UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

RELAZIONE DI SISTEMI INTELLIGENTI

Efficient Taxi Management

Progetto di Carlo Cairoli & Davide Orlando

a.a. 2024/2025

SOMMARIO:

1. Introduzione al problema:

Affrontare la gestione efficiente degli spostamenti dei taxi nei pressi di una stazione ferroviaria rappresenta una sfida cruciale nell'ambito dei sistemi intelligenti.

In scenari urbani ad alta densità, i flussi dei taxi risultano spesso disorganizzati: la concentrazione di più veicoli nello stesso punto, i tempi di attesa non coordinati e l'assenza di un sistema di smistamento ottimizzato possono generare congestione e inefficienze, con ripercussioni sia sui conducenti sia sui passeggeri.

Il nostro obiettivo è stato quello di progettare e simulare un sistema in grado di ottimizzare i movimenti dei taxi, riducendo i tempi di percorrenza e massimizzando l'efficienza degli spostamenti, in modo che i veicoli possano viaggiare, quando possibile, con il numero massimo di clienti consentito.

Per rappresentare il contesto, la città è stata modellata come una griglia 15x10: ogni cella corrisponde a un incrocio; il quadrato verde, posto in alto a sinistra (0, 9), identifica la stazione ferroviaria e rappresenta sia il punto di partenza sia il punto di arrivo di ciascun taxi; i cerchi azzurri indicano i punti in cui vengono localizzati i clienti al momento della prenotazione della corsa, mentre il quadrato rosso rappresenta il taxi, che si muove con spostamenti ortogonali. Sono stati introdotti, inoltre, ostacoli strutturali (celle bloccate) rappresentati da quadrati neri, che obbligano il pathfinding a deviare, rendendo necessaria l'integrazione dell'algoritmo di ricerca A* per il calcolo del percorso ottimale.

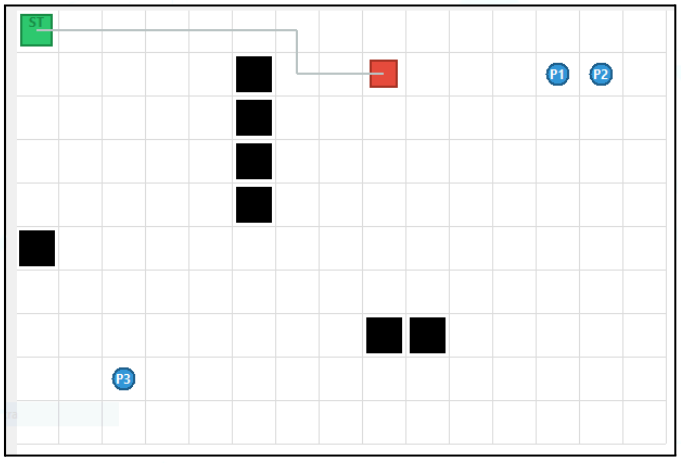

Figura 1.1 - Griglia di rappresentazione del contesto

L'obiettivo rimane la minimizzazione dei passi, calcolato step per step, garantendo la corretta gestione delle operazioni di pick-up e drop-off.

Un aspetto centrale del sistema è la distinzione tra taxi normali, adibiti al trasporto di un solo cliente, e taxi condivisibili (*shareable*), che per semplicità progettuale possono trasportare al massimo due clienti.

Questo approccio consente di rappresentare in modo chiaro la quantità di clienti in attesa, la loro posizione e la distanza rispetto alla stazione ferroviaria.

2. Modellazione del problema:

La formalizzazione del sistema è stata sviluppata in PDDL (Planning Domain Definition Language) e si articola in un file di dominio (*domain.pddl*) e tre istanze di problema (*problem1.pddl, problem2.pddl, problem3.pddl*).

Il dominio contiene la definizione astratta delle azioni disponibili. Sono previste tre tipologie di azione:

- il movimento del taxi tra celle adiacenti, limitato a spostamenti ortogonali;
- l'operazione di pick-up, che consente al taxi di prelevare un cliente;
- l'operazione di drop-off, che permette il rilascio dei passeggeri in stazione.

I file problem istanziano invece lo scenario concreto: definiscono la posizione della stazione, la collocazione dei taxi e dei clienti sulla griglia.

Lo spazio degli stati rappresenta l'insieme di tutte le configurazioni possibili di taxi e clienti sulla griglia. Lo stato iniziale corrisponde alla disposizione in cui tutti i taxi si trovano in stazione e i clienti sono distribuiti in posizioni differenti, mentre lo stato obiettivo è definito dalla condizione in cui ciascun passeggero è stato trasportato e rilasciato correttamente in stazione. Le azioni disponibili sono le mosse ortogonali, i pick-up e i drop-off, e la funzione di costo assegna un valore unitario ad ogni spostamento, considerando gratuite le azioni di prelievo e rilascio.

La modellazione è stata strutturata per consentire la descrizione di scenari con crescente livello di complessità: nel primo scenario è presente un solo cliente, nel secondo due clienti da servire in una corsa condivisa, mentre nel terzo compaiono tre clienti, con vincoli di capacità più articolati che richiedono strategie di gestione più avanzate. Questa progressione consente di valutare come il sistema si comporti al crescere della complessità del problema e rappresenta una base solida per eventuali estensioni future.

La sequenza di azioni pianificate è generata esternamente con FastDownward e viene letta dal programma tramite una routine di parsing. La proiezione sul dominio spaziale avviene attraverso i file di configurazione *locationsX.json*, che forniscono la mappatura tra simboli del piano e coordinate della griglia.

Ogni movimento astratto è quindi espanso dall'algoritmo A*, implementato nella funzione *find_path*, che determina il percorso minimo cella-per-cella rispettando i vincoli di movimento ortogonale e la presenza di ostacoli.

3. Algoritmo risolutivo A*:

Il nucleo del sistema è costituito dall'algoritmo A*, utilizzato come motore di pathfinding per garantire che i percorsi calcolati siano sempre ottimali, anche in presenza di ostacoli.

L'algoritmo mantiene una frontiera di nodi ordinata in base alla funzione di valutazione

$$f(n) = g(n) + h(n)$$

dove $g(n)$ rappresenta il costo effettivamente accumulato dal nodo iniziale fino al nodo n mentre $h(n)$ fornisce una stima euristica della distanza residua fino al traguardo.

