UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA

Progetto di Fondamenti di Intelligenza Artificiale

SmartCargo

Roksana Duda

0512114326

Mariapia Sorrentino

0512113750

Paolo Murino

0512116057

Andreea C.C. Oprisescu

0512114104

ANNO ACCADEMICO 2023/2024

Link repository GitHub

Contents

1 Introduzione

Per maggiori informazioni sul contesto trattato, si riporta la documentazione completa del progetto: **Link Drive:** 2023-C07-Final-IS-SmartCargo

1.1 Obiettivi

Il Porto di Valencia sta pianificando l'implementazione di un sistema avanzato finalizzato a migliorare la sicurezza, l'efficienza operativa e a ridurre l'impatto ambientale delle attività portuali.
L'obiettivo primario è sviluppare un'applicazione web basata su tecnologie all'avanguardia, integrando l'intelligenza artificiale in modo da individuare i percorsi ottimali, ovvero i più brevi, per gli autotrasportatori che intendono accedere al porto. La focalizzazione sull'ottimizzazione dei percorsi mira a migliorare la gestione del traffico portuale, riducendo i tempi di transito ed attesa e contribuendo a un ambiente più sicuro e regolamentato.

1.2 Sistema attuale

La situazione attuale nel Porto di Valencia è caratterizzata da una notevole lacuna causata dall'assenza di un sistema sofisticato di monitoraggio e tracciamento dei percorsi dei veicoli impiegati nelle operazioni di carico e scarico delle merci. Questa carenza rappresenta un rischio reale, poiché i mezzi di trasporto, in particolare gli autocarri, potrebbero intraprendere percorsi non autorizzati, mettendo a rischio l'integrità delle operazioni portuali. L'assenza di un sistema di tracciamento dei percorsi costringe l'autista a seguire un percorso predeterminato basato esclusivamente sulla segnaletica e sulle indicazioni disponibili, causando frequenti ritardi e inefficienze nel raggiungimento della destinazione. La mancanza di tale sistema rappresenta una criticità che potrebbe compromettere la sicurezza e l'efficienza complessiva delle operazioni portuali.

1.3 Specifica PEAS

L'ambiente è stato descritto utilizzando la specifica PEAS, ovvero Performance, Environment, Actuators, Sensors.

- **P:** sono le misure di prestazione adottate per valutare l'operato di un agente. In questo caso vogliamo che l'agente sia in grado di calcolare i percorsi ottimali per raggiungere un punto di destinazione.
- **E:** descrizione degli elementi dell'ambiente. Il nostro ambiente è una mappa rappresentata come un grafo in cui i nodi rappresentano coordinate sulla mappa e gli archi rappresentano le connessioni tra le coordinate.
- **A:** gli attuatori a disposizione dell'agente per intraprendere le azioni. Nel nostro caso l'attuatore è rappresentato dall'interfaccia utente, la quale consente la visualizzazione del percorso risultante.
- **S:** i sensori attraverso i quali riceve gli input percettivi. In questo caso sono rappresentati dalla scelta del punto di destinazione e dal dataset di punti con le relative coordinate geografiche.

1.3.1 Caratteristiche dell'ambiente

- **Completamente osservabile:** L'agente ha accesso a tutte le informazioni necessarie per prendere decisioni informate. In questo caso, il grafo rappresenta l'intero ambiente e tutte le informazioni sui nodi e sugli archi sono note all'agente. Non ci sono informazioni nascoste o inaccessibili.
- **Deterministico:** Le azioni dell'agente e le loro conseguenze sono ben definite e non soggette a casualità o incertezza.
- **Episodico:** Ogni episodio consiste nel calcolare un percorso da un punto all'altro sulla mappa.
- **Statico:** L'ambiente non cambia nel corso del tempo mentre l'algoritmo A* sta individuando il percorso ottimale. Le informazioni sul grafo, inclusi i nodi e gli archi, rimangono costanti durante l'esecuzione dell'algoritmo.

- **Discreto:** Le azioni prese dall'agente nell'ambiente sono discrete e definite dal movimento da un nodo del grafo a un altro. L'agente può scegliere solo tra un insieme discreto di azioni.
- **Singolo agente:** Un unico agente cerca il percorso ottimale sulla mappa.

2 Formulazione del problema

- **Stato iniziale:** È rappresentato dalla radice del grafo, identificata come il punto di accesso al porto.
- **Azioni:** Dato il nodo corrente, le azioni sono le possibili scelte attuabili dall'agente.
- **Modello di transizione:** Descrive il nodo raggiungibile per ogni azione attuabile dall'agente nel nodo corrente.
- **Test obiettivo:** Determina se il nodo corrente è lo stato obiettivo, ossia il punto di destinazione.
- **Costo di cammino:** Funzione che determina il costo numerico per raggiungere la destinazione, rappresentato in chilometri.

