eligible_accounts_per_stock = {}
    for _ in range(n_eligible_accounts):
```

```
        stock_id, account_id = input().strip().split(',')
        eligible_accounts_per_stock.setdefault(stock_id, set()).add(account_id)


    # Read eligible flows per stock
    n_eligible_flows = int(input().strip())
    eligible_flows_per_stock = {}
    for _ in range(n_eligible_flows):
        stock_id, source_account_id, dest_account_id = input().strip().split(',')
        eligible_flows_per_stock.setdefault(stock_id, []).append((source_account_id, dest_account_id))

    # Read balances
    n_balances = int(input().strip())
    balances = {}
    for _ in range(n_balances):
        stock_id, account_id, quantity = input().strip().split(',')
        quantity = int(quantity)
        balances.setdefault(stock_id, {})[account_id] = quantity

    return stocks, accounts, eligible_accounts_per_stock, eligible_flows_per_stock, balances, n_accounts


def process_stock(stock_id, stocks, accounts, eligible_accounts, eligible_flows, balances, n_accounts):
    account_info = accounts
    graph = {}
    P = n_accounts + 1
    base_intra = 1
    base_inter = 10
    incremental_cost = 0.5
    movements_count = defaultdict(int)  # Track number of transactions between accounts

    for source, dest in eligible_flows:
```

```
        # Only consider flows between accounts that are eligible to hold the stock
        if source not in eligible_accounts or dest not in eligible_accounts:
            continue

        if source not in graph:
            graph[source] = []
        # Determine cost
        if account_info[source]['parent'] == account_info[dest]['parent']:
            cost = base_intra + incremental_cost * movements_count[(source, dest)]
        else:
            cost = base_inter + incremental_cost * movements_count[(source, dest)]
        graph[source].append((dest, cost))

    # Build a list of sources (positive balances) and demands (negative balances)
    stock_balances = balances.get(stock_id, {})
    sources = {}
    demands = {}
    triparty_accounts = set()
    for account_id in eligible_accounts:
        balance = stock_balances.get(account_id, 0)
        if balance > 0:
            sources[account_id] = balance
        elif balance < 0:
            demands[account_id] = -balance  # We take the absolute value for demands
        # Identify triparty accounts
        if accounts[account_id]['type'] == 'TRIPARTY':
            triparty_accounts.add(account_id)

    movements = []

    # While there are demands
    while demands and sources:
        min_total_cost = float('inf')
        best_source = None
        best_demand = None
```

```
        best_path = None
        best_qty = 0

        # For each demand
        for demand_account, demand_qty in demands.items():
            # For each source
            for source_account, source_qty in sources.items():
                max_qty = min(source_qty, demand_qty)
                # Find shortest path from source to demand
                path, cost = dijkstra(graph, source_account, demand_account)
                if path is None:
                    continue
                # Check if the path stays within the same parent account
                crosses_parent = any(accounts[path[i]]['parent'] != accounts[path[i+1]]['parent']
for i in range(len(path)-1))
                if crosses_parent:
                    total_cost = cost + P  # Penalize paths crossing parent accounts
                else:
                    total_cost = cost
                if total_cost < min_total_cost:
                    min_total_cost = total_cost
                    best_source = source_account
                    best_demand = demand_account
                    best_path = path
                    best_qty = max_qty

        if best_path is None:
            # Cannot satisfy any more demands
            break

        # Generate movements along the best path
        qty = best_qty
        for i in range(len(best_path) - 1):
            src = best_path[i]
            dst = best_path[i+1]
```

```
            movements.append((stock_id, src, dst, qty))
        # Update balances
        sources[best_source] -= qty
        if sources[best_source] == 0:
            del sources[best_source]
        demands[best_demand] -= qty
        if demands[best_demand] == 0:
            del demands[best_demand]

    # Move excess stocks to triparty accounts
    for source_account in list(sources.keys()):
        source_qty = sources[source_account]
        while source_qty > 0:
            min_total_cost = float('inf')
            best_path = None
            best_triparty_account = None

            for triparty_account in triparty_accounts:
                path, cost = dijkstra(graph, source_account, triparty_account)
                if path is None:
                    continue
                # Check if the path crosses parent accounts
                crosses_parent = any(accounts[path[i]]['parent'] != accounts[path[i+1]]['parent']
for i in range(len(path)-1))
                if crosses_parent:
                    total_cost = cost + P  # Penalize paths crossing parent accounts
                else:
                    total_cost = cost
                if total_cost < min_total_cost:
                    min_total_cost = total_cost
                    best_path = path
                    best_triparty_account = triparty_account

            if best_path is None:
                # Cannot move to triparty account
```

```
                break

            # Generate movements along the best path
            qty = source_qty
            for i in range(len(best_path) - 1):
                src = best_path[i]
                dst = best_path[i+1]
                movements.append((stock_id, src, dst, qty))
            # Update balances
            source_qty -= qty
            sources[source_account] = source_qty
            if source_qty == 0:
                del sources[source_account]

    return movements

def dijkstra(graph, start, end):
    queue = []
    heapq.heappush(queue, (0, start, [start]))
    distances = {start: 0}
    while queue:
        (cost, node, path) = heapq.heappop(queue)
        if node == end:
            return path, cost
        for neighbor, edge_cost in graph.get(node, []):
            total_cost = cost + edge_cost
            if neighbor not in distances or total_cost < distances[neighbor]:
                distances[neighbor] = total_cost
                heapq.heappush(queue, (total_cost, neighbor, path + [neighbor]))
    return None, None

def main():
    stocks, accounts, eligible_accounts_per_stock, eligible_flows_per_stock, balances, n_accounts =
read_input()

```

```
    all_movements = []
    # Process stocks in order of decreasing price to maximize cash value when moving to triparty
accounts
    sorted_stocks = sorted(stocks.items(), key=lambda x: -x[1]['price'])
    for stock_item in sorted_stocks:
        stock_id = stock_item[0]
        movements = process_stock(
            stock_id,
            stocks,
            accounts,
            eligible_accounts_per_stock.get(stock_id, set()),
            eligible_flows_per_stock.get(stock_id, []),
            balances,
            n_accounts
        )
        all_movements.extend(movements)

    # Sort the movements
    all_movements.sort(key=lambda x: (x[0], x[1], x[2]))

    # Output the movements
    for movement in all_movements:
        stock_id, source, dest, qty = movement
        print(f"{stock_id},{source},{dest},{qty}")

main()
```

Line: 4 Col: 18

Upload Code as File Test against custom input Run Code Submit Code