Nel contesto della griglia 15×10 adottata, il taxi può muoversi esclusivamente nelle quattro direzioni ortogonali (alto, basso, sinistra, destra). Per questo motivo si è scelto di utilizzare come euristica la distanza di Manhattan, che misura il numero minimo di passi necessari per connettere due celle lungo movimenti ortogonali. Tale scelta garantisce proprietà fondamentali: l'ammissibilità, che impedisce di sovrastimare il costo reale, e la consistenza, che assicura che la stima non diminuisca mai più del costo effettivo di un passo.

```python
# definizione euristica Manhattan
def manhattan(a: Coord, b: Coord) -> int:
    """Distanza di Manhattan tra due celle (movimento ortogonale)."""
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

Figura 3.1 - Schermata funzione distanza di Manhattan

Gli ostacoli vengono esclusi direttamente durante l'espansione dei nodi vicini, cosicché l'algoritmo ignori automaticamente le celle non attraversabili e calcoli deviazioni minime solo quando strettamente necessario. Una volta individuato l'obiettivo, il cammino viene ricostruito risalendo la catena dei predecessori, restituendo la sequenza ordinata delle celle che compongono il percorso.

La funzione *find_path(start, goal)* rappresenta l'implementazione concreta dell'algoritmo A* all'interno del progetto. Essa prende in ingresso una cella di partenza e una di arrivo, inizializza la frontiera con il nodo iniziale e mantiene due strutture di supporto: *g_score*, che memorizza il costo minimo noto per raggiungere ciascuna cella, e came_from, che registra il predecessore di ogni nodo esplorato. Durante l'esecuzione, i nodi vengono estratti dalla frontiera secondo il valore minimo di $f(n)$.

Ogni volta che viene individuato un percorso più conveniente verso un vicino, i valori vengono aggiornati e il nodo reinserito nella coda di priorità. Una volta raggiunta la cella obiettivo, la funzione ricostruisce il cammino risalendo la catena dei predecessori, restituendo la sequenza delle celle da percorrere.

Questo output viene poi integrato nel TripPlan e animato dalla GUI, permettendo di visualizzare il percorso passo per passo.

```python
# definizione algoritmo A*
def find_path(start: Coord, goal: Coord) -> List[Coord]:

    if start == goal:
        return [start]

    openq: List[Tuple[int, Coord]] = []
    heapq.heappush(openq, (0, start))

    came_from: Dict[Coord, Optional[Coord]] = {start: None}
    g_score: Dict[Coord, int] = {start: 0}
```

Figura 3.2 - (Schermata 1) funzione find_path

```python
    while openq:
        _, current = heapq.heappop(openq)

        if current == goal:
            #ricostruisci il percorso retrocedendo con came_from
            path: List[Coord] = [goal]
            while came_from[path[-1]] is not None:
                path.append(came_from[path[-1]])
            path.reverse()
            return path[:-1]  # esclude goal

        x, y = current
        for nx, ny in ((x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)):
            if 0 <= nx < GRID_W and 0 <= ny < GRID_H:
                neigh = (nx, ny)
                if neigh in OBSTACLES:
                    continue

                tentative_g = g_score[current] + 1
                if neigh not in g_score or tentative_g < g_score[neigh]:
                    g_score[neigh] = tentative_g
                    f = tentative_g + manhattan(neigh, goal)
                    came_from[neigh] = current
                    heapq.heappush(openq, (f, neigh))

    #nessun percorso trovato
    return []
```

Figura 3.3 - (Schermata 2) funzione find_path

4. Spiegazione dell'algoritmo risolutivo

Il piano generato da Fast Downward è costituito da una sequenza di azioni astratte in linguaggio PDDL. Tali azioni definiscono l'ordine logico delle operazioni da compiere, ma non specificano né i passaggi intermedi da percorrere sulla griglia né i vincoli reali legati alla capacità del taxi e al calcolo dei costi.

Il programma Python ha quindi il compito di tradurre queste istruzioni in una simulazione concreta, integrando l'algoritmo A* per determinare i percorsi minimi passo per passo.

Il flusso parte con la funzione *read_plan*, che legge il file *sas_plan* ed estrae esclusivamente le azioni effettive.

```python
def read_plan(filepath: str) -> List[str]:
    """Legge un piano generato da Fast Downward e restituisce solo le azioni tra parentesi."""
    actions: List[str] = []
    with open(filepath, "r", encoding="utf-8") as f:
        for raw in f:
            s = raw.strip()
            if not s or s.startswith(";"):
                continue
            if "(" in s and ")" in s:
                s = s[:s.rfind(")") + 1]
            if s.startswith("(") and s.endswith(")"):
                actions.append(s.lower())
    if not actions:
        raise ValueError(f"Nessuna azione valida nel piano: {filepath}")
    return actions
```

Figura 4.1 - Schermata funzione read_plan

In parallelo, la funzione *load_locations* importa i file *locations.json* e associa i simboli logici del piano alle coordinate reali della griglia.

```python
def load_locations(filepath: str) -> Dict[str, Coord]:
    """Carica file location da JSON"""
    with open(filepath, "r", encoding="utf-8") as f:
        data = json.load(f)

    locs: Dict[str, Coord] = {}
    for k, v in data.items():
        if k.lower() == "st":
            continue
        locs[k.lower()] = (int(v[0]), int(v[1]))
    locs["st"] = STATION
    return locs
```

Figura 4.2 - Schermata funzione load_location

Successivamente la funzione *parse_sas_plan_to_trip* utilizza queste informazioni per espandere ciascuna azione di movimento in una sequenza di celle consecutive calcolata con A*, e per associare gli eventi di prelievo e rilascio a posizioni precise lungo il percorso.

```python
def parse_sas_plan_to_trip(actions: List[str], locs: Dict[str, Coord]) -> Tuple[TripPlan, Dict[str, Coord]]:
    """Traduce azioni SAS (move/pickup/dropoff) in TripPlan"""
    full: List[Coord] = []
    picks: Dict[int, List[str]] = {}
    drops: Dict[int, List[str]] = {}
    labels: Dict[str, Coord] = {}
    current: Optional[Coord] = None

    def coord_of(label: str) -> Coord:
        key = label.lower()
        if key not in locs:
            raise KeyError(f"Location '{label}' non definita nel file locations.")
        return locs[key]

    for raw in actions:
        tokens = raw.strip("()").split()
        if not tokens:
            continue
        op = tokens[0]
```

Figura 4.3 - (Schermata 1) funzione parse_sas_plan_to_trip

