Politecnico di Torino

Dipartimento di Ingegneria Gestionale e della Produzione

Laurea Triennale in Ingegneria Gestionale – Classe L8

A.A. 2026/27

PreviStock - Previsione domanda e riordino intelligente per un negozio

Relatore Prof. Giuseppe Bruno Averta

Candidato Mattia Barbero, s311274

Sommario

1. Proposta di progetto

1.1. Titolo della proposta

PreviStock - Previsione domanda e riordino intelligente per un negozio

1.2. Descrizione del problema proposto

L'applicazione mira a risolvere il problema della pianificazione dei riordini in un negozio, avente più rifornitori, dove solitamente le decisioni prese sono spesso manuali e poco ottimizzate, soprattutto per negozi di piccole dimensioni. In assenza di strumenti utili alla predizione della domanda, si verificano solitamente stockout o overstock, per colpa dei quali si ha una perdita a livello di guadagno. L'applicazione verge ad utilizzare lo storico delle vendite per stimare la domanda relativa al mese successivo, proponendo una quantità di riordino consigliato, lasciando sempre la possibilità di override manuale. Inoltre, permette mediante un algoritmo ricorsivo, di trovare i migliori fornitori per effettuare i riordini in modo ottimizzato, ovvero tenendo conto di diversi fattori come:

- diversi costi dei prodotti, a seconda del fornitore
- distanza negozio fornitore, per ridurre il costo relativo alla consegna

Ovviamente anche in quest'ultimo caso l'applicazione permetterà un override manuale.

1.3. Descrizione della rilevanza gestionale del problema

Dal punto di vista gestionale l'applicazione mira a ottimizzare il costo dell'approvvigionamento andando a considerare più aspetti: prezzo prodotto e costo di trasporto, da parte dei fornitori, in base alle quantità di prodotto ordinate. Migliorando così i margini e il capitale, mediante l'utilizzo di scorte più "aderenti" alla domanda. Così facendo si andrà ad incrementare anche il livello di servizio con un impatto diretto su vendite e soddisfazione del cliente. Infine, l'applicazione è scalabile e integrabile con altre realtà, ovvero partendo da un negozio si potrà replicare su più punti vendita, potendone implementare altre opzioni aggiuntive.

1.4. Descrizione dei data-set per la valutazione

I dati utilizzati in questa prova finale, sono dati non reali, non provenienti da fonti esterne, bensì sono stati generati artificialmente tramite script. Nonostante ciò, si avrà un database di dimensioni reali con più di 200 prodotti e 50 fornitori con circa 600 offerte per i prodotti da parte dei fornitori, eventualmente non fosse sufficiente la dimensione del database potrà essere aumentata senza problemi modificando l'input dello script. Entrambi i file saranno presenti all'interno della cartella database.

1.5. Descrizione preliminare degli algoritmi coinvolti

I principali problemi algoritmici che si dovranno affrontare sono i seguenti:

- Controllo dati di input (formato, coerenza, limiti di disponibilità) con l'obiettivo di garantire che i dati usati per previsioni e riordini siano corretti, coerenti e privi di duplicati.
- Compilazione automatica dei campi cercando di ridurre carico e tempi di inserimento, proponendo quantità e campi derivati consistenti utilizzando giuste formule di previsione della domanda.
- Ricerca del fornitore migliore con un algoritmo iterativo con l'obiettivo di selezionare, per ogni prodotto, il fornitore (o il mix di fornitori) che minimizza il costo totale, dovuto a diversi aspetti, rispettando i vincoli operativi.
- Storico dei dati senza sovraccaricare il database fornendo storici senza duplicazione di dati e senza intaccare le performance dell'applicazione.
- Scalabilità e prestazioni con l'obiettivo mantenere tempi di risposta accettabili con decine di migliaia di prodotti e centinaia di migliaia di offerte.

