POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Matematica

Tesi di Laurea Magistrale

Dynamic Multiperiod Capacitated Vehicle Routing Problem with Probabilistic Information

Relatore
prof. Paolo Brandimarte

Candidato
Luca Bajardi

Anno Accademico 2021-2022

Sommario

Il *Dynamic Multiperiod Capacitated Vehicle Routing Problem with Probabilistic Information* ha come obiettivo quello di individuare i percorsi migliori per la consegna degli ordini ai clienti, partendo da un unico deposito, con un numero limitato di veicoli disponibili. Gli ordini vengono generati dinamicamente ogni giorno e quotidianamente tra tutti gli ordini aperti ne vengono selezionati alcuni per le consegne giornaliere. Questi ordini selezionati vengono consegnati ottimizzando un CVRP deterministico.

L'obiettivo della Tesi consiste nell'analisi del problema e nell'elaborazione di un algoritmo che ottimizzi il costo dei percorsi utilizzati per le consegne. Il problema si divide in due parti e vengono effettuate le seguenti analisi: la prima analisi riguarda l'ottimizzazione deterministica della consegna giornaliera, nella quale vengono confrontati l'algoritmo *Or-Tools* di Google e la *Tabu Search* sviluppata dall'Autore; la seconda analisi riguarda la selezione giornaliera degli ordini da consegnare, nella quale vengono confrontate le 5 policy per la selezione create dall'Autore.

Il caso studiato nell'elaborato di Tesi ha come riferimento un caso reale relativo a un'azienda di consegna di mobili.

I risultati dello studio portano alla conferma di un'equivalenza tra i solutori analizzati sul problema considerato e confermano che la scelta di una politica di selezione dei clienti adeguata riduca significativamente i costi delle consegne.

The *Dynamic Multiperiod Capacitated Vehicle Routing Problem with Probabilistic Information* aims to find the best routes to deliver orders to customers from a single depot with a limited number of available vehicles. Orders are dynamically generated each day, and each day among all open orders, a few are selected for daily deliveries. These selected orders are delivered by optimizing a deterministic CVRP.

The objective of this Thesis is to analyze the problem and create an algorithm that optimizes the cost of the paths used for deliveries. The two parts of the problem are analyzed: the first analysis concerns the deterministic optimization of daily delivery, in which Google's *Or-Tools* algorithm and *Tabu Search* developed by the Author are compared; and the second analysis concerns the daily selection of orders to be delivered, in which the 5 policies for selection created by the Author are compared.

The case studied in the Thesis starts from the idea of a real case related to a furniture delivery company.

The results of the study lead to the confirmation of an equivalence between the analyzed solvers on the considered problem and confirm that the choice of an appropriate customer selection policy reduces delivery costs significantly.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Al giorno d'oggi utilizziamo algoritmi di ottimizzazione in diversi campi, tra i quali abbiamo:

- Distribuzione dell'energia elettrica: la New York ISO utilizza l'ottimizzazione per scegliere il modo più conveniente per fornire elettricità ai clienti.
- Finanza: Betterment utilizza l'ottimizzazione per scegliere il mix ottimale di asset, che massimizza i rendimenti al netto delle imposte riducendo al minimo il rischio.
- Logistica: FedEx riduce i costi ottimizzando la consegna dei pacchi attraverso la propria rete di spedizione.
- Produzione: SAP utilizza l'ottimizzazione per programmare in modo efficiente la produzione di merci nelle fabbriche al fine di soddisfare gli ordini dei clienti.
- Programmazione sportiva: la NFL utilizza l'ottimizzazione per pianificare al meglio i programmi del campionato. [4]

In particolare nel settore logistico troviamo il Vehicle Routing Problem, in breve VRP. Il VRP mira a trovare i percorsi ottimali, data una flotta di veicoli che partono da un deposito e consegnano un certo bene a un insieme di clienti.

A partire dalla prima definizione di VRP, sono state progettate molte varianti per affrontare le situazioni che si presentano nel mondo reale: l'oggetto di questa tesi è il *Dynamic Multiperiod Capacitated Vehicle Routing problem with probabilistic information*. Nel Capitolo 2 troveremo le varianti che portano alla definizione del problema in considerazione in questa tesi.

Lo studio e la definizione di questo problema derivano dal problema reale di un'azienda di consegna di mobili: ogni giorno si presenta un insieme di clienti stocastici e l'azienda deve decidere quali clienti servire nel giorno successivo, in base alle richieste deterministiche pendenti fino a quel giorno e alle informazioni stocastiche sul futuro. Dopodiché, è necessario trovare percorsi convenienti per programmare il percorso di ciascun veicolo utilizzato per servire i clienti giornalieri.

Attualmente esistono numerosi solutori per il VRP e le sue varianti deterministiche, come Or-Tools, CPLEX e Gurobi. Ognuno di questi solutori ha un proprio approccio al problema di ottimizzazione e consente all'utente di impostare diversi tipi di vincoli e variabili decisionali. In ogni caso, essi possono essere utilizzati per risolvere solo il problema deterministico e non tengono conto delle informazioni stocastiche, che non possono essere modellate.

Il primo obiettivo della tesi (analizzato nel Capitolo 3) è confrontare un solutore open source, in questo caso è stato scelto Or-Tools, con un solutore basato sull'approccio Tabu Search creato dall'Autore per trovare l'algoritmo più efficiente per risolvere il problema considerato.

Il secondo obiettivo della tesi (analizzato nel Capitolo 4) è trovare una politica per scegliere l'insieme di clienti da servire ogni giorno al fine di minimizzare il costo totale del viaggio e del servizio nel lungo periodo, sfruttando l'informazione stocastica. L'insieme dei clienti selezionati viene poi passato al solutore scelto precedentemente per trovare percorsi fattibili e convenienti.

Capitolo 2

Dynamic Multiperiod Capacitated Vehicle Routing Problem with Probabilistic Information

In questo capitolo è presentato il *Dynamic Multiperiod Capacitated Vehicle Routing Problem with Probabilistic Information*. É un classico problema multiperiodale e multistadio. Il caso studio preso in considerazione in questa Tesi ha come periodo 1 giorno, la parte stocastica è relativa alla domanda futura: non c'è un insieme fisso di clienti futuri ma viene generato di giorno in giorno ed è stocastica anche la parte relativa al peso degli ordini e alla posizione dei clienti.

Per ogni giorno di simulazione il problema si suddivide in due parti: la selezione dei clienti da servire secondo specifiche politiche e l'ottimizzazione del percorso. Nel Capitolo 3 vengono confrontati due algoritmi di ottimizzazione del percorso, nel Capitolo 4 vengono confrontate le politiche di selezione dei clienti.

Di seguito vi è l'analisi del problema partendo dalla versione base e aggiungendo vari vincoli.

2.1 Traveling Salesman Problem (TSP)

Il *Traveling Salesman Problem (Problema del commesso viaggiatore)* è il più semplice tra i problemi di instradamento. L'obiettivo del problema consiste nel trovare il percorso più breve tra un insieme di clienti che devono essere visitati. Nella formulazione classica del problema, i nodi sono le città che un venditore potrebbe visitare e l'obiettivo del venditore è quello di ridurre al minimo sia i costi di viaggio sia la distanza percorsa.

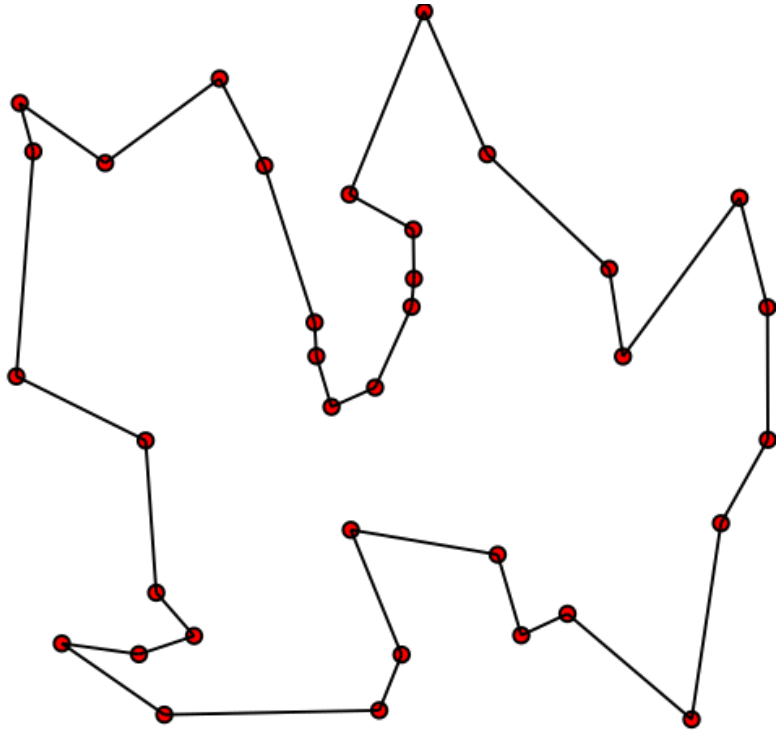


Figura 2.1: Esempio di soluzione del TSP

2.1.1 Formulazione matematica del TSP

Nella formulazione di Dantzig–Fulkerson–Johnson [13], il problema TSP è associabile a un grafo $G = (V, A)$, in cui V è l'insieme degli n nodi (o clienti) e A è l'insieme degli archi (o strade tra due clienti). Si indica con c_{ij} il costo dell'arco per andare dal nodo i al nodo j (dove il costo è solitamente la distanza tra i nodi). Viene considerato il TSP simmetrico dove $c_{ij} = c_{ji}\ \forall(i, j)$. Detta x_{ij} la generica variabile binaria tale che $x_{ij} = 1$ se l'arco (i, j) appartiene al percorso e $x_{ij} = 0$ altrimenti, una possibile formulazione matematica del problema è:

$$\min \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij} \tag{2.1a}$$

tale che

$$\sum_{i \in V} x_{ij} = 1 \qquad j \in V \tag{2.1b}$$

$$\sum_{j \in V} x_{ij} = 1 \qquad i \in V \tag{2.1c}$$

$$\sum_{i \in Q} \sum_{j \in V \setminus Q} x_{ij} \geq 1 \qquad \forall Q \subset V, |Q| \geq 1 \tag{2.1d}$$

$$x_{ij} \in \{0,1\} \qquad i, j \in V \tag{2.1e}$$

La funzione obiettivo che rappresenta la minimizzazione del costo del percorso è indicata dalla relazione (2.1a). Il fatto che in ogni nodo entra solo un arco e ne esce solo uno è indicato dai vincoli (2.1b) e (2.1c). Tali vincoli non assicurano che la soluzione sia costituita da un unico circuito, quindi per assicurarsi l'assenza di sottopercorsi vengono inseriti anche i vincoli (2.1d). In particolare essi definiscono che deve esistere almeno un arco che colleghi un nodo di Q con un nodo non appartenente a Q comunque si scelga un sottoinsieme proprio di nodi Q in V.