```python
        if op == "move":
            _, _taxi, src, dst = tokens
            src_c = coord_of(src)
            dst_c = coord_of(dst)
            if current is None:
                current = src_c
                full.append(current)
            seg = find_path(current, dst_c)
            full += seg
            full.append(dst_c)
            current = dst_c

        elif op == "pickup":
            _, _taxi, p, l = tokens
            label = p.upper()
            labels[label] = coord_of(l)
            if not full:
                current = coord_of(l)
                full.append(current)
            picks.setdefault(len(full) - 1, []).append(label)

        elif op == "dropoff":
            _, _taxi, p, _l = tokens
            label = p.upper()
            if not full:
                current = STATION
                full.append(current)
            drops.setdefault(len(full) - 1, []).append(label)

    trip = TripPlan(full, picks, drops)
    return trip, labels
```

Figura 4.4 - (Schermata 2) funzione parse_sas_plan_to_trip

Il risultato è un oggetto *TripPlan*, che contiene la traiettoria completa del taxi e le mappe di pick-up e drop-off indicizzate sugli indici del path.

Nel caso dello scenario con tre clienti, viene attivata una logica aggiuntiva di ricalcolo automatico. La funzione *extract_pickups* individua dal piano l'associazione tra ciascun passeggero e la relativa location, mentre *build_share_two_closest_trip* determina la coppia di clienti con distanza Manhattan minima e pianifica una corsa condivisa, riservando eventuali corse aggiuntive ai rimanenti. In questo modo l'ordine delle visite non è fisso, ma si adatta dinamicamente alle posizioni specificate nei file di configurazione, rendendo la simulazione più realistica.

```python
def extract_pickups(actions: List[str]) -> Dict[str, str]:
    """Estrae dal piano la prima associazione passeggero->location (es. 'P1'->'l1')."""
    mapping: Dict[str, str] = {}
    for raw in actions:
        t = raw.strip("()").split()
        if len(t) == 4 and t[0] == "pickup":
            p = t[2].upper()
            l = t[3].lower()
            if p not in mapping:
                mapping[p] = l
    return mapping
```

Figura 4.5 - Schermata funzione extract_pickups

```python
def build_share_two_closest_trip(pass_to_loclabel: Dict[str, str], locs: Dict[str, Coord]) -> Tuple[TripPlan, Dict[str, Coord]]:
    """Costruisce un TripPlan che preleva i due passeggeri più vicini tra loro ed una seconda corsa per il rimanente"""
    # Prendi solo i passeggeri che hanno una location definita
    items = [(p, pass_to_loclabel[p]) for p in sorted(pass_to_loclabel.keys()) if pass_to_loclabel[p] in locs]
    if len(items) < 2:
        # troppo pochi per lo share → costruisci un trip banale fedele al piano
        raise ValueError("Per il ricalcolo automatico servono almeno due pickup nel piano.")

    # Costruisci lista [(label, coord)]
    pts: List[Tuple[str, Coord]] = [(p, locs[l]) for p, l in items]
    labels_map: Dict[str, Coord] = {p: c for p, c in pts}

    # Trova la coppia più vicina (Manhattan)
    best_pair = None
    best_d = None
    for i in range(len(pts)):
        for j in range(i + 1, len(pts)):
            d = manhattan(pts[i][1], pts[j][1])
            if best_pair is None or d < best_d:
                best_pair = (pts[i], pts[j])  # ((p,coord),(p,coord))
                best_d = d
    assert best_pair is not None
    (pA, cA), (pB, cB) = best_pair

    # Il resto (se sono 3) è il "solo"
    remaining = [p for p, _ in pts if p not in (pA, pB)]
```

Figura 4.6 - Schermata 1 funzione build_share_two_closest_trip

```
    # Ordina i due del carpool: prima quello più vicino alla stazione
    first, second = (pA, cA), (pB, cB)
    if manhattan(cB, STATION) < manhattan(cA, STATION):
        first, second = (pB, cB), (pA, cA)

    full: List[Coord] = []
    picks: Dict[int, List[str]] = {}
    drops: Dict[int, List[str]] = {}

    # Corsa 1: ST -> first -> second -> ST
    seg = find_path(STATION, first[1])
    full += seg
    full.append(first[1]); picks[len(full) - 1] = [first[0]]
    seg = find_path(first[1], second[1])
    full += seg
    full.append(second[1]); picks.setdefault(len(full) - 1, []).append(second[0])
    seg = find_path(second[1], STATION)
    full += seg
    full.append(STATION); drops[len(full) - 1] = [first[0], second[0]]
```

Figura 4.7 - Schermata 2 funzione build_share_two_closest_trip

```
    # Corsa 2: per ogni rimanente (di solito 1)
    for p in remaining:
        c = labels_map[p]
        seg = find_path(STATION, c)
        full += seg
        full.append(c); picks[len(full) - 1] = [p]
        seg = find_path(c, STATION)
        full += seg
        full.append(STATION); drops.setdefault(len(full) - 1, []).append(p)