1.6. Descrizione preliminare delle funzionalità previste per l’applicazione software

Al fine di garantire una gestione ordinata e funzionale dell’applicazione, il software sarà strutturato in quattro pagine principali, ognuna dedicata a una specifica funzione seguendo il seguente schema:

1. Pagina relativa alle vendite effettuate nel mese corrente dal negozio:
   - in questa pagina l'utente andrà ad inserire tutti i prodotti venduti e in quale quantità.
2. Pagina relativa alla previsione della domanda e riordini:
   - in questa pagina, finita la mensilità, si potranno effettuare i riordini della merce venduta manualmente oppure mediante un pulsante di previsione della domanda, grazie al quale verranno compilati in automatico i campi relativi alle quantità dei prodotti da riordinare.
3. Pagina relativa alla scelta del fornitore:
   - una volta confermati gli ordini nella seconda pagina, si potrà scegliere, in modo manuale, il fornitore relativo ad ogni prodotto da acquistare oppure mediante un pulsante di auto compilazione, che sfrutta un algoritmo ricorsivo per scegliere i migliori fornitori secondo diversi criteri.
4. Pagina per lo storico delle vendite
   - In questa pagina potremo visualizzare mese per mese tutti gli storici delle vendite (prodotto, quantità venduta, prezzo di vendita)

2. Descrizione dettagliata del problema affrontato

Nel settore commerciale delle piccole-medie aziende, la gestione efficiente del magazzino rappresenta una delle sfide più critiche e dispendiose. Un'azienda operante in questo settore, come un supermercato o un piccolo negozio, solitamente basa la sua sostenibilità economica sulla capacità di soddisfare la domanda dei clienti, senza pensare una modalità per avere una gestione delle vendite e rifornimenti migliore. Infatti, spesso si arriva a una situazione di stock out, che causa una perdita di vendite, d'altro canto un magazzino sovradimensionato genera costi di stoccaggio, rischiando la scadenza o l'obsolescenza dei prodotti. Il processo di riordino dei prodotti è un'attività spesso ricorrente, gestita manualmente o con sistemi poco automatizzati. Il processo richiede di bilanciare due fattori economici principali:

- Il costo di acquisto dei prodotti, che varia a seconda del fornitore.
- I costi logistici, che includono il trasporto e la gestione degli ordini.

In questo contesto, un'operazione apparentemente semplice come "fare un ordine" diventa un complesso problema decisionale. Infatti, ci si pone davanti ad una semplice domanda: da quale dei tanti fornitori disponibili conviene acquistare ciascun prodotto per minimizzare il costo totale, comprensivo sia del prezzo della merce che delle spese di viaggio?

Il progetto si focalizza sul seguente sotto problema: ottimizzazione del costo di approvvigionamento dei prodotti, seguendo la seguente formula:

$$C_{tot} = \left(\sum_{p \in P} q_p \cdot prezzo(p, f_p)\right) + \left(k \cdot \sum_{i=0}^{n} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}\right)$$

ovvero la sommatoria del prezzo di acquisto del prodotto_Pesimo in base al fornitore, sommando a quest'ultimo un valore k, relativo al prezzo della benzina, moltiplicato per la sommatoria delle distanze euclidee del fornitore_Iesimo con il fornitorie_Ieismo+1

L'obiettivo è sviluppare un'applicazione software in grado di assistere il gestore del negozio nel determinare la strategia di riordino ottimale. Dato un insieme di prodotti da riordinare, ognuno disponibile presso più fornitori a prezzi diversi, il sistema deve identificare la combinazione di fornitori da visitare e l'assegnazione prodotto-fornitore che minimizzi un costo totale in base alla formula sopra citata.

Input del Problema:

- Un data-set di fornitori, ciascuno con le proprie coordinate geografiche.
- Un data-set di prodotto-fornitore, che specifica quale fornitore vende quale prodotto e a quale prezzo di acquisto.

- Un data-set di prodotti-magazzino, che specifica le scorte attuali in magazzino, disponibili per la vendita.
- La posizione del negozio (punto di partenza e di arrivo del percorso x=0, y=0).
- Lista dei prodotti da riordinare e le relative quantità in base alle vendite effettuate nel mese/i corrente/i.

Output Desiderato:

- Il percorso ottimale: una sequenza ordinata di fornitori da visitare, partendo dal negozio in posizione x=0 y=0.
- Assegnazione ottimale: da quale fornitore acquistare un determinato prodotto.
- Il costo totale minimo: valore numerico che rappresenta il costo totale.
- Storico delle vendite effettuate nel mese i_esimo, con quantità e prezzo di vendità.

