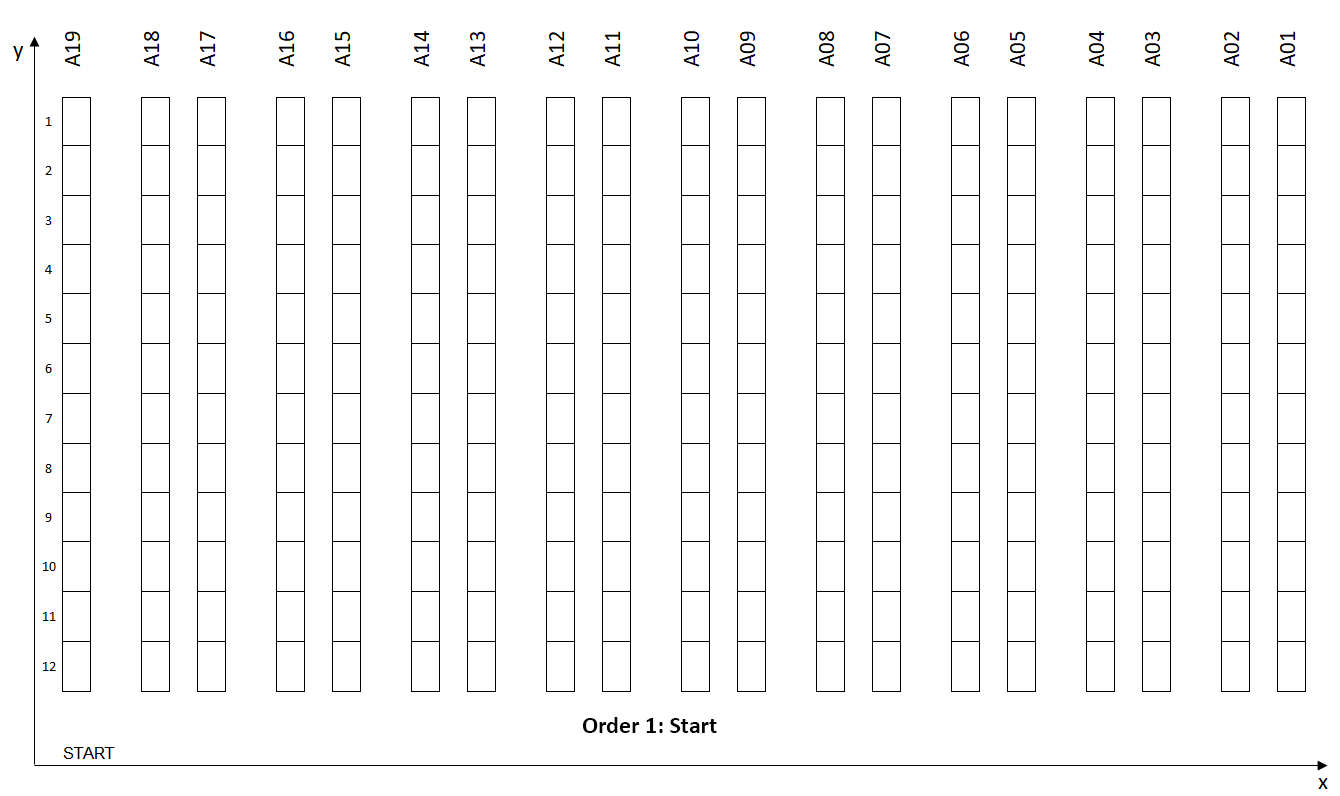
**Improve Werahouse Productivy: using Order Batching (Parte 1)**

*Progettare un modello di simulazione per stimare l'impatto di diverse strategie di Single Picker Routing Problem nella Picking Productivity.*

In un centro di distribuzione (DC), il tempo di percorrenza da un luogo all'altro durante il percorso di prelievo può rappresentare dal 60% al 70% dell'orario di lavoro dell'operatore.

Ridurre questo tempo di percorrenza è il modo più efficace per aumentare la produttività complessiva del DC.



*Figura 1: SCENARIO 1 - Picking routes with 1 order picked per wave*

1. **Obiettivo**

Questo lavoro ha l’obiettivo di determinare come progettare un modello per simulare l'impatto di diversi processi di prelievo e metodi di instradamento per l’ottimale order picking utilizzando il **Single Picker Routing Problem (SPRP)** per un modello di magazzino bidimensionale (asse-x, asse-y).

SPRP è un'applicazione specifica del problema generale del **Commesso Viaggiatore (TSP)**.

TSP risponde alla seguente domanda:

*"Dato un elenco di storage locations e le distanze tra ciascuna coppia di locations, qual è il percorso più breve possibile per visitare ciascuna posizione di stoccaggio e tornare al deposito?"*

SPRP viene utilizzato per determinare il percorso minimo nel processo di prelievo per preparare uno o più ordini.

1. **Che cos’è il ‘Wave Picking’?**

Per questo studio, viene impiegato l'esempio di un E-Commerce tipo DC in cui gli articoli sono immagazzinati in scaffali a 4 livelli. Questi scaffali sono organizzati in più righe (Riga#: 1 … n) e corridoi (Corridoio#: A1,…,A\_n).

**Premesse**:

Immagine che contiene carretto, trasporto

Descrizione generata automaticamente

*Figura 2: Werahouse Picking Carts*

1. **Dimensione degli articoli**: Articoli di dimensioni piccole e leggere;
2. **Carrello di prelievo:** Carrello di prelievo leggero con una capacità di 10 ordini;
3. **Percorso di prelievo**: Il percorso di prelievo inizia e finisce nella stessa posizione.

Lo Scenario 1 (*Figura 1*), il peggiore in termini di produttività, può essere facilmente ottimizzato grazie alle seguenti:

* Locations: Gli ordini n. 1 e n. 2 hanno posizioni di prelievo comuni;
* Zone: Gli ordini hanno punti di prelievo in una zona comune;
* Ordini a riga singola: L'efficienza items\_picked/walking\_distance è molto bassa.

Immagine che contiene testo, armadietto

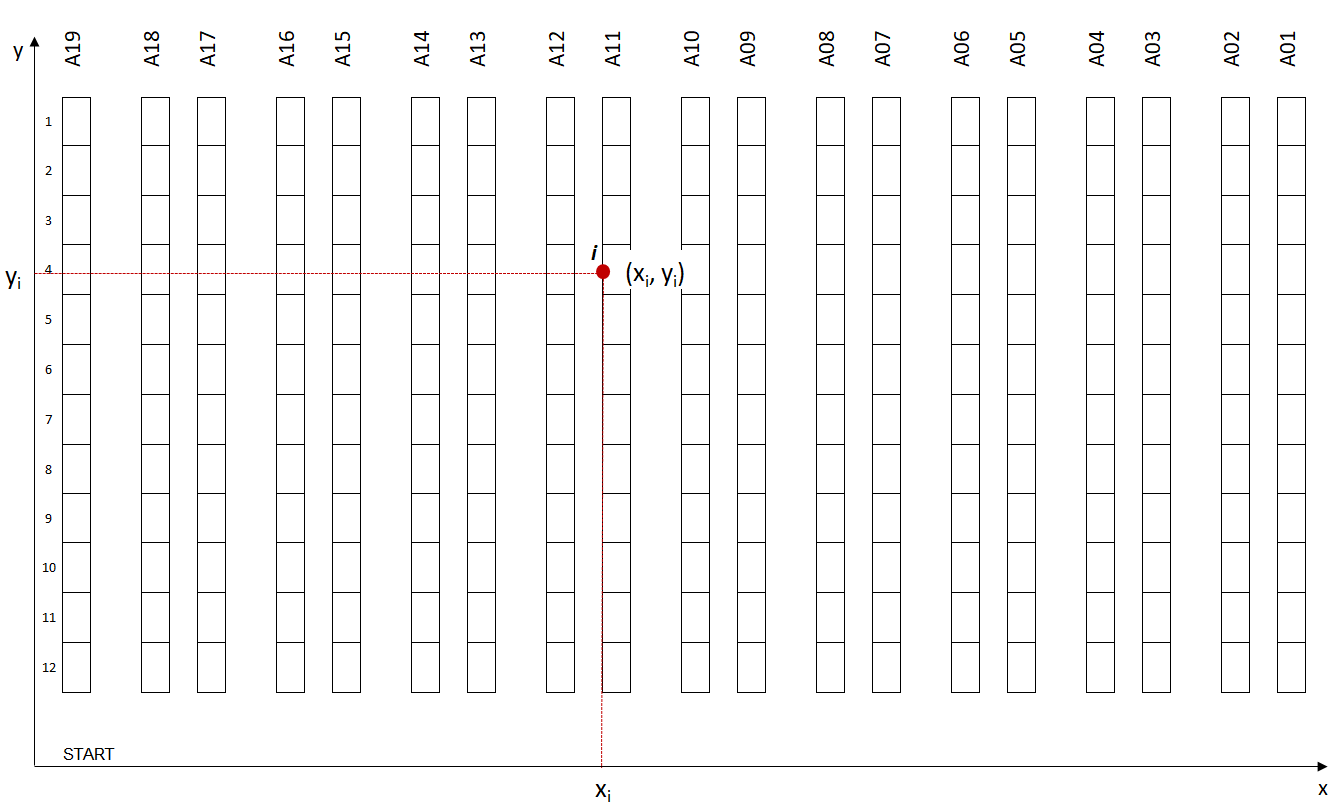
Descrizione generata automaticamente

*Figura 3: SCENARIO 2 – Wave Picking applicata allo Scenario 1*

Il primo modo intuitivo per ottimizzare questo processo è combinare questi tre ordini in un unico percorso di prelievo: questa strategia è comunemente chiamata **Wave Picking**.

Quindi, come detto in precedenza, quello che si andrà a fare è di costruire un modello per simulare l'impatto di diverse strategie di Wave Picking sulla distanza totale percorsa per uno specifico insieme di ordini da preparare.

1. **Test di diversi Algoritmi di Ottimizzazione**
   1. **Layout di magazzino con mappatura delle posizioni di stoccaggio**

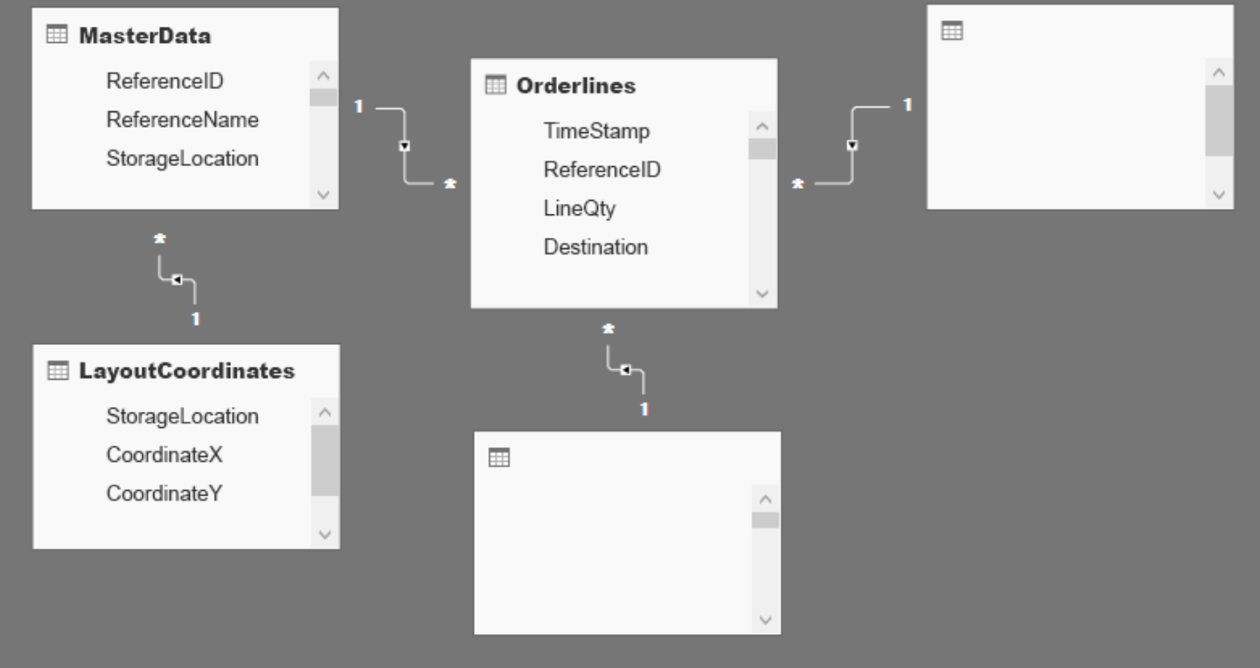


*Figura 4: Warehouse Layout con coordinate 2D*

In base al layout effettivo del magazzino, le posizioni di stoccaggio vengono mappate con coordinate 2-D (x, y) che verranno utilizzate per misurare la distanza percorsa.

