*Università degli Studi di Pisa*
*Facoltà di Scienze matematiche, fisiche, naturali*

CdL in Informatica

Relazione di stage in azienda

# *Analisi ed implementazione di un modulo per la soluzione di un problema di logistica*

Realizzato presso la Siemens VDO Automotive SpA

**Candidato:** *Andrea Giannini*
**Tutore interno:** *prof. Antonio Frangioni*
**Tutore aziendale:** *dott. Vittorio Abbiuso*

# *Introduzione*

La Siemens VDO Automotive è una delle principali aziende produttrici di iniettori per motori a benzina, con centri dislocati in varie parti del mondo. In particolare, in Toscana, questo avviene negli stabilimenti di S. Piero a Grado (PI) e Fauglia (PI), dove è stato svolto il tirocinio formativo.

Gli iniettori sono uno dei componenti con più alta tecnologia nei motori di nuova generazione, in quanto devono garantire il perfetto afflusso di carburante nella camera di combustione mantenendo alte prestazioni nel rispetto delle normative sulle emissioni inquinanti. L'obiettivo dell'azienda è quello di studiare soluzioni tecnologiche avanzate, capaci di adattarsi agli standard tecnici dei vari tipi di autoveicolo, facendo fronte alle richieste sempre più esigenti di marchi leader nel settore come BMW, Audi, Mercedes ecc.
Per offrire sul mercato mondiale prodotti di qualità ad un certo livello di concorrenza risulta fondamentale un'accurata pianificazione della produzione. Tale pianificazione è l'obiettivo del presente progetto.

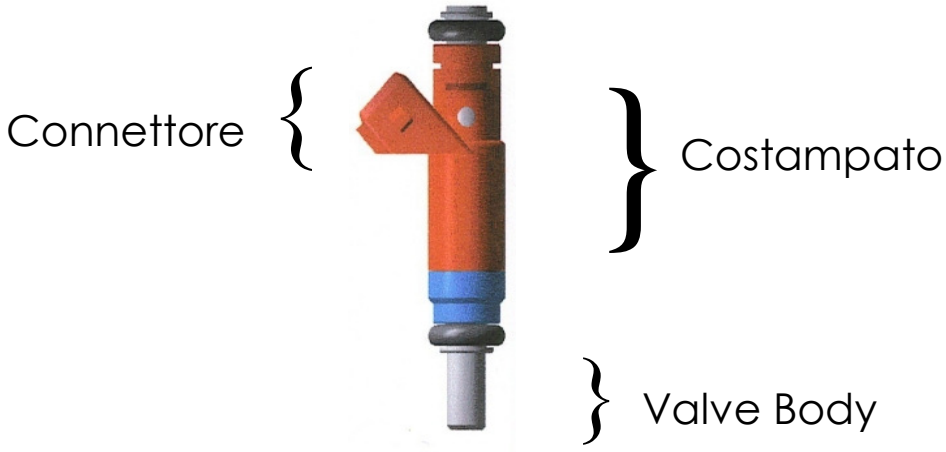
Il tirocinio formativo ha previsto, in un primo momento, l'acquisizione e l'analisi dei metodi e di tutti i dati inerenti alla pianificazione, con l'obiettivo di proporre un modello di programmazione matematica che ottimizzi la sequenza di produzione. In una seconda fase, seguendo il modello proposto, è stato definito un approccio risolutivo implementato in C++ e testato su istanze fornite dall'azienda.
Va evidenziata la difficoltà della messa a punto di un adeguato modello analitico in quanto molti dei dati rilevati possono non essere facilmente quantificabili numericamente. Si è richiesta quindi una interpretazione di tali dati grazie alla collaborazione con gli addetti del settore.

# *Analisi del processo produttivo*

Il processo produttivo si articola per gruppi omogenei ed è suddiviso in 5 cleanroom (chiamate così per l'atmosfera controllata in cui operano), dove si realizzano rispettivamente iniettori dei gruppi Deka 1 (cleanroom 11), Deka 1 (cleanroom 12), Deka 2 (cleanroom 13), Deka 4 (cleanroom 14) e Deka 7 (cleanroom 15).

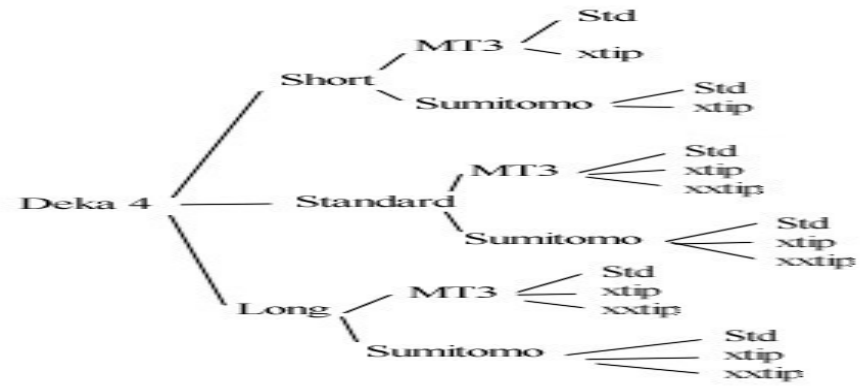
La nostra attenzione si focalizzerà sul Deka 4, in quanto il problema potrà poi essere facilmente adattato agli altri gruppi vista la stretta somiglianza dei componenti e dei processi produttivi.

Il gruppo Deka 4, mostrato in figura, comprende tre grandi famiglie di iniettori identificabili dalla dimensione totale del corpo (o costampato): SHORT, STANDARD e LONG,




e dalla tipologia di valve body montata: STANTARD, EXTENDED TIP (xtip) o EXTRA EXTENDED TIP (xxtip). Anche il connettore può variare ed essere di tipo MT3 (rettangolare) o SUMITOMO (ovale).

Il prodotto finale, quindi, sarà una combinazione dei tre elementi fondamentali: famiglia, valve body e connettore. Il seguente albero mostra chiaramente le possibili combinazioni che possiamo trovare su di un iniettore.




Il passaggio dalla fabbricazione di un certo ***tipo*** ad un altro necessita di cambiamenti delle impostazioni sui macchinari: se ad esempio è in produzione un pezzo della famiglia SHORT e si vuole passare ad uno della famiglia LONG , è indispensabile intervenire sulla linea tarando le macchine per i nuovi pezzi.

Il processo sopra descritto, chiamato ***cambio tipo****,* può provocare dei ritardi a livello di produzione che dipendono da quale parte dell'iniettore si va a modificare. Esistono dei cambi, chiamati in gergo ***trasparenti,*** che non vanno a rallentare la catena di montaggio: al fine della definizione di un modello matematico non verranno presi in considerazione.

L'analisi del processo produttivo e il confronto con i tecnologi (gli addetti alla produzione) ha consentito di creare un resoconto affidabile delle problematiche di cambio tipo.

| | |
|---|---|
| **Cambio Tipo fra famiglie** | • Richiede mediamente 2 ore di tempo in quanto deve essere totalmente cambiato lo stampo per il costampato.<br>• Va tassativamente eseguito nell'arco di tempo che va dalle 6.00 alle 14.00. |

| | |
|---|---|
| **Cambio Tipo connettore** | • Richiede mediamente 1 ora e mezzo di tempo.<br>• Se si effettua insieme al cambio di famiglia non si prende in considerazione perché viene comunque cambiato lo stampo dedicato (engel)<br>• Va effettuato dalle 6.00 alle 22.00 |
| **Cambio Tipo Valve Body** | • Richiede mediamente 1 ora di tempo<br>• Va effettuato dalle 6.00 alle 22.00 |
| **Cambio Tipo tra iniettori del solito gruppo (Famiglia, V/B connettore uguali)** | • Richiede indicativamente mezz'ora di tempo<br>• Può essere effettuato anche la notte |

Come si può notare, la variazione fra famiglie deve avvenire in un determinato arco di tempo più ristretto rispetto al cambio V/B e connettore perché, vista la difficoltà dell'operazione, si richiede che questo cambio tipo avvenga in presenza dei tecnologi che possono intervenire in caso di guasti o imprevisti che potrebbero rallentare, se non addirittura interrompere, la produzione pianificata.

In generale, cambio stampo connettore e V/B potrebbero teoricamente essere effettuati anche durante i turni notturni in quanto la loro difficoltà non è al pari del cambio costampato; si preferisce comunque rimandare tale operazione ai primi due turni giornalieri.

E' utile,a tal proposito, la conoscenza dei turni di lavoro, riassunti nella seguente tabella:

| | Lunedì | Martedì | Mercoledì | Giovedì | Venerdì | Sabato | Domenica |
|---|---|---|---|---|---|---|---|
| 1° 6.00 -14.00 | A | A | B | B | C | C | X |
| 2° 14.00 - 22.00 | D | D | A | A | B | B | X |
| 3° 22.00 - 6.00 | C | C | D | D | D | X | A man |

La prima colonna indica l'orario giornaliero dei tre turni lavorativi, le lettere le 4 squadre di lavoro. Va ricordato che dei 18 turni settimanali uno (A man), a

discrezione degli addetti alla linea, viene dedicato alla manutenzione della cleanroom.

I turni contrassegnati dalla X rappresentano il periodo di riposo settimanale sulla linea di produzione; va precisato che per gli iniettori Deka 4 è possibile attivare, in determinati casi, una turnazione a 21 turni settimanali. Risulta pertanto indispensabile tenerne di conto a livello progettuale e prevedere un sistema che permetta di scegliere il sistema di turnazione.

Si deve considerare il fatto che i tecnologi sono presenti in fabbrica solo dalle 8.00 alle 17.00 nei primi 5 giorni settimanali.

# Capitolo 2

## *Il SAP*

Il programma aziendale per la gestione della produzione, SAP, descrive per ogni tipo di iniettore il *Fabbisogno* (ovvero la domanda del cliente), la quantità disponibile (*stock* se è positivo, *arretrato* altrimenti) e il numero di pezzi da produrre giornalmente. Viene inoltre mostrata la sommatoria giornaliera dei tre campi. La seguente schermata mostra un valido esempio.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | Tavola pian. | | | | |
| 2 | Da data : 21.07.2005 | | A data: | 04.09.2005 | |
| 3 | | | | | |
| 4 | | | Scad | DO 21.07.05 | VE 22.07 |
| 5 | Gr.prod. DEKA4 | PZ | 33209 | 16680 | 48240 |
| 6 | Fabbisogno | PZ | 63696 | 30867 | 21960 |
| 7 | Quantità disponibile | PZ | 3738 | -17048 | 13192 |
| 8 | | | < | DO 21.07.05 | VE 22.07 |
| 9 | 7523423 Fuel Inje | | | | |
| 10 | Quantità disponibile | PZ | | | |
| 11 | T453 DEKA4 | PZ | | | |
| 12 | | | < | DO 21.07.05 | VE 22.07 |
| 13 | 7525721 Fuel Inje | | | | |
| 14 | Quantità disponibile | PZ | 5760 | -5948 | -8828 |
| 15 | Fabbisogno | PZ | | 11708 | 2880 |
| 16 | T450 DEKA4 | PZ | 2880 | | |
| 17 | | | < | DO 21.07.05 | VE 22.07 |
| 18 | 8200491439 Fuel I | | | | |
| 19 | Quantità disponibile | PZ | 31732 | 31732 | 31732 |
| 20 | Fabbisogno | PZ | | | |
| 21 | T448 DEKA4 | PZ | | | |
| 22 | | | < | DO 21.07.05 | VE 22.07 |
| 23 | 8200491741 Fuel I | | | | |
| 24 | Quantità disponibile | PZ | | | |
| 25 | Fabbisogno | PZ | | | |
| 26 | T476 DEKA4 | PZ | | | |

*Nella parte in alto viene riassunta la situazione sul Deka 4.*
*Nelle righe* **Gr.prod. DEKA4, Fabbisogno e Quantità disponibile***, sono indicate le sommatorie giornaliere dei rispettivi campi di ogni tipo di iniettore.*
*La colonna* ***Scad.*** *riassume la situazione dei 4 giorni precedenti al primo visibile.*

*Nella parte in basso ogni iniettore è suddiviso per 3 righe oltre quella indicante il codice SAP (stampigliato sul pezzo finito):* **il Fabbisogno** *(non appare se non è presente domanda),* **il tipo** *(es. T450 DEKA4) ovvero la quantità da pianificare e* **la quantità disponibile**.
*Questo ultimo campo viene calcolato automaticamente nel seguante modo: quantità disponibile precedente + quantità pianificata – fabbisogno.*

Come si può vedere, l'unico parametro sul quale si possa prendere una decisione (in bianco) è quello relativo alla quantità di pezzi che vanno prodotti.

Compito del software da elaborare sarà quello di calcolare automaticamente tale quantità in base ai valori presenti negli altri campi tenendo però sempre in considerazione la problematica dei cambi tipo.

Si deve inoltre tenere conto del fatto che sia la quantità da produrre che il fabbisogno sono multipli dell'unità di misura dello *skid*, il cassone in cui vengono raccolti gli iniettori prima di avviarli alla spedizione. Appare evidente come sia importante colmare ogni skid in quanto ognuno di questi non può essere inviato fino al completo riempimento.