La criticità principale si trova nella natura combinatoria del problema. Al crescere del numero di prodotti e di fornitori, il numero di possibili percorsi e di combinazioni di acquisto esplode in modo esponenziale. Un approccio di forza bruta, provando tutte le combinazioni, rimane molto dispendioso, nonostante un numero modesto di variabili. La sfida risiede quindi nel sviluppare un algoritmo di backtracking tutte le soluzioni in modo intelligente, trovandone una ottimale per la casistica che si sta affrontando.

La potenzialità della soluzione a questo problema, anche su piccola scala, ha un enorme effetto su diversi punti:

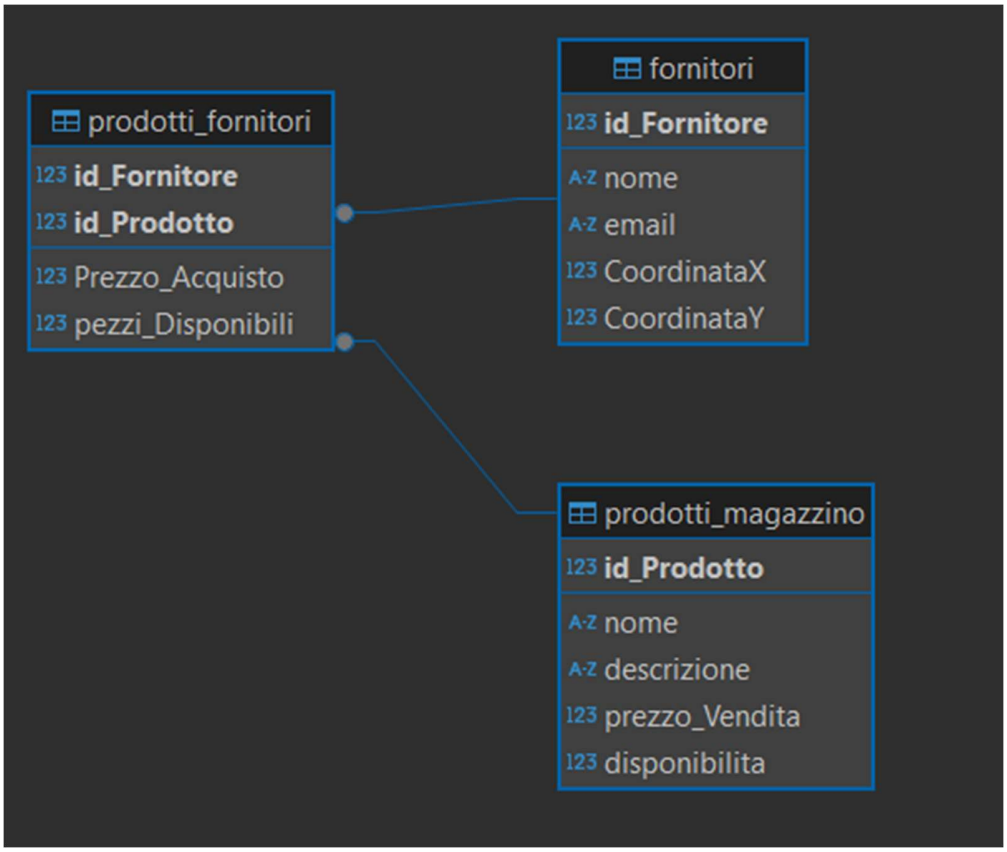
- Risparmio economico, migliore margine di guadagno.
- Supporto decisionale, automatizzando un compito completo.
- Scalabilità e generalizzazione

3. Descrizione del dataset utilizzato

Il database e l'eventuale script per generarne altri di diverse dimensioni si trovano nella cartella database al seguente link:

https://github.com/TdP-prove-finali/BarberoMattia/tree/main/database

Schema ER:



Schema logico:

- fornitori(**id_Fornitore**,nome,email,coordinatX,coordinataY)
- prodotti_fornitori(**id_Fornitore,id_Prodotto**,Prezzo_Acquisto,pezzi_disponibili)
- prodotti_magazzino(**id_Prodotto**,nome,descrizione,prezzo_Vendita,disponibilita)

4. Descrizione delle strutture dati e degli algoritmi utilizzati

Per questo progetto mi sono avvalso del pattern Model, View, Controller ovvero:

- *View:* si occupa della rappresentazione grafica dei dati. Il suo scopo è gestire l'interfaccia utente.
- *Controller:* ha il compito di gestire l'interazione tra model e view.
- *Model:* è la logica applicativa del programma.