2.2 Vehicle Routing Problem (VRP)

Il *Vehicle Routing Problem (VRP)* è un evoluzione del *TSP*. In questo problema vi sono più veicoli che partono da uno stesso deposito e devono consegnare ad un insieme di clienti. L'obiettivo è determinare un insieme di percorsi (un percorso per ogni veicolo che deve iniziare e finire presso il deposito) in modo tale che il costo di trasporto globale sia ridotto al minimo. Questo costo può essere monetario, di lunghezza o di tempo e può dipendere dal tipo di veicolo.

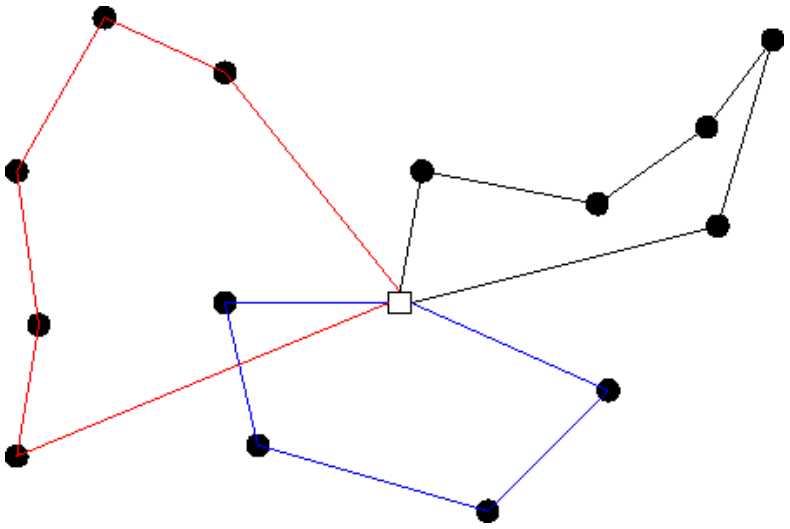


Figura 2.2: Esempio di soluzione del Vehicle Routing Problem

2.2.1 Formulazione matematica del VRP

La formulazione del TSP di Dantzig, Fulkerson e Johnson è stata estesa per creare la formulazione per il VRP

$$\min \sum_{r\in T}\sum_{i\in V}\sum_{j\in V\setminus\{i\}} c_{ij}x_{rij} \tag{2.2a}$$

tale che

$$\sum_{r\in T}\sum_{i\in V\setminus\{j\}} x_{rij} = 1 \qquad \forall j \in V\setminus\{0\} \tag{2.2b}$$

$$\sum_{j\in V\setminus\{0\}} x_{r0j} = 1 \qquad \forall r \in T, \tag{2.2c}$$

$$\sum_{i\in V\setminus\{j\}} x_{rij} = \sum_{i\in V\setminus\{j\}} x_{rji} \qquad \forall j \in V, r \in T \tag{2.2d}$$

$$\sum_{r\in T}\sum_{i\in S}\sum_{j\in S\setminus\{i\}} x_{rij} \leq |S| - 1 \qquad \forall S \subseteq V\setminus\{0\} \tag{2.2e}$$

$$x_{rij} \in \{0,1\} \qquad \forall r \in T, i, j \in V, i \neq j \tag{2.2f}$$

Come nel TSP, c_{ij} rappresenta il costo per andare dal nodo i al nodo j, x_{rij} è la variabile binaria che ha valore 1 se l'arco che va da i a j è considerato come parte della soluzione con il veicolo r e 0 altrimenti. Assumiamo che T è l'insieme dei veicoli e V è l'insieme dei clienti da servire considerando 0 il nodo deposito

La funzione obiettivo (2.2a) minimizza il costo totale del viaggio. I vincoli del modello (2.2b) sono i vincoli di grado e assicurano che ogni cliente sia visitato da un solo veicolo. I vincoli di flusso (2.2c) e (2.2d) garantiscono che ogni veicolo possa lasciare il deposito una sola volta e che il numero di veicoli che arrivano a ogni cliente ed entrano nel deposito sia uguale al numero di veicoli che escono. I vincoli di eliminazione dei sub-tour (2.2e) assicurano che la soluzione non contenga cicli disconnessi dal deposito. I restanti vincoli obbligatori (2.2f) specificano i domini di definizione delle variabili. Questo modello è noto come formulazione del flusso di veicoli a tre indici. Il numero di disuguaglianze dei vincoli di eliminazione dei sub-tour vincoli di eliminazione dei sub-tour cresce esponenzialmente con il numero di nodi.

2.3 Capacitated Vehicle Routing Problem (CVRP)

L'obiettivo del *Capacitated Vehicle Routing Problem (CVRP)* è quello di trovare un insieme di percorsi a costo totale minimo per una flotta di veicoli con capacità di carico limitata e con sede in un unico deposito, per servire un insieme di clienti con i seguenti vincoli:

1. ogni percorso inizia e termina nel deposito,
2. ogni cliente viene visitato esattamente una volta,
3. la domanda totale di ogni percorso non supera la capacità del veicolo.

2.3.1 Formulazione matematica del CVRP

La formulazione del *CVRP* è molto simile a quella del *VRP*, in questo caso vengono aggiunti i vincoli di capacità (2.3e) facendo in modo che la somma delle richieste dei clienti visitati in un percorso clienti visitati in un percorso sia inferiore o uguale alla capacità del veicolo che effettua il servizio (i veicoli potrebbero avere capacità diverse, ma in questo caso è considerata uguale per tutti).

$$\min \sum_{r\in T}\sum_{i\in V}\sum_{j\in V\setminus\{i\}} c_{ij}x_{rij} \tag{2.3a}$$

tale che

$$\sum_{r\in T}\sum_{i\in V\setminus\{j\}} x_{rij} = 1 \qquad \forall j \in V\setminus\{0\} \tag{2.3b}$$

$$\sum_{j\in V\setminus\{0\}} x_{r0j} = 1 \qquad \forall r \in T \tag{2.3c}$$

$$\sum_{i\in V\setminus\{j\}} x_{rij} = \sum_{i\in V\setminus\{j\}} x_{rji} \qquad \forall j \in V, r \in T \tag{2.3d}$$

$$\sum_{i\in V}\sum_{j\in V\setminus\{0,j\}} d_j x_{rij} \leq Q \qquad \forall r \in T \tag{2.3e}$$

$$\sum_{r\in T}\sum_{i\in S}\sum_{j\in S\setminus\{i\}} x_{rij} \leq |S| - 1 \qquad \forall S \subseteq V\setminus\{0\} \tag{2.3f}$$

$$x_{rij} \in \{0,1\} \qquad \forall r \in T, i, j \in V, i \neq j \tag{2.3g}$$

2.4 Dynamic Multiperiod Capacitated Vehicle Routing Problem with Probabilistic Information

Il CVRP può essere esteso a una versione multiperiodale se il problema viene risolto più volte, con diversi insiemi di clienti ogni volta. Se la selezione dei clienti da servire e la corrispondente ottimizzazione del percorso avvengono in periodi diversi, il problema è anche multistadio. Nel problema considerato nella Tesi, il periodo coincide con la fase dello stadio ed è di un giorno: quindi ogni giorno viene risolto un CVRP con un certo insieme di clienti.

Finora il problema ha avuto solo aspetti deterministici, ma nell'applicazione considerata in questa Tesi è inclusa un'incertezza che lo rende un *Dynamic Multiperiod Capacitated Vehicle Routing Problem with Probabilistic Information*.

Il tipo di incertezza considerata riguarda gli ordini futuri dei clienti. Infatti, ogni giorno vi è una posizione fissa del deposito, un numero fisso di veicoli disponibili (con capacità fissa) e le richieste dei clienti fino a quel giorno, ma deve essere deciso quali clienti servire. Per fare ciò, viene ottimizzata la funzione obiettivo su tutti i periodi sfruttando la distribuzione stocastica della domanda futura dei clienti.

Va osservato che, se la domanda dei clienti fosse deterministica, il problema diventerebbe un problema multiperiodale in un'unica fase, dato che l'insieme dei clienti da servire giorno per giorno potrebbe essere deciso in un primo momento.

Il problema stocastico a più stadi è molto più complicato, infatti la selezione dei clienti serviti giornalmente può basarsi solo su deboli informazioni future.

Capitolo 3

Confronto tra algoritmi per CVRP

Esistono molti algoritmi per la risoluzione del problema CVRP, ognuno di essi ha caratteristiche specifiche per alcuni problemi (ad esempio accettano vincoli particolari), altri invece hanno prestazioni molto elevate in termini di tempo.

Nello sviluppo della tesi sono utilizzati due algoritmi per la risoluzione del CVRP: Or-Tools di Google e Tabu Search sviluppato dall'Autore.

3.1 Or-Tools

Or-Tools è la suite software open-source, veloce e portatile di Google per la risoluzione di problemi di ottimizzazione combinatoria sviluppata in C++ (disponibile anche sul repository github: `https://github.com/google/or-tools`). Supporta diversi linguaggi di programmazione, tra cui C++, C#, Java e Python. Or-Tools è in grado di risolvere molti tipi di VRP, tra cui problemi con ritiri e consegne, flotte eterogenee, depositi multipli, finestre temporali, luoghi di inizio e fine diversi, dimensioni di capacità multiple, carichi iniziali, competenze, ecc.

Ovviamente, ci sono alcune limitazioni nella risoluzione dei VRP, poiché sono intrinsecamente intrattabili per istanze di grandi dimensioni. Pertanto, Or-Tools a volte restituisce soluzioni buone, ma non ottimali.