3 Ricerca Informata

Tra i numerosi algoritmi di ricerca disponibili, è stato scelto l'algortimo A* per diverse ragioni.
In primo luogo, a parità di euristica, nessun altro algoritmo espande meno nodi senza rinunciare all'ottimalità.
Utilizzando una funzione euristica per guidare la ricerca, A* può garantire una rapida convergenza verso la soluzione ottimale, riducendo il numero di nodi esplorati. Questa caratteristica è fondamentale per ottimizzare l'uso della memoria: pur mantenendo in memoria tutti i nodi generati durante la ricerca, A* non necessita di mantenere una lista di nodi così estesa, poiché esclude alcuni nodi dalla considerazione in base alla stima euristica. Ciò consente di mantenere sotto controllo l'uso della memoria, garantendo al contempo l'ottimalità della soluzione trovata.

3.1 Algoritmo di ricerca A*

A* è un algoritmo di ricerca informata che sfrutta conoscenze specifiche del problema, così da essere il più efficiente possibile.
Questa conoscenza aggiuntiva è rappresentata dall'*euristica h(n)*, una funzione che indica il costo stimato più conveniente dallo stato del nodo n allo stato obiettivo.

Il nodo da espandere viene scelto sulla base di una *funzione di valutazione f(n)*, data combinando *g(n)*, ovvero il costo di passo per raggiungere il nodo n, con *h(n)*. In pratica, $f(n) = g(n) + h(n)$, con $h(n) > 0$ e $h(goal) = 0$.

A* ha diverse proprietà utili:

1. In primo luogo, la *completezza* di A*. Poiché ci troviamo di fronte a un problema di ricerca su un grafo finito, A* è in grado di trovare una soluzione se questa esiste. Questo garantisce che, entro un numero finito di passaggi, l'algoritmo termini con successo la sua ricerca e fornisca una soluzione, se questa è raggiungibile.

2. E' *ottimale*. L'ottimalità di A* garantisce di individuare sempre il percorso più breve da un nodo di partenza a un nodo di destinazione.
   Questa dipende da due fattori:

- Il primo è che *h(n)* sia un'*euristica ammissibile*. Ciò significa che la funzione euristica fornisce sempre una stima ragionevole del costo per raggiungere la destinazione, senza mai sovrastimarlo.
- Il secondo è che *h(n)* sia un'*euristica consistente*. Per ogni nodo n e ogni successore n' di n generato da un'azione a, il costo stimato per raggiungere l'obiettivo partendo da n non è superiore al costo di passo per arrivare a n' sommato al costo stimato per andare da li all'obiettivo.

4 Applicazione dell'algoritmo

4.1 Tecnologie utilizzate

Il linguaggio di programmazione scelto per la soluzione del problema é Python. Per la realizzazione della GUI é stato impiegato il Framework Angular.
Sono state utilizzate le seguenti librerie:

- **heapq**: Questa libreria fornisce un'implementazione efficiente delle code di priorità (heap) in Python. È utile per gestire dati in coda in base a una priorità, dove gli elementi con la priorità più alta sono serviti per primi.

- **haversine:** Questa libreria calcola la distanza tra due punti sulla superficie della Terra. È utile per calcolare la distanza tra due coordinate geografiche, utilizzando latitudine e longitudine, in base al raggio medio della Terra.

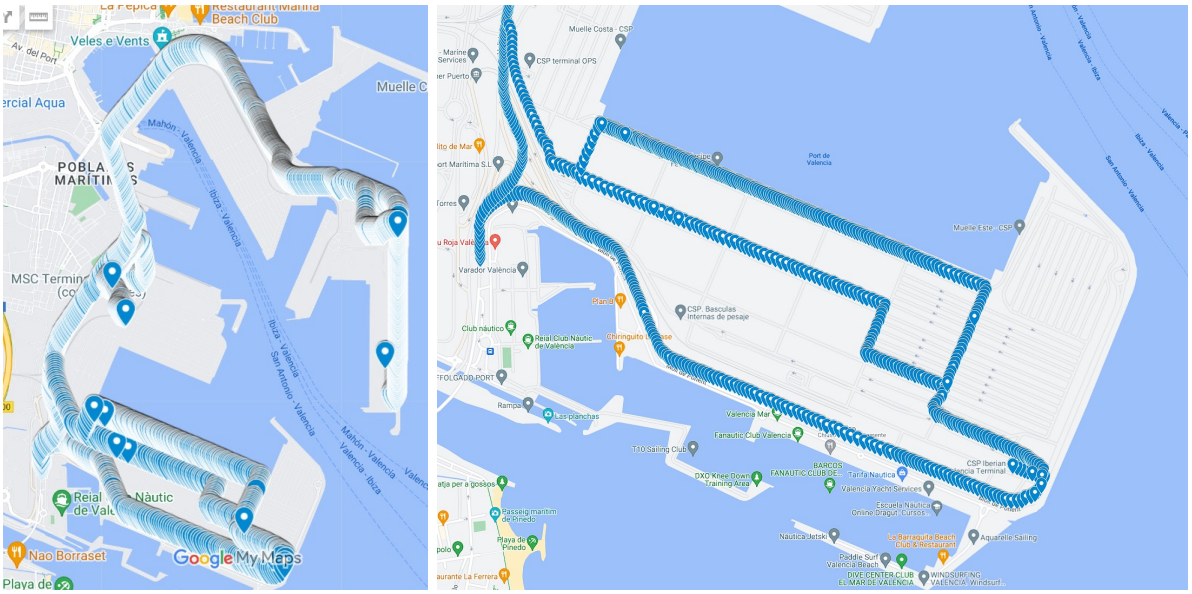
- **pandas:** Questa libreria offre strutture dati e strumenti per l'analisi dei dati in Python. In particolare, fornisce la struttura dati DataFrame, utilizzata per manipolare e analizzare dati tabellari.