Il *DAO* detiene invece lo scopo di gestire le interazioni con il database, interrogandolo con delle query SQL. I dati da qui prelevati vengono successivamente inviati al model che si occupa della gestione.

4.1 DAO

All'interno del file DAO, ho attuato le varie interrogazioni al database, che andranno a restituire tutti i record delle varie tabelle e creando degli oggetti relativi alle specifiche classi (fornitore, prodotto, prodottiFornitori), ritornando un array con la lista di tutti gli oggetti. Inoltre, nel metodo sotto mostrato, oltre a ritornare tutti i record prodotti fornitori, per facilitare varie operazioni nel file Model, ho deciso di creare due dizionari:

- idMap → che avrà come chiave il fornitore e come valore tutti i suoi prodotti associati
- idMapP → che avrà come chiave il prodotto e come valore tutti i fornitori che lo vendono

```
def getProdottiFornitori():
    conn = DBConnect.get_connection()
    cursor = conn.cursor(dictionary=True)
    query = '''select * \
             from prodotti_fornitori pf '''
    cursor.execute(query)
    returna = []
    idmap={}
    idmapP={}
    for row in cursor:
        if row["id_Fornitore"] in idmap.keys():
            idmap[row["id_Fornitore"]].append(row["id_Prodotto"])
        else:
            idmap[row["id_Fornitore"]] = []
            idmap[row["id_Fornitore"]].append(row["id_Prodotto"])
        if row["id_Prodotto"] in idmapP.keys():
            idmapP[row["id_Prodotto"]].append(row["id_Fornitore"])
        else:
            idmapP[row["id_Prodotto"]] = []
            idmapP[row["id_Prodotto"]].append(row["id_Fornitore"])
        returna.append(ProdottoFornitore(idF=row["id_Fornitore"],idP=row["id_Prodotto"],costo=row["Prezzo_Acquisto"],disponibili=row["pezzi_Disponibili"]))
    cursor.close()
    conn.close()
    return returna,idmap,idmapP
```

Per procedere all'inserimento delle chiavi e i loro valori associati, semplicemente si effettua un controllo, ove la chiave che si vuole aggiungere non sia già presente all'interno del dizionario, in quel caso creerò la chiave e associerò il primo valore ad essa. Altrimenti andrò ad aggiungere ai valori già relativi a quella chiave un nuovo valore.

4.2 Model

All'interno del file Model, troveremo la logica applicativa del programma.

```python
def ricorsione(self, parziale, arrayP, associaz):  2 usages  Mattia Barbero *
    if arrayP == []:
        self.getCosto(parziale,associaz)
    else:
        for fornitore in self.trovaCandidati(parziale, arrayP):
            # Creo una lista dei prodotti che questo fornitore può coprire,per backTrack.
            pCoperti = []
            for idP in self.prodottiFornitore[fornitore.id_Fornitore]:
                if idP in arrayP:
                    pCoperti.append(idP)
            for idP in pCoperti:
                arrayP.remove(idP)
            if fornitore.id_Fornitore not in associaz:
                associaz[fornitore.id_Fornitore] = []
            associaz[fornitore.id_Fornitore].extend(pCoperti)
            # Aggiungo il fornitore al percorso se non c'è già
            fornitore_aggiunto_ora = False
            if fornitore not in parziale:
                parziale.append(fornitore)
                fornitore_aggiunto_ora = True
            self.ricorsione(parziale, arrayP, associaz)
            #BACKTRACKING
            if fornitore_aggiunto_ora:
                parziale.pop()
            for idP in pCoperti:
                associaz[fornitore.id_Fornitore].remove(idP)
            if not associaz[fornitore.id_Fornitore]:
                del associaz[fornitore.id_Fornitore]
            arrayP.extend(pCoperti)
```