    trip = TripPlan(full, picks, drops)
    return trip, labels_map
```

Figura 4.8 - Schermata 3 funzione build_share_two_closest_trip

Il *TripPlan* costituisce anche il riferimento per il calcolo economico. Ad ogni passo generato da A*, il costo viene incrementato di un'unità; se è presente un solo cliente, quest'ultimo sostiene l'intero importo, mentre in presenza di due passeggeri la spesa viene ripartita equamente in base al tratto di percorso effettivamente condiviso a bordo del taxi.

La gestione dei costi è affidata alla GUI, che durante l'animazione aggiorna dinamicamente il totale accumulato e ne suddivide la quota tra i passeggeri presenti, garantendo una rappresentazione chiara e coerente con l'evoluzione del percorso simulato.

La fase finale è dedicata alla visualizzazione. L'interfaccia grafica, sviluppata in *Tkinter,* mostra la griglia con la stazione, i clienti e gli ostacoli, disegna il percorso calcolato e anima lo spostamento del taxi cella per cella. In parallelo, aggiorna dinamicamente la

dashboard che riporta il costo totale e la quota di ciascun passeggero, fornendo una rappresentazione chiara e immediata del meccanismo di ride sharing.

L'utente dispone inoltre di controlli interattivi per avviare, mettere in pausa, resettare o regolare la velocità della simulazione, rendendo l'analisi del comportamento del sistema intuitiva e completa.

Grazie a questa struttura modulare, il sistema mantiene coerenza tra piano astratto e simulazione concreta, garantendo un comportamento fedele sia dal punto di vista logico sia da quello operativo.

5 Panoramica sul funzionamento del programma:

All'avvio del programma, viene mostrata un'interfaccia grafica che rappresenta la griglia della città, comprensiva del quadrato verde della stazione, dei cerchi raffiguranti i clienti e dal riquadro rosso che indica la presenza del taxi. Il programma Python *taxi_visualizer.py* rappresenta la fase di simulazione e visualizzazione dei piani generati in PDDL.