Ogni posizione di stoccaggio è collegata a un riferimento utilizzando i dati anagrafici. (Ad esempio, il riferimento #123129 si trova nella coordinata (xi, yi)). È quindi possibile collegare ogni riga dell'ordine a una posizione geografica per il prelievo.

* 1. **Righe d'ordine dal tuo Warehouse Management System (WMS)**



*Figura 5: Database Schema*

Le righe di OrderLines possono essere estratte dal database WMS e questa tabella deve essere unita (join) alla tabella MasterData per collegare ogni riga ordine a un'ubicazione di stoccaggio e alle relative coordinate (x, y) nel magazzino.

È possibile aggiungere ulteriori tabelle per includere più parametri nel modello come (Destinazione, Tempi di consegna, Imballaggio speciale, etc.).

* 1. **Funzioni per il calcolo della distanza della Picking Route**
* **Funzione 1:** calcola la distanza tra due picking locations.



*Figura 6: Route differenti fra 2 storage locations nel magazzino*

La funzione 1 verrà utilizzata per calcolare la walking distance da un punto i(xi, yi) a un punto

j(xj, yj).

**Obiettivo**: restituire la walking distance più breve tra i due potenziali routes dal pnt i al pnt j.

**Parametri**:

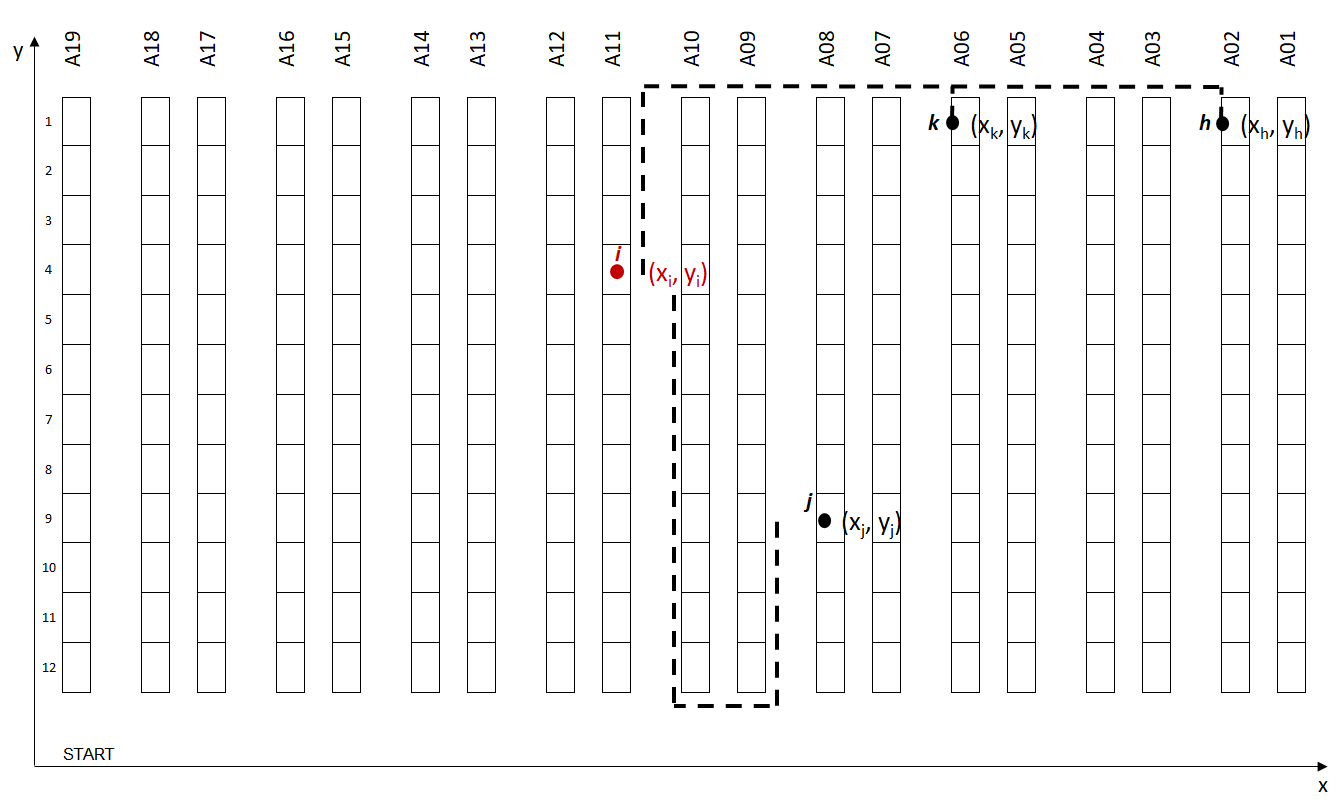
* + **y\_low** – il punto più basso dell’alley (Asse y)
  + **y\_high** – il punto più alto dell’alley (Asse y)

Codice Python:

1. import numpy as np
2. import pandas as pd
3. import ast
4. from ast import literal\_eval

7. def distance\_picking(Loc1, Loc2, y\_low, y\_high):
8. '''Calcolare Picker Route Distance fra 2 locations'''
9. # Punto di Partenza
10. x1, y1 = Loc1[0], Loc1[1]
11. # Punto di Destinazione
12. x2, y2 = Loc2[0], Loc2[1]
13. # Distanza asse x
14. distance\_x = abs(x2 - x1)
15. # Distanza asse y
16. **if** x1 == x2:
17. distance\_y1 = abs(y2 - y1)
18. distance\_y2 = distance\_y1
19. **else**:
20. distance\_y1 = (y\_high - y1) + (y\_high - y2)
21. distance\_y2 = (y1 - y\_low) + (y2 - y\_low)
22. # Distanza minina sull'asse y
23. distance\_y = min(distance\_y1, distance\_y2)
24. # Distanza Totale
25. distance = distance\_x + distance\_y
26. **return** **int**(distance)

* **Funzione 2:** la prossima location più vicina.



*Figura 7: Lo scenario della successiva Storage Location*

Questa funzione verrà utilizzata per scegliere la location successiva tra diversi candidati per continuare il picking route.

**Obiettivo**: restituire la posizione più vicina come miglior candidato.

Codice Python:

1. **def** next\_location(start\_loc, list\_locs, y\_low, y\_high):
2. '''''Trovare la closest next location'''
3. # Distanza per ogni prossimo punto candidato
4. list\_dist = [distance\_picking(start\_loc, i, y\_low, y\_high) **for** i **in** list\_locs]
5. # Distanza Minima
6. distance\_next = min(list\_dist)
7. # Location della Distanza Minima
8. index\_min = list\_dist.index(min(list\_dist))
9. next\_loc = list\_locs[index\_min]
10. list\_locs.remove(next\_loc)
11. **return** list\_locs, start\_loc, next\_loc, distance\_next

* **Funzione 3**: crea il picking route e calcola la walking distance totale.

Questa funzione viene usata per creare il picking route da una serie di ordini da preparare.

* + *Input*: un elenco di (x, y) locations in base agli elementi da prelevare per questo percorso.
  + *Output*: una sequenza ordinata di locations coperti e la walking distance totale.

Codice Python:

1. **import** pandas as pd
2. **import** numpy as np
3. **import** itertools
4. **from** ast **import** literal\_eval
5. **from** utils.routing.distances **import** \*
7. **def** create\_picking\_route(origin\_loc, list\_locs, y\_low, y\_high):
8. '''''Calcolare la distanza totale per coprire per una lista di locations'''
10. # Variabile per la distanza totale
11. wave\_distance = 0
12. # Location corrente variabile
13. start\_loc = origin\_loc
14. # Memorizzare i routes
15. list\_chemin = []
16. list\_chemin.append(start\_loc)
18. **while** len(list\_locs) > 0: # Looping fino a quando tutte le locations vengono scelte
19. # Verso la prossima location
20. list\_locs, start\_loc, next\_loc, distance\_next = next\_location(start\_loc, list\_locs, y\_low, y\_high)
21. # Update start\_loc
22. start\_loc = next\_loc
23. list\_chemin.append(start\_loc)
24. # Update distanza
25. wave\_distance = wave\_distance + distance\_next
27. # Distanza finale dall'ultima storage location dall'origine
28. wave\_distance = wave\_distance + distance\_picking(start\_loc, origin\_loc, y\_low, y\_high)
29. list\_chemin.append(origin\_loc)
31. **return** wave\_distance, list\_chemin
    1. **Funzioni per creare gli orders waves**

* **Funzione 1:** crea batches di n ordini da raccogliere contemporaneamente.
  + *Input*: order lines dataframe (df\_orderlines), numero di orders per wave(orders\_wave).
  + *Output*: dataframe mappato con il numero di wave (Column : WaveID), il numero totale di waves (waves\_number).

Codice Python:

1. **def** orderlines\_mapping(df\_orderlines, orders\_number):
2. '''''Mapping orders con il numero di wave '''
3. df\_orderlines.sort\_values(by='DATE', ascending = True, inplace = True)
4. # Lista dei Numeri univoci degli order
5. list\_orders = df\_orderlines.OrderNumber.unique()
6. dict\_map = dict(zip(list\_orders, [i **for** i **in** range(1, len(list\_orders))]))
7. # Order ID mapping
8. df\_orderlines['OrderID'] = df\_orderlines['OrderNumber'].map(dict\_map)
9. # Raggruppo per Orders by Wave del orders\_number
10. df\_orderlines['WaveID'] = (df\_orderlines.OrderID%orders\_number == 0).shift(1).fillna(0).cumsum()
11. # Conto il numero di Waves
12. waves\_number = df\_orderlines.WaveID.max() + 1
13. **return** df\_orderlines, waves\_number

* **Funzione 2:** listing delle wave\_ID picking route.
  + *Input*: order lines dataframe (df\_orderlines) e numero di wave ( WaveID ).
  + *Output*: lista delle locations i(xi,yi) incluse nel picking route.

Codice Python:

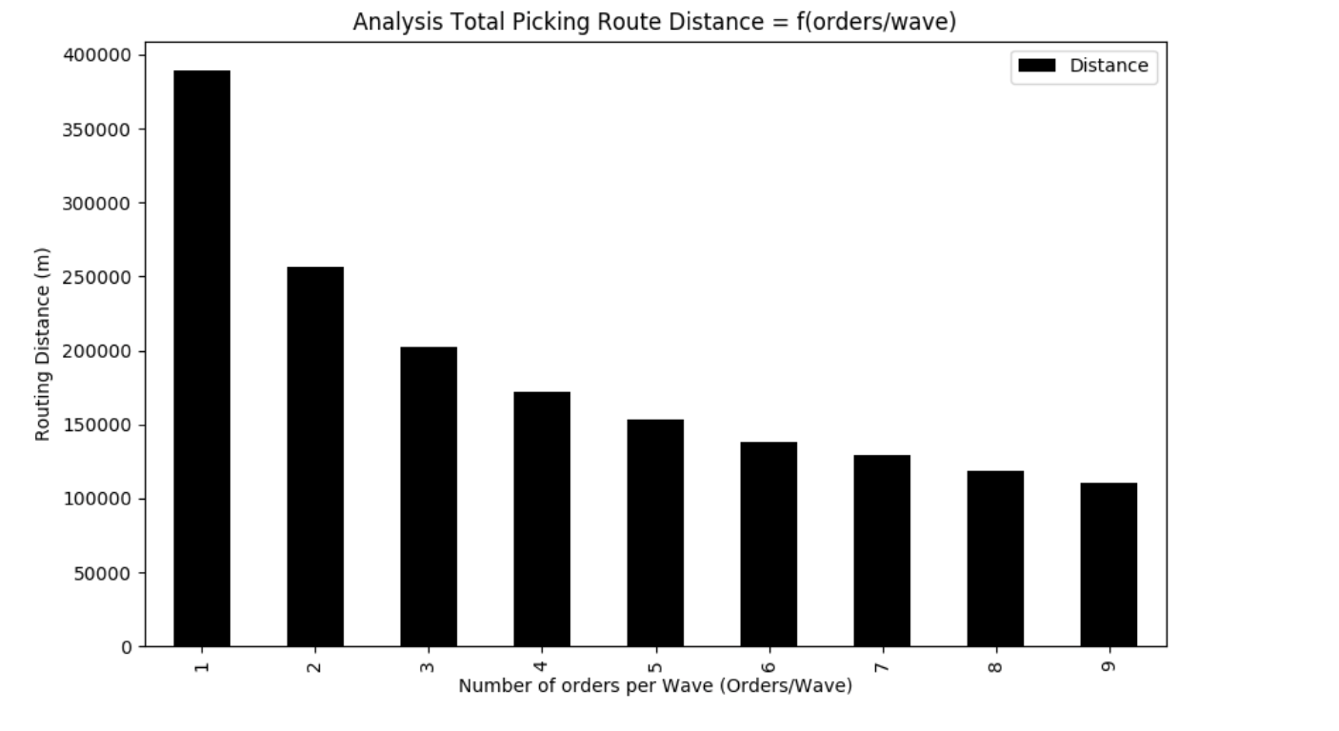
1. def locations\_listing(df\_orderlines, wave\_id):
2. '''Get le storage locations per coprire una wave di orders'''
3. df = df\_orderlines[df\_orderlines.WaveID == wave\_id]
4. # Creo un elenco di coordinate
5. list\_locs = list(df['Coord'].apply(lambda t: literal\_eval(t)).values)
6. list\_locs.sort()
7. # Lista delle coordinate univoche
8. list\_locs = list(k **for** k,\_ in itertools.groupby(list\_locs))
9. n\_locs = len(list\_locs)
10. **return** list\_locs, n\_locs
11. **Risultati e Prossimi Steps**

Dopo aver impostato tutte le funzioni necessarie per misurare la picking distance, ora si puó testare la strategia del picking route con le picking order lines.