La seguente tabella mostra gli iniettori attualmente in produzione sulla linea Deka 4 indicando, per ogni tipo, la famiglia di appartenenza, i componenti montati e la capacità del relativo Skid.

**Iniettori SHORT**

| I/T | Codice SAP | Valve Body | Connettore | Capacità Skid |
|---|---|---|---|---|
| **436** | **1446-534/FR** | **std** | **sumitomo** | **3960** |
| **413** | **1446-508/FR** | **std** | **sumitomo** | **3960** |
| **445** | **1446-538/FR** | **std** | **sumitomo** | **3960** |
| **459** | **874-502** | **xtip** | **sumitomo** | **3960** |
| **448** | **8200491439** | **std** | **MT3** | **3960** |

**Iniettori STANDARD**

| I/T | Codice SAP | Valve Body | Connettore | Capacità Skid |
|---|---|---|---|---|
| **430** | **06C133551** | **xtip** | **sumitomo** | **3240** |
| **439** | **06B133551T** | **xtip** | **sumitomo** | **3240** |
| **415** | **1446-708/FR** | **xxtip** | **MT3** | **3240** |
| **432** | **PW811635** | **std** | **MT3** | **1600** |
| **473** | **VAZ20734** | **std** | **MT3** | **1600** |
| **474** | **VAZ20735** | **std** | **MT3** | **1600** |
| **449** | **7515267** | **xxtip** | **MT3** | **3240** |

**Iniettori LONG**

| I/T | Codice SAP | Valve Body | Connettore | Capacità Skid |
|---|---|---|---|---|
| **431** | **06B133551H** | **std** | **sumitomo** | **3240** |
| **435** | **1446-531/FR** | **xtip** | **sumitomo** | **2880** |
| **457** | **874485** | **std** | **MT3** | **2880** |
| **460** | **874487** | **std** | **MT3** | **2880** |
| **470** | **874-480** | **std** | **MT3** | **2880** |
| **450** | **7525721** | **xxtip** | **MT3** | **2880** |
| **458** | **874-520** | **xxtip** | **MT3** | **2880** |

Vanno evidenziati due casi particolari.

A parità di arretrato, gli iniettori contrassegnati con /FR (Fuel Rail) hanno la priorità; tali iniettori, infatti, non vanno direttamente al cliente, ma vengono in seguito assemblati in fabbrica su appositi RAIL per il montaggio diretto sul motore.

Inoltre, i tipi 473 e 474, visto che è il cliente (Autovaz) che si occupa del trasporto, devono essere prodotti nella quantità richiesta entro la data prevista per la spedizione.

Per la programmazione della produzione si deve infine tener conto della capacità di produzione del Deka 4 che va da 5300 a 5500 pezzi con una media di 5400 iniettori a turno ovvero 16200 pezzi per giorno lavorativo. Questo dato può sensibilmente diminuire in caso di scioperi o addirittura venire azzerata nel caso di festività.

# Capitolo 3

## *Pianificare la Produzione*

Ad oggi la programmazione viene fatta manualmente dagli addetti senza l'ausilio del calcolatore, vista la presenza di un ottimizzatore poco funzionale su SAP, facendo affidamento più sull'esperienza personale che non sull'elaborazione sistematica dei dati numerici. Se questo può risultare efficiente dal momento che qualità e flessibilità umana riescono a far fronte ad eventi non prevedibili da un modello matematico, si deve però considerare il fatto che una grande quantità di variabili in gioco possono essere più facilmente gestibili da un calcolatore.

Per la definizione di un adeguato modello matematico è essenziale conoscere innanzitutto il metodo usato dagli addetti al settore per la programmazione di produzione. La procedura può essere riassunta nel seguente modo.

Innanzitutto si individuano su SAP gli iniettori con più arretrato e si va a vedere l'ultimo pezzo in produzione per determinare a quale famiglia appartiene e i componenti installati. A questo punto si crea, per tentativi, una sequenza di produzione cercando di pianificare cambi tipo più veloci possibile. Ad esempio la variazione LONG-STD-SUMI → LONG-XTIP-MT3 sarà molto meno lenta di LONG-STD-SUMI → SHORT-XTIP-MT3 dato che si vanno solamente a modificare macchinari adibiti all'assemblaggio di V/B e Connettore.

Va sempre controllato il fatto che si possa effettuare il cambio tipo durante il turno corrispondente.

Naturalmente si tiene in considerazione la possibilità di effettuare l'accorpamento di richieste (relative ad uno stesso tipo di iniettore) al fine di ridurre in maniera significativa il numero di cambi tipo. Se ad esempio il TP 430 ha un fabbisogno di 3240 pezzi il 10 di aprile, 9720 il 14 e di 6480 il 18, si può decidere di produrre tutti i 19440 elementi il 16 aprile mandando in arretrato le prime due scadenze e anticipando l'ultima.

In generale, nella determinazione manuale di un'adeguata sequenza di produzione, si considerano le seguenti priorità:

1. accorpare più possibile produzioni con uguale costampato;
2. all'interno di una stessa famiglia ridurre i cambio tipo connettore;
3. all'interno di una stessa famiglia ridurre cambi tipo V/B;
4. nei cambi fra famiglie, si cerca inoltre di accorpare produzioni che non prevedono cambi V/B (non è necessario per il connettore in quanto col cambio costampato si esegue automaticamente cambio stampo engel).

Prima di passare alla proposta di un modello matematico, è fondamentale avere chiara visione del problema. Riassumiamo quindi le caratteristiche chiave e gli obiettivi da raggiungere.

- Sulla linea Deka 4 si possono attualmente produrre 19 tipi di iniettori.
- Ogni iniettore è composto da tre parti: costampato, valve body e connettore.
- Il costampato (o famiglia) può essere Short, Standard o Long.
- La valve body può essere Standard, xtip o xxtip.
- Il connettore può essere MT3 o sumitomo.
- Ogni tipo di iniettore deve essere prodotto in multiplo della capienza del relativo skid.
- Per passare da un tipo di iniettore ad un altro (cambio tipo) si impiega una certa quantità di tempo che varia a seconda della parte dell'iniettore che si va a cambiare.
- Determinati cambi tipo hanno limitazione per quanto riguarda l'ora in cui possono essere effettuati.
- L'obiettivo è progettare un algoritmo che ottimizzi la produzione, cioè che cerchi di soddisfare i fabbisogni prima possibile minimizzando quindi l'arretrato che si va a creare quando una domanda viene soddisfatta in ritardo.

Sono necessarie alcune ulteriori considerazioni.

Innanzitutto i numerosi dati in gioco ci impongono di discretizzare il tempo, ovvero di trovare un opportuno intervallo temporale da considerare come unità di misura. Per far ciò è indispensabile analizzare sia la durata di produzione di ogni skid, sia quella dei vari cambi tipo.

La linea Deka 4 ha una media produttiva che si assesta fra i 5300 e i 5500 pezzi prodotti per turno di lavoro, ovvero circa 663/688 iniettori per ora. Vi sono inoltre quattro dimensioni standard di Skid, costituiti rispettivamente da 3960, 3240, 2880 e 1600 pezzi. Si possono quindi stimare gli intervalli di tempo necessari per produrre un singolo skid, riassunti nella tabella seguente.

*Durata in minuti*

| *Dimensione Skid* | *Durata max* | *Durata min* | *Durata media* |
|---|---|---|---|
| **3960** | 358 | 346 | 352 |
| **3240** | 293 | 283 | 288 |
| **2880** | 261 | 251 | 256 |
| **1600** | 145 | 139 | 142 |

*Durata in ore*

| *Dimensione Skid* | *Durata max* | *Durata min* | *Durata media* |
|---|---|---|---|
| **3960** | 5h 58m | 5h 46m | 5h 52m |
| **3240** | 4h 53m | 4h 42m | 4h 48m |
| **2880** | 4h 21m | 4h 11m | 4h 16m |
| **1600** | 2h 25 m | 2h 19m | 2h 22m |

Se arrotondiamo i tempi medi al quarto d'ora (ovvero 5h 45m, 4h 45m, 4h 15m e 2h 15m) vediamo che le prime tre approssimazioni rientrano o superano di poco i limiti orari calcolati; il problema si concentra sullo skid da 1600 pezzi in quanto, sia adottando 2h 15m che 2h 30m, è evidente come si superino ampiamente i valori preposti.

Appare quindi ragionevole adottare come unità di tempo 10 minuti: prendendo come tempo medio di produzione dei 4 tipi di skid 5h 50m, 4h 50m, 4h 20m e 2h 20m rimaniamo ampiamente nei limiti reali di produzione.
La seguente tabella mostra chiaramente come varia la situazione con l'arrotondamento a 10 e 15 minuti.

| | *unità di tempo 10 min* | | *unità di tempo 15 min* | |
|---|---|---|---|---|
| *Dimensione Skid* | *tempo* | *Pezzi per turno* | *tempo* | *Pezzi per turno* |
| **3960** | 5h 50m | 5431 | 5h 45m | 5509 |
| **3240** | 4h 50m | 5363 | 4h 45m | 5457 |
| **2880** | 4h 20m | 5317 | 4h 15m | 5421 |
| **1600** | 2h 20m | 5485 | 2h 15m | 5689 |
| | | | 2h 30m | 5120 |

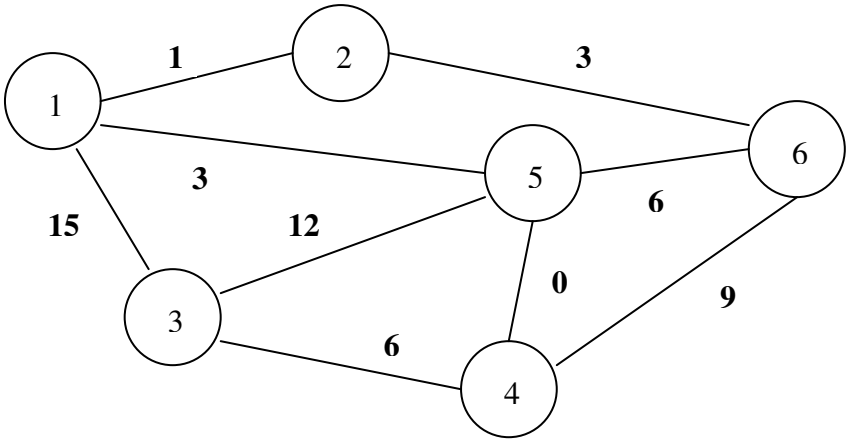
# Capitolo 4

## *Formulazione del problema*

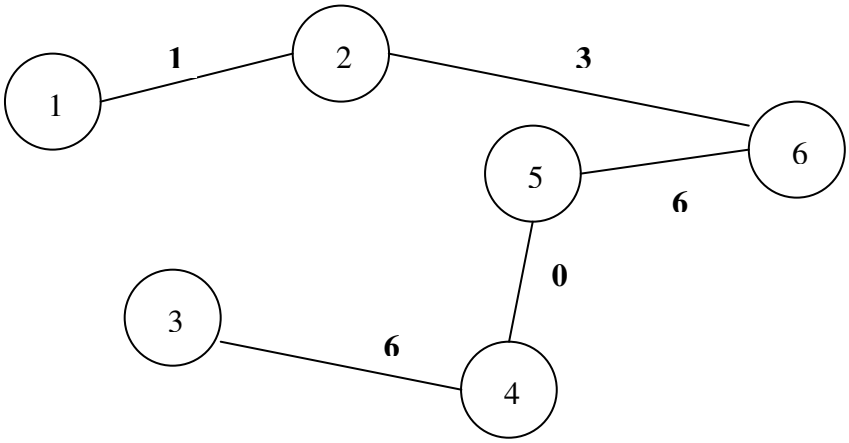
Ricordiamo che le principali informazioni che ci vengono fornite sono le *quantità di domanda* dei clienti, suddivise per tipo di iniettore. Solitamente queste richieste sono multiple della capienza dello skid relativo e vengono suddivise per giorno.
Un altro dato importante è la *quantità disponibile* prima del *periodo fisso* da pianificare.
Oltre ai dati ottenuti da SAP, vanno considerate tutte le informazioni fin qui descritte, in particolare il fatto che un cambio tipo comporti ritardi in termini di tempo con quantità prodotta uguale a zero, e che tali sospensioni di produzione siano fattibili solo in determinate ore dell'attività produttiva.
Il quadro fin qui descritto ci suggerisce di modellizzare il problema tramite la seguente struttura a grafo.
Un grafo, denominato **G=(N,A),** è caratterizzato da un insieme di nodi **(N)** ed un insieme di archi **(A)** che li collegano; tali archi possono avere un'etichetta ovvero un valore associato che ne determina il valore nel sistema di riferimento. Nel caso in cui gli archi abbiano un verso, ovvero che si possano percorrere solo in una certa direzione, sono detti orientati.

Prima di adattare la definizione di grafo al nostro problema è necessario descrivere le caratteristiche che dovrà avere un cammino su tale grafo.
Questo cammino, chiamato *Hamiltoniano*, dovrà raggiungere tutti i nodi a disposizione senza mai passare due volte dallo stesso nodo.

Facciamo un esempio per meglio capire il funzionamento:




Inizialmente il grafo si presenta in questo modo, i numeri in grassetto indicano il costo degli archi.