Per avviare la simulazione è sufficiente:

1. generare un piano tramite Fast Downward,
2. eseguire il file *taxi_visualizer.py*.

Il programma carica automaticamente il file plan e il corrispondente *locations.json*, disegna la griglia 15x10 e posiziona taxi, clienti e stazione.

Durante l'esecuzione, la simulazione mostra in tempo reale il movimento dei taxi passo per passo, le azioni di pick-up e drop-off ed il tracciato seguito su griglia.

All'avvio della GUI sarà visualizzato di default il Problem1, tuttavia, è possibile selezionare il problem desiderato attraverso gli appositi tasti, ed in automatico sulla griglia verranno visualizzati gli elementi ad esso associati.

Per garantire una visualizzazione pulita dell'animazione e dei log su terminale, è consigliato selezionare un nuovo problema solo quando l'animazione non è in esecuzione, così da resettare correttamente tutte le variabili.

Per avviare la simulazione è sufficiente premere il tasto play. Sono stati inoltre introdotti controlli base per la gestione dell'animazione, come il comando di pausa ed il comando di reset, così da rendere la simulazione più interattiva e intuitiva.

L'interfaccia grafica permette di osservare l'evoluzione delle azioni di gestione dei clienti da parte del taxi in tempo reale, evidenziando anche il tracciato seguito, oltre alle operazioni eseguite di prelievo o scarico dei clienti. Inoltre, aggiorna dinamicamente anche una **dashboard dei costi**, che visualizza il costo accumulato e, in caso di condivisione, la ripartizione equa tra i clienti a bordo.

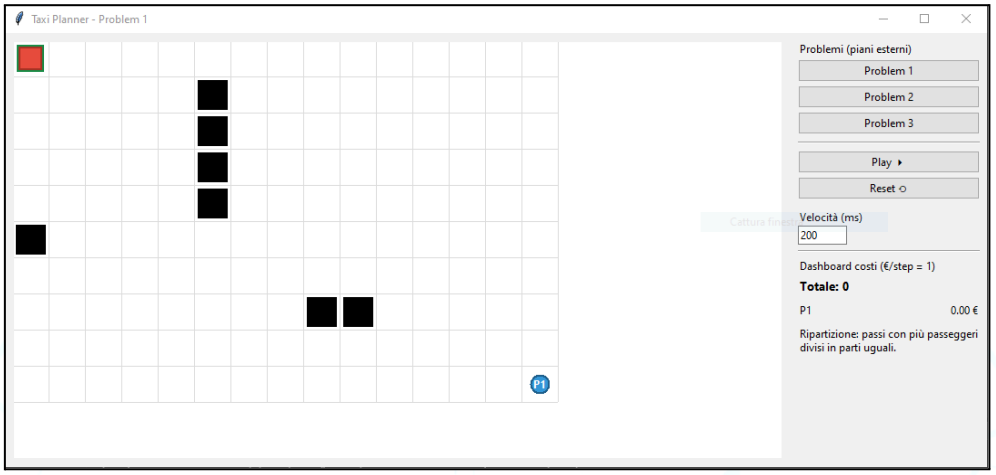

Figura 5.1 - Schermata interfaccia grafica esempio Problem1