* **Orders** **Waves**: gli ordini sono raggruppati per ordine cronologico del tempo di ricezione da OMS (TimeStamp);
* **Picking** **Route**: la strategia del picking route segue la logica della location più vicina successiva.

Per stimare l'impatto della strategia di wave picking sulla produttività, si eseguono diverse simulazioni con un numero graduale di ordini per wave:

1. **Misura totale della Walking distance**: quanta distanza percorribile viene ridotta all'aumentare del numero di ordini per percorso?
2. **Record Picking Route per Wave**: registrazione della sequenza di locations per route per ulteriori analisi.



*Figura 8: Risultati di 5000 order lines con un rapporto da 1 a 9 per route*

Questa soluzione è ben lungi dall'essere la più ottimale, i prossimi step saranno:

1. Gli ordini possono essere raggruppati per geographical clusters di picking locations per ridurre la walking distance dei picker.
2. La strategia next closest Location ha i suoi limiti che possono essere facilmente evidenziati dai picking route records.

**Improve Werahouse Productivity: using Spatial Clustering (Parte 2)**

1. **Due livelli di Ottimizzazione**

Nella parte 1 quello che si è cercato di fare è di stimare la picking route walking distance totale per un set di orders, usando:

1. **Werahouse Mapping**: Collegare ciascuna linea di ordine del database con le coordinate di picking location(x,y) nel magazzino;
2. **Calcolo della Distanza**: funzione che calcola la walking distance da 2 picking location

Inoltre, nella parte 1, si è deciso anche di adottare un approccio semplice per le seguenti:

1. **Picking Route Design**: data la possibilità di scegliere tra diverse picking locations, il werahouse picker sceglierà sempre di andare alla più vicina (Next Closest Location Strategy);
2. **Order** **Waving**: orders sono ordinati e raggruppati in waves in base al tempo di ricezione (TimeStamp).

Come si riesce a migliorare?

Sicuramente esistono algoritmi più complessi che permettono un miglioramento, ma esistono anche delle soluzioni semplici che ottimizzano l’algoritmo.

1. **Order Waves usando Picking Locations Clustering**

Gli ordini a riga singola hanno il vantaggio di trovarsi in un'unica location di stoccaggio; il raggruppamento di più ordini a riga singola per cluster può garantire che il picker rimanga in una zona delimitata e, quindi, non faccia troppi spostamenti.

Dove si trovano gli ordini a riga singola?

Immagine che contiene testo

Descrizione generata automaticamente

*Figura 1: Order Lines DataFrame*

**Funzione**: Calcolo del numero di ordini a riga singola per location di stoccaggio (%)

Codice Python:

1. # Creare un Pivot Table Orderlines a secondo della location
3. **def** lines\_locations(df\_orderlines, filename):
5. # Groupby
6. df\_count = pd.DataFrame(df\_orderlines.groupby(['Alley\_Number', 'Cellule'])['Coord'].count()).reset\_index()
7. df\_pivot = pd.pivot\_table(df\_count, values='Coord', index=['Cellule'],
8. columns=['Alley\_Number'], aggfunc=np.sum).fillna(0)
9. # Totale
10. lines\_total = df\_pivot.sum().sum()
11. df\_pivot = df\_pivot/lines\_total
12. df\_pivot.to\_excel(filename)
14. **return** df\_pivot

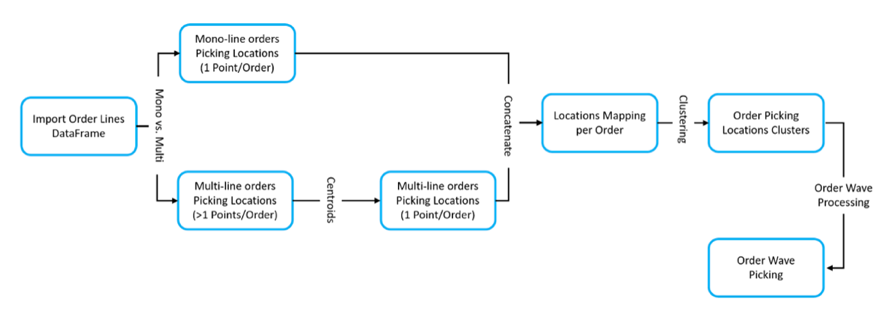
Immagine che contiene tavolo

Descrizione generata automaticamente

*Figura 2: Distribuzione di righe di ordini a riga singola per posizione di stoccaggio: 5.000 righe di ordine (%)*

**Nota:** si prende come riferimento l’esempio di distribuzione sopra

* **Scope**: 5.000 righe d'ordine per 23 corridoi;
* **Ordini a riga singola:** 49% degli ordini situati nei vicoli A11, A10 e A09.
  1. ***Picking locations clustering using Scipy***



*Figura 3: Order Lines Processing for Order Wave Picking using Clustering by Picking Location*

**Idea**: **Picking Location Clustering**

Raggruppamento delle picking locations per clusters per ridurre la walking distance per ciascuna picking route. (Esempio: la massima walking distance fra 2 locations è <15).

Il clustering spaziale è il task di raggruppare insieme un set di punti in modo che gli oggetti nello stesso cluster siano più simili tra loro rispetto agli oggetti in altri cluster.

Immagine che contiene testo, termometro, dispositivo

Descrizione generata automaticamente

*Figura 4: Esempio di tre Picking Location Clusters*

Qui, la metrica di somiglianza sarà la walking distance da una posizione all'altra.

Ad esempio, si vogliono raggruppare le locations assicurandosi che la massima walking distance tra due locations sia di 10 m.

*Euclidian Distance vs. Walking Distance*

Per il modello specifico definito, non si possono usare metodi di raggruppamento convenzionali utilizzando la distanza euclidea. In effetti, la walking distance (usando la funzione: distance\_picking) è diversa dalla distanza euclidea.

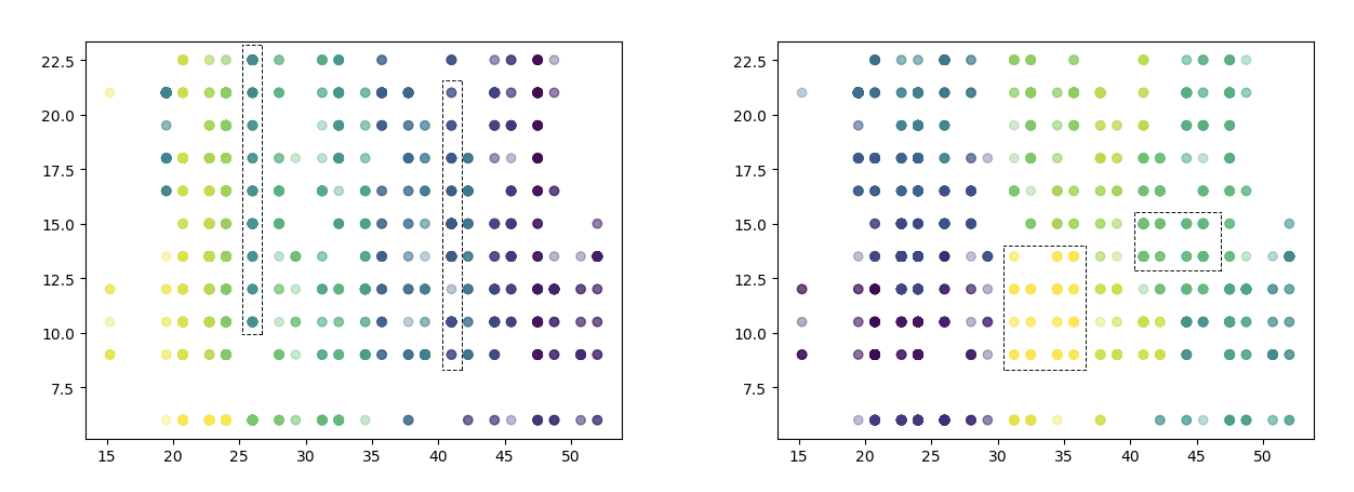
Come si evince dalla *Figura 5,* la distanza Euclidea fra i (xi,yi) e i 2 punti: p (x\_p,y\_p) e j (x\_j,y\_j) sono uguali. Invece, se si compare il picker Walking distance, p (x\_p,y\_p) è più vicino.

Per questo modello, Picker Walking Distance è la metrica specifica che si va ad usare e con l’obiettivo di ridurre al minimo. Pertanto, l'algoritmo di clustering dovrebbe utilizzare una distance\_walking personalizzata per prestazioni migliori.



*Figura 5: Esempio di Walking vs. Euclidian Distance*

**Esempio**: Locations Clustering entro 25 m di distanza (5.000 righe d'ordine)



*Figura 6: A SX: Clustering usando la Walking Distance | A DX: Clustering usando la Euclidian Distance*

L'esempio a sinistra che utilizza la Walking Distance è il raggruppamento di locations all'interno della stessa corsia riducendo la pciking route distance; mentre l'esempio a destra può raggruppare posizioni che coprono più corridoi.

**Funzione**: Clusters for Single Line Orders using Walking Distance

Per un set di ordini, le righe estraggono ordini a righe singole (df\_orderlines) e creano clusters di storage locations entro una distanza (dist\_method) usando una funzione di distanza custom (dist\_method).

Si espone di seguito il **Codice Python**, implementato usando le funzioni *ward* e *fcluster* di Scipy per creare clusters di Picking Location usando una metrica *distance\_func* (walking distance).