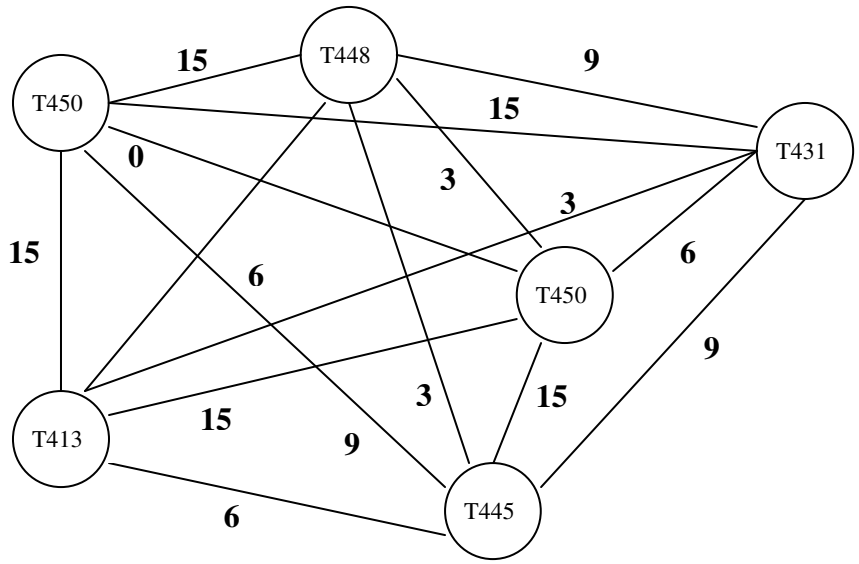

Dopo l'applicazione dell'algoritmo di ricerca ci troviamo di fronte ad un cammino Hamiltoniano.

Il modello descritto sopra può venir facilmente adattato alla nostra situazione reale in cui è presente una sequenza di produzione di skid appartenenti a varie tipologie di iniettori. Per riportarci ad una condizione in cui sia possibile individuare un cammino hamiltoniano, deve essere determinata la

composizione degli archi e dei nodi che andranno a specificare la conformazione del grafo.

La struttura presa in considerazione è quella in cui ogni skid di fabbisogno viene rappresentato da un nodo collegato agli altri elementi tramite archi la cui etichetta determina il costo di cambio tipo, in termini di istanti di tempo, fra ogni skid.

L'esempio proposto sopra applicato ad una situazione reale è il seguente:




Ogni nodo rappresenta un singolo skid di un determinato tipo di iniettore caratterizzato dal nome (es. T450). Ogni arco ha assegnata un'etichetta che identifica il tempo di cambio tipo espresso in unità di tempo

Come si può vedere dall'esempio, ognuno degli n nodi è inizialmente connesso con i restanti elementi tramite n-1 archi etichettati dal tempo di cambio tipo. Si può notare come archi che collegano iniettori dello stesso tipo (nel nostro caso T450-T450) abbiano ovviamente questo valore impostato a zero.

Come accennato sopra, un cammino sul grafo deve specificare una sequenza che stabilisca l'ordine di produzione in base ai nodi ed ai costi degli archi presenti. Per far ciò adattiamo la definizione di cammino Hamiltoniano

al nostro problema in quanto alla fine del ciclo produttivo devono essere messi in produzione tutti gli skid per cui è stata fatta un'esplicita richiesta. Per garantire l'ammissibilità di tale cammino è necessario verificare i vincoli temporali imposti sull'inizio e fine di produzione di ogni nodo, viene cioè determinato che per ogni passo (arco percorso) il cambio tipo che ne deriva vada a cadere in un orario effettivamente consentito.

Una volta descritte le caratteristiche che dovrà avere un cammino, va chiarito come individuare una soluzione ammissibile che tenga conto anche dei fabbisogni e della loro disposizione nel tempo. Non è infatti corretto calcolare una sequenza che escluda solamente i cambi tipo vietati, ciò potrebbe portare a soddisfare domande in largo anticipo o, ancor peggio, in forte ritardo.
Questo è essenzialmente il motivo che porta alla determinazione di una ***funzione obiettivo*** che valuti la "bontà" di ogni soluzione e la confronti con ogni possibile altra candidata.
Il requisito fondamentale è che le richieste vengano soddisfatte prima possibile, per tale motivo viene utilizzato il livello di backlog; questo rappresenta la quantità di arretrato di ogni iniettore per ogni giorno lavorativo . E' sostanzialmente la *quantità disponibile,* con segno opposto, che appare sul sistema SAP e viene calcolata nel seguente modo:
*backlog giorno precedente – quantità prodotta + fabbisogno*.
La somma di tutti i backlog del periodo fisso rappresenta il costo totale di backlog.
Vediamo un esempio su un tipo di iniettore:

| | | ***Giorno 1*** | ***Giorno 2*** | ***Giorno 3*** | ***Giorno 4*** | ***Giorno 5*** |
|---|---|---|---|---|---|---|
| **T450** | **backlog** | **2880** | **2880** | **0** | **-5760** | **0** |
| | fabbisogno | **2880** | | | | **5760** |
| | quantità prodotta | | | **2880** | **5760** | |

Nei primi due giorni è presente un backlog positivo provocato dal fabbisogno di 2880 pezzi del giorno uno, il terzo giorno la richiesta viene soddisfatta e ciò provoca l'azzeramento del backlog. Nel quarto viene anticipata la produzione per la domanda del giorno 5 e questo determina un arretrato negativo di -5760 pezzi.

Sommando tutti i campi backlog otteniamo un costo totale di 0 pezzi, questo ci dice che al giorno 5 noi avremo soddisfatto tutti i fabbisogni precedenti.

Il costo di backlog fornisce quindi un quadro riassuntivo fornendoci utili indicazioni riguardo la situazione globale, analizzando però ogni singolo box notiamo come il soddisfacimento della prima richiesta nel terzo giorno porti in arretrato i primi due.

Naturalmente queste situazioni devono essere evitate, la consegna in ritardo di un fabbisogno viene considerata costosa a seconda del tipo di iniettore che si va a valutare. Per ovviare a queste circostanze (il costo di backlog non se ne accorge) viene introdotto un parametro, che chiameremo *alfa*, che accosta ad ogni iniettore un valore moltiplicativo del backlog positivo. Se ad esempio forniamo un alfa di 10, 1 skid in ritardo varrà effettivamente come 10. Vediamo l'esempio sopra modificato:

**alfa=10**

| | | *Giorno 1* | *Giorno 2* | *Giorno 3* | *Giorno 4* | *Giorno 5* |
|---|---|---|---|---|---|---|
| **T450** | **backlog** | **2880**<br>**28800** | **2880**<br>**28800** | **0** | **-5760** | **0** |
| | fabbisogno | **2880** | | | | **5760** |
| | quantità prodotta | | | **2880** | **5760** | |

-in rosso i valori moltiplicati per il parametro alfa-

Questa situazione, che ha un costo virtuale di backlog pari a 51840, risulta più costosa della seguente (che verrrà preferita) anche se ha il medesimo costo di backlog effettivo:

| | | *Giorno 1* | *Giorno 2* | *Giorno 3* | *Giorno 4* | *Giorno 5* |
|---|---|---|---|---|---|---|
| **T450** | **backlog** | **2880**<br>**28800** | **0** | **0** | **-2880** | **0** |
| | fabbisogno | **2880** | | | | **5760** |
| | quantità prodotta | | **2880** | | **2880** | **2880** |

-in rosso i valori moltiplicati per il parametro alfa-

Costo virtuale di backlog = 25920

Allo stesso modo è stato inserito un parabetro, *beta*, per pesare i valori di backlog negativo: è sempre bene produrre in anticipo (non sono stati forniti vincoli a riguardo), ma ciò potrebbe portare, come mostra l'esempio sopra, a generare ritardi di produzione.
Tutti i parametri introdotti possono essere decisi dall'utente e variare per ogni tipologia di iniettore.

I miglioramenti sul ciclo produttivo tramite il calcolo del costo di backlog hanno la limitazione di non "accorgersi" di eventuali cambi tipo che vanno a cadere in orari vietati. E' indispensabile quindi cercare di ridurre prima di tutto tali cambi tipo e solo dopo, a parità di essi, provare a minimizzare il costo di backlog virtuale.
Per riprodurre questa situazione a livello matematico, deve essere introdotto un valore indicante il costo massimo di backlog che definisce il livello di arretrato come se non fosse stato messo in produzione niente. Questo fattore, numericamente elevato, andrà a dividere via via i costi di backlog trovati in modo da ridurli in forma decimale per farli "pesare" meno del numero di cambi tipo vietati. Il valore così trovato andrà minimizzato ovvero dovranno essere confrontate varie soluzioni e scelta, ad ogni passo, la migliore.
Vediamo in dettaglio la conformazione della funzione obiettivo:

**n =** numero di cambi tipo vietati

$\mathbf{k}_{ij}$ **=** livello di backlog dell'iniettore i al giorno j

**max =** massimo possibile costo di backlog

**β =** parametro per pesare il backlog negativo

$\sum_{j \in \text{giorni}} \sum_{i \in \text{iniettori}} \alpha_i (\mathbf{k}_{ij}) < 1 / \max$

costo totale di backlog ($\alpha_i$ è il parametro per pesare $\mathbf{k}_{ij}$)

**Funzione Obiettivo:**

$$\min \left[ \mathbf{n} + \left( \sum \text{costo di backlog positivo} + \beta * \sum \text{costo di backlog negativo} \right) \frac{1}{\max+1} \right]$$

# Capitolo 5

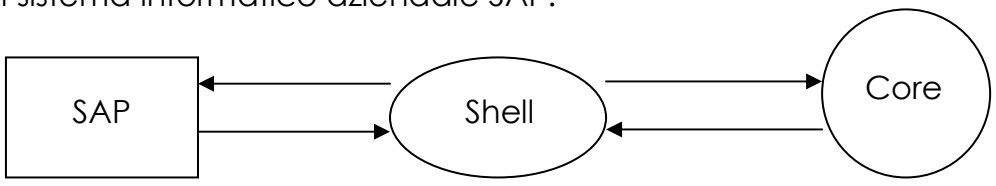
## *Un approccio algoritmico di tipo euristico*

Molti problemi di ottimizzazione combinatoria, vista la loro difficoltà e complessità, non possono essere facilmente risolti in modo esatto; si cerca in tal caso una "buona" soluzione ammissibile. Gli algoritmi relativi vengono chiamati *euristici*.

Sintetizziamo le varie fasi di risoluzione:

- In fase di inizializzazione si determina un grafo dove i nodi corrispondono ad una forte aggregazione della domanda; su tale grafo viene poi applicato l'algoritmo euristico che determinerà un primo cammino hamiltoniano, cioè una prima sequenza di produzione.
- Successivamente il grafo viene espanso per trovare, se esiste, una soluzione ammissibile con una funzione obiettivo minore rispetto alla precedente mediante un procedimento chiamato *ricerca locale*.
- Si disaggregano ulteriormente i nodi secondo uno o più metodi e si effettua sul grafo trovato una nuova ricerca locale che migliori il risultato.
- Viene ripetuto il procedimento di disaggregazione finché non sono più possibili ulteriori passi ovvero non si determinino soluzioni migliori rispetto a quelle già trovate.

L'esecuzione di algoritmi euristici e di ricerca locale sono computazionalmente molto onerosi e richiedono un numero molto elevato di operazioni, per questo è bene dividere in livelli la struttura del software in modo da alleggerire i dati che analizzerà il vero motore, chiamato *core*, rendendoli poi visibili ad una *shell* che fungerà da traduttore per l'utente e per il sistema informatico aziendale SAP:



A questo punto specifichiamo in dettaglio l'algoritmo che determina una prima soluzione tramite un procedimento euristico. Per prima cosa dobbiamo definire i nodi del grafo iniziale: l'idea è quella di creare tanti nodi quanti sono i tipi di iniettore presi in considerazione e di associare ad ogni nodo tutte le relative domande nell'orizzonte temporale di riferimento, cioè le richieste in skid di tale iniettore nel mese. Va evidenziato il fatto che la tavola di pianificazione di SAP ci fornisce traccia della quantità disponibile di ogni tipo di iniettore nel giorno immediatamente precedente al periodo da pianificare. Questo dato deve essere sommato alle richieste nel caso sia un arretrato, oppure sottratto nel caso vada a rappresentare una quantità in stock.

Il nodi possono essere ulteriormente aggregati in insiemi tali che ogni insieme contenga tutti i nodi legati da una specifica relazione. Ogni gruppo di nodi con le sue relazioni viene definito *cluster*. Un esempio concreto di *clusterizzazione* potrebbe essere quello di avere una k cluster in cui ciascun nodo ha un cambio tipo di V/B e connettore con gli altri nodi del cluster ma non di famiglia. Questa procedura può essere adottata per ridurre cambi tipo tra le tre famiglie (SHORT, STD , LONG ) che hanno una notevole limitazione riguardo gli orari di fattibilità.