- **combinations:** Questa libreria fornisce funzioni per generare tutte le possibili combinazioni di elementi da un iterabile, prendendo un numero specifico di elementi alla volta. E' stata utilizzata per rendere più efficiente la costruzione del grafo.

- **matplotlib:** Questa libreria è stata utilizzata per rappresentare dal punto di vista grafico i percorsi individuati dall'algoritmo rispetto al resto dei percorsi percorribili.

4.2 Creazione della mappa

Il nostro ambiente è rappresentato da una mappa del porto di Valencia. La mappa è stata costruita tramite l'utilizzo di MyMaps, individuando i possibili percorsi percorribili dagli autotrasportatori per il raggiungimento dei gate di destinazione all'interno del porto. I percorsi sono stati costruiti posizionanado i punti ad una distanza non superiore a 26m, ognuno dei quali rappresenta un nodo all'interno del grafo.



4.3 Implementazione

Sia per la costruzione del grafo che per il calcolo del percorso più breve è stata utilizzata **la distanza di Haversine**, un metodo matematico per calcolare la distanza tra due punti, espressa in kilometri, sulla superficie di una sfera, come la Terra, utilizzando le loro coordinate di latitudine e longitudine. Per fare ciò è stata utilizzata la libreria di Python denominata *haversine*.

```
17  # Funzione per calcolare la distanza haversine tra due coordinate
    4 usages  PaoloMurino +1
18  def distanza_haversine(lat1, lon1, lat2, lon2):
19      coord1 = (lat1, lon1)
20      coord2 = (lat2, lon2)
21      distance = haversine(coord1, coord2, unit=Unit.KILOMETERS)  # Calcolo della distanza
22      return distance
23
```

Il grafo è stato creato utilizzando un dizionario delle liste di adiacenza, costruito tramite la libreria standard di Python, *combinations*. Questo processo coinvolge la costruzione di un grafo da un insieme di coordinate di un dataset. La distanza tra ogni coppia di coordinate è calcolata utilizzando la funzione *distanza-haversine*. Se la distanza tra due coordinate è inferiore a 0.026 (corrispondente a una soglia di 26 metri), vengono aggiunti archi nel grafo tra le due coordinate.

```
def costruzione_grafo(dataset):
    # Crea il grafo rappresentato come un dizionario delle liste di adiacenza
    graph = {coord: [] for coord in dataset}

    # Trova le coppie di coordinate vicine e aggiunge gli archi al grafo
    for coord1, coord2 in combinations(dataset, 2):
        if distanza_haversine(coord1[0], coord1[1], coord2[0], coord2[1]) < 0.026:
            graph[coord1].append(coord2)
            graph[coord2].append(coord1)

    return graph
```

Per rappresentare i nodi all'interno del grafo è stata definita la **classe Node**. Ogni istanza della classe rappresenta un punto nel grafo con coordinate geografiche (latitudine e longitudine). Oltre a memorizzare le coordinate, ogni nodo mantiene informazioni sul costo del percorso calcolato finora per raggiungere il nodo corrente e sul nodo genitore, che consente di ricostruire il percorso ottimale una volta raggiunto il nodo di destinazione. Infine la classe implementa il metodo *-lt-* che permette di confrontare due istanze di Node in base al loro costo all'interno della coda a priorità.

```python
# Definizione della classe Node per rappresentare i nodi nel grafo
7 usages  Marypi02
class Node:
    Marypi02
    def __init__(self, lat, lon, cost=0, parent=None):
        self.lat = lat
        self.lon = lon
        self.cost = cost  # Costo del percorso finora per raggiungere il nodo
        self.parent = parent  # Nodo genitore nel percorso

    Marypi02
    def __lt__(self, other):
        return self.cost < other.cost  # Permette di confrontare nodi in base al costo
```