Codice Python:

1. **from** scipy.cluster.vq **import** kmeans2, whiten
2. **from** scipy.spatial.distance **import** pdist
3. **from** scipy.cluster.hierarchy **import** ward, fcluster
4. **from** calculate\_distance **import** dist\_func
5. **from** pre\_processing **import** mono\_order

8. **def** clustering\_loc(df, distance\_threshold, dist\_method, orders\_number,
10. ''''' Step 1: Clustering + Mapping'''
12. wave\_start, clust\_start, df\_type):
14. # 1. Create Clusters
15. list\_coord, list\_OrderNumber, clust\_id, df = cluster\_wave(df, distance\_threshold, 'custom', clust\_start, df\_type)
16. clust\_idmax = max(clust\_id) # Last Cluster ID
18. # 2. Mapping Order lines with clusters
19. dict\_map, dict\_omap, df, Wave\_max = lines\_mapping\_clst(df, list\_coord, list\_OrderNumber, clust\_id, orders\_number, wave\_start)
21. **return** dict\_map, dict\_omap, df, Wave\_max, clust\_idmax
23. **def** cluster\_wave(df, distance\_threshold, dist\_method, clust\_start, df\_type):
25. '''''Step 2: Creo waves by clusters'''
27. # Creo colonne per Clustering
28. **if** df\_type == 'df\_mono':
29. df['Coord\_Cluster'] = df['Coord']
30. # Mapping points
31. df\_map = pd.DataFrame(df.groupby(['OrderNumber', 'Coord\_Cluster'])['SKU'].count()).reset\_index() # Here we use Coord Cluster
32. list\_coord, list\_OrderNumber = np.stack(df\_map.Coord\_Cluster.apply(**lambda** t: literal\_eval(t)).values), df\_map.OrderNumber.values
33. # Cluster picking locations
34. clust\_id = cluster\_locations(list\_coord, distance\_threshold, dist\_method, clust\_start)
35. clust\_id = [(i + clust\_start) **for** i **in** clust\_id]
36. # List\_coord
37. list\_coord = np.stack(list\_coord)
38. **return** list\_coord, list\_OrderNumber, clust\_id, df
40. **def** cluster\_locations(list\_coord, distance\_threshold, dist\_method, clust\_start):
42. ''''' Step 3: Creo clusters per la location (x,y)'''
44. # Creo la linkage matrix
45. **if** dist\_method == 'euclidian':
46. Z = ward(pdist(np.stack(list\_coord)))
47. **else**:
48. Z = ward(pdist(np.stack(list\_coord), metric = distance\_picking\_cluster))
49. # Single cluster array
50. fclust1 = fcluster(Z, t = distance\_threshold, criterion = 'distance')
51. **return** fclust1

**Funzione**: Single Line Orders Mapping with ClusterID

Per un insieme di ordini, le righe estraggono gli ordini a righe singole (df), l'ID del cluster e il numero degli ordini, questa funzione consente di mappare il dataframe con l'ID del cluster per la creazione della wave.

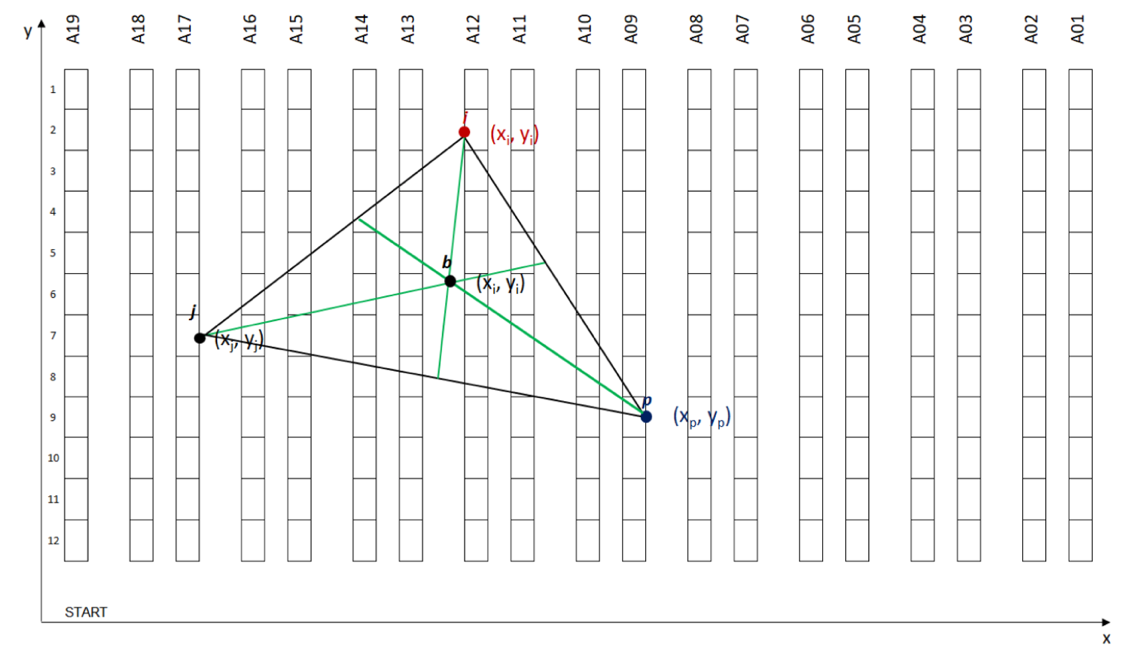
1. **def** lines\_mapping\_clst(df, list\_coord, list\_OrderNumber, clust\_id, orders\_number, wave\_start):
3. '''''Step 4: Mapping Order lines mapping with clustering '''
5. # Dictionnary for mapping by cluster
6. dict\_map = dict(zip(list\_OrderNumber, clust\_id))
7. # Dataframe mapping
8. df['ClusterID'] = df['OrderNumber'].map(dict\_map)
9. # Order by ID and mapping
10. df = df.sort\_values(['ClusterID','OrderNumber'], ascending = True)
11. list\_orders = list(df.OrderNumber.unique())
12. # Dictionnary for order mapping
13. dict\_omap = dict(zip(list\_orders, [i **for** i **in** range(1, len(list\_orders))]))
14. # Order ID mapping
15. df['OrderID'] = df['OrderNumber'].map(dict\_omap)
16. # Create Waves: Increment when reaching orders\_number or changing cluster
17. df['WaveID'] = wave\_start + ((df.OrderID%orders\_number == 0) | (df.ClusterID.diff() != 0)).shift(1).fillna(0).cumsum()
19. wave\_max = df.WaveID.max()
20. **return** dict\_map, dict\_omap, df, wave\_max

* 1. ***Picking locations clustering for Multi-line Orders***

**Funzione:** Centroide per ogni multi-line order

A differenza degli ordini a riga singola, gli ordini a più righe (multi-line) possono coprire diverse picking locations. Tuttavia, possiamo applicare la stessa metodologia applicata ai centroidi delle storage locations.

**Esempio**: Ordine con 3 linee coperto da 3 differenti picking locations



*Figura 7: Centroide di tree Picking Locations*

1. **def** centroid(list\_in):
2. '''''Centroid function'''
3. x, y = [p[0] **for** p **in** list\_in], [p[1] **for** p **in** list\_in]
4. centroid = [round(sum(x) / len(list\_in),2), round(sum(y) / len(list\_in), 2)]
5. **return** centroid

8. **def** centroid\_mapping(df\_multi):
9. '''''Mapping Centroids'''
10. # Mapping multi
11. df\_multi['Coord'] = df\_multi['Coord'].apply(literal\_eval)
12. # Raggrupo le coordinate per ordine
13. df\_group = pd.DataFrame(df\_multi.groupby(['OrderNumber'])['Coord'].apply(list)).reset\_index()
14. # Calcolo i Centroid
15. df\_group['Coord\_Centroid'] = df\_group['Coord'].apply(centroid)
16. # Definisco un dizionario per il mapping
17. list\_order, list\_coord = list(df\_group.OrderNumber.values), list(df\_group.Coord\_Centroid.values)
18. dict\_coord = dict(zip(list\_order, list\_coord))
19. # Final mapping
20. df\_multi['Coord\_Cluster'] = df\_multi['OrderNumber'].map(dict\_coord).astype(str)
21. df\_multi['Coord'] = df\_multi['Coord'].astype(str)
22. **return** df\_multi

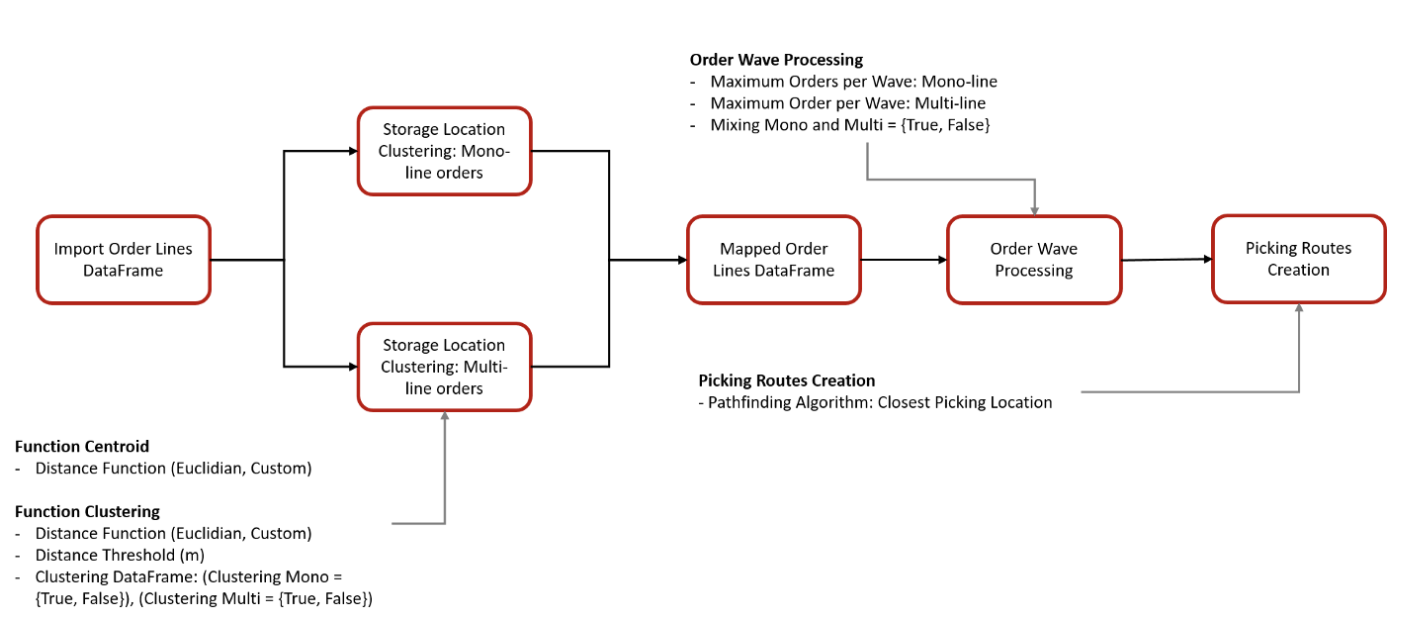
Dopo aver utilizzato questa funzione, si torna alla situazione degli ordini a riga singola con un singolo punto (x, y) per ordine.

Tornando a quel contesto, si può, quindi, applicare il clustering a questi punti cercando di raggruppare l’oridne per zona geografica con la condizione di massima distanza.

1. **Model Simulation**

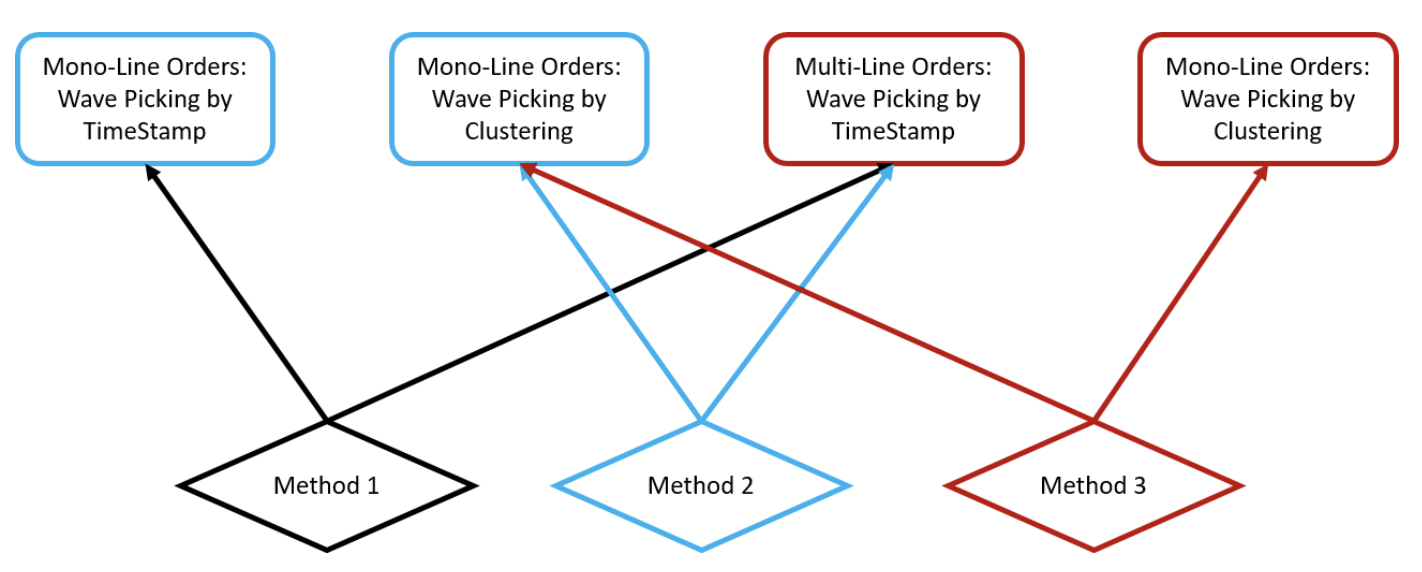
Per riassumere, la costruzione del modello, riportata nel grafico sottostante, si hanno diversi steps prima di scegliere la creazione di percorsi utilizzando l'elaborazione delle onde.