Presentiamo ora un primo algoritmo di inizializzazione che suddivide i nodi per iniettore e successivamente gli raggruppa in cluster:

```
Procedure Inizializza_grafo(G, Vr)
  begin  for  j…numero_iniettori
         {N = crea_nodo
          for r…numero_richieste
               { if  (Vr[r].id_iniettore = j)              //se l'id della richiesta è uguale
                     N.add_richiesta (Vr[r])  };          // la aggiunge al nodo

            G.add_nodo(N)                  // aggiunge il nodo al Grafo
          }
         dividi_per_cluster(G, C[k])  //divide i nodi del grafo G sul vettore dei cluster

         ordina_per_backlog(C[k])        //ordina i nodi all'interno di ogni cluster per
                                          //arretrato decrescente

         ordina_cluster(V[p], C[k])   //ordina il vettore dei cluster per arretrato
                                      // decrescente e li connette nel cammino V[p]
  end
```

- Procedura *Inizializza_grafo(G, Vr)* -

Va puntualizzato che non vi è nessuna garanzia che il cammino che deriva dalla clusterizzazione sia hamiltoniano ammissibile cioè che escluda la presenza di cambi tipo che avvengono in orari non consentiti.
L'algoritmo di inizializzazione può essere raffinato introducendo la funzione obiettivo descritta nel precedente capitolo che tiene conto sia del numero di cambi tipo vietati sia del costo totale di backlog.

Definiamo quindi un altro interessante algoritmo, chiamato *nearest neighbour*, che individui una prima valida soluzione :

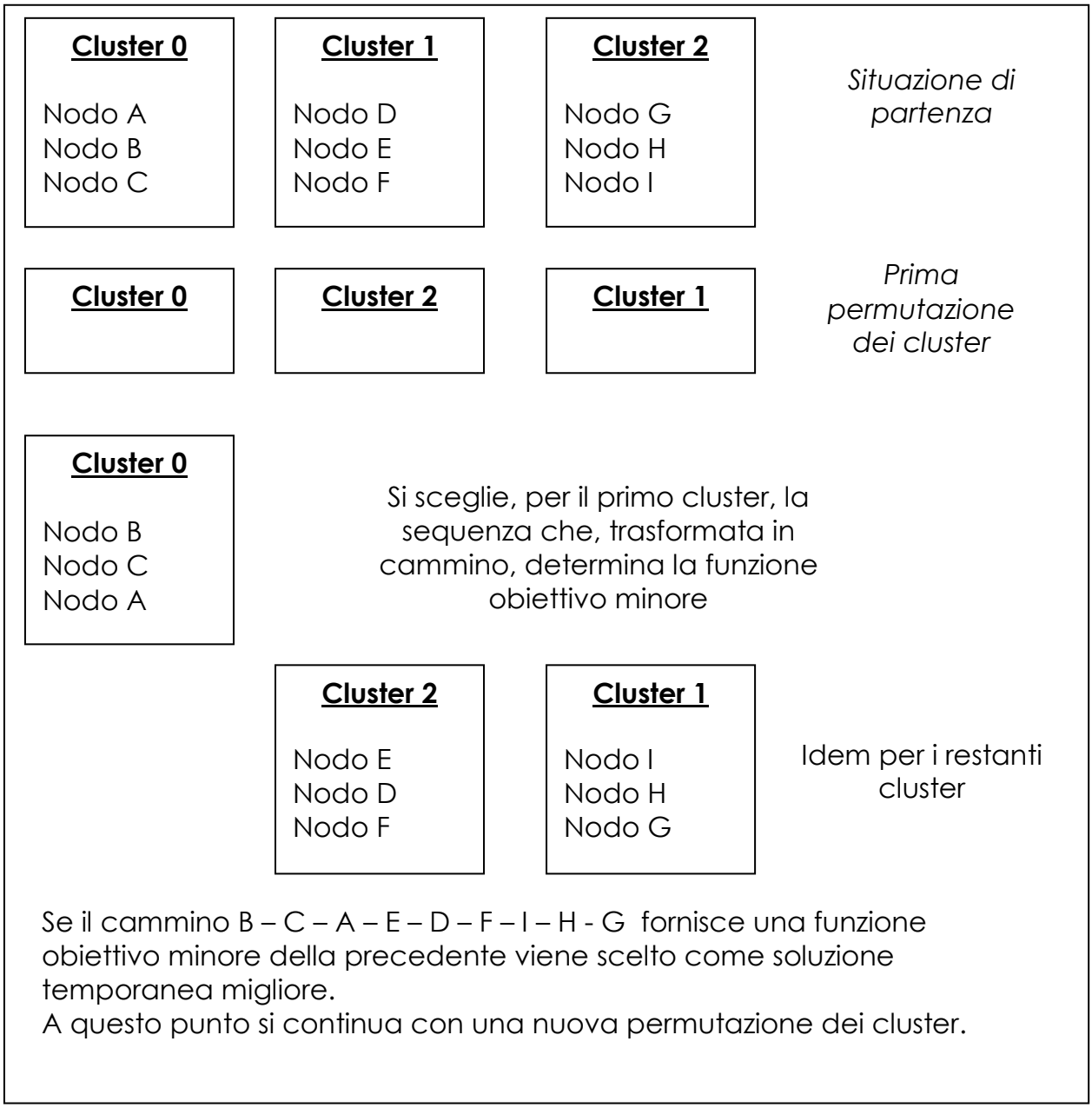
```
Procedure Nearest_neighbour (G, Vr)
  begin  for  j…numero_iniettori
         {N = crea_nodo
          for r…numero_richieste
               {
                 if  (Vr[r].id_iniettore = j)          //se l'id della richiesta è uguale
                       N.add_richiesta (Vr[r])         // la aggiunge al nodo
                  }
           G.add_nodo(N)                  // aggiunge il nodo al Grafo
          }

         V[p].inizializza()               // inizializza a zero il vettore cammino
          repeat
            {
             n = G.scegli_nodo()    // sceglie il nodo che decrementa
                                    // di più la funzione obiettivo
             V[p].add_nodo(n)       // lo aggiunge a vettore del cammino
             }
           until   V[p].hamiltoniano(G) = true  // finché non ho un cammino hamiltoniano
end
```

- Procedura *Nearest_neighbour (G, Vr )*-

Il nearest neighbour, in pratica, va a scegliere, per ogni passo, il nodo che incrementa di meno la funzione obiettivo fino a determinare un cammino hamiltoniano.
Un ulteriore ordinamento dei cluster e dei nodi all'interno di essi potrebbe essere quello di determinare tutte le possibili permutazioni tra loro. Questo sistema va prima di tutto a disporre, in tutti i modi possibili, i cluster disponibili,

poi, per ogni singola permutazione, vengono determinate tutte le possibili permutazioni all'interno dei cluster e scelta la sequenza che fornisce una funzione obiettivo minore. Vediamo con un esempio questo metodo.




Questo metodo ha la particolarità di essere computazionalmente molto costoso; analizzando infatti i dati in gioco possiamo vedere che il numero delle permutazioni da calcolare, e quindi della funzione obiettivo, risulta

$$\mathbf{k * k! * n_k!}$$

dove k è il numero di cluster ed $n_k$ il numero di nodi presenti nel relativo cluster k.

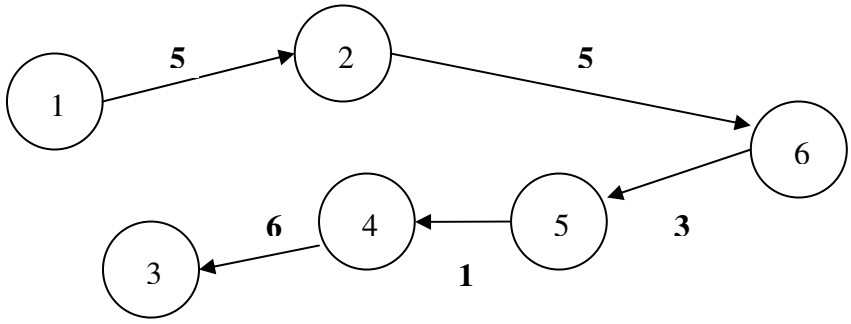
Per k=3 ed $n_1 = n_2 = n_3 = 7$, il numero totale di permutazioni sarà 90720, per k=6 arriviamo a 21772800.

Risulta quindi evidente che questo metodo, pur essendo più efficiente del nearest neighbour in termini di cammini analizzati, vada usato ogni volta valutandone la convenienza rispetto alle tempistiche di calcolo.

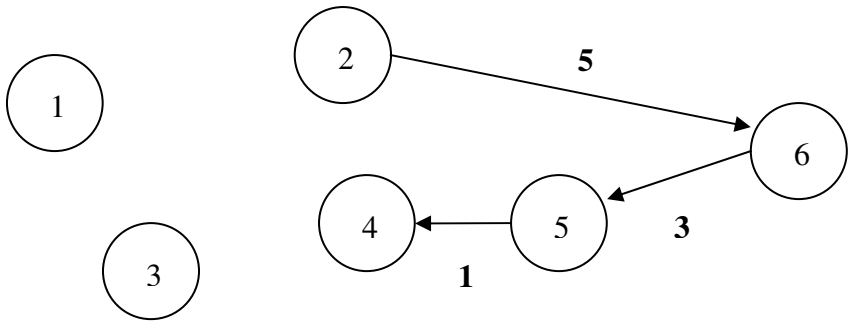
Una volta individuato un primo cammino hamiltoniano iniziale, è possibile definire un algoritmo che cerchi, una soluzione migliore. Per far ciò abbiamo proposto un algoritmo comunemente chiamato di *ricerca locale* che analizza soluzioni "vicine" alla soluzione corrente e le scarta se non producono una funzione obiettivo minore.

Nel caso in esame, l'algoritmo di ricerca locale, chiamato *2-scambio*, seleziona due archi del grafo non consecutivi, li modifica, scambiando l'ordine dei nodi adiacenti, e genera un cammino alternativo per i nodi restanti. Questo procedimento viene ripetuto ricorsivamente e termina al momento in cui sono state analizzate tutte le possibili coppie di archi.

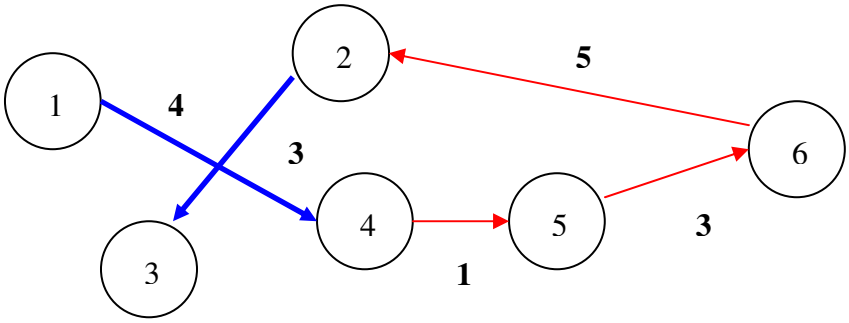
Per meglio capirne il funzionamento vediamone un esempio:




**1** - Il primo cammino produce un tempo di cambio tipo (somma delle etichette degli archi) di 20 unità di tempo.




**2**- Vengono eliminati gli archi 1-2 e 4-3




**3**- Vengono creati gli archi 1-4 e 2-3 e viene invertito il sottocammino 2-6-5-4. A questo punto il cammino ammissibile risultante ha tempo di cambio tipo 16, minore del precedente.

A livello esemplificativo è stata considerata una funzione obiettivo pari al solo tempo di cambio tipo in quanto non è possibile, in questo caso, conoscere i fabbisogni e le piazzature; la funzione obiettivo considerata risulta molto più complicata. E' molto probabile infatti che un tempo inferiore porti a cambi tipi in orari non permessi ed a piazzature molto distanti dal giorno della relativa domanda.

# Capitolo 7

## *Schema dell'algoritmo*

Per formulare l'algoritmo, si è tenuto conto del sistema di pianificazione fino ad ora usato dagli addetti alla logistica di produzione. È stata quindi introdotta una procedura denominata split_grafo() che divide a metà i nodi di ogni cluster creando, per i nodi residui, un nuovo cluster.
Se ad esempio abbiamo un cluster così composto:

**CLUSTER 0**
nodo 0 --> 3 skid
nodo 1 --> 1 skid
nodo 2 --> 4 skid

viene slittato in:

**CLUSTER 0**
nodo 0 --> 2 skid
nodo 1 --> 1 skid
nodo 2 --> 2 skid
**CLUSTER K**
nodo 0 --> 1 skid ..... (dal nodo 0 Cluster 0)
nodo 1 --> 2 skid ..... (dal nodo 2 Cluster 0)

I cluster vengono divisi per skid, nel caso di numeri dispari il sistema divide i nodi disponendo la parte maggiore nel cluster di partenza.
Questa situazione risulta molto simile ad uno scenario reale in cui ogni cluster rappresenta una famiglia di iniettori;

Utilizzando le funzioni fin qui introdotte, possiamo definire il seguente schema algoritmico:

1. Inizializzazione grafo

2. Creazione in k cluster

3. Calcolo del numero di permutazioni e della relativa stima del tempo di esecuzione. L'utente deciderà, a propria discrezione, se far procedere il sistema alla determinazione di una prima soluzione tramite nearest neighbour o tramite il calcolo delle permutazioni.

4. Splittaggio dei cluster tramite split_grafo()

5. Calcolo del numero di permutazioni e della relativa stima del tempo di esecuzione. L'utente deciderà, a propria discrezione, se far procedere il sistema alla determinazione di una prima soluzione tramite nearest neighbour o tramite il calcolo delle permutazioni.