4.3.1 Descrizione dell'Algoritmo:

Inizializzazione: L'algoritmo parte con l'inizializzazione di un insieme aperto contenente il nodo di partenza e un insieme chiuso inizialmente vuoto che conterrà i nodi esplorati.
Esplorazione: Finché ci sono nodi nell'insieme aperto, l'algoritmo continua a esplorare il grafo. Ad ogni passo, seleziona il nodo con il costo totale più basso rappresentato da *(f(n))* dall'insieme aperto. Se il nodo estratto è già stato esplorato l'algoritmo passa al nodo successivo. Analogamente se il nodo corrente è il nodo di destinazione, costruisce e restituisce il percorso individuato fino a quel punto.

```python
# Algoritmo A* per trovare il percorso ottimale tra due punti
2 usages  Roksid2002 +3 *
def a_star(start, goal, graph):
    open_set = [start]  # Inizializza l'insieme aperto con il nodo di partenza
    closed_set = set()  # Inizializza l'insieme chiuso vuoto
    nodes_in_memory = 0  # Necessario per memorizzare il numero di nodi effettivamente

    while open_set:
        current_node = heapq.heappop(open_set)  # Estrae il nodo con il costo minimo dall'insieme aperto

        if (current_node.lat, current_node.lon) in closed_set:  # Se il nodo è già stato esplorato, salta
            continue

        if (current_node.lat, current_node.lon) == (goal.lat, goal.lon):  # Verifica se il nodo corrente è il nodo di destinazione
            path = []
            while current_node:
                path.append((current_node.lat, current_node.lon))
                current_node = current_node.parent
            return path[::-1]
```

Aggiornamento: Per il nodo corrente, esplora i vicini. Se il vicino non è stato esplorato, procede con il calcolo del costo totale *f(n)* formato da *g(n) + h(n)*, dove *g* rappresenta il costo del percorso effettuato finora per raggiungere il vicino e *h* rappresenta l'euristica ovvero il costo stimato per raggiungere l'obiettivo partendo dal vicino.
Aggiornamento Insiemi: Quando un vicino è stato esaminato, crea una nuova istanza della classe Node e lo aggiunge all'insieme aperto, mentre il nodo corrente viene aggiunto all'insieme dei nodi esplorati.
Terminazione: Se l'insieme aperto è vuoto e il nodo di destinazione non è stato raggiunto, significa che non esiste un percorso valido. In tal caso, restituisce "None".

```python
    closed_set.add((current_node.lat, current_node.lon))  # Aggiunge il nodo corrente all'insieme chiuso

    for neighbor in graph[(current_node.lat, current_node.lon)]:  # Scorre i vicini del nodo corrente
        if neighbor not in closed_set:  # Se il vicino non è stato esplorato
            g_score = current_node.cost + distanza_haversine(current_node.lat, current_node.lon, neighbor[0], neighbor[1])
            h_score = distanza_haversine(neighbor[0], neighbor[1], goal.lat, goal.lon)  # Calcola l'euristica (distanza diretta al nodo di destinazione)
            f_score = g_score + h_score  # Calcola il punteggio totale (costo finora + euristica)

            neighbor_node = Node(neighbor[0], neighbor[1], f_score, current_node)  # Crea un nuovo nodo per il vicino

            if neighbor_node not in open_set:  # Se il vicino non è nell'insieme aperto, lo aggiunge
                heapq.heappush(open_set, neighbor_node)

return None  # Se nessun percorso viene trovato
```

4.4 Valutazione dell'Euristica

Come detto, l'euristica utilizzata in questo contesto è la distanza di Haversine.
Per valutare l'efficacia dell'euristica utilizzata, al fine di garantire l'ottimalità dell'algoritmo, sono stati eseguiti due test:

1. **Test di ammissibilità dell'euristica:** Questo test verifica se l'euristica fornisce una stima inferiore o uguale al costo reale del percorso per raggiungere l'obiettivo. Una stima ammissibile è cruciale per garantire che l'algoritmo di ricerca produca sempre un percorso ottimale. Durante il test, l'euristica viene confrontata con il costo effettivo del percorso, e se l'euristica risulta essere inferiore o uguale al costo reale, allora l'euristica viene considerata ammissibile.

```python
def testAmmissibilita(percorso_ottimale):
    # Calcolo dell'euristica (distanza di Haversine) dallo start all'obiettivo
    start = percorso_ottimale[0]
    goal = percorso_ottimale[-1]
    heuristic = distanza_haversine(start[0], start[1], goal[0], goal[1])

    # Calcolo del costo totale del percorso
    total_cost = 0
    for i in range(len(percorso_ottimale) - 1):
        total_cost += distanza_haversine(percorso_ottimale[i][0], percorso_ottimale[i][1], percorso_ottimale[i+1][0],
                                         percorso_ottimale[i+1][1])

    # Verifica se l'euristica è ammissibile
    if heuristic <= total_cost:
        print("ammissibile!")
    else:
        print("non ammissibile!")

    # L'euristica è ammissibile se non sovrastima il costo per raggiungere l'obiettivo
    return heuristic <= total_cost
```

