#: 1D
Vertici: Stores
Archi (condizione): Condividono almeno M clienti
Peso arco: # clienti in comune
Input utente: M
Possibili domande:
• Trova store “ponte” (highest eigenvector centrality)
#: 2D
Tema: Selezione di clienti
Vincoli ricorsivi:
Nessun cliente condivide più di X prodotti con un altro già
scelto
Obiettivo da massimizzare: Fatturato totale generato

# ◇ 1D – “Rete di Store”

## Obiettivo

Costruire un grafo non-orientato i cui:

**Vertici** sono gli store.

**Archi** collegano due store se condividono almeno **M** clienti.

**Peso** = numero di clienti in comune.

### 1.1 Query SQL

**1. Vertici**

```
SELECT DISTINCT store_id
FROM orders;
```

(oppure: FROM stores se vuoi tutti gli store anche se non hanno ordini)

**2. Archi**

```
SELECT
  o1.store_id    AS s1,
  o2.store_id    AS s2,
  COUNT(DISTINCT o1.customer_id) AS shared_clients
FROM orders o1,
    orders o2
WHERE o1.customer_id = o2.customer_id
  AND o1.store_id <  o2.store_id         -- evita doppioni e self-loop
GROUP BY o1.store_id, o2.store_id
HAVING COUNT(DISTINCT o1.customer_id) >= %s;  -- parametro M
```

Passi (M,) come bind-param.

### 1.2 Build del grafo in Python

```
import networkx as nx

def build_store_graph(dao, M):
    """
    :param dao: oggetto con i metodi get_store_nodes() e get_store_edges(M)
    :param M: soglia minima di clienti condivisi
    :return: grafo non orientato con store come nodi e peso sugli archi
    """
    G = nx.Graph()
    # 1) aggiungi i vertici
    nodes = dao.get_store_nodes()       # lista di store_id
    G.add_nodes_from(nodes)
    # 2) aggiungi gli archi
    for s1, s2, w in dao.get_store_edges(M):
        G.add_edge(s1, s2, weight=w)
    return G
```

### 1.3 Domanda: “store ponte” (highest eigenvector centrality)

```
def find_bridge_store(G):
    """
    Restituisce lo store con centralità eigenvector massima.
    """
    centrality = nx.eigenvector_centrality(G, weight='weight')
    # prende la chiave con valore massimo
    return max(centrality, key=centrality.get), centrality
```

# ◇ 2D – “Selezione di clienti”

## Obiettivo

Scegliere un **insieme** di clienti che massimizza il **fatturato totale** generato,
con il **vincolo** che nessun cliente scelto condivida più di **X** prodotti con uno già presente nel set.

## 2.1 Preparazione dei dati

**1. Pre-calcola** per ogni cliente:

revenue[c] = Σ quantity*list_price*(1-discount) (DAO + Python)

**2. Pre-calcola** la matrice (o dict di dict)
shared[c1][c2] = # prodotti condivisi
con query SQL implicite su order_items+orders.

## 2.2 Codice Python in 3 funzioni

```
import copy

class ClientSelector:
    def __init__(self, shared_map, revenue_map, X, max_k):
        """
        :param shared_map: dict[c1][c2] -> # prodotti condivisi
        :param revenue_map: dict[c] -> revenue totale del cliente
        :param X: soglia massima di prodotti condivisi ammessa
        :param max_k: numero massimo di clienti da selezionare
        """
        self.shared = shared_map
        self.rev    = revenue_map
        self.X      = X
        self.K      = max_k
        self.best_set   = []
        self.best_score = 0.0

    def getBest(self, all_clients):
        """
        Funzione madre: inizializza la ricerca.
        :param all_clients: lista di tutti i customer_id disponibili
        """
        self.best_set = []
        self.best_score = 0.0
        # partire da ognuno come seed (o da un seed scelto)
        for c in all_clients:
            self._ricorsione(path=[c])
        return self.best_set, self.best_score

    def _ricorsione(self, path):
        """
        Backtracking: prova a estendere `path` con un nuovo cliente
        che non violi il vincolo shared<=X e non superi K.
        """
        score = self.getScore(path)
```

```
        # aggiorna soluzione migliore
        if score > self.best_score:
            self.best_score = score
            self.best_set   = path.copy()

        # pruning: se ho già K clienti, interrompo
        if len(path) >= self.K:
            return

        # prova a ciascun cliente non ancora in path
        for cand in self.shared:
            if cand in path: continue
            # verifica vincolo condivisi con TUTTI i già in path
            if all(self.shared[cand].get(p, 0) <= self.X for p in path):
                path.append(cand)
                self._ricorsione(path)
                path.pop()

    def getScore(self, path):
        """
        Obiettivo: somma delle revenue dei clienti nel path.
        """
        return sum(self.rev[c] for c in path)
```

### 2.3 Come usarlo

```
# 1) Dalla DAO prendi:
#    shared_map = dao.get_shared_products_map()
#    revenue_map = dao.get_customer_revenue()
#    all_clients = list(revenue_map.keys())

selector = ClientSelector(shared_map, revenue_map, X=5, max_k=10)
best_clients, best_revenue = selector.getBest(all_clients)
print("Clienti scelti:", best_clients)
print("Revenue totale:", best_revenue)
```