Di seguito i tasti utilizzabili:

- **Problem 1**: seleziona il primo caso in cui è presente un unico cliente
- **Problem 2**: seleziona il secondo caso in cui sono presenti due clienti
- **Problem 3**: seleziona il terzo caso in cui sono presenti tre clienti
- **Play**: avvia la simulazione
- **Pause**: arresta la simulazione
- **Reset**: effettua il reset delle variabili per il corretto avvio del programma
- **Velocità (ms)**: è possibile gestire la velocità di movimento del taxi

Il diagramma di flusso illustra le principali fasi operative del sistema. A partire dalla generazione del piano in PDDL tramite FastDownward, il programma esegue il parsing delle azioni, associa le posizioni concrete tramite i file *locationsX.json*, calcola i percorsi minimi con A* e costruisce un oggetto *TripPlan*.

Quest'ultimo è utilizzato per guidare l'animazione grafica e aggiornare in tempo reale la dashboard dei costi. La sequenza evidenzia la modularità del sistema e la chiara separazione tra pianificazione simbolica, pathfinding e visualizzazione interattiva.

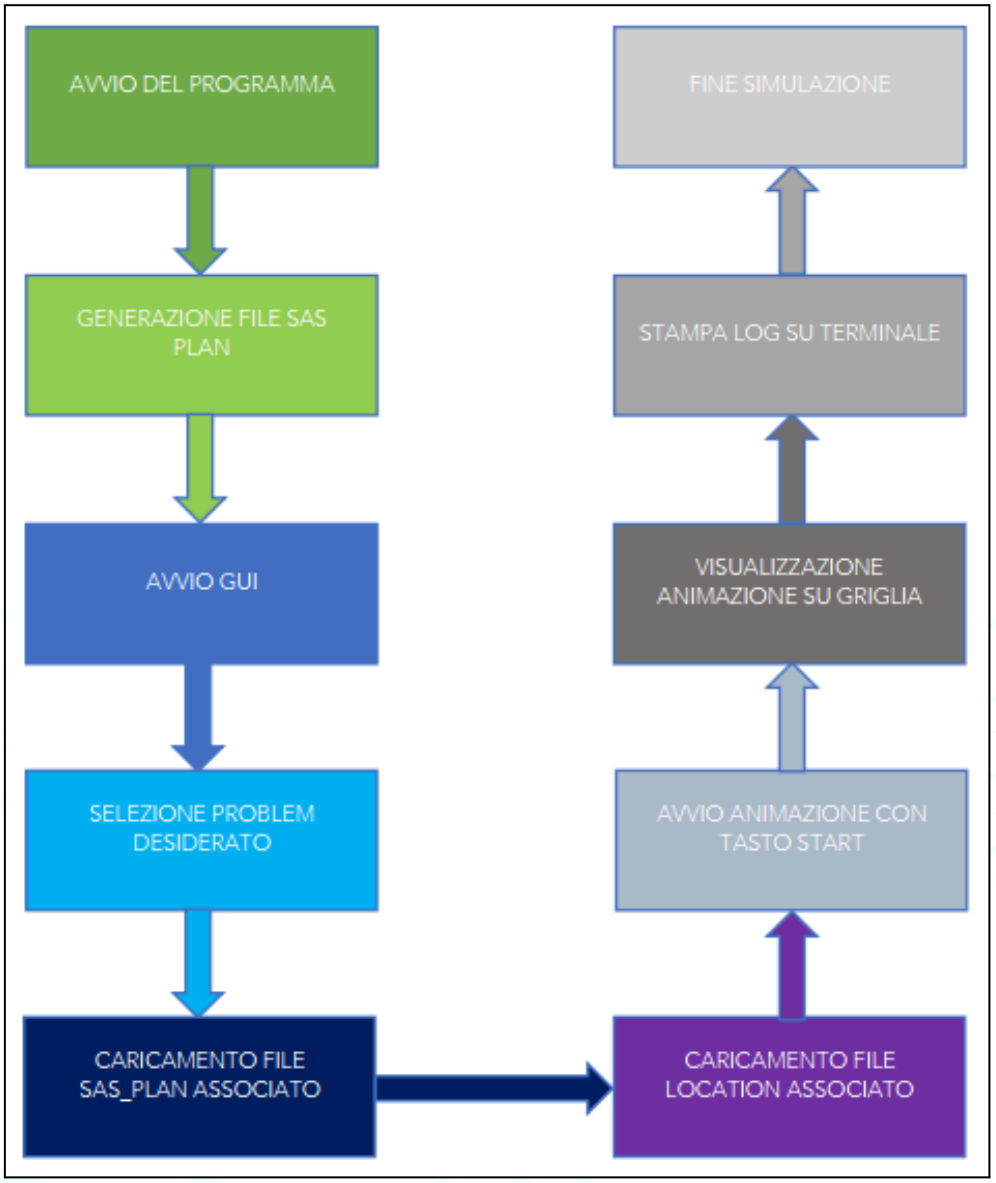

Figura 5.2 - Schermata diagramma di flusso

6. Esempi di esecuzione:

Per validare il sistema sono stati realizzati tre scenari di test che differiscono per numero di clienti e modalità di gestione delle corse.

Nel primo scenario viene simulata la richiesta di un unico cliente. Il taxi parte dalla stazione, raggiunge il passeggero collocato nell'angolo in basso a destra della griglia e lo riconduce in stazione. Sebbene elementare, questo caso permette di verificare il corretto funzionamento della catena completa: lettura del piano PDDL, traduzione in *TripPlan*, calcolo del percorso con A* e aggiornamento della dashboard dei costi.

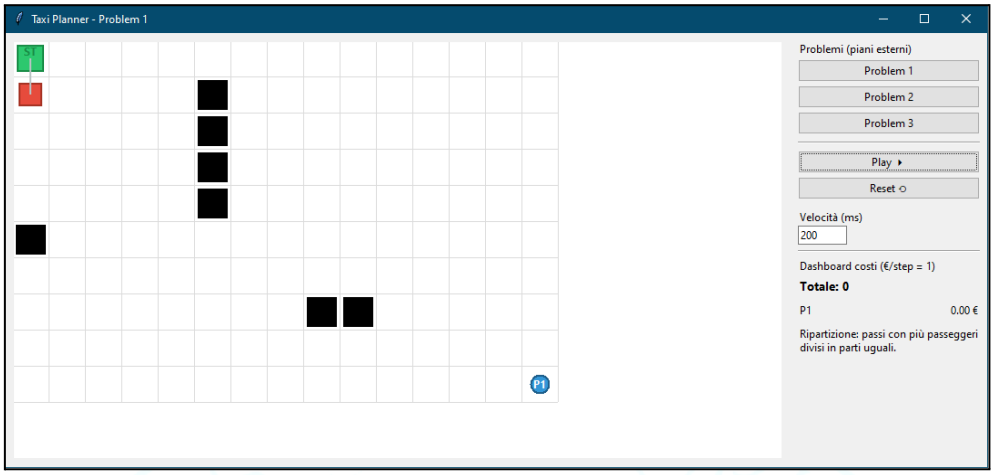

Figura 6.1 - Schermata avvio Problem1