6. Se dopo l'esecuzione di 4. e 5. la funzione obiettivo risulta peggiore del passo 3, ritorno alla situazione precedente

7. Aggregazione dei cluster in un cammino

8. Finché vi sono ct vietati o è possibile migliorare la funzione obiettivo
   a. splittaggio dei nodi che li introducono
   b. ricerca locale

# Capitolo 8

## *Strutture dati Shell*

La struttura dati della shell, dichiarata nel file allegato *shell.h*, rappresenta la parte incaricata di leggere e di tradurre tutte le informazioni necessarie al core per eseguire l'algoritmo. L'interazione col sistema SAP avviene esclusivamente tramite un file DAT di appoggio, un criterio che è risultato, dal confronto con gli addetti al settore, semplice e sicuro visto la criticità degli elementi che si vanno ad analizzare.
Di seguito sono rappresentate i contenitori fondamentali:

| ***iniettore*** |
|---|
| string name;<br>string codice;<br>string famiglia;<br>string vb;<br>string connettore;<br>int skid; |

La classe *iniettore* rappresenta la struttura dove caricare il tipo iniettore caratterizzato da nome, codice, famiglia, valve-body, connettore e capacità dello skid corrispondente.

| ***inietarray*** |
|---|
| vector<iniettore><br>array_inj; |

E' semplicemente un vettore di iniettori. Con questa classe si caricano tutti gli iniettori da prendere in considerazione per le operazioni.

| ***fabbi*** |
|---|
| iniettore in;<br>data d;<br>int pezzi; |

Introduce la struttura di un singolo fabbisogno definito dall'iniettore, dalla data e dal numero di pezzi. Questa classe viene anche usata per modello e, a seconda del contesto, identifica un backlog o una piazzatura.

| ***fabbisogni*** |
| --- |
| vector<fabbi> farray; |

Fabbisogni, così come *inietarray*, è semplicemente un vettore di fabbisogni.

| ***backlog*** |
| --- |
| vector<fabbi><br>back_array; |

Il vettore *backlog*, come introdotto sopra, utilizza il modello di *fabbi* per caricare un vettore di backlog. Viene usata soltanto per caricare i backlog del primo giorno del periodo fisso usati poi per calcolare il backlog complessivo.

Come si può vedere si è preferito l'uso dello standard *vector* al posto dei classici array, questo per avere maggior versatilità nelle operazioni di inserimento, rimozione e calcolo sui vettori.

# Capitolo 9

## *Funzioni Shell*

Le funzioni della shell non presentano particolari problemi in termini di accesso alla memoria o di costo computazionale, queste infatti vengono impiegate per la lettura e la scrittura di dati solamente nelle operazioni che precedono la vera e propria elaborazione dell'algoritmo affidata, come già detto, al core.

Vediamo le principali funzioni che operano sulla struttura dati della shell:

| ***iniettore*** |
|---|
| void setname(string name); // setta il nome |
| void setcodice(string co); // setta il codice |
| void setfamiglia(string fam); // setta la famiglia |
| void setvb(string v); // setta la valve body |
| void setconnettore(string con); // setta il connettore |
| void setskid(int sk); // setta lo skid |
| |
| string getname(); //restituisce il nome |
| string getcodice(); //restituisce il codice |
| string getfamiglia(); //restituiscela famiglia |
| string getvb(); //restituisce la v/b |
| string getconnettore(); //restituisce il connettore |
| int getskid(); //restituisce lo skid |
| int cambio_tipo(iniettore i); // ritorna le unità di tempo necessarie per il cambio tipo |

Questa classe non presenta particolari problemi visto che i metodi di lettura (contraddistinti da get) e quelle di scrittura (contraddistinti da set) hanno un accesso del tutto lineare alla struttura dati.
Il metodo cambio_tipo(iniettore i) restituisce la durata del cambio tipo con un altro iniettore i.

| ***inietarray*** |
| --- |
| void carica_iniet(); //carica l'inietarray da file |
| string injcod(string n); //ritorna il codice dato il nome |
| string injfam(string n); //ritorna la famiglia dato il nome |
| string injvb(string n); //ritorna la valve body dato il nome |
| string injcon(string n);//ritorna il connettore dato il nome |
| int injskid(string s); //ritorna lo skid dato il nome |
| void remove(string n); //rimuove da array_inj l'iniettore |
| void stampa_inietarray(); // stampa tutti i dati dell'iniettore |
| int is_equal(string st); //restituisce l'indice se trova un iniettore col solito nome altrimenti restituisce 1000 |
| iniettore what_inj(string n); // restituisce l'iniettore dato il nome |
| iniettore what_inj_ind(int n); // restituisce l'iniettore dato l'indice |
| vector<iniettore> get_vector(){return array_inj;}; //restituisce l'array |

La vettore *inietarray*, definito da un array di iniettori, serve principalmente per conoscere informazioni sui tipi di iniettori usati. Per far questo il metodo *carica_iniet()*le prende da un file precaricato.
I metodi di ricerca, vista la conformazione di vector in c++, sono anche in questo caso lineari.

| ***fabbi*** |
| --- |
| iniettore get_iniettore(); //restituisce l'iniettore |
| data get_data(); //restituisce la data |
| int get_pezzi(); //restituisce il numero dei pezzi |
| void set_pezzi(int p); //*setta i pezzi* |

Le operazioni anche in questo caso vengono fatte con semplici accessi alla memoria.

| ***fabbisogni*** |
| --- |
| void carica_fabb();//carica i fabbisogni da file<br>vector<fabbi> get_fabb(iniettore i); //dato il nome restituisce un vettore con tutti i fabbisogni<br>vector<fabbi> get_arr(); //restituisce l'array dei fabbi<br>vector<fabbi> get_by_fam(string fam , data d); //data la famiglia mi restituisce tutti i fabbisogni fino alla data d<br>vector<fabbi> get_by_con(string con , data d);//dato il connettore mi restituisce tutti i fabbisogni fino alla data d<br>vector<fabbi> get_by_ve(data d); //mi restituisce tutti i fabbisogni fino alla data d<br>int get_by_date(data d); //data la data d mi restituisce l'indice del primo fabbisogno che trova, -1 se non trova niente<br>void set_fabbi(int indice , int pezzi); //dato l'indice e i pezzi mi setta il fabbisogno |

Come *inietarray*, *fabbisogni* carica da file tutte le richieste dei clienti e le inserisce nel vettore *farray*; il costo degli accessi e delle operazione su quest'ultimo sono di ordine O(n).

| ***backlog*** |
| --- |
| void carica_back(); //carica i backlog da file<br>vector<fabbi> get_fabb(iniettore i);//dato il nome restituisce un vettore con tutti i backlog<br>vector<fabbi> get_array(); //ritorna il back_array |

La classe backlog prende da file l'arretrato di tutti i tipi di iniettore del periodo immediatamente precedente il primo giorno di periodo fisso.
Questo viene poi usato dal core per calcolare, in automatico, la quantità disponibile di prodotto distribuita su tutti i giorni del periodo fisso.

# Capitolo 10

## *Strutture dati Core*

Il core, come già descritto prima, è la parte principale del programma; visto l'ampio numero di calcoli e la grande quantità di dati, è stato strutturato in modo da leggere ed elaborare soltanto dati numerici che vengono tradotti da quelli forniti dalla shell.
Definiamo quindi quali saranno i dati di input da passare al core:

- **i = numero giorni del periodo fisso + 1 (per tenere conto della quantità disponibile di prodotto del giorno precedente a quello iniziale)**
- **k = istanti di tempo in un giorno k = 0 … 143**
- **$\alpha_i$ = peso del backlog per iniettore i**
- **$\beta_i$ = peso del backlog negativo per iniettore i**
- **j = numero iniettori (iniettori indicizzati da 0 a j-1)**
- **V[j] = vettore di j elementi (indice vettore) rappresentante la *durata* in termini di istanti di tempo della produzione di 1 skid**
- **M[ j , j ] = matrice di j * j elementi rappresentate le *durate* di cambio tipo in termini di tempo**
- **r = numero richieste nel periodo fisso**
- **Vr[r] = vettore delle richieste di r elementi ognuno dei quali rappresenta il *giorno* della richiesta (0 se prima dell'inizio del periodo fisso), l'*indice* dell'iniettore e il *numero di skid* richiesti**

- **Mc[ j , j ] = matrice di j * j elementi rappresentante l'ultimo *istante* di tempo disponibile in cui è possibile terminare il cambio tipo**

- **i-start = istante partenza programmazione**

- **g-start = giorno partenza programmazione**

Analogamente vengono definiti i dati di output che il core passerà alla shell dopo aver elaborato un'adeguata sequenza di produzione:

- **p = numero di piazzature**

- **V[p] = vettore di p elementi ognuno dei quali identifica l'*indice* dell'iniettore e il *numero di skid* da produrre (l'ordine del vettore rappresenta la sequenza di produzione finale)**

Per avere una struttura dati che riproduca fedelmente un grafo su cui applicare l'algoritmo, sono state create le seguanti classi :

| **class *nodo*** |
|---|
| skid sk ;<br>indice ind; |

Rappresenta, come dice il nome, un nodo del grafo. Viene definita dai due elementi: *sk*, che rappresenta la dimensione in skid del nodo e *ind* che identifica l'indice j del nodo. Skid e indice sono definiti come int.
Questa classe serve essenzialmente per storare tutte le richieste di ogni iniettore j dentro il solito contenitore.

| **class *grafo*** |
|---|
| vector<nodo> graph; |

Classe definita semplicemente da un vettore *graph* di nodi.

# Capitolo 11

## *Funzioni Core*

Analogamente a come è stato fatto per la shell, verranno descritte le principali funzioni del core, ognuna delle quali sarà ampiamente descritta per meglio capirne il funzionamento. Per una più ampia visione dei parametri passati alle funzioni e per evitare di ripeterli ogni volta nella descrizione delle stesse, iniziamo col descriverli nel dettaglio.

- **grafo g** ; grafo creato dall'aggregazione delle domande che verrà modificato solamente per le operazioni di split

- **vector<vector<int> > cluster** ; vettore di vettori di int che rappresenta una matrice di ordine k (numero cluster) per j (numero nodi per cluster). Gli elementi sono indici al grafo g.

- **vector<int> path** ; vettore di interi i cui elementi rappresentano gli indici del grafo g. La sequenza del path da 0 a path.size()-1 rappresenta il vero e proprio cammino sul grafo g. Gli elementi di questo vettore, così come quelli di cluster non vanno confusi con gli indici del nodo che identificano solamente il tipo di elemento storato.

- **vector<vector<int> > Vr** ; Vettore bidimensionale di interi in cui ogni riga rappresenta una richiesta composta da giorno i, indice iniettore j e quantità richiesta.

- **vector<int> V_skid**; Vector di interi che associa all'indice di V_skid la dimensione dello skid di quell'iniettore:
  es. V_skid[j]= 3250; ovvero la dimensione dello skid dell'iniettore j è 3240.

- **vector<vector<int> > m ;** Vettore bidimensionale di int le cui righe i e colonne j sono rappresentate dagli indici degli iniettori. Gli elementi di questa matrice indicano i tempi di cambio tipo tra i e j espressi in unità di tempo.

- **vector<vector<int> > Mc ;** Vettore bidimensionale di int le cui righe i e colonne j sono rappresentate dagli indici degli iniettori. Ogni elemento della matrice Mc indica l'ultima unità di tempo disponibile della giornata per cui è possibile effettuare cambio tipo fra i e j

- **skid min_nodo[j];** Array di j interi che rappresentano la dimensione minima, in termini di skid, che deve avere ogni singola piazzatura di ogni tipo di iniettore j. Ogni elemento di questo vettore viene automaticamente inizializzato a 1, l'utente ha la facoltà di settarlo a proprio piacimento.

- **double a[j];** Array di j double indicanti il parametro, per singolo iniettore, che serve per pesare il costo di backlog. Vettore settabile dall'utente.

- **double b[j];** Array di j double indicanti il parametro, per singolo iniettore, che serve per pesare il costo di backlog negativo. Vettore settabile dall'utente.