L'algoritmo creato dell'Autore per il confronto tra algoritmi si collega ad Or-Tools tramite la libreria *ortools.constraint_solver* scaricabile tramite *pip*. Usando la classe *pywrapcp* è possibile creare il modello da risolvere. Infatti, tramite la funzione *SetArcCostEvaluatorOfAllVehicles* si aggiungono le informazioni sul costo di ciascuno spostamento da un cliente all'altro, tramite la funzione *RegisterUnaryTransitCallback* si aggiungono le informazioni relative alla dimensione di ciascun ordine e tramite la funzione *AddDimensionWithVehicleCapacity* si aggiungono i vincoli di capacità. [7]

3.2 Tabu Search

Il secondo algoritmo studiato è la tecnica della Tabu Search. La forma di Tabu Search ideata da Fred Glover è quella oggi più in uso [10], anche se idee simili erano state abbozzate da P. Hansen in contemporanea. La Tabu Search è una tecnica di ottimizzazione affermata e ciò è dimostrato da numerosi esperimenti computazionali effettuati. Inoltre, grazie alla sua flessibilità, può trovare soluzioni ottimali migliori rispetto a molte delle procedure classiche di ottimizzazione [14]. In questa Tesi l'analisi e la scrittura dell'algoritmo sono basate sugli articoli e tesi [5, 6, 10].

Come primo step viene trovata una soluzione iniziale ammissibile, poi si applica una procedura iterativa che ha lo scopo di ridurre i costi di percorrenza dei percorsi. Il numero di iterazioni è controllato implicitamente impostando un limite di tempo per l'esecuzione dell'algoritmo o direttamente fissando un numero massimo di iterazioni.

L'approccio Tabu Search viene utilizzato per evitare di rimanere bloccati in minimi locali: data la soluzione corrente, si genera una soluzione nel suo "vicinato" (*neighborhood*), cioè una soluzione che è una leggera modifica di quella attuale. Nel nostro caso, una soluzione vicina si ottiene modificando leggermente 2 routes. Quindi, si valuta il costo della soluzione vicina e si decide se accettare o meno questa soluzione come nuova soluzione corrente.

Nello pseudo-codice 1 è possibile vedere la struttura dell'algoritmo che è analizzato di seguito.

Nella *Tabu Search* i due elementi fondamentali sono la *tabu list* e l'*aspiration criterion*:

- La *tabu list* è un elenco che contiene le mosse tabu, ovvero le variazioni della soluzione corrente che portano ad una soluzione già visitata. Ogni volta che generiamo dei nuovi vicini possiamo facilmente controllare se i nuovi percorsi corrispondono ad una soluzione già visitata. La tabu list non è infinita ma ogni volta che raggiunge la sua lunghezza massima (*tabu_size*) la lista viene aggiorna in modalità FIFO, salvando quindi solo le mosse più recenti e perdendo la memoria delle mosse tabu più vecchie[1].

- L'*aspiration criterion* viene usato per determinare l'accettazione di una soluzione. Infatti data una soluzione vicina viene accettata se non viola la tabu list e la differenza nei costi di viaggio tra la soluzione corrente e quella vicina è maggiore di 0 oppure se è associata al minimo costo di viaggio trovato finora anche se viola la tabu list.

[1]Nell'implementazione della Tabu Search sviluppata dall'Autore invece che una lista sequenziale delle mosse, è utilizzato un dizionario dove la chiave è la mossa mentre il valore corrispondente è il numero di iterazioni per cui quella mossa è ancora nella *tabu list*

Algorithm 1 Tabu Search

```
1: G := Insieme di clienti
2: soluzione_ammissibile ← False
3: while not soluzione_ammissibile do
4:     S ← generate_random_solution(G)
5:     if solution_feasible(S) then
6:         soluzione_ammissibile ← True
7:         initialize_solution(S)
8:     else
9:         Rimuove l'ultimo cliente in G
10:    end if
11: end while
12: best_solution ← S
13: best_cost ← solution_cost(S)
14: tabu_list ← ∅
15: n_iters ← 5000
16: while n_iters > 0 do
17:    N, M ← find_neighborhood(S)          ▷ N è l'insieme delle soluzioni vicine
18:                                ▷ M è l'insieme delle mosse per arrivare a ciascun vicino
19:    for each neighbor, move in N, M do
20:        if solution_cost(neighbor) < candidate_cost then      ▷ Aspiration criteria
21:            if move not in tabu_list then
22:                best_candidate ← neighbor
23:                candidate_cost ← solution_cost(neighbor)
24:                temp_move ← move
25:            else
26:                if solution_cost(neighbor) < best_cost then
27:                    best_candidate ← neighbor
28:                    candidate_cost ← solution_cost(neighbor)
29:                    temp_move ← move
30:                end if
31:            end if
32:        end if
33:    end for
34:    if candidate_cost < best_cost then
35:        best_solution ← best_candidate
36:        best_cost ← candidate_cost
37:    end if
38:    Aggiungere temp_move alla tabu_list
39:    n_iters ← n_iters − 1
40: end while
```

Un altro aspetto fondamentale della *Tabu Search* è la ricerca dei vicini. Questo viene effettuato tramite la funzione *find_neighborhood*. Per ogni coppia di route presente[2] si presentano 3 tipologie di swap che possono accadere:

1. se i due percorsi hanno entrambi dei clienti da servire allora si scelgono due clienti casualmente (uno su ciascuno dei due percorsi) e si scambiano, ovvero il cliente selezionato sul primo percorso viene assegnato al secondo e viceversa (vedi Fig. 3.1),
2. se vengono selezionati un percorso con dei clienti e uno vuoto allora si sceglie casualmente un cliente da spostare nel percorso vuoto, aumentando quindi il numero di veicoli utilizzati (vedi Fig. 3.2),
3. se viene selezionato un percorso con un solo cliente allora si sposta questo cliente in una posizione casuale di un altro percorso, riducendo quindi il numero di veicoli utilizzati (vedi Fig. 3.3).

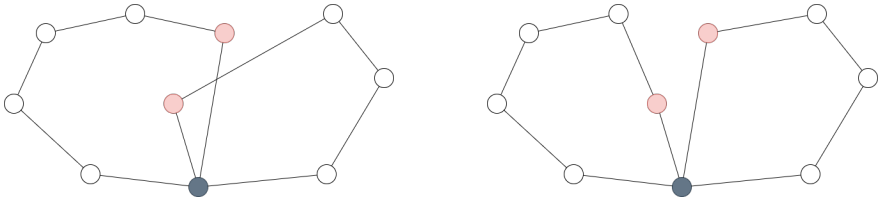


Figura 3.1: Swap semplice

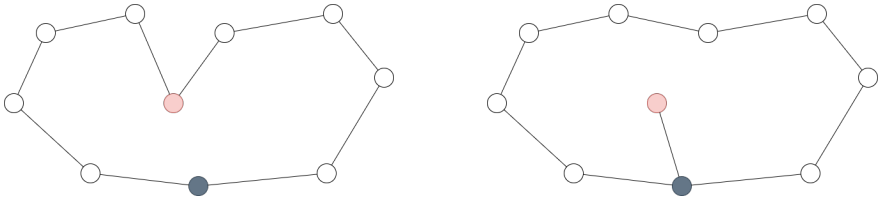


Figura 3.2: Aggiungi in percorso vuoto

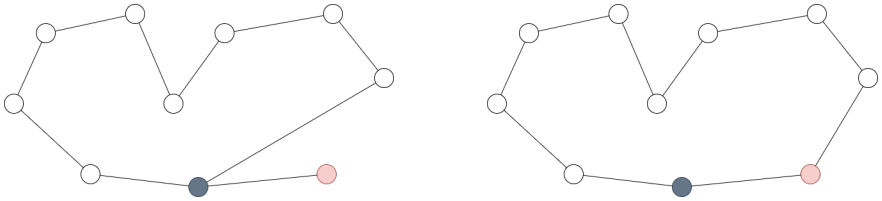


Figura 3.3: Rimuovi percorso con 1 cliente

[2]Bisogna tenere presente che i percorsi corrispondono ai veicoli uno a uno quindi sono presenti anche route/veicoli senza clienti da servire e che quindi rimarranno nel deposito

3.3 Fine-tuning Tabu Search

Data la definizione dell'algoritmo Tabu Search, abbiamo due parametri da regolare: il numero di iterazioni massimo dell'algoritmo, *n_iters*, e la dimensione massima della tabu list, *tabu_size*.

Per il fine-tuning vengono utilizzati dei problemi dalla libreria CVRPLIB [3] sviluppata da Uchoa et al. vista la mancanza di una buona serie di istanze di riferimento per il problema CVRP [11].

Per vedere l'errore percentuale al variare dei parametri scelti abbiamo utilizzato l'approccio *grid search* e i possibili valori sono:

- Numero di iterazioni $\in \{1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000\}$
- Lunghezza della tabu list $\in \{20, 40, 60, 80, 100, 120, 140, 160\}$

Nella Fig. 3.4 possiamo vedere la variazione dell'errore percentuale medio a seconda dei parametri scelti. Possiamo notare che nell'intorno delle 5000 iterazioni c'è un gomito nell'andamento della diminuzione dell'errore quindi prendiamo quello come valore ottimale per il numero di iterazioni. Invece, per la lunghezza ottimale della tabu list scegliamo come valore 120.

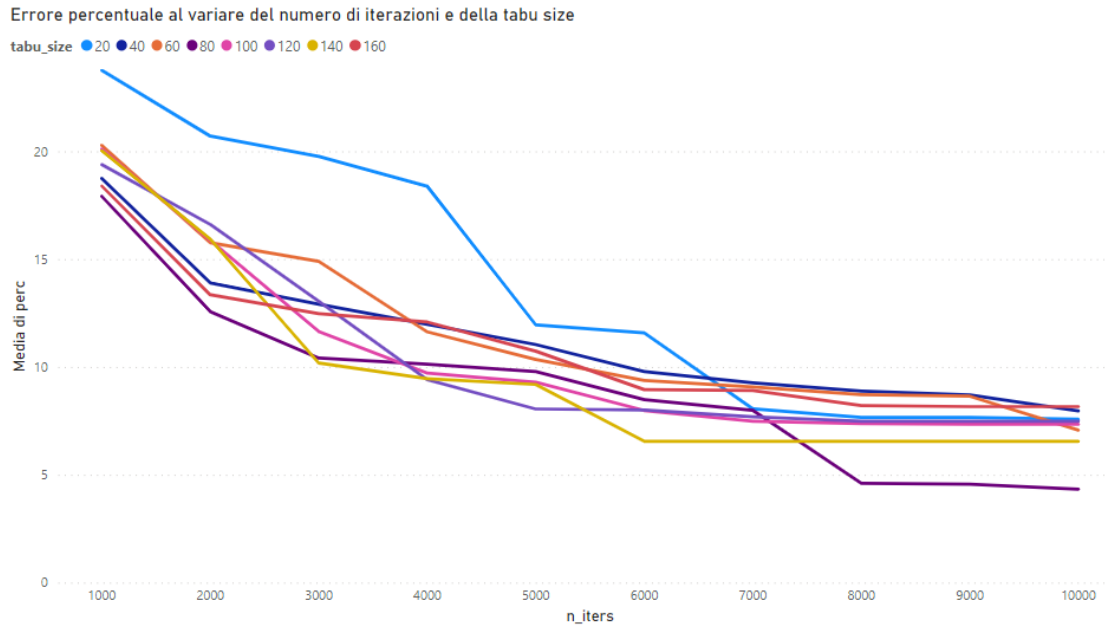


Figura 3.4: Fine-tuning Tabu Search

3.4 Confronto tra algoritmi

Per il confronto delle performance tra Or-Tools e Tabu Search vengono utilizzati dei problemi dalla libreria CVRPLIB [3] sviluppata da Uchoa et al. [11].

