#: 1E
Vertici: Staff
Archi (condizione): Hanno gestito ordini dello stesso cliente
Peso arco: # clienti condivisi
Input utente: Store opzionale
Possibili domande:
• Staff più “collaborativo” (grado)
• Cammino più lungo tra due membri
#: 2E
Tema: Turno ideale di staff
Vincoli ricorsivi: Staff non può appartenere allo stesso store già in
turno
• peso = ordini gestiti
Obiettivo da massimizzare: Numero di ordini coperti

1E – “Rete di Staff collaborativo”

1E.1 Vertici: tutti gli staff (opzionalmente filtrati per store)

```
SELECT DISTINCT s.staff_id
FROM staff s,
    orders o
WHERE s.staff_id = o.staff_id
  AND o.store_id = COALESCE(%s, o.store_id);
```

Parametro: (store_id_or_NULL,)

Restituisce i staff_id che hanno gestito almeno un ordine (in quel store, se non NULL).

1E.2 Archi e peso: staff A–B con # clienti in comune

```
SELECT
   o1.staff_id      AS s1,
   o2.staff_id      AS s2,
   COUNT(DISTINCT o1.customer_id) AS weight
FROM orders o1,
    orders o2
WHERE o1.customer_id = o2.customer_id
  AND o1.staff_id   < o2.staff_id   -- evita (A,A) e duplicati (A,B)/(B,A)
  AND o1.store_id   = COALESCE(%s, o1.store_id)
GROUP BY o1.staff_id, o2.staff_id;
```

Parametro: (store_id_or_NULL,)

Ogni riga (s1,s2,weight) diventa un arco non orientato con weight = # clienti condivisi.

1E.3 Python: costruzione grafo e domande

```
import networkx as nx

def build_staff_graph(dao, store=None):
    G = nx.Graph()
    # 1) Nodi
    nodes = dao.get_staff_nodes(store)     # lista di staff_id
    G.add_nodes_from(nodes)
    # 2) Archi
    for s1, s2, w in dao.get_staff_edges(store):
        G.add_edge(s1, s2, weight=w)
    return G

def most_collaborative_staff(G):
    """
    Restituisce lo staff con il grado (degree) massimo.
    """
    deg = dict(G.degree())
    return max(deg, key=deg.get), deg

def longest_collaboration_path(G):
    """
    Restituisce il cammino più lungo (in termini di nodi)
    tra due staff, misurato sul più breve percorso.
    """
    # calcola tutte le distanze minime
    dists = dict(nx.all_pairs_shortest_path_length(G))
    best = (None, None, -1)
    for u, dd in dists.items():
        for v, dist in dd.items():
            if dist > best[2]:
                best = (u, v, dist)
    if best[0] is None:
        return [], 0
    path = nx.shortest_path(G, best[0], best[1])
    return path, best[2]
```

dao.get_staff_nodes(store) esegue la query 1E.1

dao.get_staff_edges(store) esegue la query 1E.2

most_collaborative_staff usa il grado del nodo per il “grado di collaborazione”.

longest_collaboration_path trova la coppia di nodi più “lontana”
e ne restituisce il cammino.

2E – “Turno ideale di staff”

Scegliere un sottoinsieme di staff che **non contenga due membri dello stesso store** e **massimizzi il totale di ordini gestiti**.

2E.1 Struttura dati

staff_store[s] → store_id di ogni staff

staff_orders[s] → numero totale di ordini gestiti da s

2E.2 Codice in 3 funzioni

```
import copy

class ShiftPlanner:
    def __init__(self, staff_store: dict[int,int], staff_orders: dict[int,int], max_k: int):
        """
        :param staff_store: mappa staff_id → store_id
        :param staff_orders: mappa staff_id → ordini gestiti
        :param max_k: numero massimo di staff nel turno (opzionale)
        """
        self.staff_store  = staff_store
        self.staff_orders = staff_orders
        self.max_k        = max_k
        self.best_team    = []
        self.best_score   = 0

    def getBest(self, all_staff: list[int]):
        """
        Funzione madre: prova ogni staff come seed e lancia il backtracking.
        """
        self.best_team  = []
        self.best_score = 0
        for s in all_staff:
            self._ricorsione([s])
        return self.best_team, self.best_score

    def _ricorsione(self, team: list[int]):
        """
```

```
        Backtracking:
        - non aggiunge due staff dello stesso store
        - rispetta max_k
        """
        score = self.getScore(team)
        if score > self.best_score:
            self.best_score = score
            self.best_team  = team.copy()

        if len(team) >= self.max_k:
            return

        used_stores = { self.staff_store[s] for s in team }
        for cand in self.staff_store:
            if cand in team:
                continue
            if self.staff_store[cand] in used_stores:
                continue
            team.append(cand)
            self._ricorsione(team)
            team.pop()

    def getScore(self, team: list[int]) -> int:
        """
        Punteggio = somma degli ordini gestiti dal team
        """
        return sum(self.staff_orders[s] for s in team)
```

2E.3 Uso pratico

```
# 1) DAO fornisce:
staff_store  = dao.get_staff_store_map()    # {staff_id: store_id}
staff_orders = dao.get_staff_orders_map()   # {staff_id: total_orders}
all_staff    = list(staff_store.keys())

planner = ShiftPlanner(staff_store, staff_orders, max_k=5)
team, covered = planner.getBest(all_staff)
print("Team ottimale:", team)
print("Ordini coperti:", covered)
```

1E – “Rete di Staff collaborativo”

1E.1 Vertici: tutti gli staff (opzionalmente filtrati per store)

```
SELECT DISTINCT s.staff_id
FROM staff s,
    orders o
```

```
WHERE s.staff_id = o.staff_id
  AND o.store_id = COALESCE(%s, o.store_id);
```

Parametro: (store_id_or_NULL,)

Restituisce i staff_id che hanno gestito almeno un ordine (in quel store, se non NULL).