Veniamo ora alla descrizione delle funzioni principali del core discutendo come sono state implementate e le scelte algoritmiche usate per la loro realizzazione:

```
void inizializza_grafo(grafo &g, vector<vector<int> > &Vr , const vector<int> &V_skid ,
                       const vector<vector<int> > &m,  const vector<vector<int> > &Mc,istante i_start ,
                       giorno g_start, const double *a ,const double *b  );
```

*La funzione inizializza_grafo riceve in ingresso un grafo vuoto; prima di tutto raggruppa tutte le richieste di Vr per tipo in modo da avere tanti nodi quanti sono gli iniettori con almeno un fabbisogno attivo ed avere la massima aggregazione della domanda su di ogni nodo. In seguito i nodi vengono aggiunti al grafo g.*
*La funzione ha complessità O(n) in quanto per determinare i nodi esegue un'unica scansione del vettore Vr.*

```
void dividi_per_cluster(grafo &g,vector<vector<int> > &cluster, int ct ,
                                const vector<vector<int> > &m);
```

*Dividi_per_cluster riceve come parametro un grafo g già processato dalla precedente funzione di inizializzazione. L'algoritmo per la creazione dei cluster è il seguente:*

1. *Si seleziona il primo nodo*
2. *Viene creato un nuovo cluster vuoto ed inserito al suo interno il nodo*
3. *Si seleziona il prossimo nodo j, se viene trovato un nodo all'interno di un cluster che ha tempo di cambio tipo con j minore o uguale al parametro ct, si inserisce nel cluster relativo, altrimenti si riparte dal punto 2.*

*Questa funzione ha ordine lineare in quanto viene scandito una sola volta il grafo g e (per il controllo dell'appartenenza ai cluster) controllato soltanto il primo elemento di ogni cluster.*

```
void ordina_per_backlog(grafo &g ,vector<vector<int> > &cluster)   ;
```

*Questa funzione ordina semplicemente gli iniettori all'interno dei cluster per arretrato crescente ovvero viene data maggior priorità a quei nodi che presentano un maggior livello di domanda.*

```
int n_cambi_tipo( grafo &g , const vector<vector<int> > &cluster, istante i_start ,
                  const vector<vector<int> > &m,const vector<vector<int> > &Mc) ;
```

*Come dice il nome questa funzione determina il numero di cambi tipo vietati dato il cammino sui cluster. Tale percorso viene determinato aggregando i cluster secondo il loro indice:*

*Cluster 0 : A – B – C - D –E*
*Cluster 1 : F – G – H – I – L*

*Il cammino risultante è ovviamente A – B – C - D –E - F – G – H – I – L*
*Il numero di ritorno viene determinato calcolando il tempo di produzione di ciascun nodo e il proprio tempo di cambio tipo con il successivo*

```
int n_cambi_tipo( const int &half, grafo &g , const vector<int>  &path, istante i_start,
                  const vector<vector<int> > &m,const vector<vector<int> > &Mc)
```

*Altra versione di n_cambi_tipo che riceve come parametro il vettore del cammino path invece del cluster. Determina il numero di cambi tipo vietati dalla prima piazzatura del cammino fino al nodo con indice half non compreso*

```
int n_cambi_tipo( grafo &g ,const vector<int>  &path, istante i_start ,
                  const vector<vector<int> > &m,const vector<vector<int> > &Mc,
                   const int &half)
```

*Ultima versione di n_cambi_tipo simile alla precedente ma che calcola il numero di cambi tipo vietati sul vettore path dal nodo con indice half compreso sino alla fine del cammino.*

```
double check_back( grafo &g ,  const vector<vector<int> > &cluster,
                   const vector<vector<int> > &Vr  ,  istante  i_start,
                giorno g_start,const double *a,const double *b,const vector<vector<int> > &m);
```

*Questa funzione serve per calcolare parte della funzione obiettivo ed in particolare il valore relativo al costo di backlog ovvero il numero di iniettori i in backlog il giorno j.*

*Inizialmente crea le due matrici Kij (quantità disponibile iniettore j giorno i)*
*e Pij (piazzatura iniettore j giorno i) e le setta a 0.*
*La matrice dei fabbisogni viene poi determinata scandendo il vettore delle richieste.*

*Il passo successivo è quello di calcolare il cammino (cambi tipo compresi) sul grafo definito dal vector cluster e vedere in quale giorno vanno a cadere i nodi in modo da posizionarli nella matrice piazzature Pij.*
*A questo punto viene aggiornata la matrice Kij tramite Pij in questo modo:*
*Kij[i][z]=Kij[i-1][z]+Pij[i][z]-Kij[i][z]*
*ovvero per ogni giorno i la quantità disponibile di prodotto è uguale alla quantità disponibile del giorno i-1 più la piazzatura del giorno i meno la quantità residua del giorno i (nel caso di matrici inizializzate l'ultimo valore è zero).*
*I valori di Kij negativi, cioè che determinano un backlog non positivo, vengono moltiplicati per i rispettivi valori del vettore b.*
*Per finire viene calcolato il costo di backlog complessivo che è uguale alla sommatoria di tutti gli elementi della matrice per un coefficente a predefinito.*

```
double check_back( grafo &g , const vector<int>  &path,
                   const vector<vector<int> > &Vr  ,istante  i_start,
                 giorno g_start, const double *a, const double *b, const vector<vector<int> > &m);
```

Funzione identica alla precedente, ha solo la particolarità di usare, per la determinazione di un cammino su grafo, il vettore path anziché la matrice dei cluster

```
vector<double> check_back(giorno &half, grafo &g , const vector<int>  &path,
                   const vector<vector<int> > &Vr  ,istante  i_start,
                 giorno g_start, const double *a, const double *b, const vector<vector<int> > &m);
```

*Funzione per calcolare il costo di backlog, viene però passato per parametro, rispetto alla precedente versione, un nuovo valore half, giorno fino al quale calcolare la somma delle quantità disponibili. La funzione restituisce, anziché un unico valore, un vettore di j+1 elementi in cui i primi j sono la somma delle righe mentre l'ultimo elemento è il costo totale di backlog*

Es:

| | **giorno 0** | **giorno 1** | **giorno2** | **giorno 3** | **giorno 4** | **giorno 5** |
|---|---|---|---|---|---|---|
| **Inj A** | 3 skid | | 5 skid | | 3 skid | |
| **Inj B** | | 2 skid | | 3 skid | | 4 skid |
| **Inj C** | | 4 skid | | | 2 skid | 1 skid |

*Per half =2 viene calcolata solo la seguente matrice e restituito il vettore H:*

| | **giorno 0** | **giorno 1** | **giorno2** |
|---|---|---|---|
| **Inj A** | 3 skid | | 5 skid |
| **Inj B** | | 2 skid | |
| **Inj C** | | 4 skid | |

| **Vettore H** |
|---|
| 8 skid |
| 2 skid |
| 4 skid |
| $\sum$ costo backlog |

```
double check_back( grafo &g , const vector<int>  &path, const vector<vector<int> > &Vr,
                   istante  i_start,  giorno g_start,const double *a,const double *b,
                   const vector<vector<int> > &m, const int &half, const vector<double> &prec)
```

Funzione gemella che riceve come parametro, oltre agli elementi fondamentali, anche il vettore prec calcolato dalla precedente funzione. Chiariamo con l'esempio uguale a quello appena visto il suo funzionamento:

Es:

| | **giorno 0** | **giorno 1** | **giorno2** | **giorno 3** | **giorno 4** | **giorno 5** |
|---|---|---|---|---|---|---|
| **Inj A** | 3 skid | | 5 skid | | 3 skid | |
| **Inj B** | | 2 skid | | 3 skid | | 4 skid |
| **Inj C** | | 4 skid | | | 2 skid | 1 skid |

*Per half =2 viene calcolata solo la seguente matrice prendendo in considerazione il vettore prec per il calcolo complessivo del costo di backlog:*

| | **Vettore *prec*** | **giorno 3** | **giorno 4** | **Giorno5** |
|---|---|---|---|---|
| **Inj A** | 8 skid | | 3 skid | |
| **Inj B** | 2 skid | 3 skid | | 4 skid |
| **Inj C** | 4 skid | | 3 skid | 1 skid |

$\sum$ costo backlog

*Per il calcolo si procede determinando il valore di costo di backlog normalmente, sommandoci poi quello fornito dal vettore prec.*

*Va sicuramente aperta una parentesi per spiegare il motivo della scelta di calcolo del costo di backlog parziale: questo metodo, visto l'uso di una funzione dispendiosa in termini di costo computazionale come è la ricerca locale, permette di evitare il ricalcalo di un cammino precedentemente valutato, cosa che il 2-scambio effettua regolarmente.*
*L'uso di un vettore di interscambio tra le due funzioni risulta essere necessario dato che la determinazione del costo di backlog su una matrice parziale richiede la conoscenza delle quantità disponibili precedenti.*

```
double calcolo_fo( grafo &g, const vector<vector<int> > &cluster,
                   const vector<vector<int> >  &Vr, const vector<vector<int> > &m , const
                   vector<vector<int> > &Mc , const double *a,const double *b, istante i_start ,
                   giorno g_start)
```

*Funzione che determina il valore vero e proprio della funzione obiettivo del cammino specificato dal vettore cluster.*
*Per prima cosa calcola il massimo costo di backlog tramite la set_max_fo() (descritta in seguito), poi individua il numero di cambi tipo e costo di backlog tramite le relative funzioni, infine calcola il valore finale con secondo la formula già descritta ovvero:* ***ct vietati +(costo backlog /(max fo+1)).***

*Questo determina come risultato un numero decimale la cui parte intera rappresenta il numero di cambi tipo vietati, mentre la parte decimale il costo di backlog.*

```
double calcolo_fo( grafo &g, const vector<int>  &path,
                   const vector<vector<int> >  &Vr, const vector<vector<int> > &m , const
                   vector<vector<int> > &Mc , const double *a,const double *b, istante i_start ,
                   giorno g_start)
```

*Funzione identica alla precedente ma che usa il cammino path al posto del vettore dei cluster.*

```
double calcolo_fo( grafo &g,vector<int> &path, const vector<vector<int> > &Vr,
                   const vector<vector<int> > &m , const vector<vector<int> > &Mc ,
                   const double *a,const double *b, istante i_start , giorno g_start,
                   giorno &half, const vector<double> &prec)
```

*Questa versione di calcolo_fo() riceve in ingresso, rispetto alle altre, anche l'indicazione del giorno half e del vettore di appoggio prec. Questo per determinare la funzione obiettivo dell'intero cammino deducendola da prec, array precedentemente calcolato da check_back. La sequenza di istruzioni è la medesima delle funzioni già viste con l'eccezione che le chiamate a check_back() e n_cambi_tipo() inglobano i parametri per il calcolo parziale.*

```
void set_max_fo( grafo &g, vector<vector<int> > cluster, const vector<vector<int> > &Vr,
                 istante i_start , giorno g_start,  const double *a,const double *b,
                 const vector<vector<int> > &m )
```

*Questa funzione, già vista in calcolo_fo(), determina il massimo valore della funzione obiettivo. Per far ciò richiama semplicemente calcolo_fo() passando come parametro un cluster vuoto, questo per simulare una situazione in cui non viene prodotto nessun iniettore pur avendo richieste contenute in Vr.*

```
void set_max_fo( grafo &g, vector<int>  path, const vector<vector<int> > &Vr,
                 istante i_start , giorno g_start,  const double *a,const double *b,
                 const vector<vector<int> > &m )
```

*Seconda versione per il calcolo del massimo valore della funzione obiettivo. Rispetto alla precedente riceve come parametro un cammino path anziché un vettore cluster.*

```
void ordina_cluster( grafo &g, vector<vector<int> > &cluster, const vector<vector<int> > &Vr,
                     const vector<vector<int> > &m , const vector<vector<int> > &Mc,
                     const double *a, const double *b, istante i_start , giorno g_start);
```

*Questa funzione, al contrario di ordina_per_backlog() che opera soltanto sui nodi, ordina i cluster. Per determinare la sequenza non viene scelto un algoritmo di semplice ordinamento per dimensione, bensì vengono calcolate tutte le possibili permutazioni dei cluster mantenendo inalterata la sequenza dei nodi all'interno di essi; alla fine viene scelta la sequenza che produce una funzione obiettivo minore. Si è stabilito di usare questo metodo per il semplice fatto che il numero di cluster rimane sempre molto basso e quindi un uso delle possibili permutazioni, che è n fattoriale, risulta essere sempre fattibile ed efficiente.*

```
void nearest_neighbour(grafo &g,  vector<vector<int> > &cluster,
                       const vector<vector<int> > &Vr, const vector<vector<int> > &m,
                       const vector<vector<int> > &Mc , istante i_start , giorno g_start,
                       const double *a, const double *b)
```

*Funzione che determina la disposizione dei nodi all'interno di ogni cluster cercando, tramite l'algoritmo nearest neighbour, la sequenza di piazzature migliore.*
*Per far ciò opera su una sequenza di cluster individuata da ordina_cluster(), per ogni cluster sceglie ricorsivamente il nodo che introduce la funzione obiettivo minore.*

```
void permutazioni(grafo &g,  vector<vector<int> > &cluster, const vector<vector<int> > &Vr,
                  const vector<vector<int> > &m , const vector<vector<int> > &Mc ,
                   istante i_start , istante g_start, const double *a,const double *b)
```

*Funzione simile alla precedente, distribuisce però i nodi all'interno di ogni cluster provando tutte le possibili permutazioni e scegliendo quella che fornisce la minor funzione obiettivo. L'uso di permutazioni() è molto critico, è possibile infatti che la sua esecuzione provochi un pesante rallentamento nello svolgimento dell'algoritmo dovuto all'alto numero di elementi che può portare ad un livello molto elevato la quantità di permutazioni. Per questo motivo è stato previsto un controllo sull'algoritmo finale che fornisce numero di permutazioni e tempo stimato di esecuzione, ciò dà la possibilità all'utente di conoscere a priori la situazione per scegliere se è più conveniente questa funzione o la nearest_neighbour().*

```
void split_grafo(grafo &g, vector<vector<int> > &cluster, const vector<int> &V_skid,
                 const skid *min_nodo );
```