2. **Test di consistenza dell'euristica:** Questo test verifica se l'euristica mantiene una relazione di consistenza lungo il percorso ottimale. In particolare, si controlla se il valore dell'euristica da un nodo corrente all'obiettivo è sempre minore o uguale alla somma del costo effettivo per raggiungere un nodo successivo e dell'euristica dal nodo successivo all'obiettivo. La consistenza dell'euristica aiuta a garantire una convergenza efficiente dell'algoritmo di ricerca, accelerando il processo di individuazione del percorso ottimale.

```python
def testConsistenza(percorso_ottimale):
    # Individuazione del nodo obiettivo
    goal = percorso_ottimale[-1]

    # Iterazione su tutti i nodi del percorso ottimale
    for i in range(len(percorso_ottimale) - 1):
        # Individua il nodo corrente e il nodo successivo
        current_node = percorso_ottimale[i]
        next_node = percorso_ottimale[i + 1]

        # Calcola il costo di passo tra il nodo corrente e il nodo successivo
        g_score = distanza_haversine(current_node[0], current_node[1], next_node[0], next_node[1])

        # Calcolo dell'euristica dal nodo corrente all'obiettivo e dal nodo successivo all'obiettivo
        h_score_current = distanza_haversine(current_node[0], current_node[1], goal[0], goal[1])
        h_score_next = distanza_haversine(next_node[0], next_node[1], goal[0], goal[1])

        # Se il costo di passo più l'euristica dal nodo successivo è maggiore dell'euristica dal nodo corrente, l'euristica non è consistente
        if h_score_current > g_score + h_score_next:
            print("non consistente!")

    # Se nessuna incosistenza è stata trovata, l'euristica è consistente
    print("consistente!")
    return True
```

Entrambi i test sono terminati con esito positivo.

4.5 Complessità temporale e spaziale

Le complessità temporale e spaziale forniscono una valutazione del costo computazionale dell'algoritmo A* in termini di tempo e memoria.

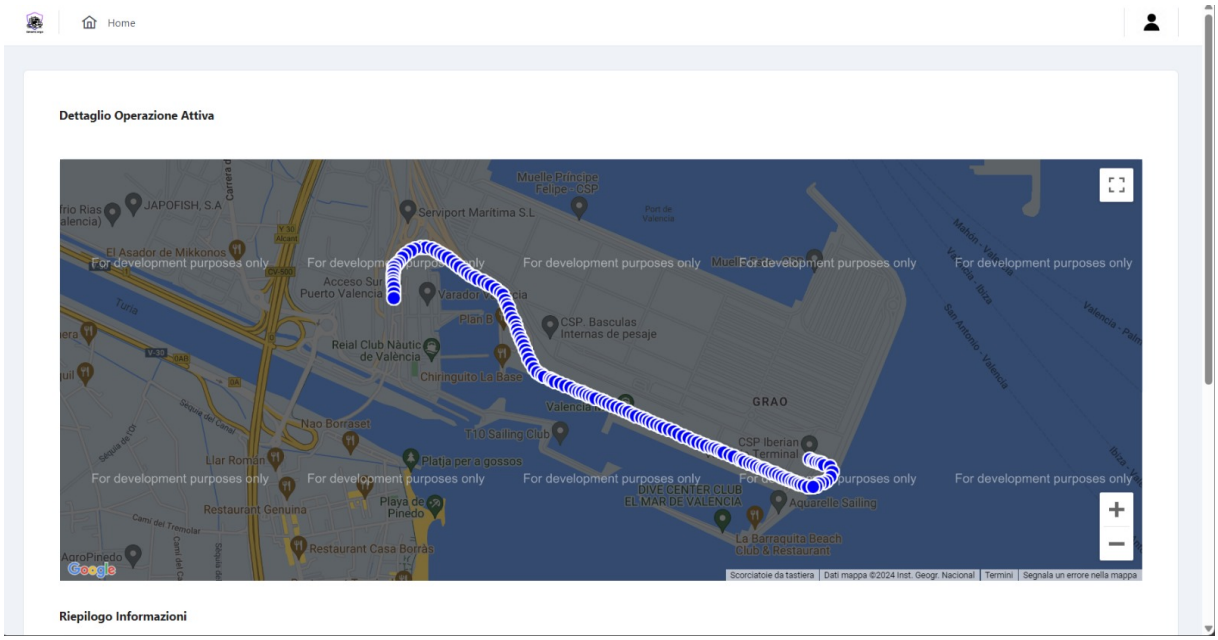
Sulla base della dimensione dell'input e della complessità del problema che stiamo risolvendo, risulta che la complessità temporale, calcolata in base al numero di nodi (V) e archi (E) nel grafo, è stimata a *O(88084.6877)*, mentre la complessità spaziale, influenzata dal numero totale di nodi nel grafo e dalla struttura dati utilizzata durante l'esecuzione dell'algoritmo, è stimata a *O(334)*, entrambe a rappresentare quanto tempo e spazio di memoria l'algoritmo richiede per eseguire le varie operazioni.