Questo metodo viene chiamato ogni qual volta che si necessita di trovare la migliore combinazione, che riduce il costo totale di rifornimento. Ad esso viene passato una lista parziale contenete il punto di partenza(negozio), arrayP ovvero i prodotti, sui quali si vuol effettuare il rifornimento ed infine un dizionario associaz vuoto, ove si assocerà ad un determinato fornitore tutti i prodotti che fornirà. Il metodo fornirà tutte le possibili combinazioni possibili, fornitore prodotti, rimuovendo il prodotto da arrayP ogni qual volta che un prodotto sarà disponibile da un fornitore, andando a creare o ad aggiungere il nuovo valore al dizionario associazione ed aggiungendo a parziale il fornitore. Una volta che una soluzione sarà valida, ovvero controllando che non ci siano più prodotti all'interno di arrayP, si effettuerà un backTracking ovvero si andrà a svuotare la soluzione, parziale, a riempire nuovamente l'arrayP e rimuovere le varie associaz. Il metodo utilizza altri due metodi:

- trovaCandidati(), ove il quale restituirà tutti i candidati possibili, che non siano già all'interno di parziale e che possano offrire un prodotto non ancora rimosso da arrayP
- getCosto(), ove calcola il costo totale della soluzione trovata, se essa è minore della soluzione precedentemente trova allora salva associaz, il costo e parziale.

```python
def provaSetGrafo(self, ddf):  1 usage  Mattia Barbero
    self.grafo.clear()
    self.grafo.add_node(self.negozio)
    for idP, f in ddf.items():
        for fornitore in f.options:
            idF = fornitore.key
            if self.idMapF[idF] not in self.grafo.nodes():
                self.grafo.add_node(self.idMapF[idF])

    for f1 in self.grafo.nodes():
        for f2 in self.grafo.nodes():
            if f1 != f2:
                peso = math.sqrt((f2.CoordinataX - f1.CoordinataX) * (f2.CoordinataX - f1.CoordinataX) + (
                        f2.CoordinataY - f1.CoordinataY) * (f2.CoordinataY - f1.CoordinataY))
                self.grafo.add_edge(f2, f1, weight=peso*2)
                #costo benzina *2(sto contando in linea d'aria)
```

Ogni volta che questo metodo verrà chiamato, si effettuerà un reset del grafo, creandone uno nuovo, ove all'interno verranno inseriti solo i nodi utili all'esecuzione della ricorsione, ovvero tutti i fornitori possibili per i prodotti selezionati per effettuare un riordino. Ogni nodo sarà collegato con tutti gli altri nodi, ed il peso dell'arco equivarrà alla distanza euclidea tra i due nodi moltiplicata per un coefficiente, in questo caso 2, il quale indica il costo della "benzina".