*Funzione di splittaggio nodi all'interno dei cluster.*
*Prova a dividere i nodi (> 1 skid) esattamente a metà, se la dimensione è pari, in modo sbilanciato se dispari(es per 5 skid --> 3-2). A questo punto viene creato, per ogni cluster, un nuovo cluster dove inserire i nodi splittati e vengono risettati i nodi del vecchio cluster.*
*Vediamo un esempio:*
*CLUSTER 0*
*nodo 0 --> 3 skid*
*nodo 1 --> 1 skid*
*nodo 2 --> 4 skid*

*viene slittato in:*

*CLUSTER 0*
*nodo 0 --> 2 skid*
*nodo 1 --> 1 skid*
*nodo 2 --> 2 skid*
*CLUSTER K*
*nodo 0 --> 1 skid ..... (dal nodo 0 Cluster 0)*
*nodo 1 --> 2 skid ..... (dal nodo 2 Cluster 0)*

*Per dare possibilità all'utente di avere un certo livello di controllo sullo splittaggio è stato definito un vettore di vincoli (in numero di skid),* ***min_nodo****, che indica quale è la quantità minima di cassoni che deve avere ogni piazzatura di ogni iniettore j. Il funzionamento è semplice, se la divisone di un nodo porta a due nuovi elementi entrambi maggiori di min_nodo, la funzione provvede a splittarli, altrimenti passa ad esaminare il successivo.*

```
void due_scambio(grafo &g, vector<int>  &path, const vector<vector<int> > &Vr,
                 const vector<vector<int> > &m , const vector<vector<int> > &Mc ,
                  istante i_start , giorno g_start, const double *a,const double *b  )
```

*Come dice il nome, questa funzione implementa il 2-scambio su di un cammino path. Ad ogni passo vengono selezionati i due archi non consecutivi necessari per l'esecuzione dell'algoritmo.*
*La particolarità di questa ricerca locale prevede che da un certo punto in poi, per ogni iterazione venga ricalcolata dall'inizio parte della funzione obiettivo sul di un cammino. Facciamo un esempio per capire il problema:*
*Sia dato il camino*
*0 – 1 – 2 – 3 – 4 – 5 – 6*
*La ricerca locale produce le seguenti iterazioni:*
*0 – 2 – 1 – 3 – 4 – 5 – 6*
*0 – 3 – 2 – 1 – 4 – 5 – 6*
*0 – 4 – 3 – 2 – 1 – 5 – 6  ecc.*
*Ad un certo passo ci troveremo nella situazione seguente:*
*0 – 1 – 2 – 3 – 5 – 4 – 6*
*0 – 1 – 2 – 3 – 6 – 5 – 4*
*Per questi ultimi due passi viene calcolata due volte la parte di funzione obiettivo relativa al cammino 0 – 1 – 2 – 3 con conseguente dispendio di risorse e di rallentamenti nell'esecuzione. Per questo, il 2-scambio qui implementato, fa uso della funzione, introdotta sopra, per il calcolo parziale della f.o.*

```
void due_scambio(grafo &g, vector<int>  &path, const vector<vector<int> > &Vr,
                 const vector<vector<int> > &m , const vector<vector<int> > &Mc ,
                  istante i_start , giorno g_start, const double *a,const double *b, int j )
```

*In questa funzione di 2-scambio è stato incluso un indice j che identifica un nodo specifico del cammino path. Questo algoritmo, a differenza del 2-*

*scambio classico opera solamente sulla coppia di archi (j - j+1 ; *) dove * rappresenta tutti i possibili archi del grafo, escluso (j - j+1) e quelli consecutivi. Questa funzione è stata introdotta specificatamente per cercare di ottimizzare il 2-scambio dopo lo splittaggio di un singolo nodo j che porta alla creazione del nuovo arco j – j+1. E' bene precisare che pur essendo molto più veloce, può non portare ai risultati sperati, ovvero ad un miglioramento della funzione obiettivo; per questo, durante l'esecuzione dell'algoritmo, verrà adoperato un controllo sui risultati ottenuti da questa funzione.*

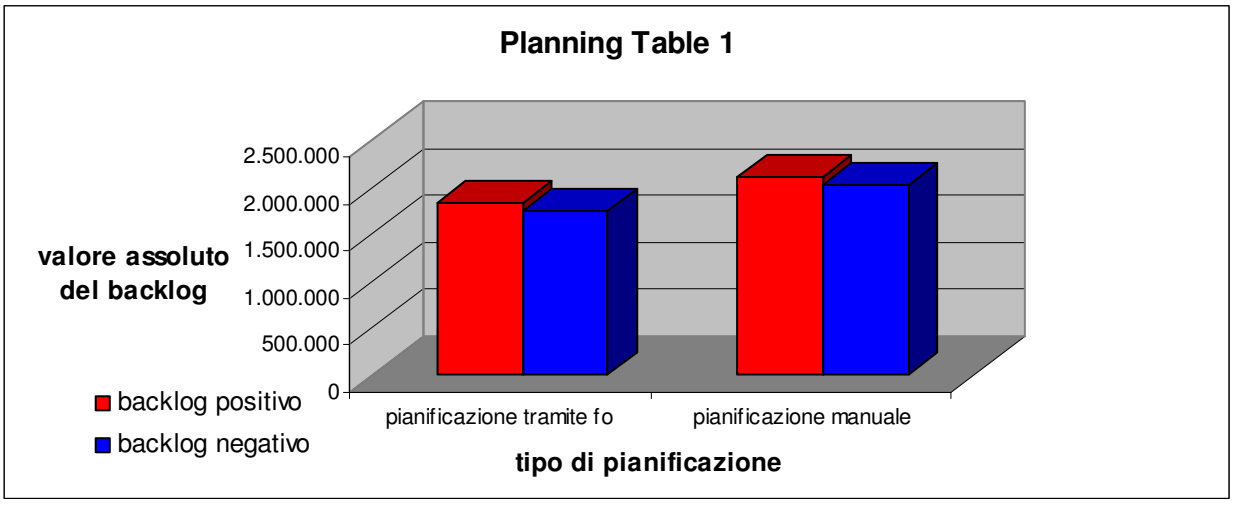
# Capitolo 12

## *Valutazione delle prestazioni*

Per valutare la bontà del modello elaborato, si è confrontata la soluzione da esso determinata con quella redatta dagli addetti alla pianificazione.
Il programma è stato testato in azienda prendendo in esame sei istanze i cui dati derivano direttamente da relative planning tables di un mese ciascuna.
La comparazione è avvenuta sia analizzando direttamente la sequenza di produzione ottenuta, sia confrontando il valore della funzione obiettivo prodotta dall'algoritmo con quello ricavato dalle tavole di pianificazione compilate dal reparto della logistica di produzione.
Dal confronto dei risultati, riassunto nelle seguenti tabelle, è emerso che le soluzioni elaborate dal modello, producono, in tutti i casi esaminati, risultati migliori, ossia che è stato apportato un miglioramento valutabile secondo i parametri che compongono la funzione obiettivo studiata per il problema.

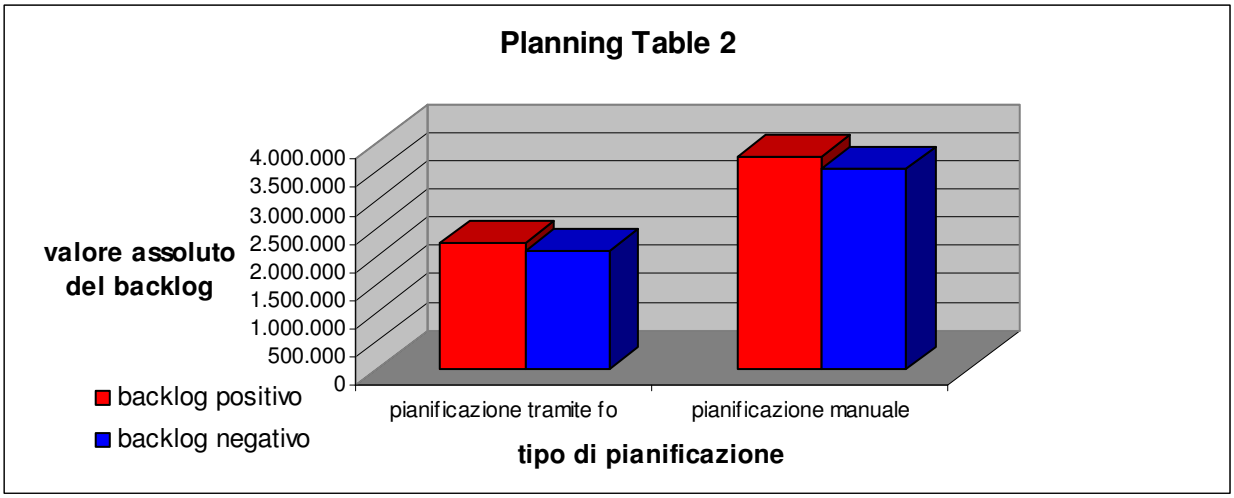
## Planning Table 1

| | |
|---|---|
| ***Tempo di esecuzione algoritmo*** | 2 minuti 43 secondi |
| ***Funzione Obiettivo*** | 0.209574 |
| ***Funzione Obiettivo di confronto*** | 0.241253 |
| ***Tipi di iniettori*** | 16 |
| ***Quantità pianificata*** | 430240 pezzi |
| ***Cambi tipo totali*** | 27 |
| ***Backlog positivo*** | 1831860 |
| ***Backlog negativo*** | -1747550 |
| ***Percentuale giorni in backlog*** | 36% |
| ***Backlog positivo di confronto*** | 2109500 |
| ***Backlog negativo di confronto*** | -2037060 |
| ***Percentuale giorni in backlog di confronto*** | 19% |



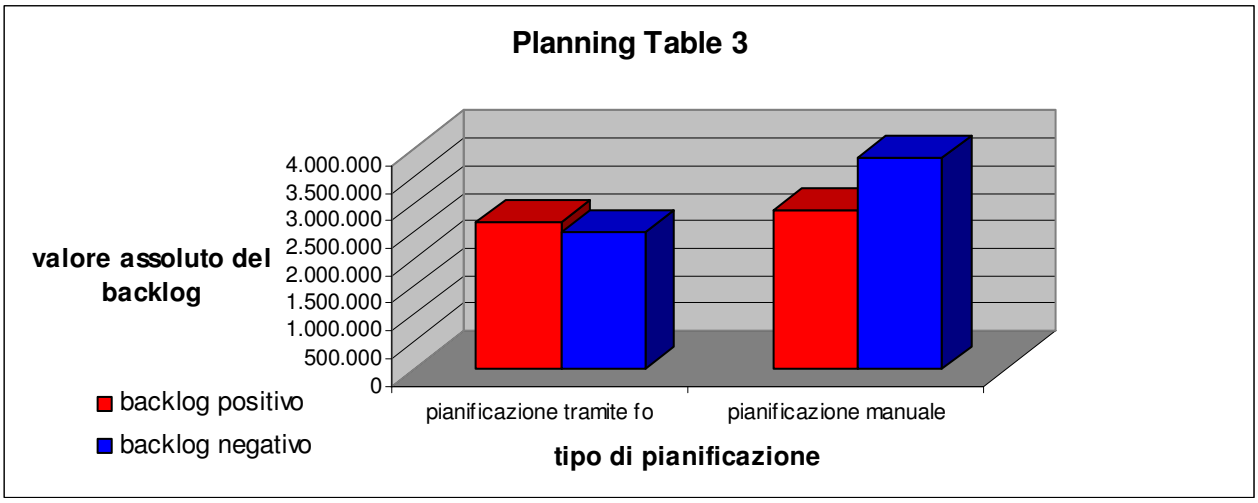
## Planning Table 2

| | |
|---|---|
| ***Tempo di esecuzione algoritmo*** | 2 minuti 32 secondi |
| ***Funzione Obiettivo*** | 0.191483 |
| ***Funzione Obiettivo di confronto*** | 0.321679 |
| ***Tipi di iniettori*** | 17 |
| ***Quantità pianificata*** | 506360 pezzi |
| ***Cambi tipo totali*** | 29 |
| ***Backlog positivo*** | 2240420 |
| ***Backlog negativo*** | -2086810 |
| ***Percentuale giorni in backlog*** | 33% |
| ***Backlog positivo di confronto*** | 3767320 |
| ***Backlog negativo di confronto*** | -3556020 |
| ***Percentuale giorni in backlog di confronto*** | 18% |