Ad ogni passaggio, abbiamo un set di parametri che possono essere regolati per migliorare le prestazioni:



*Figura 8: Model Costruction with Parameters*

* 1. ***Confronto fra 3 metodi di Wave Processing***



*Figura 9: 3 Metodi per Wave Processing*

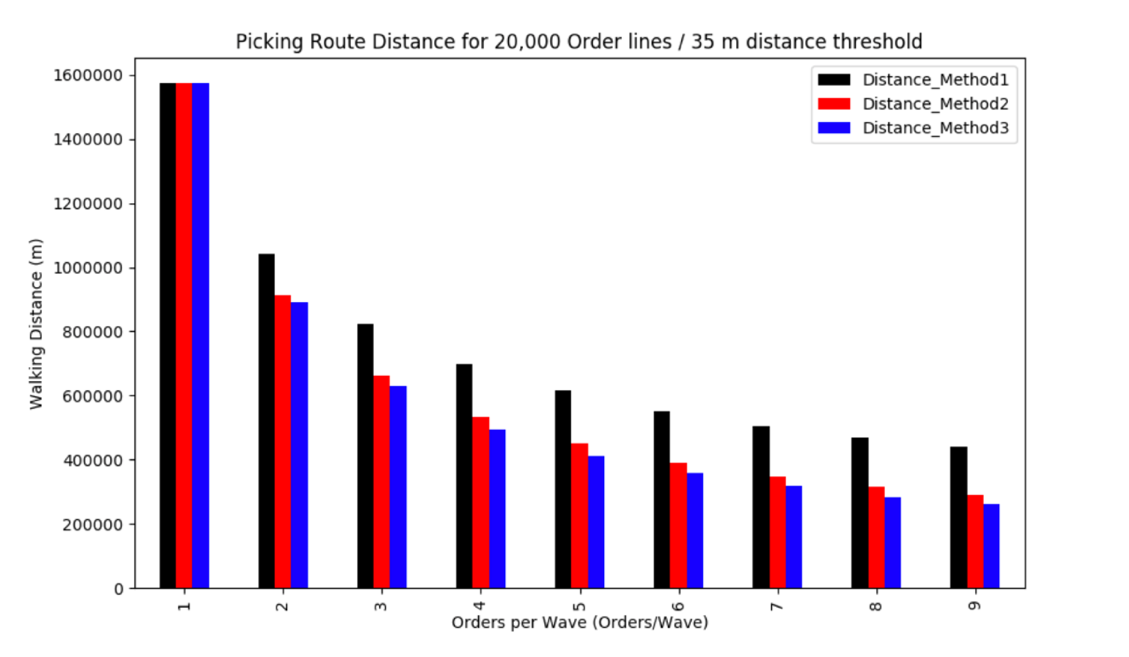
Quello che si va a fare è valutare l’impatto dell’elaborazione Order Wave by clusters di picking locations sulla walking distance totale.

Questo viene fatto testando tre diversi metodi:

* **Metodo 1**: non si applica il clustering (es. Scenario iniziale);
* **Metodo 2**: si applica il clustering solo agli ordini a Riga Singola;
* **Metodo 3**: si applica il clustering agli ordini a Riga Singola e ai Centroidi degli ordini multilinea.

Scenario della Simulation:

* **Order Lines**: 20,000 Lines;
* **Distance Threshold**: Distanza Massima fra 2 picking locations (distance\_threshold = 35 m);
* **Orders per Wave**: orders\_number in [1,9].

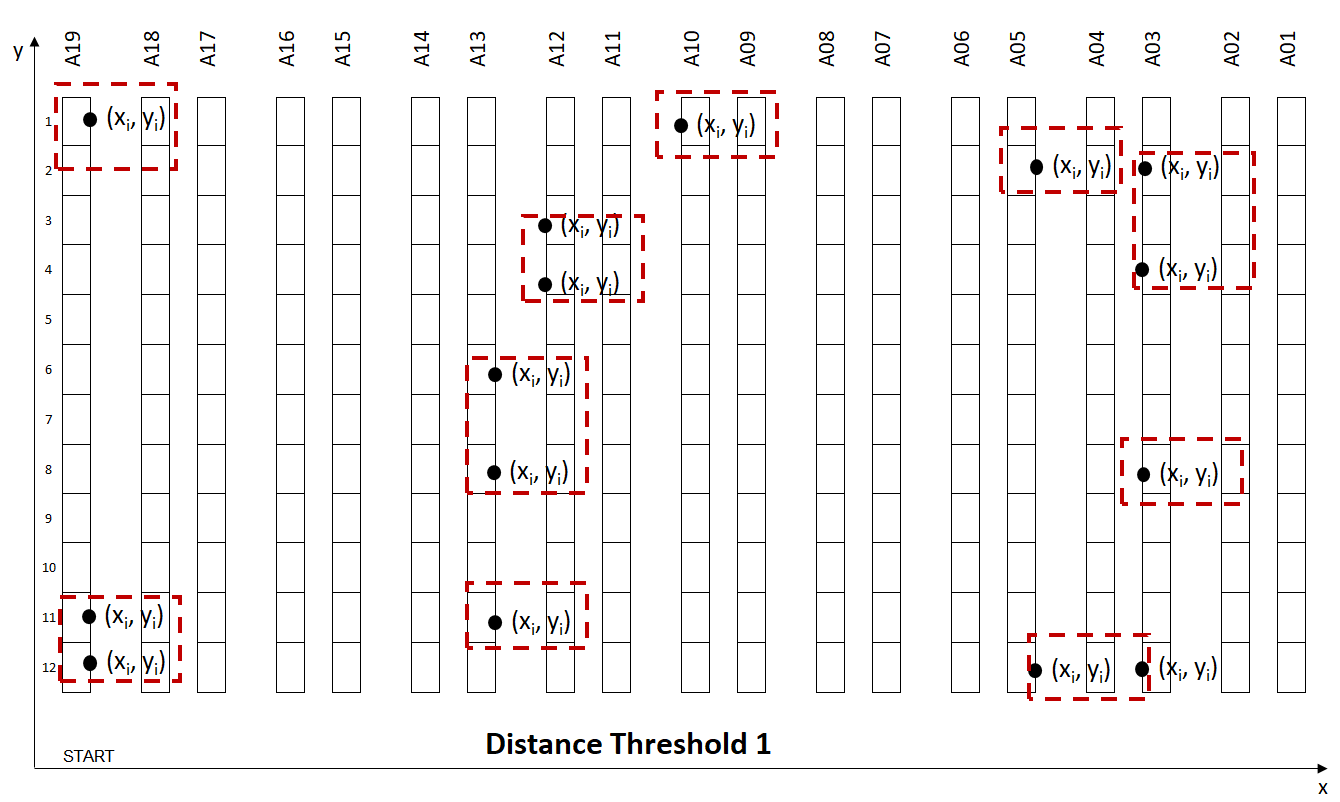


*Figura 10: Test 1:20.000 Order Lines/35 m distance Threshold*

Risultati:

* **Best** **Performace**: Metodo 3 per 9 orders/Wave con riduzione dell'83% della walking distance.
* **Metodo 2 vs. Metodo 1**: Clustering per ordini mono-linea riduce la walking distance del 34%;
* **Metodo 3 vs. Metodo 2**: Clustering per ordini mono-linea riduce la walking distance del 10%.
  1. ***Tuning Distance Threshold for Clustering***

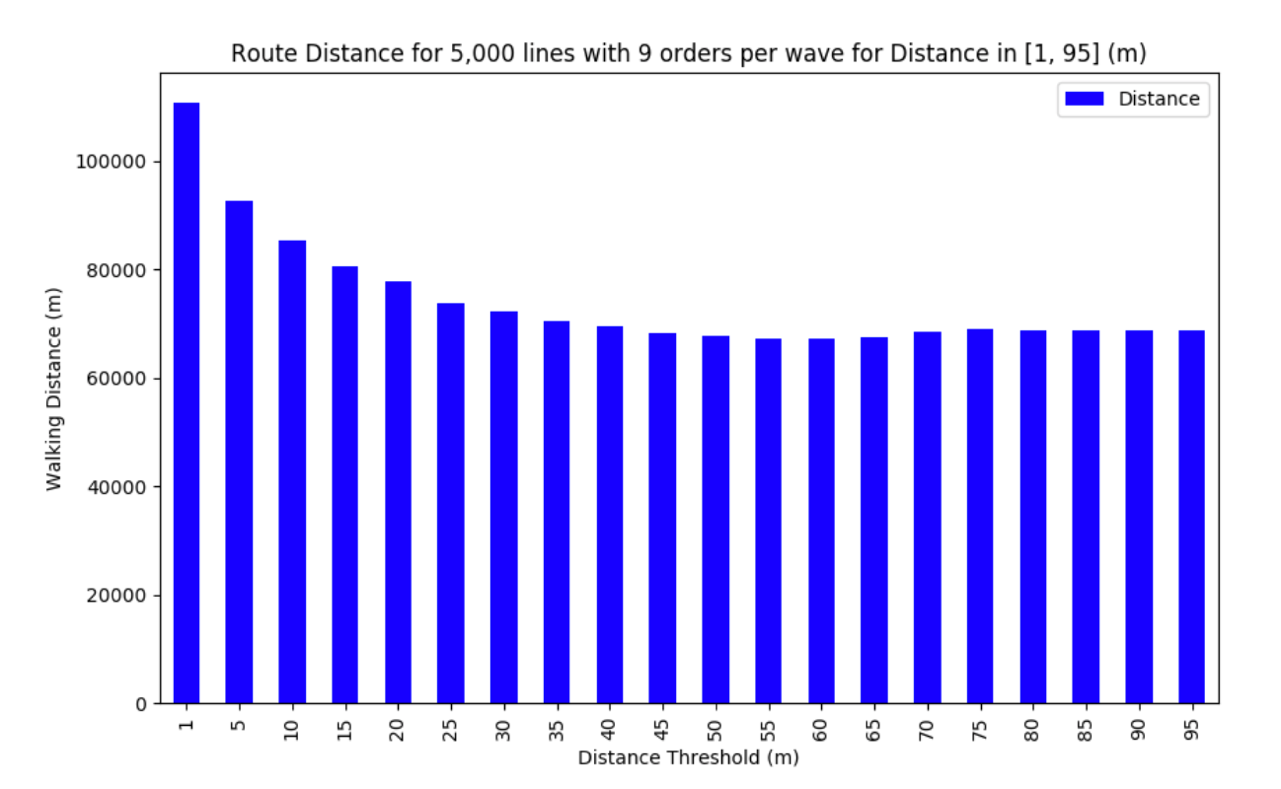
La prima ipotesi che viene fatta è che il Metodo 3 è il miglior scenario (20.000 order lines, 35 m Distance Threshold). Quindi, si va, di seguito, ad osservare la Distance Threshold sulla Walking Distance totale.