1E.2 Archi e peso: staff A–B con # clienti in comune

```
SELECT
    o1.staff_id      AS s1,
    o2.staff_id      AS s2,
    COUNT(DISTINCT o1.customer_id) AS weight
FROM orders o1,
     orders o2
WHERE o1.customer_id = o2.customer_id
  AND o1.staff_id   < o2.staff_id   -- evita (A,A) e duplicati (A,B)/(B,A)
  AND o1.store_id   = COALESCE(%s, o1.store_id)
GROUP BY o1.staff_id, o2.staff_id;
```

Parametro: (store_id_or_NULL,)

Ogni riga (s1,s2,weight) diventa un arco non orientato
con weight = # clienti condivisi.

1E.3 Python: costruzione grafo e domande

```
import networkx as nx

def build_staff_graph(dao, store=None):
    G = nx.Graph()
    # 1) Nodi
    nodes = dao.get_staff_nodes(store)     # lista di staff_id
    G.add_nodes_from(nodes)
    # 2) Archi
    for s1, s2, w in dao.get_staff_edges(store):
        G.add_edge(s1, s2, weight=w)
    return G

def most_collaborative_staff(G):
    """
    Restituisce lo staff con il grado (degree) massimo.
    """
    deg = dict(G.degree())
    return max(deg, key=deg.get), deg

def longest_collaboration_path(G):
```

```
    """
    Restituisce il cammino più lungo (in termini di nodi)
    tra due staff, misurato sul più breve percorso.
    """
    # calcola tutte le distanze minime
    dists = dict(nx.all_pairs_shortest_path_length(G))
    best = (None, None, -1)
    for u, dd in dists.items():
        for v, dist in dd.items():
            if dist > best[2]:
                best = (u, v, dist)
    if best[0] is None:
        return [], 0
    path = nx.shortest_path(G, best[0], best[1])
    return path, best[2]
```

dao.get_staff_nodes(store) esegue la query 1E.1

dao.get_staff_edges(store) esegue la query 1E.2

most_collaborative_staff usa il grado del nodo per il "grado di collaborazione".

longest_collaboration_path trova la coppia di nodi più "lontana" e ne restituisce il cammino.

2E – "Turno ideale di staff"

Scegliere un sottoinsieme di staff che
non contenga due membri dello stesso store e **massimizzi il totale di ordini gestiti**.

2E.1 Struttura dati

staff_store[s] → store_id di ogni staff

staff_orders[s] → numero totale di ordini gestiti da s

2E.2 Codice in 3 funzioni

```
import copy

class ShiftPlanner:
    def __init__(self, staff_store: dict[int,int], staff_orders: dict[int,int], max_k: int):
        """
        :param staff_store: mappa staff_id → store_id
        :param staff_orders: mappa staff_id → ordini gestiti
```

```
    :param max_k: numero massimo di staff nel turno (opzionale)
    """
    self.staff_store  = staff_store
    self.staff_orders = staff_orders
    self.max_k        = max_k
    self.best_team    = []
    self.best_score   = 0

def getBest(self, all_staff: list[int]):
    """
    Funzione madre: prova ogni staff come seed e lancia il backtracking.
    """
    self.best_team  = []
    self.best_score = 0
    for s in all_staff:
        self._ricorsione([s])
    return self.best_team, self.best_score

def _ricorsione(self, team: list[int]):
    """
    Backtracking:
    - non aggiunge due staff dello stesso store
    - rispetta max_k
    """
    score = self.getScore(team)
    if score > self.best_score:
        self.best_score = score
        self.best_team  = team.copy()

    if len(team) >= self.max_k:
        return

    used_stores = { self.staff_store[s] for s in team }
    for cand in self.staff_store:
        if cand in team:
            continue
        if self.staff_store[cand] in used_stores:
            continue
        team.append(cand)
        self._ricorsione(team)
        team.pop()

def getScore(self, team: list[int]) -> int:
    """
    Punteggio = somma degli ordini gestiti dal team
    """
    return sum(self.staff_orders[s] for s in team)
```

2E.3 Uso pratico

```
# 1) DAO fornisce:
staff_store  = dao.get_staff_store_map()    # {staff_id: store_id}
staff_orders = dao.get_staff_orders_map()   # {staff_id: total_orders}
all_staff    = list(staff_store.keys())

planner = ShiftPlanner(staff_store, staff_orders, max_k=5)
team, covered = planner.getBest(all_staff)
print("Team ottimale:", team)
print("Ordini coperti:", covered)
```

Con questo schema hai:

1E: due query SQL (solo join impliciti), plus tre funzioni Python pronte per NetworkX.

2E: tre metodi Python (getBest, _ricorsione, getScore) che realizzano il backtracking col vincolo “uno per

store” e massimizzano il numero di ordini coperti.