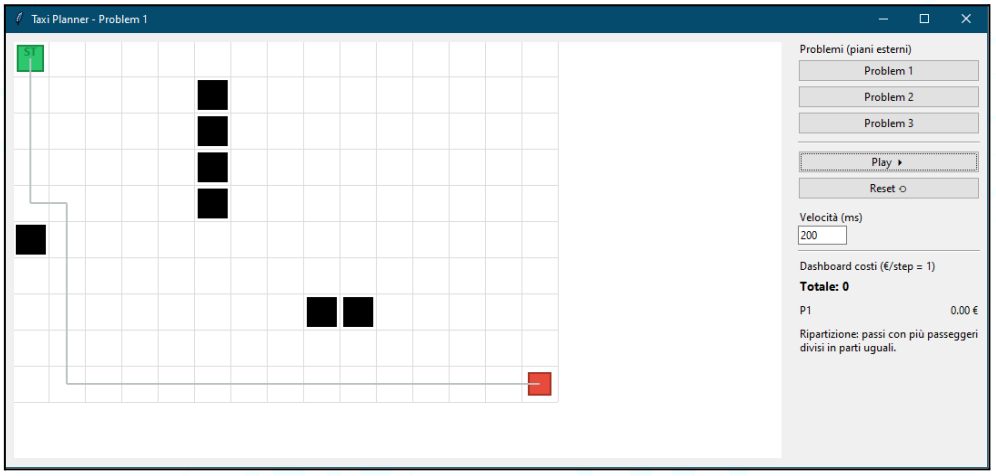

Figura 6.2 - Schermata prelievo clienti P1

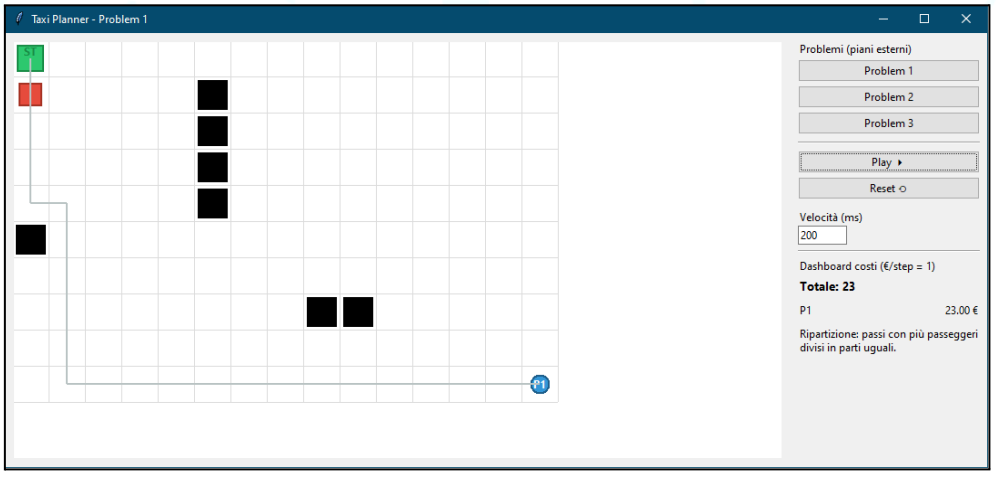

Figura 6.3 - Schermata scarico cliente P1

Il **secondo scenario** prevede la gestione condivisa di due clienti. In questo caso il piano indica due operazioni di pick-up che avvengono in sequenza e un drop-off simultaneo in stazione.

L'animazione conferma il rispetto del vincolo di capacità massima, mentre la dashboard dei costi evidenzia la ripartizione equa della tariffa tra i due passeggeri durante i tratti percorsi in condivisione.

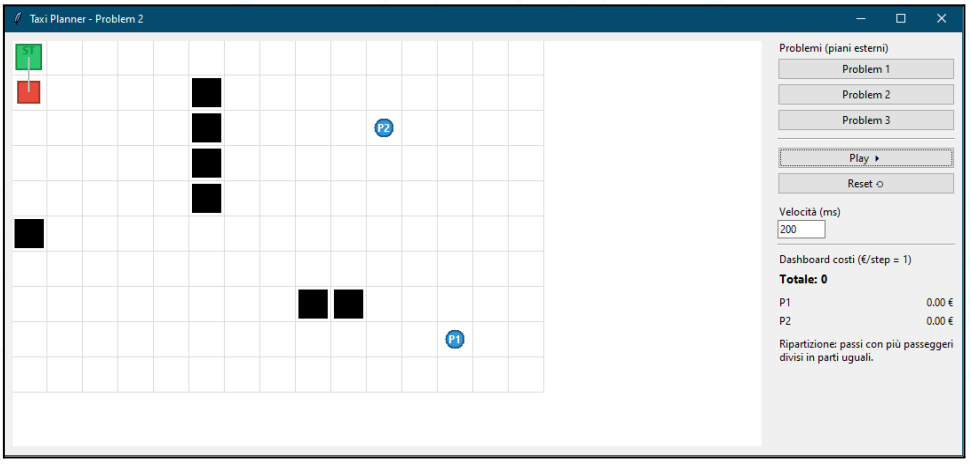

Figura 6.4 - Schermata avvio Problem2

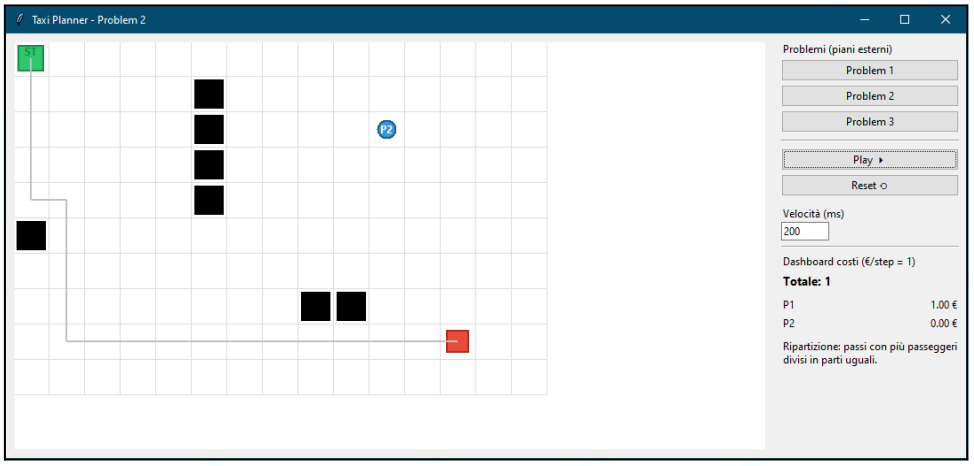

Figura 6.5 - Schermata prelievo cliente P1

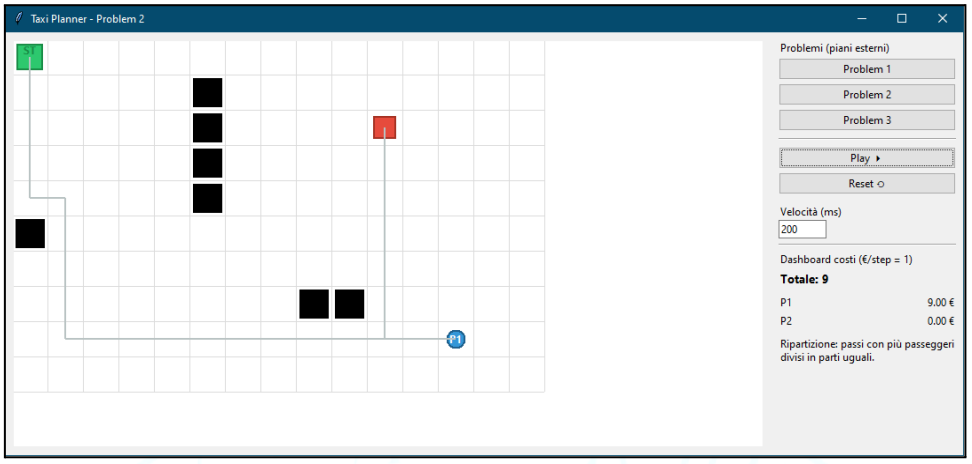

Figura 6.6 - Schermata prelievo cliente P2

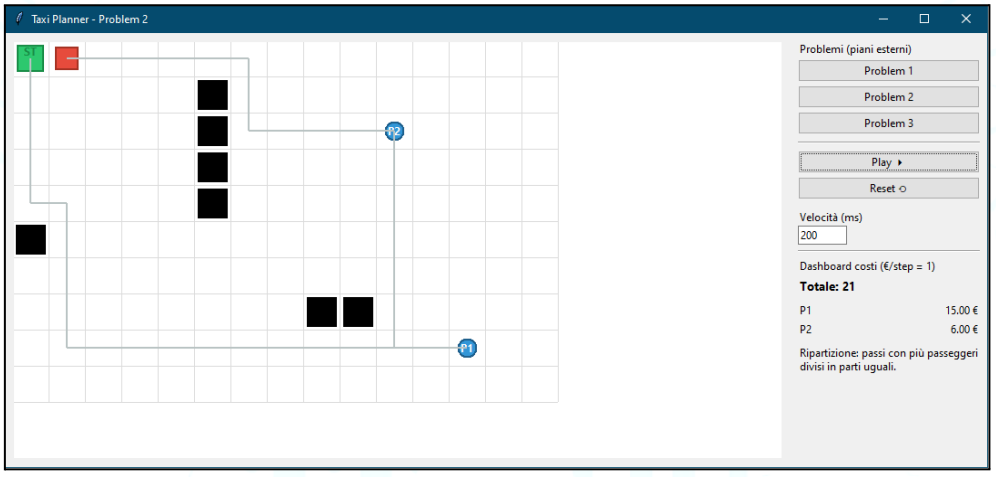

Figura 6.7 - Schermata scarico clienti P1 e P2

Il terzo scenario è dedicato a tre clienti. Il sistema utilizza la funzione di ricalcolo automatico per individuare la coppia di passeggeri con distanza Manhattan minima e li serve nella stessa corsa condivisa, lasciando il passeggero rimanente a un viaggio dedicato. Questo test dimostra la capacità del programma di adattare dinamicamente l'ordine dei pick-up in base alla configurazione corrente e di combinare taxi "normali" e taxi "shareable" nello stesso contesto simulativo.

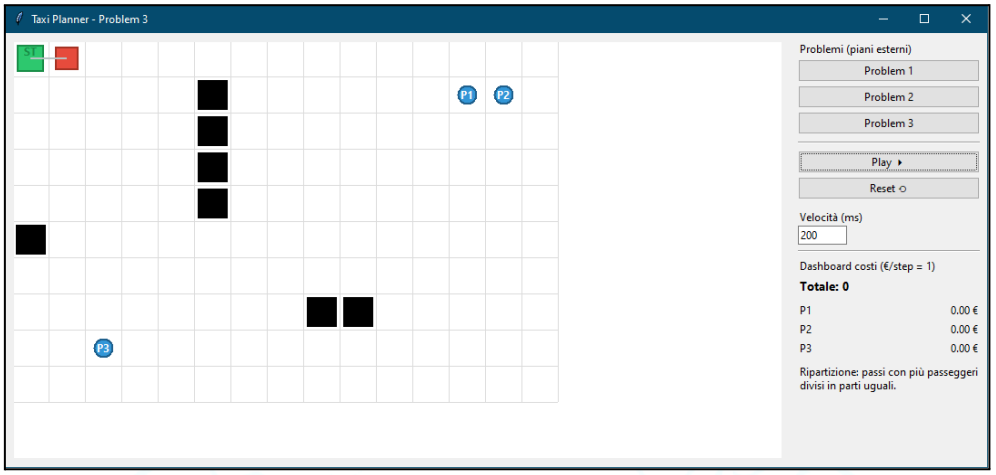

Figura 6.8 - Schermata avvio Problem3

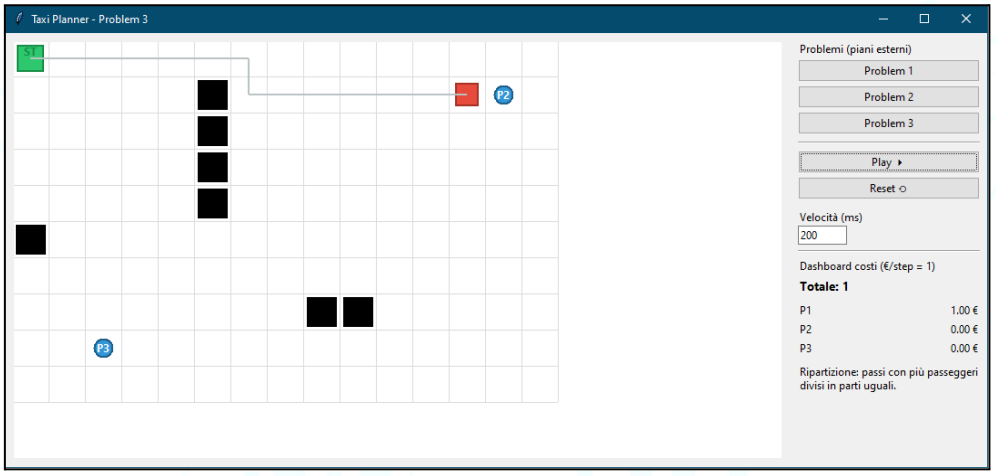

Figura 6.9 - Schermata prelievo cliente P1

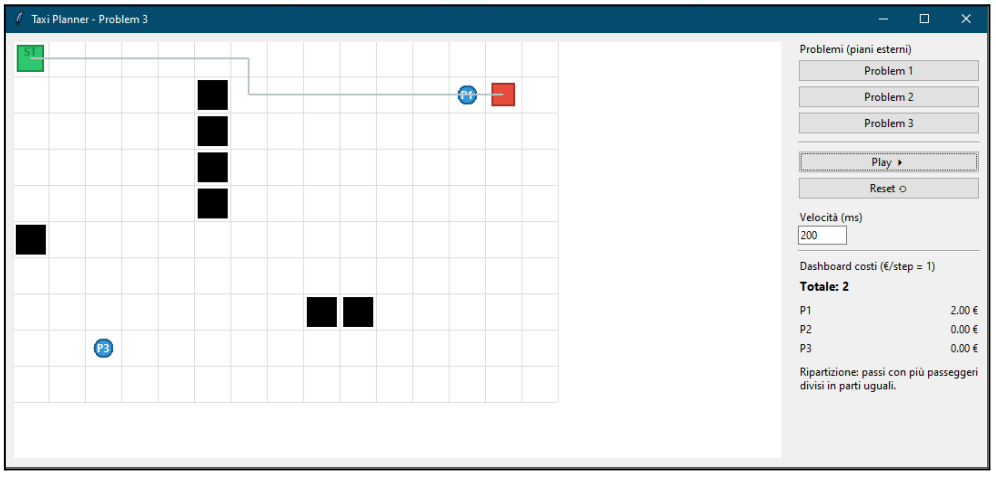

Figura 6.10 - Schermata prelievo cliente P2

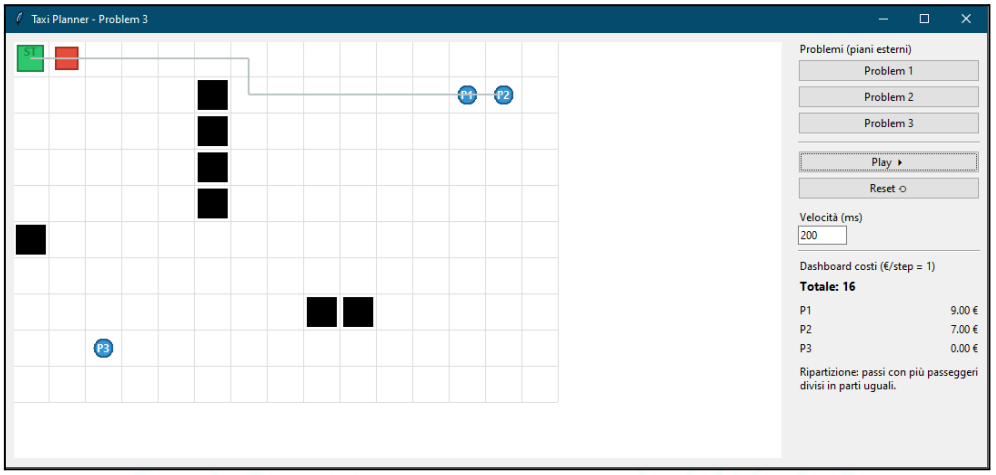

Figura 6.11 - Schermata scarico clienti P1 e P2

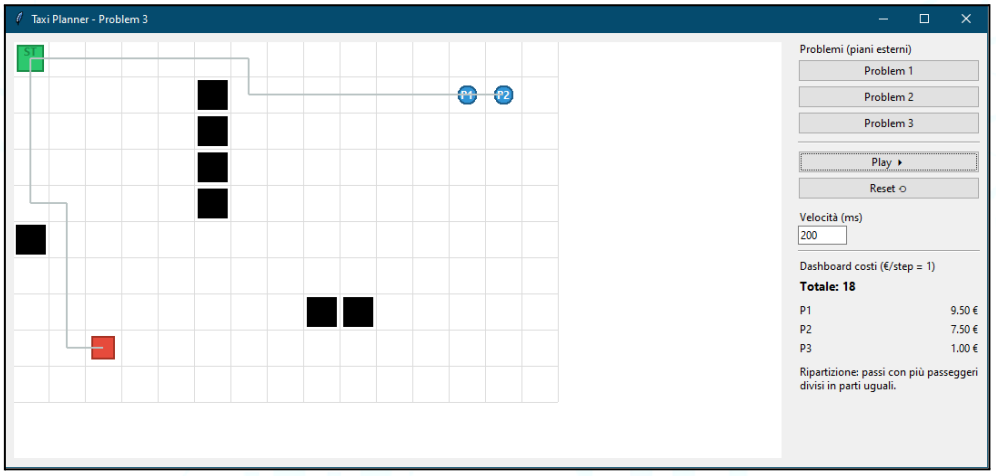

Figura 6.12 - Schermata prelievo cliente P3