*Figura 11: Differenti distance threshold per Picking Location Clustering*

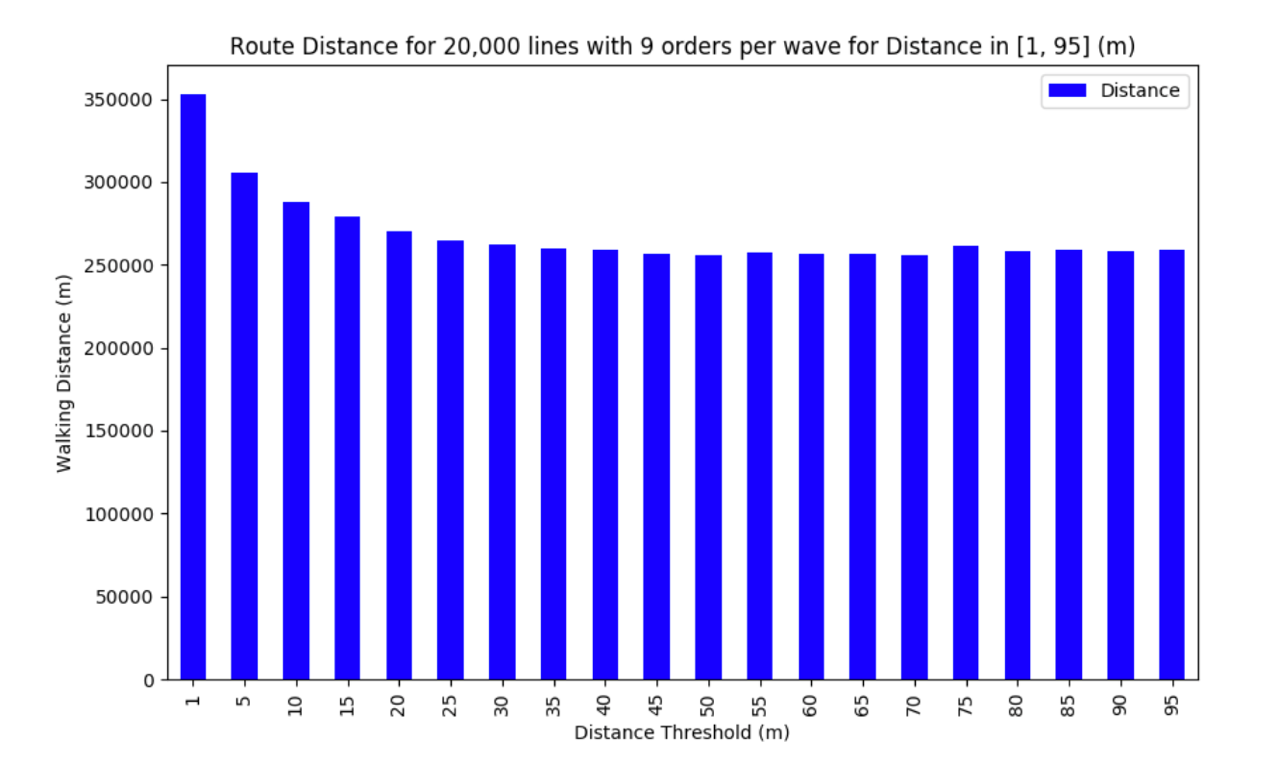
Quando il trade-off fra Walking Distance per due locations e la Wave Size è caratterizzato da:

* **Low** **Distance**: La walking distance fra due locations è bassa ma si hanno meno ordini per wave (più waves);
* **High** **Distance**: La walking distance fra due locations è alta ma si hanno molti ordini per wave (meno waves).



*Figura 12: Risultati per 5.000 linee raggruppate in Waves di 9 ordini con una distance threshold in [1,95]*

Si può osservare un minimo locale, per Distance\_Threshold = 60 m, dove la distanza è ridotta del 39% rispetto alla Distance\_Threshold = 1 m.



*Figura 12: Risultati per 20.000 linee raggruppate in Waves di 9 ordini con una distance threshold in [1,95]*

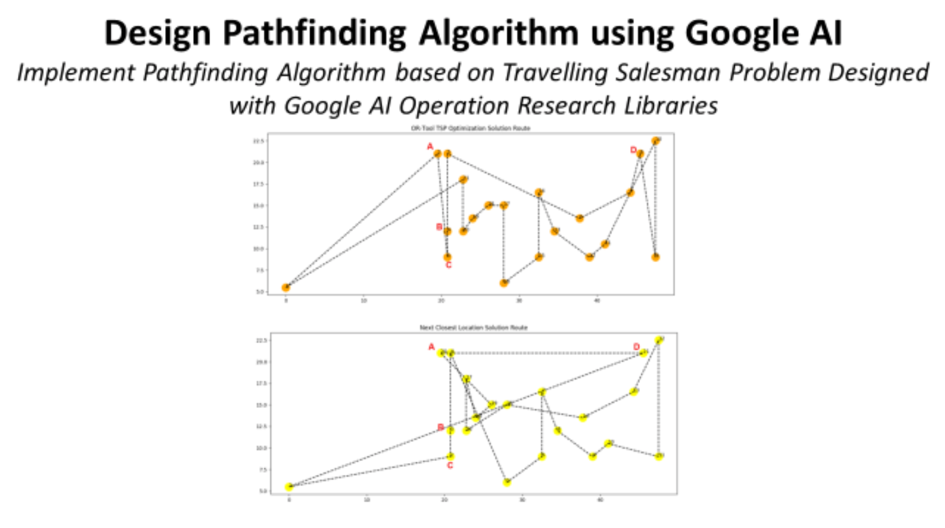
Si può osservare un minimo locale, per Distance\_Threshold = 50 m, dove la distanza è ridotta del 27% rispetto alla Distance\_Threshold = 1 m.

1. **Prossimo Step**

Il prossimo passo è la creazione del **Picking Route**, cioè per una lista di Picking Locations da coprire, come si può trovare il perc

**Improve Werahouse Productivity using Pathfinding with Python (Parte 3)**

Implementazione dell’Algoritmo Pathfinding basato sul Travelling Salesman Problem modellato attraverso Google AI Operation Research libraries.



1. **Pathfinding Algorithm for Picking Route Creation**

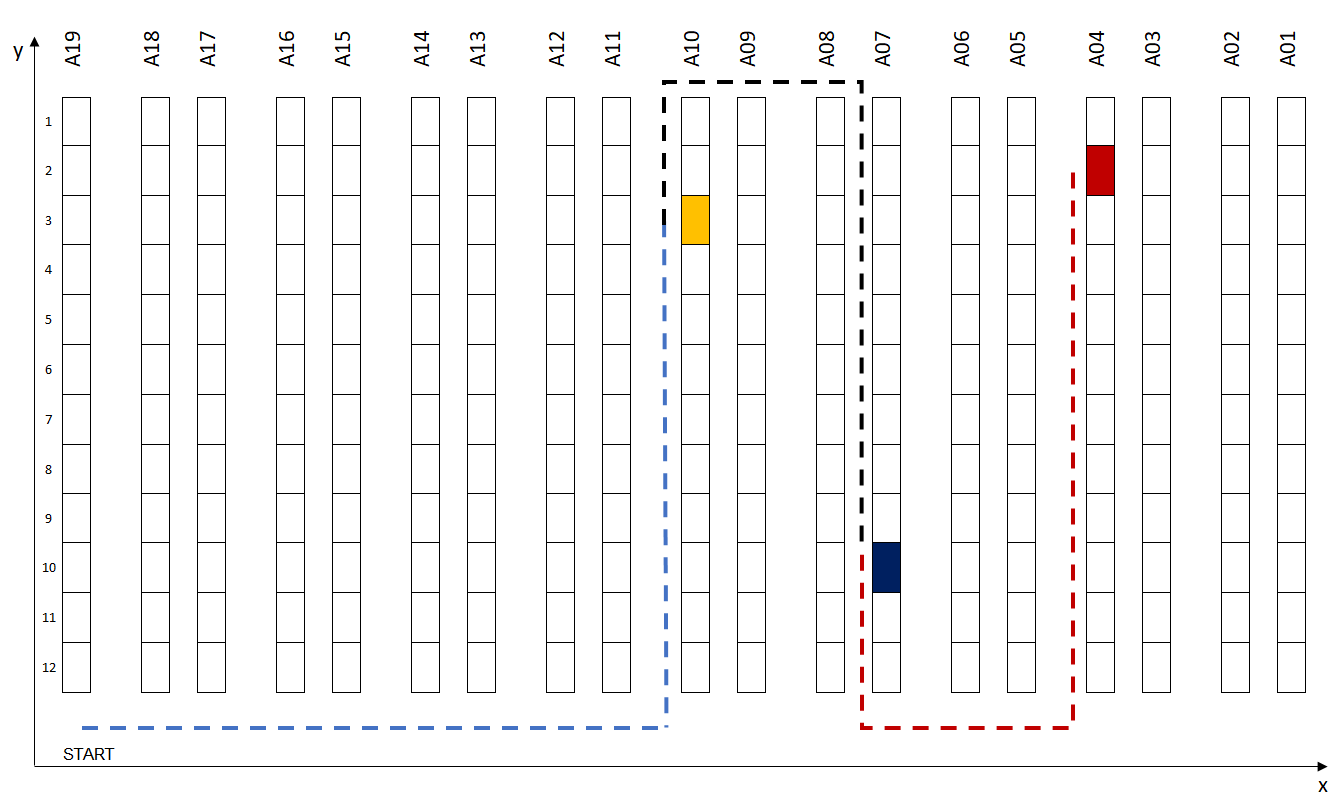
Nella prima parte si è costruito un modello per testare diverse strategie di creazione di orders wave per ridurre la walking distance per il picking attraverso:

* Warehouse Mapping: join per ogni linea d’ordine con le associate coordinate(x,y) di picking location nel warehouse;
* Distanza di calcolo: una funzione che calcola la walking distance fra due picking locations;
* Picking Locations Clustering: funzione che raggruppa orders by waves usando il clustering geografico (x,y) per ridurre la massima distanza fra due locations nello stesso picking route.



*Figura 1: Process Flow per il calcolo della picking route*

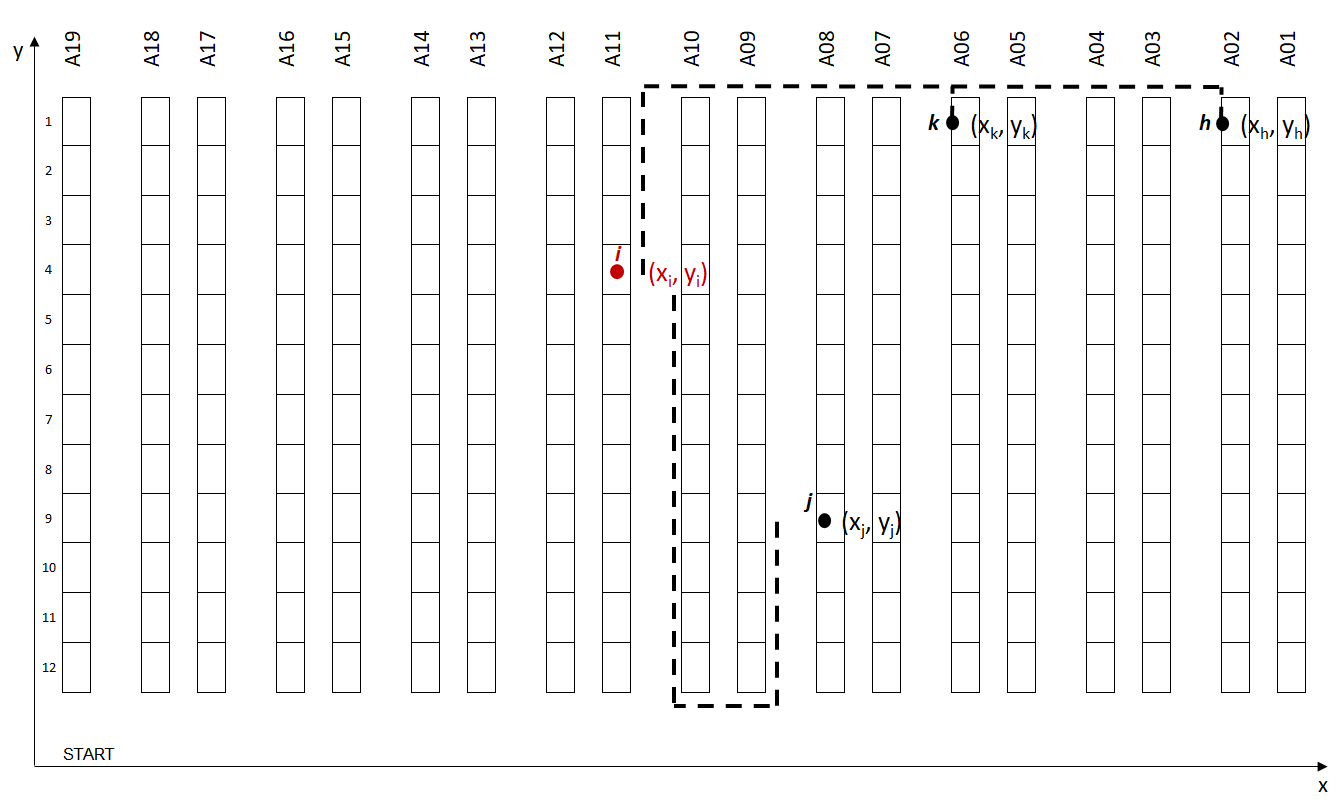
In questa fase, si vanno a raggruppare gli orders by waves per essere prelevati insieme. Per ogni wave, si ha una lista delle picking locations che necessita di essere coperta dal werahouse picker.

Il prossimo obiettivo è quello di definire una funzione per trovare la sequenza delle locations che minimizzerà la walking distance.