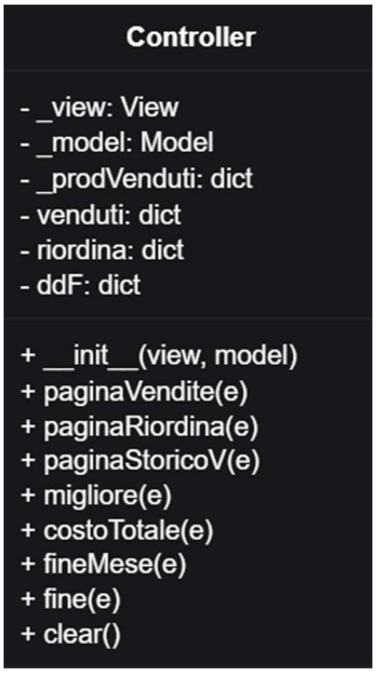
4.3 Controller

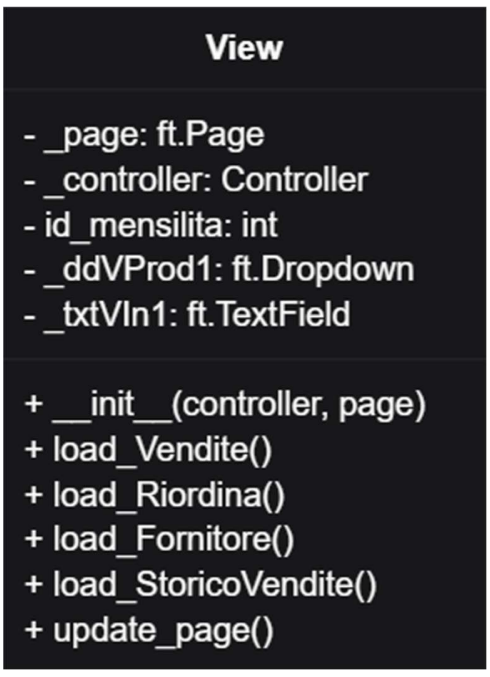
```python
def costoTotale(self, e):  1 usage (1 dynamic)  Mattia Barbero
    flag = True
    self._view._rowPF2.controls = []
    #controllo siano stati sel tutti i dd
    for idP, f in self.ddF.items():
        if f.value == None:
            flag = False
            break
    if flag:
        fornitori_selezionati = {}  # Dizionario: {idFornitore: [lista_prodotti]}
        costo_prodotti = 0.0
        for id_prodotto, dropdown in self.ddF.items():
            id_fornitore = int(dropdown.value)
            # Raggruppa i prodotti per fornitore
            if id_fornitore not in fornitori_selezionati:
                fornitori_selezionati[id_fornitore] = []
            fornitori_selezionati[id_fornitore].append(id_prodotto)
            #costoTot Prodotti
            quantita = int(self.riordina[id_prodotto].value)
            for pf in self._model.prodottiFornitori:
                if pf.idF == id_fornitore and pf.idP == id_prodotto:
                    costo_prodotti += quantita * pf.costo
                    break

        grafo = self._model.grafo
        idmapF = self._model.idMapF
        #nodi da visitare unici
        nodi_da_visitare = {idmapF[idF] for idF in fornitori_selezionati.keys()}
        costo_viaggio = 0.0
        nodo_corrente = self._model.negozio
        while len(nodi_da_visitare)!=0:
            nodo_piu_vicino = None
            distanza_minima = 999999999
            for nodo_candidato in nodi_da_visitare:
                distanza = grafo.get_edge_data(nodo_corrente, nodo_candidato)['weight']
                if distanza < distanza_minima:
                    distanza_minima = distanza
                    nodo_piu_vicino = nodo_candidato

            costo_viaggio += distanza_minima
            nodo_corrente = nodo_piu_vicino
            nodi_da_visitare.remove(nodo_corrente)

        costo_finale = costo_prodotti + costo_viaggio
        testo = ft.Text(f'Costo totale: {round(costo_finale, 2)}')

        # funzione model costo totale
    else:
        testo = ft.Text( value: "Inserisci Tutti i fornitori!", color="red")
    self._view._rowPF2.controls.append(testo)
    self._view.update_page()
```

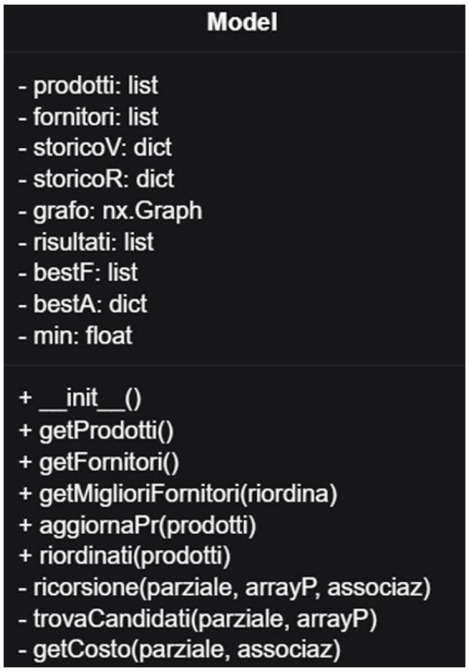
Lo scopo di questo metodo è calcolare il costo totale di un percorso di riordino basato sulla selezione manuale dei fornitori. Come primo step, il metodo esegue un controllo per assicurarsi che l'utente abbia selezionato un fornitore per ogni prodotto da riordinare. Se non tutti i fornitori sono stati selezionati, stamperà un messaggio di errore. Nel caso in cui invece, siano stati