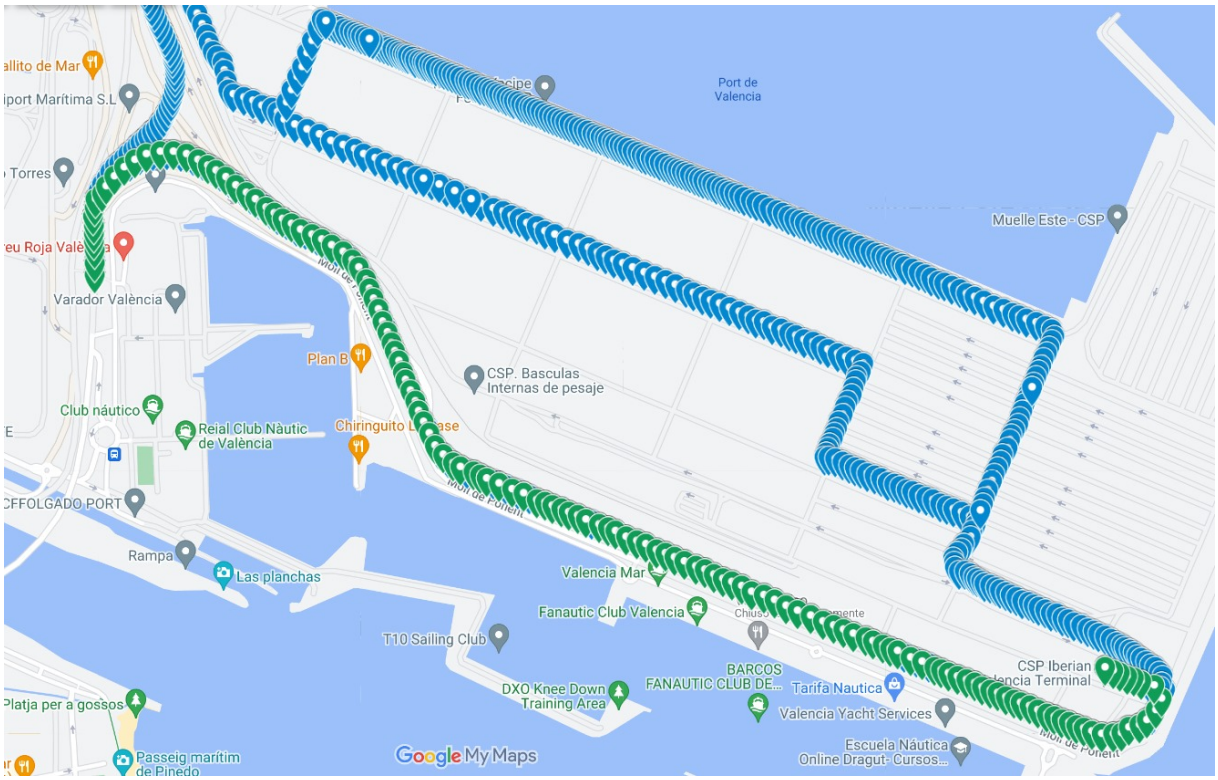
Un valore più basso della complessità temporale sarebbe preferibile, poichè indicherebbe una maggiore efficienza temporale, ciò non toglie che l'algoritmo sia in grado di calcolare il percorso ottimale.

5 Esecuzione dell'algoritmo

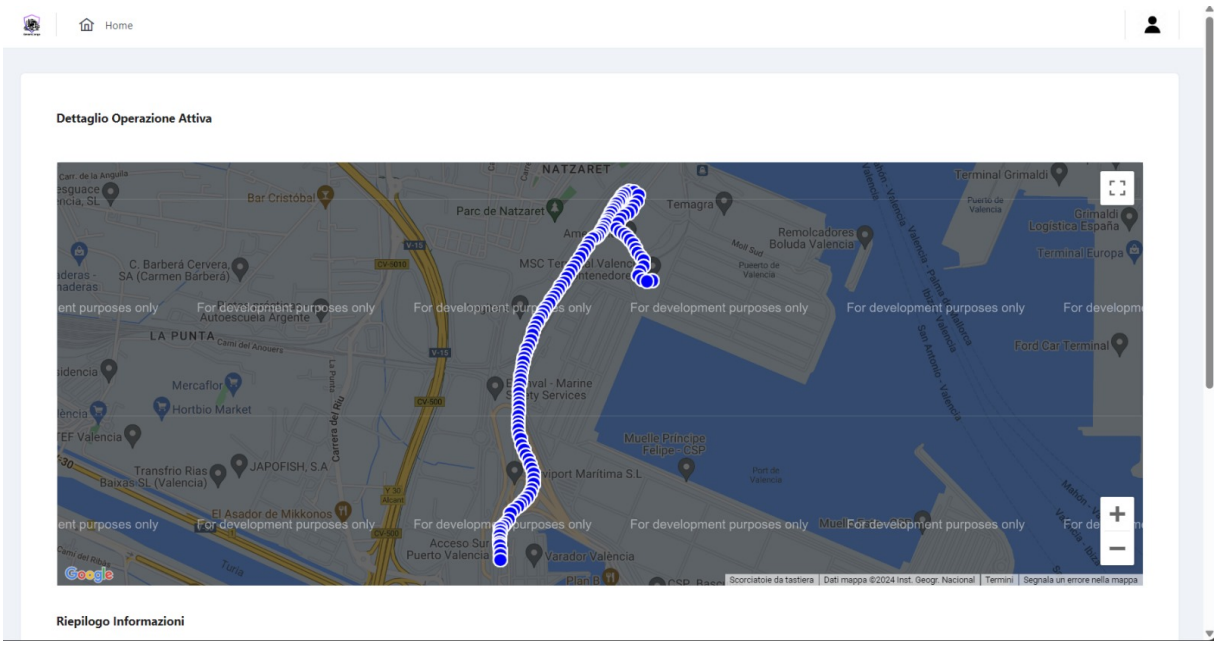
Di seguito sono riportate le immagini che illustrano i percorsi individuati dall'esecuzione dell'algoritmo quando una nuova operazione di un autotrasportatore all'interno del porto viene registrata, a confronto con i possibili percorsi.