Per ogni istanza vengono eseguiti sia Or-Tools che Tabu Search e vengono confrontati la soluzione ottimale del problema con le due trovate dai solutori.

In termini di tempo la soluzione più veloce viene trovata da Or-Tools in quanto scritto e ottimizzato in C++, mentre la Tabu Search è stata scritta in Python e quindi è più lenta ad essere eseguita[3]. Questo però non è un problema visto che lo scopo della tesi non è la risoluzione istantanea ma una volta al giorno.

In termini quantitativi le soluzioni mediamente hanno un errore percentuale simile come è possibile vedere in Figura 3.5.

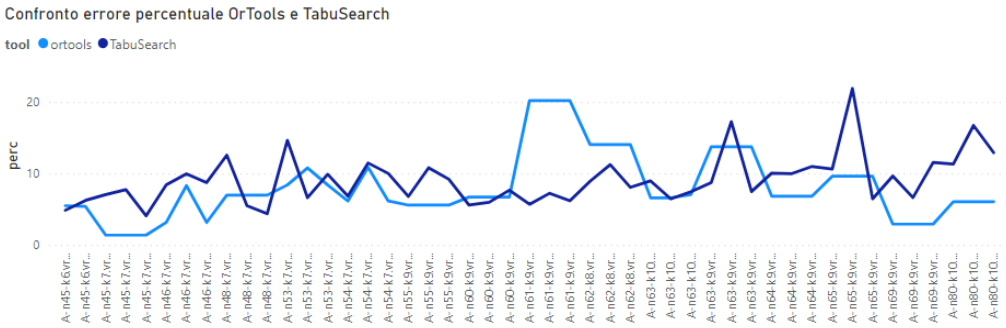


Figura 3.5: Confronto Or-Tools e Tabu Search

Vista la similitudine negli errori e la differenza nel tempo di risoluzione è stato scelto di fare l'analisi su più giorni (descritta nel Capitolo 4) utilizzando il solutore Or-Tools.

[3]Secondo analisi su vari problemi, C++ è in media 30 volte più veloce di Python [9]

Capitolo 4

Politiche per la selezione dei clienti

Nella risoluzione del problema multiperiodale si presenta la necessità di selezionare i clienti da servire quel giorno e quelli da seguire nei giorni seguenti sapendo che nel frattempo si aggiungeranno altri clienti.

Sono state create dall'Autore 5 politiche per la selezione dei clienti (Random, ASAP, ASAP_2, ALAP, Neighbour) che vengono descritte di seguito.

L'obiettivo è dimostrare quali sono le politiche più efficienti per l'utilizzo nell'analisi multiperiodale.

4.1 Definizione delle politiche

Per ciascuna politica di selezione dei clienti si opera un processo iterativo in quanto vengono selezionati dei clienti, si cerca la soluzione e nel caso questa sia non ammissibile viene cancellato un cliente e si cerca una nuova soluzione e così via.

L'ultimo giorno della simulazione vengono serviti tutti i clienti rimanenti anche se si supera il numero massimo di clienti servibili quel giorno. Ciò viene fatto per poter rendere comparabili le analisi sulle varie politiche. Nella realtà questo caso non accadrebbe mai perché non ci sarebbe una durata limitata del periodo di simulazione ma sarebbe un algoritmo che viene eseguito tutti i giorni.

Ogni giorno, ad eccezione degli ultimi due, viene stilato un elenco di clienti servibili, ma viene selezionato solo un numero limitato di clienti pari al numero massimo di ordini consegnabili quel giorno. Questa limitazione serve perché nella realtà non si possono consegnare un numero illimitato di ordini. Bisogna tener presente che questa selezione rischia di far scadere alcuni ordini che sono già arrivati all'ultimo giorno di consegna ammissibile.

Algorithm 2 Selezione dei clienti

```
1: Selezione della politica
2: G:= Insieme di clienti (all'inizio è vuoto)
3: for each day t in T do
4:     N ← Insieme dei nuovi clienti
5:     G ← Insieme dei clienti dei giorni precedenti con nuovi clienti
6:     S ← Insieme dei clienti selezionati per essere serviti
7:     G ← Insieme dei clienti posticipati
8:     Soluzione ← False
9:     while Soluzione == False do
10:        Risolve CVRP
11:        if CVRP è ammissibile then
12:            Soluzione ← True
13:        end if
14:        if Soluzione == False then
15:            Rimuove ultimo cliente in S e lo riaggiunge a G
16:        end if
17:    end while
18: end for
```

4.2 Random Policy

La *Random Policy* consiste in una scelta randomica dei clienti da servire tra tutti quelli disponibili. La priorità viene data ai clienti che hanno una data di consegna massima il giorno stesso così da ridurre al minimo gli ordini rimandati.

La selezione si divide in due step:

1. Selezione di clienti con data di consegna massima oggi o nei giorni precedenti
2. Selezione randomica tra i clienti rimanenti fino a riempire il numero massimo di clienti che si possono servire quel giorno

Qui di seguito possiamo vedere il codice Python usato per la selezione dei clienti:

```
1  def _random_customer_selection(df: pd.DataFrame, day: int, num_days:
       int, num_max_customers: int, seed: int = None):
2
3      # selezione clienti all'ultimo giorno o gia' scaduti
4      df_last_day = df[df['last_day'] <= day]
5      df_not_last_day = df[df['last_day'] > day]
6
7      # tra i clienti non all'ultimo giorno ne prendo il numero giusto
           per completare il numero prefissato
8      df_sample = df_not_last_day.sample(n=max(min(num_max_customers -
           len(df_last_day), len(df_not_last_day)), 0),random_state=seed)
9      # concateno i clienti all'ultimo giorno e quelli campionati
10     df_customer_selected = pd.concat([df_last_day, df_sample])
11     if day != num_days-1:
12        # prende solo il numero massimo di clienti prendibili
13        df_customer_selected=df_customer_selected[0:num_max_customers]
14     else:
15        # l'ultimo giorno vengono serviti tutti i clienti
16        df_customer_selected = df
17
18     # estraggo i clienti non selezionati e quindi posticipati
19     df_postponed=df[~df['index'].isin(df_customer_selected['index'])]
20
21     return df_customer_selected, df_postponed
```

Come anticipato nella Sezione 4.1, il controllo alla riga 11 serve a imporre che l'ultimo giorno tutti i clienti rimasti vengano serviti, anche se si supera il numero massimo di clienti servibili quel giorno.

Il numero di clienti da campionare nel comando alla riga 8 viene selezionato come differenza tra il numero di clienti che si possono servire quel giorno e il numero di clienti all'ultimo giorno già selezionati. Nel caso questa differenza sia maggiore del numero di clienti con ordini aperti vengono selezionati tutti i clienti disponibili. Viene infine aggiunto il calcolo del massimo tra il numero calcolato prima e 0 per evitare che ci sia un errore nel caso in cui il numero di clienti all'ultimo giorno di consegna disponibile sia maggiore del numero di clienti selezionabili.

Nel comando alla riga 13 vengono selezionati solo un numero limitato di clienti pari al numero massimo di ordini consegnabili quel giorno.

4.3 ASAP Policy

La *ASAP Policy (As Soon As Possible)* consiste nella selezione dei clienti da servire tra tutti quelli disponibili in modo tale da servire i clienti appena possibile.

Vengono selezionati i clienti dando priorità a quelli che hanno un giorno di consegna massima più vicino al giorno corrente per ridurre il più possibile il rischio di superare il giorno di consegna massimo. Nel caso di ordini da consegnare entro lo stesso giorno viene data priorità a chi è già stata rimandata la consegna così da ridurre l'insoddisfazione dei clienti.

L'idea principale dell'*ASAP Policy* è quindi di ridurre il tempo di attesa per ogni cliente. Questo può risultare una politica molto conveniente per i clienti e potrebbe essere una politica intelligente anche per l'azienda se la distribuzione della domanda dei clienti clienti fosse abbastanza uniforme nella regione o i costi di inventario fossero così elevati da incoraggiare una rapida eliminazione.

```
1  def _asap_customer_selection(df: pd.DataFrame, day: int, num_days:
       int, num_max_customers: int):
2      # ordina i clienti in base all'ultimo giorno di consegna
           disponibile e se sono gia' stati posticipati o no
3      df = df.sort_values(by=['last_day', 'yet_postponed'], ascending=[
           True, False])
4      if day != num_days-1:
5          # prende solo il numero massimo di clienti selezionabili
6          df_customer_selected = df[0:num_max_customers]
7      else:
8          # l'ultimo giorno vengono serviti tutti i clienti
9          df_customer_selected = df
10     # estraggo i clienti non selezionati e quindi posticipati
11     df_postponed=df[~df['index'].isin(df_customer_selected['index'])]
12
13     return df_customer_selected, df_postponed
```

4.4 ASAP_2 Policy

La *ASAP_2 Policy (As Soon As Possible)* consiste nella selezione dei clienti da servire tra tutti quelli disponibili in modo tale da servire i clienti appena possibile.

Vengono selezionati i clienti dando priorità a quelli a cui la consegna è già stata rimandata così da ridurre al minimo l'insoddisfazione dei clienti. Nel caso di ordini già rimandati viene data priorità a chi ha una data di consegna massima più vicina al giorno corrente per ridurre il più possibile il rischio di superare il giorno di consegna massimo.