selezionati tutti i fornitori, procede a organizzare i dati e a calcolare la prima componente del costo totale, ovvero i costi dei singoli prodotti. Successivamente calcola il costo del tragitto, iniziando dal primo nodo(negozio) cerca il nodo più vicino ad esso, somma la distanza al costo totale e rimuove il nodo visionato dalla lista dei nodi da visitare, successivamente setta il nodo corrente uguale nodo precedentemente visitato e ripete l'iter cercando il nodo più vicino, ecc.. finché la lista dei nodi da visitare non sarà vuota.

5. Diagramma delle classi principali






Model
- prodotti: list
- fornitori: list
- storicoV: dict
- storicoR: dict
- grafo: nx.Graph
- risultati: list
- bestF: list
- bestA: dict
- min: float
+ __init__()
+ getProdotti()
+ getFornitori()
+ getMiglioriFornitori(riordina)
+ aggiornaPr(prodotti)
+ riordinati(prodotti)
- ricorsione(parziale, arrayP, associaz)
- trovaCandidati(parziale, arrayP)
- getCosto(parziale, associaz)

6. Videate dell'applicazione

Fig. 6.1 Screenshot della videata "Vendite"

Fig. 6.2 Screenshot della videata "Riordina"

Fig. 6.3 Screenshot della videata "Seleziona Fornitori"

Fig. 6.4 Screenshot della videata "Storico Vendite"

Il video dimostrativo dell'applicazione software è disponibile al seguente link: https://www.youtube.com/watch?v=blcibbBbAZs

7. Risultati sperimentali

Per testare l'applicazione sviluppata, sono stati condotti una serie di test sperimentali volti a misurare due aspetti fondamentali:

- capacità dell'algoritmo di trovare una soluzione che abbia un costo totale inferiore rispetto alle altre soluzioni scelte manualmente.
- tempo di esecuzione richiesto dall'algoritmo al crescere della complessità del problema, ovvero aumentando numero di prodotti da riordinare.

Si ricorda che i seguenti test sono stati eseguiti su un pc portatile, dunque un pc con basse specifiche tecniche.

- Tabella costi totali

Scenario	**Strategia Utilizzata**	**Percorso**	**Costo**
Acqua#01(2pz) Acqua#21(2pz)	Ottimale	Negozio Palermo29 Palermo19	83.23€
Acqua#01(2pz) Acqua#21(2pz)	Manuale	Negozio Torino11 Palermo19	101.36€

- Tabella tempi di esecuzione

Pezzi da Riordinare	**Tempo**
pz<5	~0.1s
5<pz<10	~0.2s ~15s
Pz>10	Crescita non lineare

8. Valutazioni sui risultati ottenuti e conclusioni

L'analisi dei risultati ci permette di validare l'efficacia e le caratteristiche della soluzione proposta sotto due aspetti principali. Dal punto di vista dell'efficacia economica, i test hanno dimostrato i risultati attesi dalla creazione dell'applicazione. Questo risultato, sebbene basato su uno scenario specifico, conferma che un approccio algoritmico può portare a risparmi diretti e significativi, minimizzando sia il costo di acquisto della merce, sia i costi logistici legati al trasporto. Dal punto di vista delle performance computazionali, l'analisi ha confermato la natura la complessità del problema affrontato. I tempi di esecuzione crescono all'aumentare del numero di prodotti da riordinare. Se per problemi di piccola entità il sistema risponde in tempi interattivi e pienamente accettabili, per scenari più complessi l'attesa diventa considerevole. Questo risultato non è un difetto dell'implementazione, ma una caratteristica nota di questa classe di problemi di ottimizzazione. Dimostra che l'approccio esaustivo basato su ricorsione e backtracking è praticabile ed efficace per la scala operativa di una piccola impresa, ma evidenzia anche i suoi limiti naturali.

9. Licenza