*Figura 2: Routes potenziali di una wave per coprire 3 picking locations*

**Soluzione Iniziale:** Uso della strategia di prossima location più vicina.

Nella parte 1 di questo lavoro è stata esposta una soluzione iniziale che usava la strategia di Prossima Location più vicina.



*Figura 3: Next Storage Location Scenario*

Questa è una semplice strategia che non richiede uno sforzo computazionale di alto livello.

Dalla location j (xj,yj), questa funzione prenderà come input una lista di potenziali prossime locations [(xj, yj), (xk, yk), (xh, yh)].

Questo output sarà vicino alla locatiomn, usando una funzione di walking distance custom, come il miglior candidato per la prossima location da coprire.

A questo punto, sorge la domanda: Qual è la soluzione ottimale per la creazione della Picking Route?

1. **Travelling Salesman Problem applicato al warehouse picking route design**

Obiettivo: trovare l’algoritmo ottimale per la creazione della picking route usando SPRP – Single Picker Routing Problem per un werahouse bidimensionale.

Quindi, operativamente, quello che si va a fare è costruire un modello di Travelling Salesman usando Google AI’s OR-Tools.

**OR-Tools**: è una collezione open-source di tools per l’ottimizzazione combinatoria. Da un vastissimo insieme di possibili soluzioni, l’obiettivo è di trovare la migliore soluzione.

Immagine che contiene testo, clipart

Descrizione generata automaticamente

*Figura 4: OR-Tools part of Google AI Solutions*

* 1. **Adattamento del Modello Google-OR**

Input Data:

* Lista delle coordinate di picking locations:

**List\_Coord = [(xi, yi), (x2,y2), .. , (x\_n, y\_n)]**

* Funzione di Walking Distance:

**F: ((x\_i, y\_i), (x\_j, y\_j)) -> distance(i,j)**

Codice Python:

1. # Creo il Data Model
2. data = {}
3. data['distance\_matrix'] = [[**int**(distance\_picking(List\_Coord[i], List\_Coord[j], y\_low, y\_high)) **for** i in range(len(List\_Coord))] **for** j in range(len(List\_Coord))]
4. data['num\_vehicles'] = 1
5. data['depot'] = 0

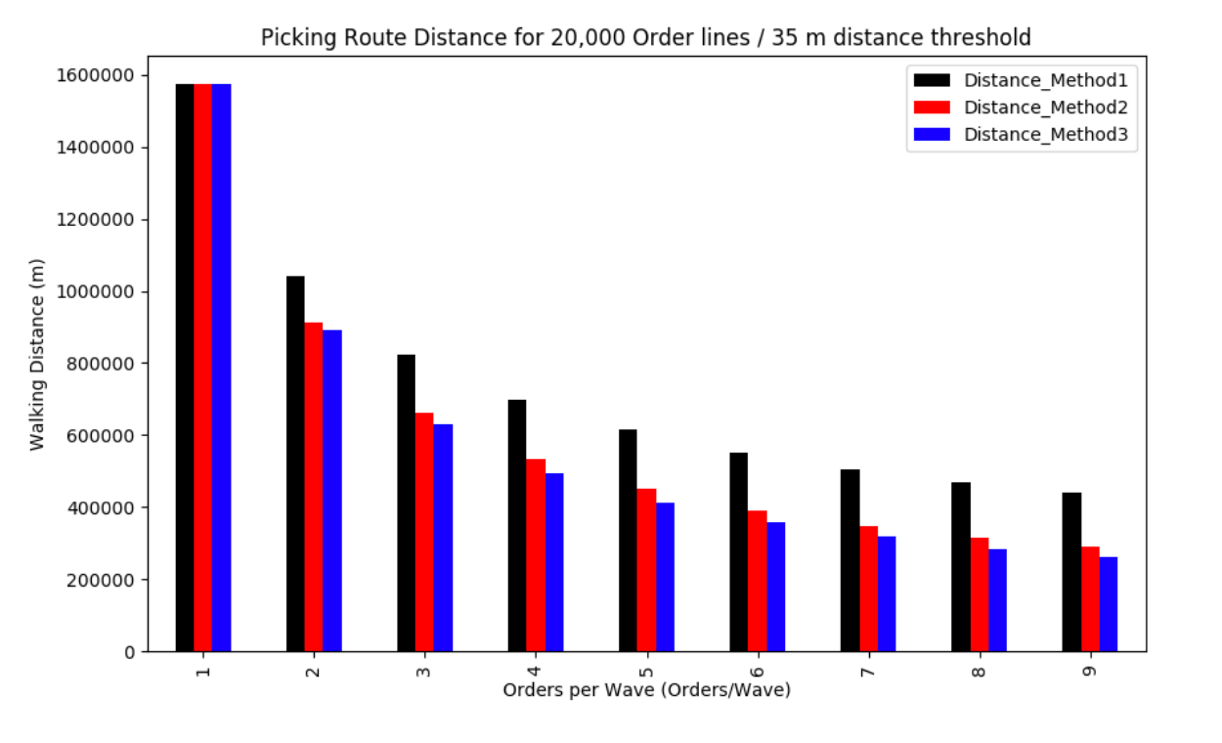
Nell’esempio di Google OR, un dizionario viene creato con:

* Matrice delle distanze: **[ distance(i,j) for i in [1,n] for j in [1,n] ]**
* Numero di veicoli: **1 veicolo per ogni scenario**
* Location per depot – start e end: start= end= **List\_Coord[0]**
  1. **Simulazione: OR-Tool Solution vs. Prossima Location più vicina**

L’obiettivo è quello di stimare quale impatto ha la soluzione ottimizzata del TSP di Google-OR, sul totale della walking distance, rispetto alla soluzione iniziale costruita sulla prossima location più vicina.

Verrà impiegato il metodo usato nella parte due per simulare diversi scenari:

* **Order** **Lines**: 20.000 Linee;
* **Threshold sulla distanza:** Massima distanza fra due picking locations in ogni cluster (distanza\_threshold = 35 m);
* **Orders per Wave:** order\_number in [1,9];
* **3 metodi per il Clustering:**
  + **Metodo 1 -** Non si applica il Clustering;
  + **Metodo 2** – Si applica il Clustering solo su ordini a linea singola;
  + **Metodo 3** – Si applica il Clustering su ordini a linea singola e sui centroidi degli ordini multi\_linea.

****

*Figura 5: 20.000 Orders Lines su 35 m come threshold della distanza*

Le performance migliori sono ottenute dal metodo 3 per 9 orders/wave con l’83% di riduzione della Walking Distance.

Sorge, a questo punto, la seguente domanda:

Riusciamo a ridurre la walking distance di più dell’ 83% applicando le soluzioni ottimizzate del problema TSP implementate da Google-OR?

1. # Risultati per Scenario
2. list\_OPW, list\_MM, list\_DistanceOr, list\_DistanceIt = [], [], [], []
3. # Risultati per Wave
4. l\_chor, l\_dstor, l\_chit, l\_dstit = [], [], [], []
6. **for** mm\_param in MM\_Unique:
7. **for** opw\_param in OPW\_Unique:
9. # Dataset mapping
10. dist\_waveor = 0
11. dist\_waveit = 0
13. # Creo percorsi
14. List\_chemins = create\_chemin(df\_3m, mm\_param, opw\_param)
16. # Loop nei percorsi per (OrderPerWave, MonoMulti Methods)
17. **for** chemin in List\_chemins:
19. # Creo il data model
20. data = {}
21. data['distance\_matrix'] = [[**int**(distance\_picking(chemin[i], chemin[j], y\_low, y\_high)) **for** i in range(len(chemin))] **for** j in range(len(chemin))]
22. data['num\_vehicles'] = 1
23. data['depot'] = 0
25. # 1. Creo routing model con OR-Tools
26. manager = pywrapcp.RoutingIndexManager(len(data['distance\_matrix']),
27. data['num\_vehicles'], data['depot'])
28. routing = pywrapcp.RoutingModel(manager)
30. # 2. Creo la distance callback
31. transit\_callback\_index = routing.RegisterTransitCallback(distance\_callback)
33. # 3. Setto il costo del viaggio
34. routing.SetArcCostEvaluatorOfAllVehicles(transit\_callback\_index)
36. # 4. Setto i parametri di ricerca
37. search\_parameters = pywrapcp.DefaultRoutingSearchParameters()
38. search\_parameters.first\_solution\_strategy = (
39. routing\_enums\_pb2.FirstSolutionStrategy.PATH\_CHEAPEST\_ARC)
41. # 5. Calcolo e mostro la soluzione
42. solution = routing.SolveWithParameters(search\_parameters)

45. **if** solution:
46. distance\_or, chemin\_or = out\_or\_solution(manager, routing, solution, chemin)
47. distance\_init, chemin\_init, distance\_max\_init = create\_picking\_route(Loc\_dep, chemin, y\_low, y\_high)

50. # Aggiungo la distanza
51. dist\_waveor += distance\_or
52. dist\_waveit += distance\_init
54. # Append per Wave
55. l\_chor.append(chemin\_or)
56. l\_dstor.append(distance\_or)
57. l\_chit.append(chemin\_init)
58. l\_dstit.append(distance\_init)
60. # Append
61. list\_OPW.append(opw\_param)
62. list\_MM.append(mm\_param)
63. list\_DistanceOr.append(dist\_waveor)
64. list\_DistanceIt.append(dist\_waveor)
66. print("OR {:,} m / It {:,} m for {} with {} Orders/Wave".format(dist\_waveor, dist\_waveit, mm\_param, opw\_param))
    1. **OR-Tool Solution vs. Closest Next Location**

Dopo aver eseguito waves usando questi tre metodi, si vanno a creare percorsi di prelievo utilizzando:

* **Next Closet Location:** distance\_init
* **Soluzione Ottimizzata del TSP di Google-OR:** distance\_or
* **Riduzione della distanza (%):** 100 \* (distance\_init – distance\_or) / distance\_init

Immagine che contiene tavolo

Descrizione generata automaticamente

*Figura 6: 20.000 Orders Lines su 35 m come threshold della distanza*

Osservazioni:

* **Orders per Wave**: quando il numero di ordini per wave è maggiore OR-Tool ha un impatto maggiore;
* **Metodi**: quando viene aggiunto il clustering per processare le waves allora OR-Tool ha un impatto minore.

Risultati per il miglior scenario:

Il metodo 3 (il clustering per tutti gli ordini) ha un impatto di -1.23% sul totale della walking distance.

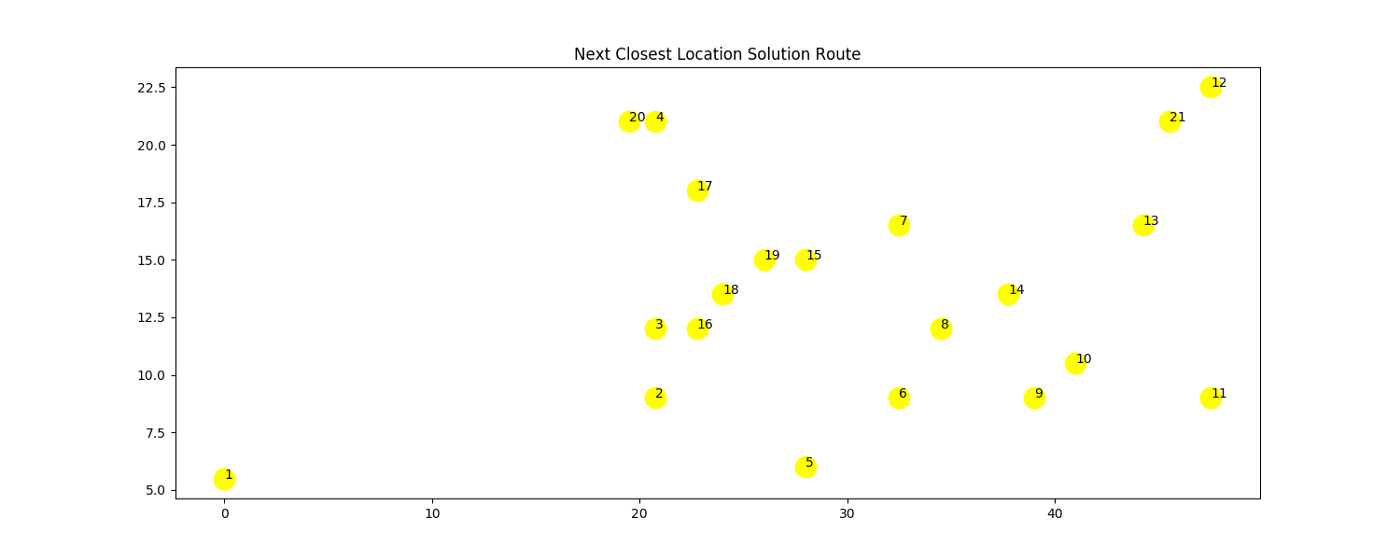
1. **Definizione del problema del TSP**

Vediamo, di seguito, come, praticamente, l’OR-Tool permette di ridurre la Walking Distance totale del 1.23%.