Rispetto alla politica *ASAP* precedente, in questa viene data priorità all'insoddisfazione del cliente nel vedere il proprio ordine posticipato, invece che all'insoddisfazione dovuta al vedere il proprio ordine consegnato dopo la data massima di consegna.

```
def _asap_2_customer_selection(df: pd.DataFrame, day: int, num_days:
    int, num_max_customers: int):
    # ordina i clienti in base a se sono gia' stati posticipati o no
        e all'ultimo giorno di consegna disponibile
    df = df.sort_values(by=['yet_postponed', 'last_day'], ascending=[
        False, True])
    if day != num_days-1:
        # prende solo il numero massimo di clienti selezionabili
        df_customer_selected = df[0:num_max_customers]
    else:
        # l'ultimo giorno vengono serviti tutti i clienti
        df_customer_selected = df
    # estraggo i clienti non selezionati e quindi posticipati
    df_postponed=df[~df['index'].isin(df_customer_selected['index'])]

    return df_customer_selected, df_postponed
```

4.5 ALAP Policy

La *ALAP Policy (As Late As Possible)* consiste nella selezione dei clienti da servire tra tutti quelli la cui data di scadenza è il giorno stesso o uno dei precedenti. Viene data priorità agli ordini già rimandati e viene preso al massimo un numero di ordini pari al numero massimo di clienti servibili quel giorno.

Con questa politica il rischio di avere ordini che superano la data massima di consegna è elevato, però può essere conveniente nel caso di pochi ordini giornalieri, per cui gli ordini in sospeso vengono ritardati il più possibile, sperando che gli ordini futuri sfruttino meglio la capacità dei veicoli disponibili.

```
def _alap_customer_selection(df: pd.DataFrame, day: int, num_days:
    int, num_max_customers: int):
    # ordina i clienti in base a se sono gia' stati posticipati o no
    df = df.sort_values(by=['yet_postponed'], ascending=[False])
    # selezione clienti all'ultimo giorno o gia' "scaduti"
    df_customer_selected = df[df['last_day'] <= day]

    if day != num_days-1:
       # prende solo il numero massimo di clienti selezionabili
       df_customer_selected=df_customer_selected[0:num_max_customers]
    else:
       # l'ultimo giorno vengono serviti tutti i clienti
       df_customer_selected = df
    # estraggo i clienti non selezionati e quindi posticipati
    df_postponed=df[~df['index'].isin(df_customer_selected['index'])]

    return df_customer_selected, df_postponed
```

4.6 Neighbour Policy

La *Neighbour Policy (NP)* consiste nella selezione dei clienti da servire secondo un indice di compatibilità tra i vari ordini.

L'indice utilizzato è proposto in [1] e [12] ed esprime la convenienza di includere un cliente nell'insieme dei clienti selezionati, dato l'insieme dei clienti in attesa, la presenza/assenza di altri clienti nel vicinato e i giorni rimanenti da servire.

Gli indici di compatibilità sono usati per prendere in considerazione i potenziali risparmi derivanti dal servire insieme coppie di clienti i e j vicini tra loro, così da decidere se non sia conveniente rimandare il servizio di un tale cliente e servire i due clienti nello stesso periodo di tempo.

$$I_{ij} = \frac{c_{0i} + c_{0j} - c_{ij}}{2\,(c_{0j} + c_{0i})} \tag{4.1}$$

Analizzando l'indice possiamo notare che il numeratore indica il risparmio derivante dall'aver servito i clienti i e j sullo stesso percorso, anziché servire ciascuno di essi singolarmente. Il denominatore indica invece il costo dei due percorsi individuali, quindi I_{ij} è un indicatore relativo delle potenzialità di risparmio per servire i e j insieme. Un valore elevato di I_{ij} indica che potrebbe essere conveniente servire i clienti i e j sullo stesso percorso.

L'utilizzo di questo indice però non tiene in considerazione la posizione dei futuri ordini. Il calcolo delle distanze tra gli n clienti è costoso dal punto di vista computazionale, in quanto scala come $\mathcal{O}(n^2)$ anche se non tutti i clienti verranno poi serviti.

Per superare questi problemi si utilizza il centro delle celle a cui appartengono gli ordini, che è un numero costante, invece che la posizione esatta dei clienti.

Quindi, dopo aver calcolato le coppie di distanze $\widetilde{c}_{ij}$, con $i, j \in V_0 = \{0\} \cup V$ dove 0 è la posizione esatta del deposito e V è l'insieme delle celle, l'indice di risparmio approssimato diventa:

$$\widetilde{I}_{ij} = \frac{\widetilde{c}_{0i} + \widetilde{c}_{0j} - \widetilde{c}_{ij}}{2\,(\widetilde{c}_{0i} + \widetilde{c}_{0j})} \quad i, j \in V \tag{4.2}$$

Dato $I_{ij}, \forall i, j \in V$, è possibile calcolare l'insieme di celle vicine per una determinata cella $i \in V$:

$$V_\rho(i) = \left\{ j \in V : \widetilde{I}_{ij} \geq \rho \right\} \tag{4.3}$$

A un certo giorno $t \in T$, l'insieme V viene suddiviso nell'insieme di celle attive V^t, ossia le celle che sono associate alle posizioni dei clienti in attesa, e nell'insieme delle celle inattive $V \backslash V^t$. Quindi, stimiamo la convenienza a servire il cliente i al tempo t, calcolando l'indice $conv_i^t$, in base alle seguenti considerazioni:

- i clienti con pochi giorni di servizio disponibili possono essere molto urgenti da servire;
- conosciamo la probabilità p_{ij}^t che un cliente della cella j nelle vicinanze della cella i richieda un servizio il giorno successivo. Questo è semplicemente la probabilità associata alla cella j nella distribuzione multinomiale (vedi Sezione A.1.1);

- può essere conveniente rimandare il cliente j se p_{ij}^t è sufficientemente grande;
- la priorità tra gli ordini rinviati e quelli in scadenza deve essere mantenuta.

Queste osservazioni sono riassunte nella definizione di $conv_i^t$:

$$conv_i^t = \begin{cases} \frac{1}{b_i - t}\left(1 + \frac{1}{|V_\rho(i)\backslash V^t|}\sum_{j \in V_\rho(i)\backslash V^t}\left(1 - p_{ij}^t\right)\right) & b_i > t, V_\rho(i)\backslash V^t \neq \emptyset \\ \frac{M}{b_i - t} & b_i > t, V_\rho(i)\backslash V^t = \emptyset \\ M & b_i \leq t, \text{ non ancora posticipato} \\ M + \gamma & b_i \leq t, \text{ già posticipato} \end{cases} \tag{4.4}$$

dove b_i è l'ultimo giorno disponibile per l'ordine i, M è un parametro che garantisce che, quando possibile, tutti gli ordini che hanno raggiunto la loro data di scadenza saranno selezionati nel giorno t, e γ è un termine di rinforzo che dà la massima priorità agli ordini già posticipati.

Infine, la selezione dei clienti al giorno t viene effettuata ordinando i clienti in base al valore decrescente di $conv_i^t$ e prendendo i primi k elementi fino a riempire il numero massimo di clienti servibili.

```
def _neighbour_customer_selection(df: pd.DataFrame, day: int,
    num_days: int, num_max_customers: int, list_compatible_cells,
    cell_probabilities, M, gamma):
    # insieme delle celle con un cliente da servire
    cells_with_orders = set(df['cell_id'])
    # calcolo dell'indice di compatibilita' della cella
    df = df.apply(lambda row: _index_compatibility_cell(row, day,
        list_compatible_cells[int(row['cell_id'])], cells_with_orders,
         cell_probabilities, M, gamma), axis=1)
    # ordine i clienti da servire secondo l'indice di compatibilita'
    df = df.sort_values(by=['index_compatibility'],ascending=[False])

    if day != num_days-1:
        # prende solo il numero massimo di clienti selezionabili
        df_customer_selected = df[0:num_max_customers]
    else:
        # l'ultimo giorno vengono serviti tutti i clienti
        df_customer_selected = df

    # estraggo i clienti non selezionati e quindi posticipati
    df_postponed=df[~df['index'].isin(df_customer_selected['index'])]

    return df_customer_selected, df_postponed
```

```
def _index_compatibility_cell(row, day, compatible_cells,
    cells_orders, cell_probabilities, M, gamma):
    # numero di giorni che rimangono per la consegna
    time_distance = row['last_day'] - day
    # celle compatibili che al momento non hanno ordini attivi
    compat_no_orders_cells = set(compatible_cells).difference(
        cells_orders)
    # se oggi e' l'ultimo giorno disponibile per la consegna o e' gia
        ' passato
    if time_distance <= 0:
        # suddivisione della casistica di ordini gia' posticipati e
            nuovi
        if row['yet_postponed']:
            index_compatibility = M + gamma
        else:
            index_compatibility = M
    else:  # se oggi non e' l'ultimo giorno per la consegna
        num_cells_no_orders = len(compat_no_orders_cells)
        # se tutte le celle compatibili hanno ordini attivi
        if num_cells_no_orders == 0:
            index_compatibility = M/time_distance
        # se alcune celle compatibili non hanno ordini attivi
        else:
            index_compatibility = 1/time_distance*(1+1/
                num_cells_no_orders*(num_cells_no_orders-
                cell_probabilities[list(compat_no_orders_cells)].sum()
                ))
    # aggiunge l'indice di compatibilita' alla riga
    row['index_compatibility'] = index_compatibility

    return row
```

4.6.1 Fine-tuning dei parametri

I parametri da scegliere nella policy *neighbour* sono M, γ e ρ.

Per il fine-tuning vengono generati dei problemi al variare del *seed*. Il problema ha una durata di 7 giorni e ogni giorno vengono generati nuovi clienti.

Per vedere il valore migliore della funzione obiettivo al variare dei parametri scelti è utilizzato l'approccio *grid search* e i possibili valori sono:

- $M \in \{4, 4.5, 5\}$
- $\gamma \in \{0.3, 0.5, 0.7, 0.9\}$
- $\rho \in \{0.1, 0.15, 0.2, 0.25, 0.3\}$

Nella Tabella 4.1 possiamo vedere il valore medio della funzione obiettivo al variare di γ e M. Nella maggior parte dei seed possiamo notare che i valori migliori si ottengono quando abbiamo $\gamma = 0.3$ e $M = 4$ e quindi prendiamo quelli come valori ottimali.