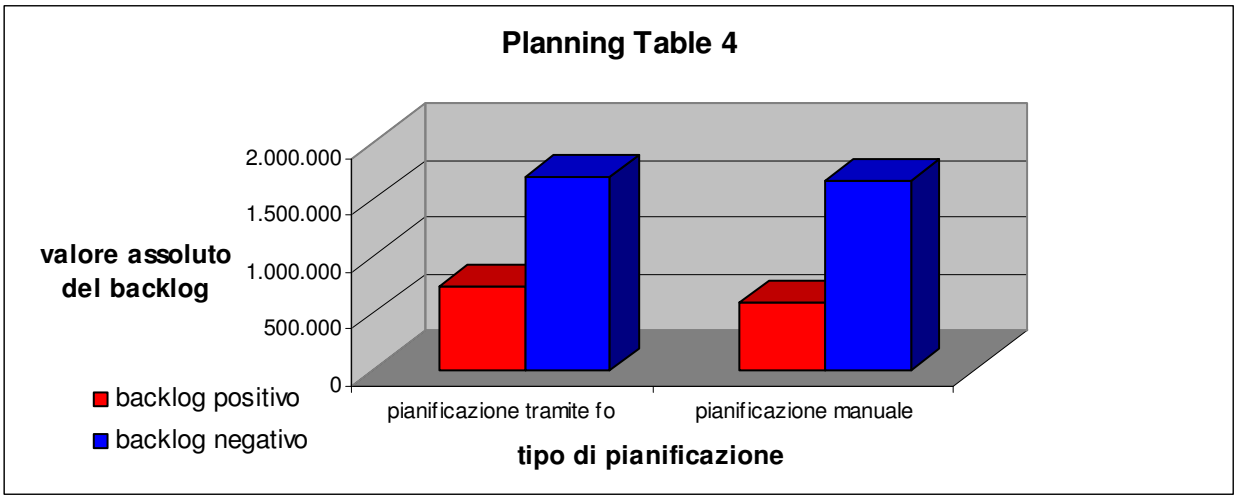
## Planning Table 3

| | |
|---|---|
| ***Tempo di esecuzione algoritmo*** | 8 minuti 50 secondi |
| ***Funzione Obiettivo*** | 0.217983 |
| ***Funzione Obiettivo di confronto*** | 0.236447 |
| ***Tipi di iniettori*** | 18 |
| ***Quantità pianificata*** | 527120 pezzi |
| ***Cambi tipo totali*** | 31 |
| ***Backlog positivo*** | 2672030 |
| ***Backlog negativo*** | -2504420 |
| ***Percentuale giorni in backlog*** | 39% |
| ***Backlog positivo di confronto*** | 2899690 |
| ***Backlog negativo di confronto*** | -3837760 |
| ***Percentuale giorni in backlog di confronto*** | 17% |



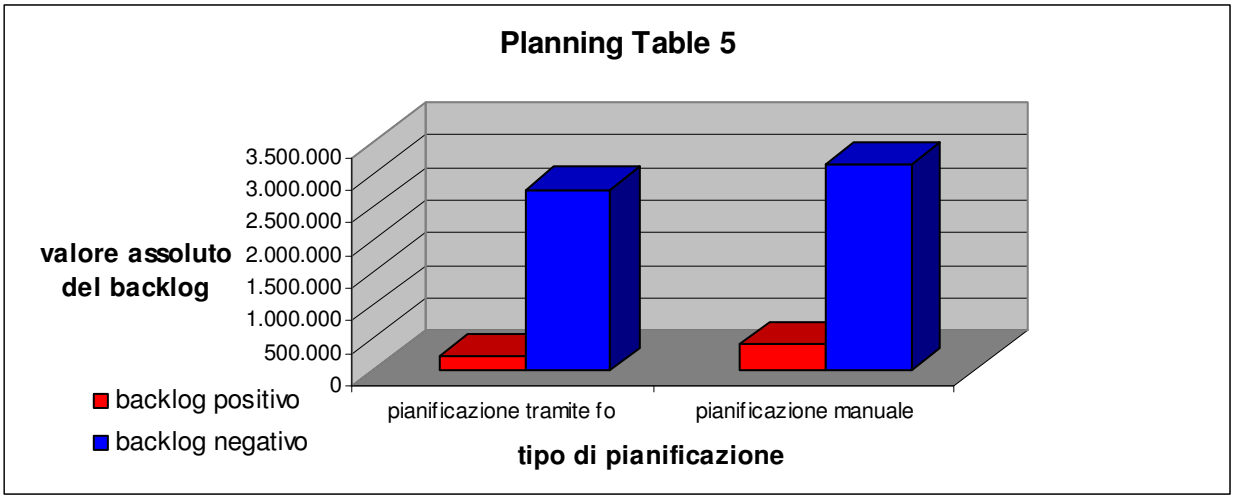
## Planning Table 4

| | |
|---|---|
| ***Tempo di esecuzione algoritmo*** | 11 minuti 23 secondi |
| ***Funzione Obiettivo*** | 0.153270 |
| ***Funzione Obiettivo di confronto*** | 0.165354 |
| ***Tipi di iniettori*** | 22 |
| ***Quantità pianificata*** | 316000 pezzi |
| ***Cambi tipo totali*** | 43 |
| ***Backlog positivo*** | 724201 |
| ***Backlog negativo*** | 1694450 |
| ***Percentuale giorni in backlog*** | 20% |
| ***Backlog positivo di confronto*** | 586846 |
| ***Backlog negativo di confronto*** | -1652650 |
| ***Percentuale giorni in backlog di confronto*** | 22% |



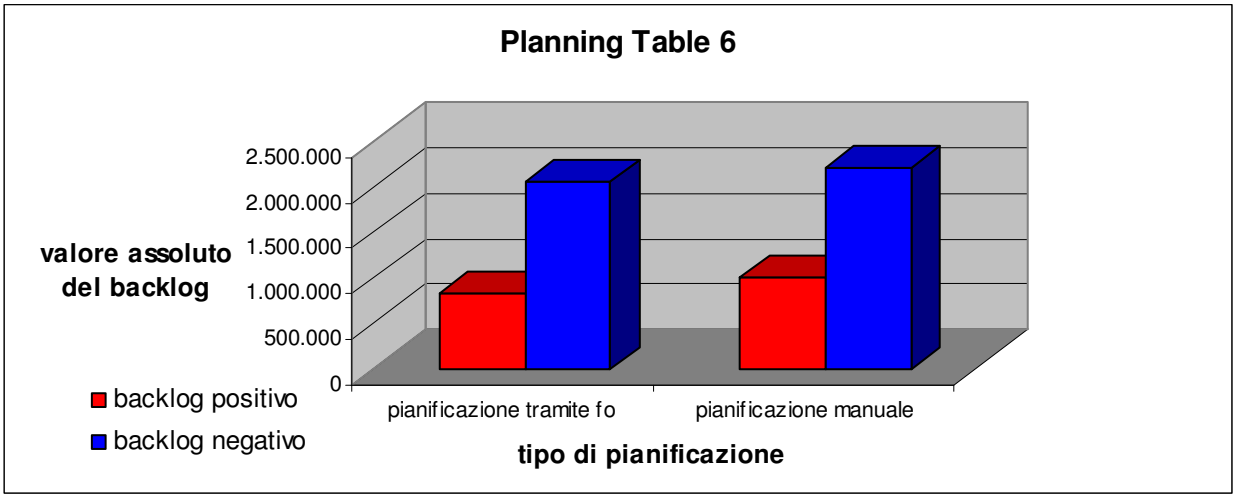
## Planning Table 5

| | |
|---|---|
| ***Tempo di esecuzione algoritmo*** | 0 minuti 44 secondi |
| ***Funzione Obiettivo*** | 0.0470328 |
| ***Funzione Obiettivo di confronto*** | 0.0906997 |
| ***Tipi di iniettori*** | 14 |
| ***Quantità pianificata*** | 334040 pezzi |
| ***Cambi tipo totali*** | 23 |
| ***Backlog positivo*** | 206431 |
| ***Backlog negativo*** | -2761770 |
| ***Percentuale giorni in backlog*** | 12% |
| ***Backlog positivo di confronto*** | 395814 |
| ***Backlog negativo di confronto*** | -3152330 |
| ***Percentuale giorni in backlog di confronto*** | 2% |



## Planning Table 6

| | |
|---|---|
| ***Tempo di esecuzione algoritmo*** | 2 minuti 04 secondi |
| ***Funzione Obiettivo*** | 0.0990778 |
| ***Funzione Obiettivo di confronto*** | 0.120885 |
| ***Tipi di iniettori*** | 16 |
| ***Quantità pianificata*** | 449960 pezzi |
| ***Cambi tipo totali*** | 36 |
| ***Backlog positivo*** | 823848 |
| ***Backlog negativo*** | -2046140 |
| ***Percentuale giorni in backlog*** | 24% |
| ***Backlog positivo di confronto*** | 1005060 |
| ***Backlog negativo di confronto*** | -2207320 |
| ***Percentuale giorni in backlog di confronto*** | 16% |



Oltre alla funzione obiettivo sono stati confrontati ulteriori elementi per valutare la conformazione delle soluzioni. Innanzitutto è stato posto in evidenza il fatto che funzioni obiettivo minori portino a backlog reali minori; ciò va a discapito del backlog negativo (quantità prodotta in anticipo) che risulta in tutti i casi minore rispetto alle soluzioni elaborate manualmente. Questo dato risulta molto importante perché dimostra che l'algoritmo è riuscito a minimizzare la quantità di pezzi in arretrato, diminuendo così i pezzi prodotti prima della relative richieste.
Il dettaglio delle tavole di pianificazione e delle sequenze di produzione proposte vengono allegate, vista l'impossibilità di inserirle in questa relazione, insieme ai files sorgenti del programma.
Da un confronto delle percentuali dei giorni in cui la domanda è in arretrato, emerge che il programma genera una soluzione in cui tale numero è maggiore rispetto a quello della pianificazione manuale. Il valore più alto di questo fattore non deve però tranne in inganno: benché la funzione obiettivo dia origine ad una quantità maggiore di giorni in arretrato, il valore totale degli arretrati stessi risulta in ogni caso più basso, perché è l'ammontare giornaliero dei singoli arretrati ad essere inferiore.

Per quanto riguarda l'analisi diretta della sequenza di produzione, i responsabili aziendali hanno avanzato alcune proposte per rendere la soluzione più aderente al loro attuale modello produttivo.

- Mentre l'algoritmo assume come punto di partenza una piazzatura, gli addetti hanno suggerito di iniziare la pianificazione dal cambio tipo che introduce alla nuova sequenza. Per far ciò, è quindi necessario tener conto di quale sia effettivamente l'ultimo iniettore prodotto.

- E' stato inoltre richiesto che all'interno di sottosequenze inerenti alla stessa famiglia, venga minimizzato anche il tempo di cambio tipo totale. In tal senso, è stato suggerito di introdurre una nuova funzione obiettivo che minimizzi tale quantità, da applicare solamente ai gruppi di famiglia che vengono rilevati al termine dell'esecuzione dell'algoritmo.

  Determinare una famiglia all'interno di una sequenza non risulta particolarmente difficoltoso a livello di core in quanto la famiglia stessa viene delimitata fra due cambi tipo "lunghi". Questo approccio può però introdurre nuovi cambi tipo vietati vanificando così la soluzione trovata in precedenza. Il migliore sviluppo a questo problema, è quello di individuare un parametro da aggiungere alla funzione obiettivo originaria, che sia in grado di dare un peso al tempo di cambio tipo totale. La valutazione di tale parametro andrà ricercata all'interno della strategia aziendale facendo riferimento al costo di cambio tipo, costo che non andrà più valutato esclusivamente in termini di tempo.

- Il modulo sviluppato considera i limiti di cambio tipo in maniera rigorosa, se ad esempio una piazzatura termina alle ore 22,10, questa viene considerata, al fine del calcolo della funzione obiettivo, fuori orario consentito. Va considerato il fatto che la produzione sulla linea non è sempre costante; come spiegato nel capitolo 3, questa può variare da 5300 a 5500 pezzi orari. Il programma, dal canto suo, tiene conto, per tutto l'arco della pianificazione, del valore preimpostato dall'utente senza prendere in esame variazioni della velocità di fabbricazione. E' stata pertanto avanzata la richiesta di creare un *limite di tolleranza* entro il quale un cambio tipo viene considerato valido. Tale limite andrà determinato non solo seguendo la capacità dell'impianto ma anche andando a studiare la rigorosità con cui vengono effettivamente fatti i cambi tipo rispetto alla planning table.

- Le pianificazioni, generate dal modulo sviluppato, riportano gruppi di famiglie che possono risultare molto piccole sia in termini temporali che di quantità. E' possibile, cioè, che vengano pianificati anche più cambi tipo famiglia nel medesimo giorno andando così ad aumentare il numero globale di tali cambio tipo. Ciò è dovuto, più che dalla funzione obiettivo, dalla conformazione dell'algoritmo ed in particolare dall'aggregazione delle richieste e del successivo splittaggio in singoli nodi. Questo nuovo approccio alla pianificazione si discosta dallo schema abitualmente utilizzato in azienda che cerca di minimizzare il backlog mantenendo fisso il numero di cambi tipo famiglia. Ciò ha indotto gli addetti a consigliare di inserire un controllo sul numero di cambi tipo famiglia effettuati per ricreare le stesse modalità operative da loro utilizzate.

  A tal proposito, è stato fornito un limite minimo, ovvero quello di non creare gruppi di famiglie che abbiano dimensione inferiore ai 32000 pezzi (2 giorni lavorativi).

  L'introduzione di questi nuovi vincoli non ha una precisa giustificazione teorica ma servirebbe a ricalcare in maniera più fedele gli attuali orientamenti aziendali.

Concludendo, il modello elaborato è stato ritenuto, da parte dei responsabili della logistica di produzione, come aderente ai sistemi di gestione aziendale ed in grado di soddisfare le problematiche poste dagli obiettivi del tirocinio stesso. Le soluzioni risultanti dal programma si sono rivelate una buona base di partenza per lo sviluppo di uno strumento di pianificazione professionale capace di soddisfare le loro esigenze produttive.