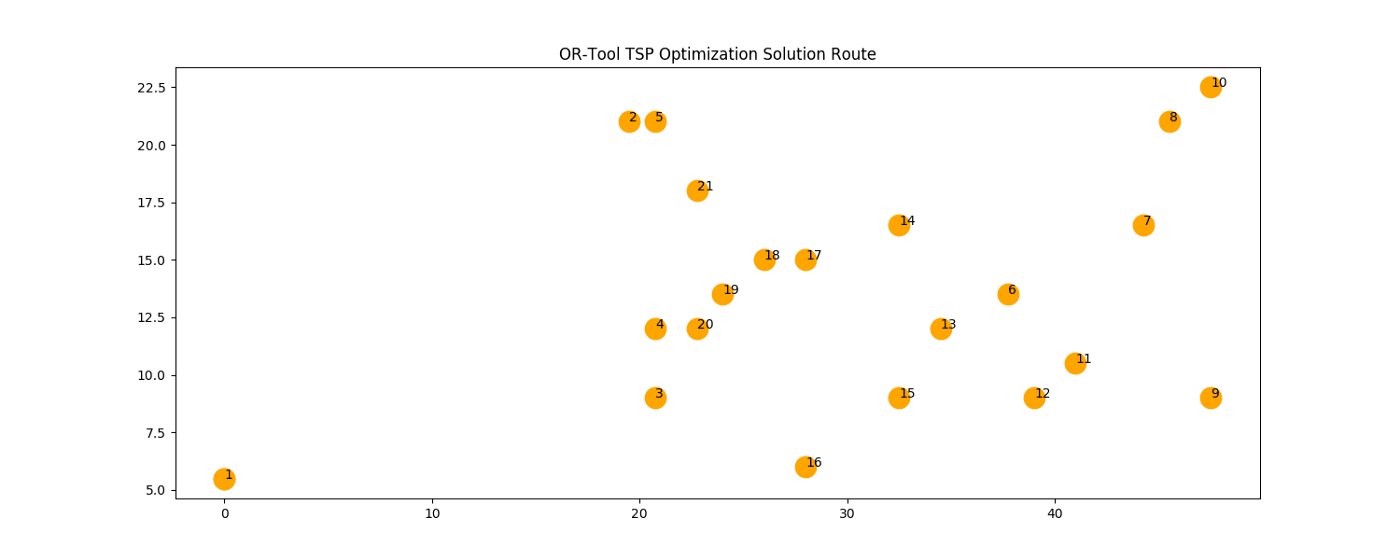
Gap più grande: 60 m per 21 Picking Locations.

In questo esempio si hanno da coprire 21 diverse locations in una wave.

* **OR-Tool TSP**: distanza = 384m.
* **Next Closest Location**: distanza = 444m.
* **Riduzione della distanza**: 60m.



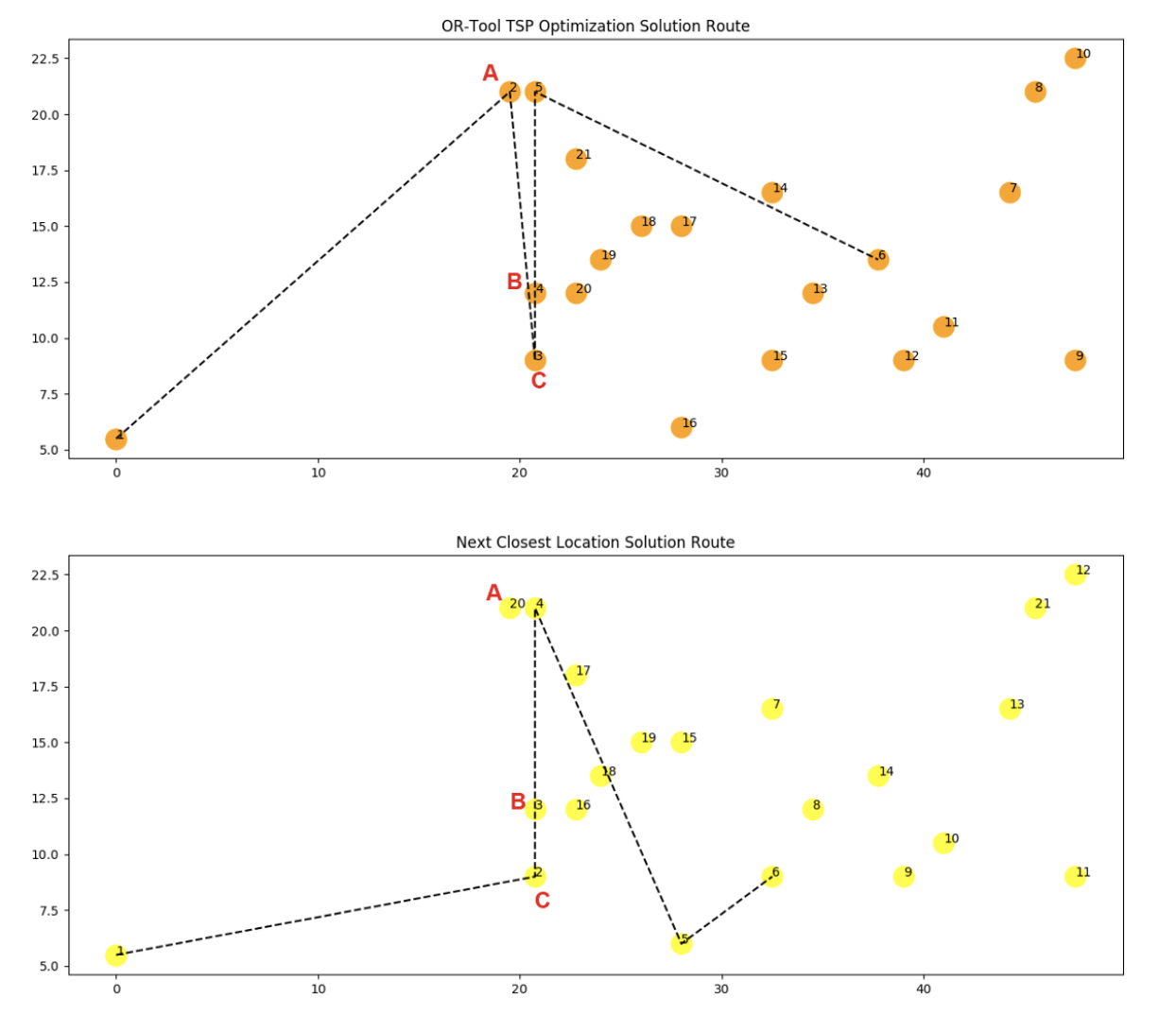
*Figura 7: Next Closest Location Solution Route per 21 locations (distanza: 444m)*



*Figura 8: OR-Tool TSP Optimization Solution Route per 21 locations (distanza: 384m)*

**Limiti dell’algoritmo Next Closest Location (NCL):**

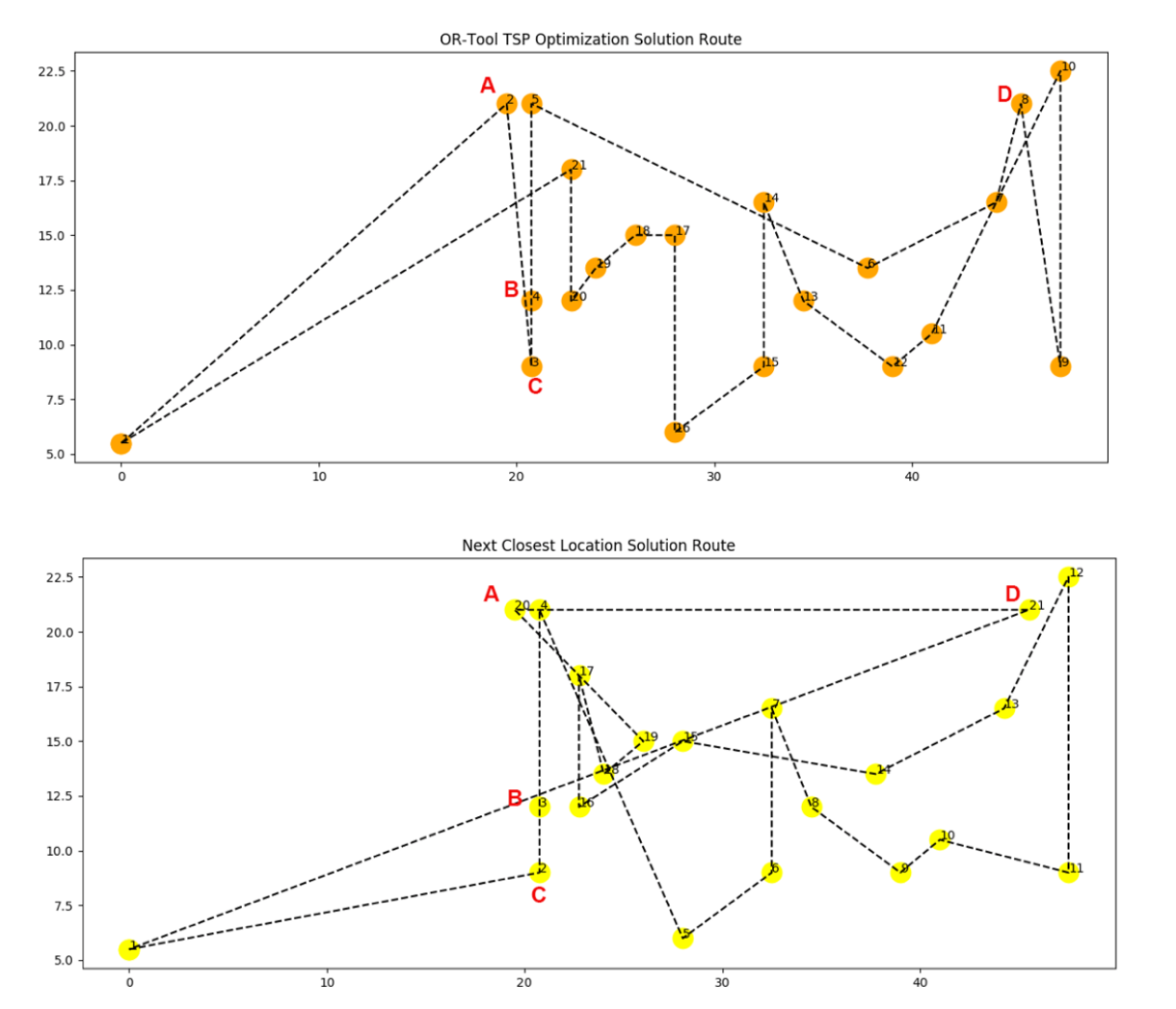
*TSP Point 2 (Point A):*

In questo specifico esempio, si può osservare i benefici della soluzione TSP ottimizzata per il secondo punto (19.5, 21).

*Figura 9: Esempio di confronto NCL vs. TSP*

Per il punto 1 di start ci sono 3 punti vicini {A, B, C}.

* 1. Il punto più vicino dal punto 1 è A.
  2. Punto NCL sarà diretto verso C, TSP partirà da A.
  3. TSP copre {A, B, C} in una sola volta mentre NCL copre solo {B, C} e lascia A per dopo.



*Figura 10: Specifico Esempio di NCL vs TSP*

NCL per il punto 21:

Nell'esempio sopra, dopo il punto 20 (punto A) il Warehouse Picker deve ancora coprire il punto 21 (punto D) prima di andare al deposito (punto 1). Questa distanza extra influisce notevolmente sull'efficienza dell'NCL per questa wave.

**Conclusioni**

L'applicazione della soluzione del problema del commesso viaggiatore alla creazione del percorso di prelievo può ridurre la distanza percorsa totale e migliorare la produttività complessiva.

Insieme alla smart Order Wave Processing, può ottimizzare la sequenza di prelievo per ondate con un numero elevato di ubicazioni di prelievo da coprire, trovando il percorso più breve per coprire un numero massimo di locations.