Average of objective_value		Seed			
γ	M	100	202	97	15
0,3	4	24731,2	17409,2	15597,0	24152,6
0,3	4,5	24778,0	17443,8	15643,2	24142,8
0,3	5	24739,8	17468,4	15629,0	24147,4
0,5	4	24735,0	17416,6	15653,2	24114,8
0,5	4,5	24754,8	17496,8	15638,2	24140,2
0,5	5	24743,6	17446,4	15632,8	24148,8
0,7	4	24776,6	17415,6	15670,0	24277,2
0,7	4,5	24800,8	17461,2	15641,0	24106,4
0,7	5	24743,2	17433,6	15723,2	24305,6
0,9	4	24784,0	17420,0	15638,2	24172,0
0,9	4,5	24731,0	17436,6	15625,6	24111,6
0,9	5	24796,4	17429,8	15638,0	24213,4

Tabella 4.1: Fine-tunining della neighbour policy su γ e M

objective_value			Seed			
M	γ	ρ	100	202	97	15
4	**0,3**	**0,1**	24898	17891	15839	24042
4	**0,3**	**0,15**	24614	17468	15418	24105
4	**0,3**	**0,2**	24965	16696	15997	23749
4	**0,3**	**0,25**	24623	17475	15525	24270
4	**0,3**	**0,3**	24556	17516	15206	24597

Tabella 4.2: Fine-tunining della neighbour policy su ρ con $\gamma = 0.3$ e $M = 4$

Nella Tabella 4.2 possiamo vedere che considerando $\gamma = 0.3$ e $M = 4$ il valore della funzione obiettivo al variare di ρ ha i valori migliori quando $\rho = 0.3$.

Di conseguenza scegliamo come migliore configurazione della policy *neighbour* per le successive analisi:

- $\gamma = 0.3$
- $M = 4$
- $\rho = 0.3$

4.7 Confronto tra policy

Confrontiamo infine i risultati delle varie policy che si possono usare per selezionare i clienti ogni giorno.

Il problema su cui abbiamo confrontato le varie policy ha la seguente struttura:

- La durata della simulazione è 7 giorni
- Ogni giorno ci sono 200 nuovi ordini, tranne negli ultimi due dove non arrivano nuovi ordini
- I primi due giorni vengono consegnati al massimo 150 ordini, dal terzo al penultimo giorno al massimo 200 ordini, l'ultimo giorno vengono consegnati tutti gli ordini rimanenti
- Gli ordini hanno una finestra per la consegna che va dal giorno di creazione dell'ordine fino a 4 giorni dopo o fino all'ultimo giorno della simulazione
- Gli ordini vengono consegnati anche dopo la data massima di consegna

Nella Figura 4.3a possiamo vedere la distribuzione di probabilità dell'arrivo di un nuovo ordine e in basso a sinistra possiamo vedere la posizione del deposito segnato con una croce (+). Nella Figura 4.3b possiamo vedere gli ordini creati il giorno 1. Nella Figura 4.3c possiamo vedere gli ordini che sono aperti il giorno 2, ovvero gli ordini non consegnati il giorno 1 uniti agli ordini creati il giorno 2 e così via nelle Figure successive. Fino ad arrivare alla Figura 4.3h dove possiamo vedere che al giorno 7 non ci sono più ordini aperti.

Una prima analisi va fatta sulla consegna entro la scadenza degli ordini. Secondo il problema creato utilizzando tutte le policy tutti gli ordini vengono consegnati entro la scadenza, ad eccezione della *ALAP* (vedi Figura 4.1). Con quest'ultima policy mediamente ci sono 34 ordini consegnati dopo la data massima di consegna. Di conseguenza non è considerabile come una policy accettabile.

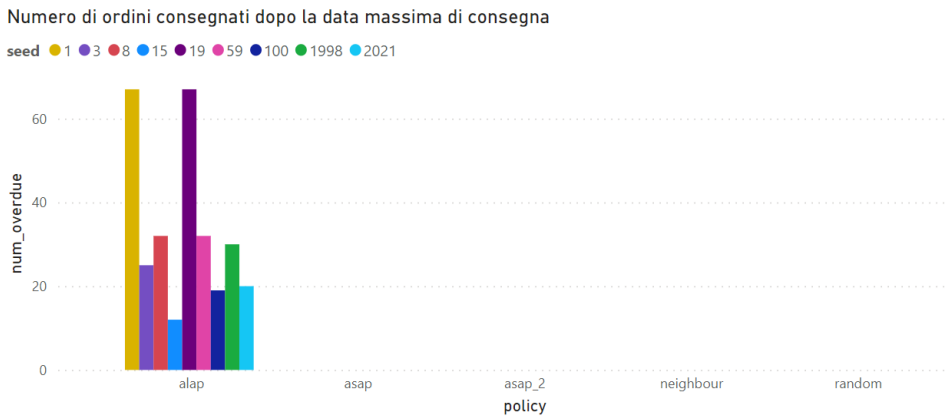


Figura 4.1: Numero di ordini consegnati dopo la data massima di consegna

Come seconda analisi possiamo confrontare il valore della funzione obiettivo al variare delle policy. Per il confronto utilizziamo l'improvement rispetto alla policy random:

$$impr_{policy} = \frac{cost_{random} - cost_{policy}}{cost_{policy}} \tag{4.5}$$

Dal grafico in Figura 4.2 e dalla Tabella 4.3 possiamo notare come la policy *neighbour* sia quella con l'improvement maggiore rispetto a tutte le altre. Questo avviene perché non è presa in considerazione solo la data massima di consegna, ma è considerata anche la posizione degli ordini e la probabilità di ordini futuri.

Quindi visto l'improvement possiamo considerare la *Neighbour* policy come la miglior policy per la selezione dei clienti.

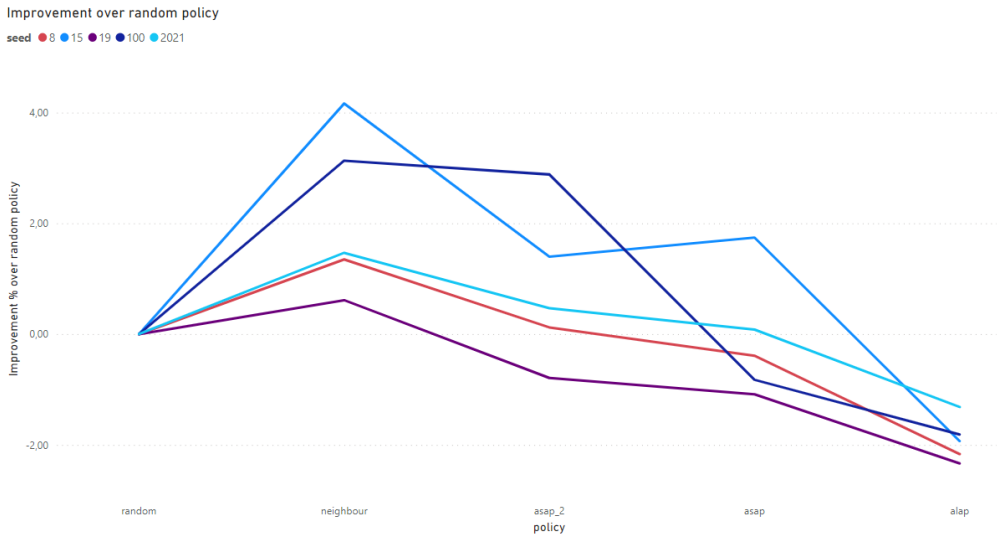
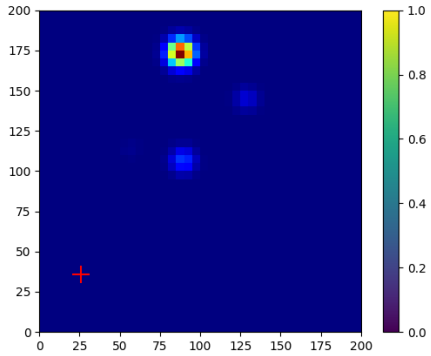


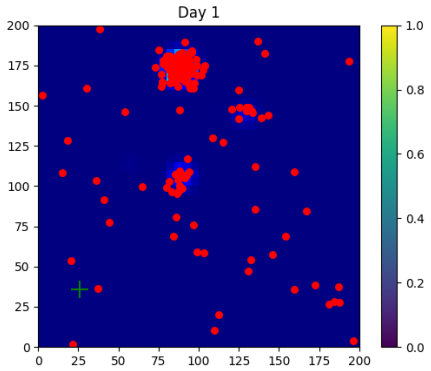
Figura 4.2: Confronto del miglioramento percentuale rispetto alla politica random

Policy\Seed	1	3	8	15	19	59	100	1998	2021
alap	28046	34672	26815	24653	33640	22239	24796	31659	35489
asap	28189	34594	26337	23762	33215	22265	24549	31255	34993
asap_2	28410	34904	26202	23843	33116	22541	23665	31314	34859
random	27648	34665	26234	24177	32854	21933	24347	31312	35022
neighbour	27376	34087	25885	23210	32654	21917	23608	31188	34515

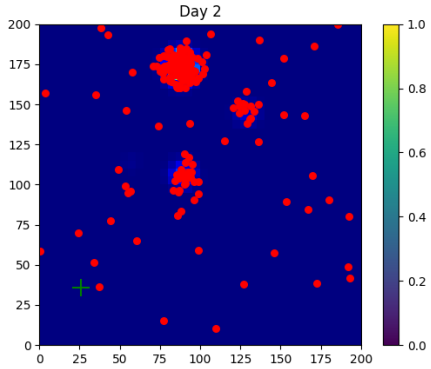
Tabella 4.3: Valore della funzione obiettivo al variare del seed e della policy