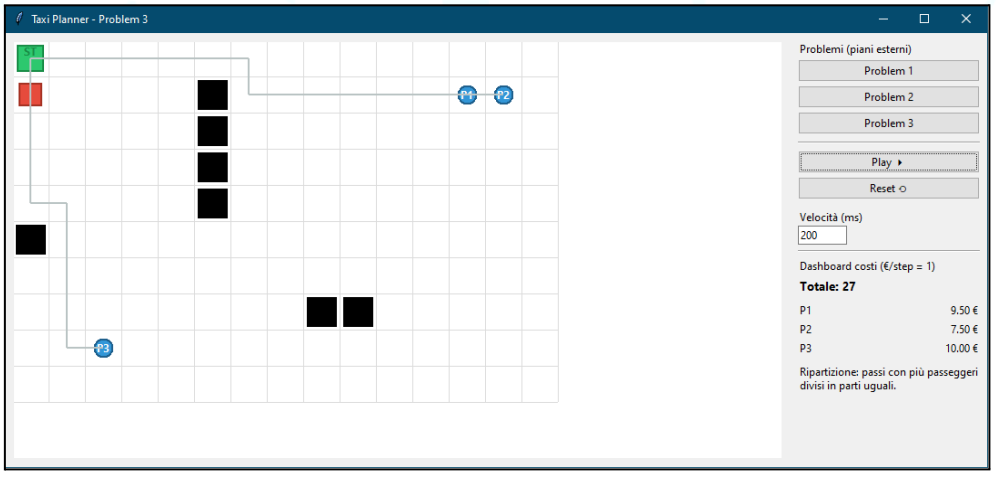

Figura 6.13 - Schermata scarico cliente P3

In tutti gli scenari, la simulazione grafica ha mostrato in tempo reale lo spostamento passo per passo del taxi, l'attivazione delle operazioni di pick-up e drop-off e il tracciato effettivamente seguito sulla griglia. Parallelamente, i log prodotti a terminale hanno fornito una validazione puntuale del comportamento del sistema: per ogni passo vengono riportate la posizione corrente del taxi, i passeggeri a bordo e l'aggiornamento del costo totale.

Questo doppio livello di visualizzazione, grafico e testuale, ha permesso di verificare la coerenza tra il piano astratto in PDDL e la sua traduzione operativa sulla griglia, oltre a confermare la corretta contabilizzazione dei costi minimi.

7. Sviluppi e implementazioni future

Il progetto, pur offrendo una base solida per la simulazione di un sistema di gestione dei taxi con algoritmi intelligenti, presenta alcune opportunità di evoluzione.

Un primo ambito di miglioramento riguarda sicuramente la precisione della simulazione grafica: questa potrebbe essere resa più realistica attraverso l'implementazione di una mappa reale di una città, così da ridurre le ambiguità tra la logica di funzionamento e il rendering visivo.

Un secondo ambito di miglioramento potrebbe essere l'introduzione di un sistema di priorità, ossia l'integrazione di logiche che diano precedenza a determinate categorie di clienti, come ad esempio emergenze, persone con disabilità o utenti con prenotazione anticipata.

Un ulteriore ambito di miglioramento potrebbe essere l'integrazione di un algoritmo di previsione della domanda, basato su tecniche di machine learning, in grado di stimare in anticipo dove e quando si registrerà una maggiore richiesta di taxi, così da posizionare preventivamente i veicoli in zone strategiche.