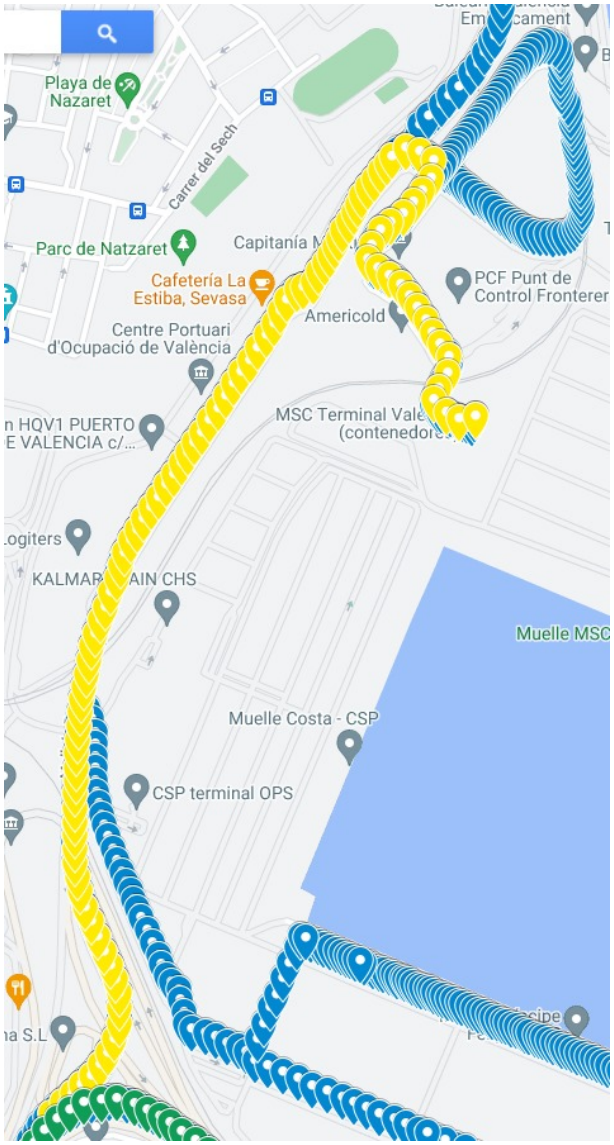
Gate1:



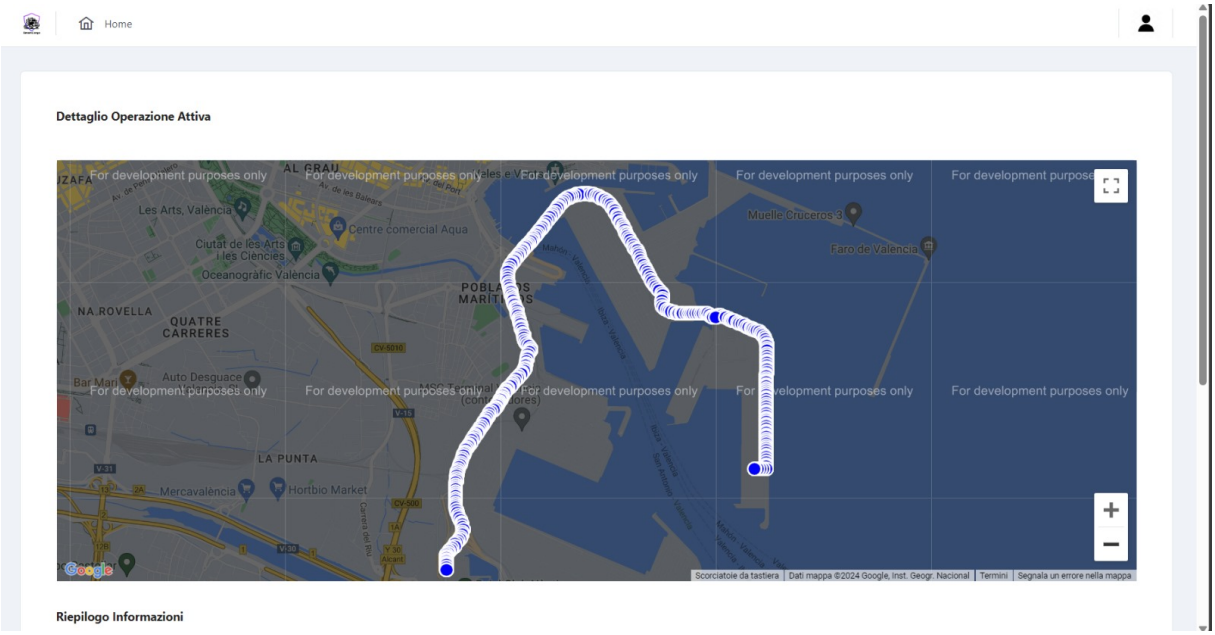


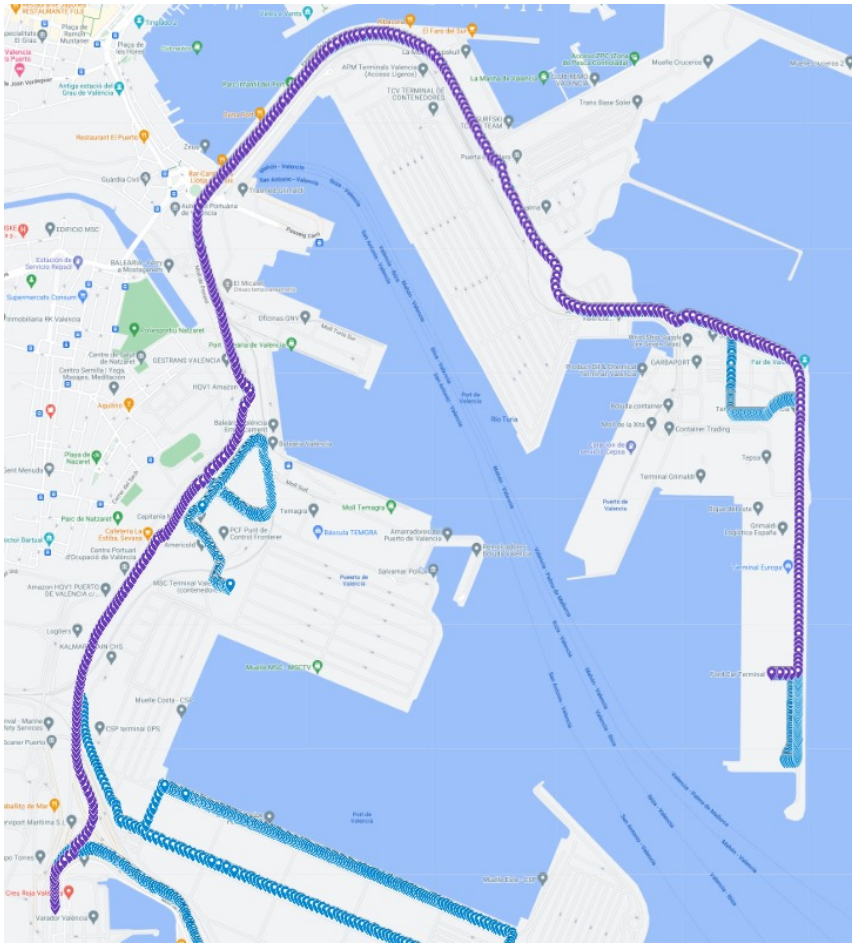
Gate2:


Home
Dettaglio Operazione Attiva
NATZARET
Parc de Natzaret
Temagra
Terminal Grimaldi
Remolcadores Boluda Valencia
Grimaldi Logística España
Terminal Europa
C. Barberá Cervera, SA (Carmen Barberá)
Bar Cristóbal
LA PUNTA
Mercaflor
Hortbio Market
Transfrio Rias Baixas SL (Valencia)
JAPOFISH, S.A
Ford Car Terminal
Muelle Príncipe Felipe - CSP
El Asador de Mikkonos
Acceso Sur Puerto Valencia
Varador València
For development purposes only
Riepilogo Informazioni


Playa de Nazaret
Carrer del Sech
Parc de Natzaret
Capitanía
Cafetería La Estiba, Sevasa
PCF Punt de Control Fronterer
Americold
Centre Portuari d'Ocupació de València
MSC Terminal
(contenedores)
HQV1 PUERTO
E VALENCIA c/...
Muelle MSC
Muelle Costa - CSP
CSP terminal OPS

Gate3:


Home
Dettaglio Operazione Attiva
For development purposes only
Riepilogo Informazioni



6 Considerazioni finali

Il lavoro dietro questo progetto è stato davvero entusiasmante in quanto ci ha permesso di applicare le nozioni apprese durante il corso in un contesto pratico approfondendo di molto una branca dell'intelligenza artificiale occupata dalla ricerca, non solo da un punto di vista teorico ma osservando i comportamenti dell'algoritmo e agendo di conseguenza. Inoltre integrandolo in un progetto più ampio, sviluppato durante l'esame di Ingegneria del Software, ci ha permesso di collocarlo in un ambito reale e che potrebbe fornire una vera utilità, per quanto sia solo un abbozzo.