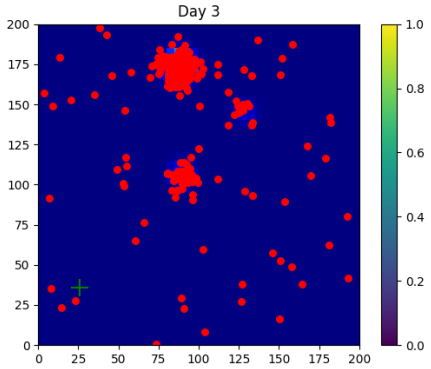
(a) Distribuzione di probabilità



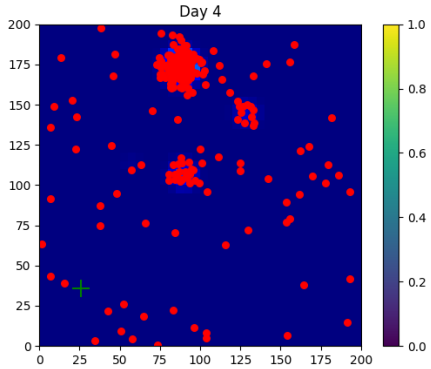
(b) Clienti da servire il giorno 1



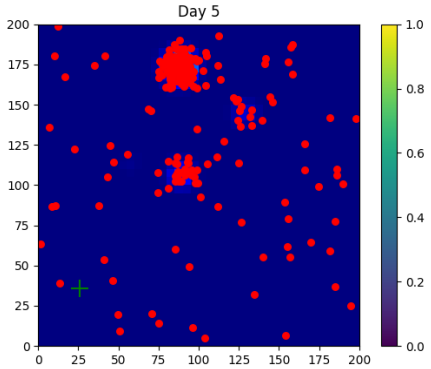
(c) Clienti da servire il giorno 2



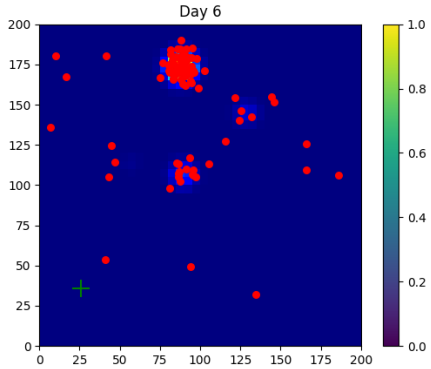
(d) Clienti da servire il giorno 3



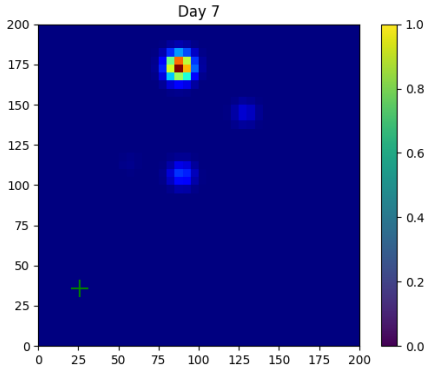
(e) Clienti da servire il giorno 4



(f) Clienti da servire il giorno 5



(g) Clienti da servire il giorno 6



(h) Clienti da servire il giorno 7

Figura 4.3: Simulazione su 7 giorni

Capitolo 5

Sviluppi futuri

Il lavoro svolto in questa Tesi offre alcuni punti di partenza per futuri sviluppi che possono essere studiati sia come argomento di un futuro dottorato di ricerca sia come applicazione a dataset reali.

Come primo sviluppo potrebbe esserci l'implementazione del codice dell'algoritmo Tabu Search scritto in un linguaggio compilato come C++. In questo modo il tempo di esecuzione sarebbe ridotto e di conseguenza il confronto analizzato nel Capitolo 3 potrebbe essere effettuato anche sul tempo e non solo sulle performance quantitative. Inoltre, ciò permetterebbe uno sviluppo più agevole di un algoritmo più affinato che potrebbe superare le performance di Or-Tools.

Come altro sviluppo potrebbe esserci il miglioramento della *neighbour policy* e la creazione di nuove policy per la selezione dei clienti. Inoltre, l'utilizzo di dati reali permetterebbe di affinare l'algoritmo nella parte di generazione di nuovi clienti e nella selezione di un numero più calibrato di clienti.

Capitolo 6

Conclusione

Per quanto riguarda il primo obiettivo della tesi, relativo al confronto tra algoritmi, è stato constatato che sul problema preso in considerazione gli algoritmi Or-Tools e Tabu Search performano in maniera similare dal punto di vista quantitativo, mentre l'implementazione in linguaggi diversi rende più veloce Or-Tools. La traduzione dell'algoritmo Tabu Search in C++ ridurrebbe il tempo di esecuzione e lo renderebbe equivalente a Or-Tools.

Il secondo obiettivo della tesi, relativo alla creazione e al confronto delle policy per la selezione dei clienti, è stato raggiunto con successo in quanto è stata sviluppata la *neighbour policy* che permette la diminuzione media dei costi di oltre il 2% rispetto alla policy di riferimento. Questa policy apre la strada a ulteriori diminuzioni dei costi in caso di fine-tuning con dati reali.

Appendice A

Analisi del codice

I vari processi di ottimizzazione e le varie analisi effettuate vengono controllati dalla funzione *main* con l'utilizzo di varie funzioni che vengono analizzate nel dettaglio di seguito.

Nella funzione *main* il codice è diviso in 6 parti.

La prima parte è un esempio di creazione della distribuzione di probabilità dell'arrivo di un nuovo ordine. La funzione principale utilizzata è *create_distribution* descritta nella sezione A.1.1.

La seconda parte è relativa al fine-tuning dell'algoritmo Tabu Search sviluppato dall'Autore. Avviene una grid search sui parametri relativi alla lunghezza della tabu list e al numero di iterazioni dell'algoritmo. La classe contenente tutte le funzioni per utilizzare l'algoritmo si chiama *TabuSearch* ed è descritta nella sezione A.4.1. I risultati di ciascuna esecuzione dell'algoritmo sono salvati in file *.csv* che diventa poi input per un report *PowerBI* da cui è estratto il grafico in Figura 3.4. I risultati del fine-tuning sono descritti nella Sezione 3.3.

La terza parte è relativa al confronto tra l'algoritmo Or-Tools di Google e l'algoritmo Tabu Search sviluppato dall'Autore. Sui CVRP esistenti (vedi dettagli nella sezione A.1.4) i due algoritmi performano in modo similare. I risultati dettagliati sono descritti nella Sezione 3.4.

Nella quarta parte viene effettuato il fine-tuning della policy *neighbour*. Avviene una grid search sui parametri M, γ e ρ utilizzati nelle funzioni *_index_compatibility_cell* e *_compatible_cells* descritte nelle Sezioni A.2.3 e A.1.3. I risultati del fine-tuning sono descritti nella Sezione 4.6.

La quinta parte della funzione *main* è relativa al confronto tra le 5 policy descritte nel Capitolo 4. Vengono confrontate su problemi generati autonomamente: viene prima generata la distribuzione della popolazione (descritta nella Sezione A.1.1), poi per ogni

giorno vengono simulati dei nuovi ordini e scelti quali ordini tra i nuovi e i passati consegnare quel giorno preso in considerazione seguendo una policy specifica (la descrizione delle funzioni utilizzate è nella Sezione A.2). La migliore policy è la *neighbour*. I risultati dettagliati sono descritti nella Sezione 4.7.

Nell'ultima parte vi è il codice per l'ottimizzazione multiperiodale con già inseriti i parametri migliori ottenuti nelle analisi precedenti.

A.1 File: input.py

A.1.1 Funzione: create_distribution

La funzione va usata quando non vi è un file con una distribuzione di probabilità prestabilita e bisogna crearne una.

Se non vengono dati parametri in input si considera un'area di 200 km per 200 km suddivisa in quadrati (celle) di lato 5 km.

Vengono create in posizioni casuali 5 città (essendo casuali potrebbero essere parzialmente sovrapposte), ogni città è rappresentata da una distribuzione normale multivariata. Vengono campionati un numero di persone equivalente alla dimensione della popolazione (numero anch'esso casuale).

Vengono contati per ogni cella tutti gli elementi campionati dalle varie distribuzioni che rappresentano le città. Viene aggiunta una densità minima di abitanti per km quadrato così da avere una probabilità di ordini anche al di fuori delle città.

Normalizzando il numero di elementi per cella viene trovata la probabilità di avere un ordine nuovo da quella cella.

É possibile creare l'immagine delle città simulate e l'immagine della distribuzione di probabilità con inserita anche la posizione del deposito.

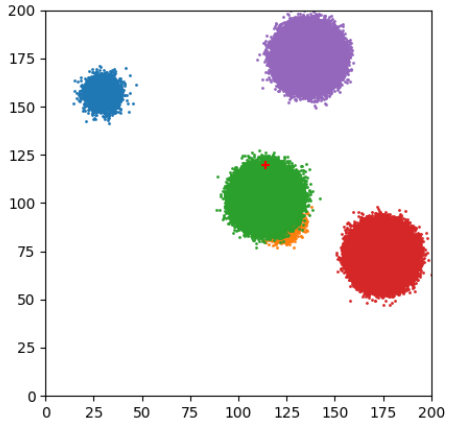


Figura A.1: Esempio città simulate

A.1.2 Funzione: print_heat_map_and_orders

La funzione crea l'heat map della distribuzione di probabilità e sopra di essa vengono inserite le posizioni degli ordini e del deposito. Esempi degli output della funzione si possono vedere nelle Figure A.2 e A.3.

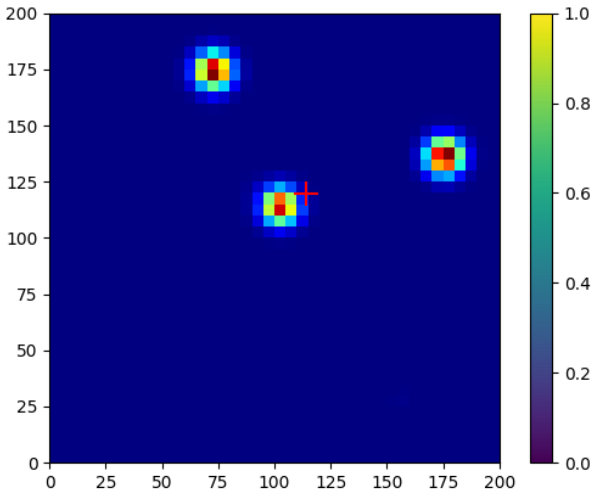


Figura A.2: Distribuzione di probabilità degli ordini

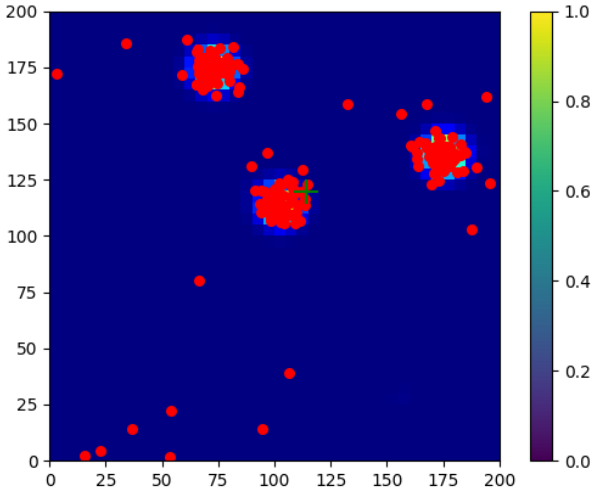


Figura A.3: Posizione ordini simulati

A.1.3 Funzione: compatible_cells

La funzione crea per ogni cella una lista di celle che hanno un indice di compatibilità sopra un threshold ρ dato in input. L'inserimento in questa lista significa che includere gli ordini di quelle celle nella stessa route di quelli della cella in considerazione porta un risparmio in termini di distanza percorsa.

L'indice viene calcolato seguendo l'indice di compatibilità proposto nell'articolo [1]

$$I_{ij} = \frac{c_{0i} + c_{0j} - c_{ij}}{2\,(c_{0j} + c_{0i})} \tag{A.1}$$

e impostando che due ordini nella stessa cella sono sempre compatibili.

A.1.4 Funzione: load

Vengono caricati dei problemi dalla libreria CVRPLIB [3].

L'output della funzione è un DataFrame con tutti gli ordini presenti nel problema caricato e un dizionario con tutte le informazioni di base del problema (superficie del problema, dimensione e numero delle celle, posizione del deposito, ...).

A.2 File: Customer.py

A.2.1 Classe: Customer

La classe contiene tutte le informazioni relative ai clienti: id, domanda e posizione.

É presente anche una funzione che simula i nuovi ordini. Data la distribuzione multinomiale viene campionato quanti ordini in ciascuna cella ci sono. Poi per ogni cella vengono campionate da due distribuzioni uniformi le posizioni precise degli ordini all'interno della cella.

A.2.2 Funzione: selection_customers

Questa funzione gestisce la selezione dei clienti per il CVRP tra tutti gli ordini aperti in quello specifico giorno. Per ogni giorno vengono selezionati tra i 150 e i 200 ordini da consegnare.

Ci sono 5 politiche per la selezione dei clienti da servire:

- *Random*: vengono selezionati tutti gli ordini che devono essere consegnati entro il giorno in considerazione e poi casualmente vengono selezionati tutti gli altri clienti necessari per raggiungere il numero massimo di clienti a cui è possibile consegnare nel giorno considerato.
- *ASAP (As Soon As Possible)*: vengono selezionati il numero massimo di ordini che è possibile consegnare. Viene data priorità agli ordini che hanno una data massima di consegna ravvicinata e sono già stati posticipati.

- *ASAP_2 (As Soon As Possible)*: vengono selezionati il numero massimo di ordini che è possibile consegnare. Viene data priorità agli ordini che sono già stati posticipati e poi a quelli che hanno una data massima di consegna ravvicinata.

- *ALAP (As Late As Possible)*: vengono selezionati tutti i clienti la cui data di consegna massima è nel giorno considerato o è già passata. Nel caso non fosse possibile consegnarli quel giorno perché ci sono troppi ordini da consegnare allora verranno consegnati il giorno successivo.

- *Neighbour*: vengono selezionati i clienti in base all'indice di compatibilità proposto nell'articolo [1] e in [12] (e calcolato nella funzione *_index_compatibility_cell*) che valuta l'inclusione di un cliente in base alla presenza o all'assenza di clienti nelle celle vicine e ai giorni rimanenti per la consegna.

A.2.3 Funzione: _index_compatibility_cell

La funzione calcola l'indice di compatibilità $conv_i^t$ proposto nell'articolo [1] e in [12]:

$$conv_i^t = \begin{cases} \frac{1}{b_i - t}\left(1 + \frac{1}{|V_\rho(i)\backslash V^t|}\sum_{j \in V_\rho(i)\backslash V^t}\left(1 - p_{ij}^t\right)\right) & b_i > t, V_\rho(i)\backslash V^t \neq \emptyset \\ \frac{M}{b_i - t} & b_i > t, V_\rho(i)\backslash V^t = \emptyset \\ M & b_i \leq t, \text{ non ancora posticipato} \\ M + \gamma & b_i \leq t, \text{ già posticipato} \end{cases} \tag{A.2}$$

dove t è il numero del giorno attuale e b_i è il giorno di consegna massimo, M è un valore elevato tale che appena possibile l'ordine sia consegnato perché è in considerazione il giorno massimo di consegna o è già passato. γ è un valore tale che venga data priorità agli ordini già posticipati almeno una volta. $V_\rho(i)$ è l'insieme delle celle compatibili e V^t è l'insieme delle celle con un ordine aperto nel giorno in considerazione. p_{ij}^t è la probabilità che ci sia un ordine in una delle celle vicine nel giorno seguente.

A.3 File: OrTools_solver.py

A.3.1 Funzione: create_distance_matrix

Date in input le posizioni del deposito e dei clienti, la funzione calcola la *distance matrix* utilizzando la metrica euclidea.

A.3.2 Funzione: ortools_solver

La funzione trova la soluzione ottima del problema CVRP utilizzando il solver *Or-Tools* di Google [8].

L'obiettivo dell'ottimizzazione è la minimizzazione della lunghezza del percorso dei veicoli nella consegna dei pacchi. I vincoli presenti sono relativi al carico massimo che ciascun veicolo può trasportare in ciascun momento del tragitto.

Or-Tools lavora con numeri interi quindi all'inizio tutte le distanze vengono trasformate da kilometri a metri così da avere risultati più precisi e a fine calcolo il risultato viene trasformato nuovamente in kilometri.

A.3.3 Funzione: print_solution

La funzione stampa sulla console il risultato dell'ottimizzazione che utilizza Or-Tools.

Vengono stampati il valore della funzione obiettivo e il tragitto seguito da ciascun veicolo.

A.4 File: TabuSearch_solver.py

A.4.1 Classe: TabuSearch

La classe contiene tutte le informazioni necessarie alla creazione del modello di ottimizzazione e tutte le funzioni necessarie a trovare la soluzione ottimale usando l'algoritmo Tabu Search.

A.4.2 Funzione: generate_random_solution

La funzione trova una soluzione iniziale che rispetta tutti i vincoli.

A.4.3 Funzione: solution_feasible

Data in input una soluzione trovata, viene analizzata e viene restituito *True* o *False* in base a se rispetta tutti i vincoli e quindi se è ammissibile o meno.

A.4.4 Funzione: solution_cost

La funzione calcola il costo della soluzione sommando le tappe del tragitto di tutti i veicoli utilizzando la soluzione trovata.

A.4.5 Funzione: find_neighborhood

La funzione ha lo scopo di trovare delle soluzioni vicine a quella ottimale trovata in questo momento.

Per trovare soluzioni candidate vicine avviene una delle seguenti azioni:

- lo swap di due ordini presenti sui percorsi di due veicoli differenti
- lo spostamento di un ordine ad un veicolo attualmente senza ordini
- lo spostamento di un ordine "posseduto" da un veicolo con solo quell'ordine ad un veicolo con già altri ordini

A.4.6 Funzione: search

La funzione trova la soluzione ottima del problema CVRP utilizzando Tabu Search.

- Inizialmente viene generata una soluzione ammissibile causale.
- Poi inizia un loop con 5000 iterazioni (valore di default che può essere modificato a seconda della priorità tra velocità e precisione).
- Vengono trovate delle soluzioni candidate vicine e si analizza se non erano state precedentemente trovate e se sono migliori della soluzione ottimale attuale.
- Una volta trovata una soluzione migliore viene salvata e si prosegue con l'iterazione successiva.
- Infine vengono stampate sulla console il risultato dell'ottimizzazione e il percorso dei veicoli utilizzati.

A.5 Qualità del codice

Per analizzare la qualità del codice scritto dall'Autore è stato utilizzato il tool DeepSource [2]. Il suo utilizzo ha permesso di migliorare e pulire il codice fino a non ottenere più problemi di performance, sicurezza, ... come è possibile vedere della Figura A.4.

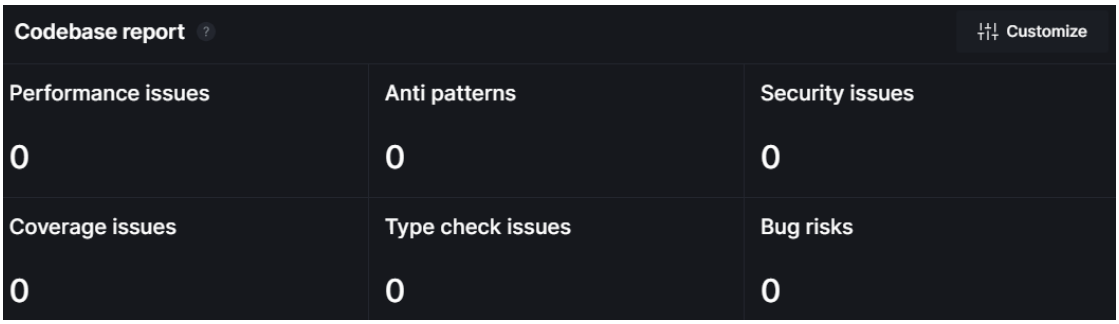


Figura A.4: Code quality analysis by DeepSource

Bibliografia

[1] Maria Albareda-Sambola, Elena Fernández, and Gilbert Laporte. The dynamic multiperiod vehicle routing problem with probabilistic information. *Computers & Operations Research*, 48:31–39, 2014. URL https://www.sciencedirect.com/science/article/pii/S0305054814000458.

[2] DeepSource Corp. deepsource. 2022. URL https://deepsource.io/about/.

[3] CVRPLIB. Capacitated Vehicle Routing Problem Library. URL http://vrp.galgos.inf.puc-rio.br/index.php/en/.

[4] Engali. Mathematical optimization. 2022. URL https://www.engati.com/glossary/mathematical-optimization.

[5] Pier Giribone. La metaeuristica della Tabu Search: Teoria ed applicazioni. 2015. URL https://www.researchgate.net/publication/287815616_La_metaeuristica_della_Tabu_Search_Teoria_ed_applicazioni.

[6] Fred Glover and Manuel Laguna. Tabu Search I. *Informs Journal on Computing*, 1999.

[7] Google. Capacity Constraints - Or-Tools. 2022. URL https://developers.google.com/optimization/routing/cvrp.

[8] Google. Or-Tools. 2022. URL https://developers.google.com/optimization.

[9] Jonas Mellin. How fast is C++ compared to Python? 2021. URL https://www.quora.com/How-fast-is-C-compared-to-Python-4/answer/Jonas-Mellin.

[10] Alessandro Soldano. L'algoritmo "Tabu Search" per l'ottimizzazione, teoria ed applicazioni. *Master's thesis, Università degli Studi di Padova, Facoltà di Ingegneria*, 2011.

[11] Eduardo Uchoa, Diego Pecin, Artur Pessoa, Marcus Poggi, Thibaut Vidal, and Anand Subramanian. New benchmark instances for the Capacitated Vehicle Routing Problem. *European Journal of Operational Research*, 257(3):845–858, 2017. URL https://www.sciencedirect.com/science/article/pii/S0377221716306270.

[12] Chiara Vercellino. A two-step optimization approach for a stochastic multi-stage capacitated vehicle routing problem. *Master's thesis, Politecnico di Torino*, 2020.

[13] Wikipedia. Problema del commesso viaggiatore. 2022. URL `https://it.wikipedia.org/wiki/Problema_del_commesso_viaggiatore`.

[14] Wikipedia. Tabu search. 2022. URL `https://it.wikipedia.org/wiki/Tabu_search`.