PROGETTO VOL*AI*

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Corso di Laurea in *Ingegneria Informatica*

Corso di *Intelligenza Artificiale*

a.a. *2024/2025*

Prof. *Giancarlo Sperlì*

Antonio Marco Vanacore – N46007063

Chiara Improta – N46006980

Obiettivo del progetto

L’obiettivo di questo progetto è la realizzazione di un'applicazione intelligente in grado di calcolare in modo ottimale un percorso aereo tra due città, utilizzando un dataset di collegamenti realmente ispirati alla rete globale dei voli. Il problema è affrontato tramite l’algoritmo di ricerca informata A*, una strategia ampiamente utilizzata in Intelligenza Artificiale per la pianificazione di percorsi ottimali.

L’applicazione consente a un utente di selezionare una città di partenza e una di arrivo, e ottenere:

- Il *percorso più breve* in termini di distanza, con penalizzazione per eventuali scali
- Il *costo* totale del viaggio, ottenuto sommando i costi di ciascun volo
- Una stima della *durata* complessiva in ore
- Una *visualizzazione geografica* del percorso sulla mappa interattiva

L'interfaccia grafica è stata sviluppata per essere intuitiva e funzionale, permettendo anche a un utente inesperto in ambito informatico di esplorare il sistema e ottenere un risultato significativo in pochi secondi.

Dal punto di vista tecnico, il cuore del sistema è basato su un *grafo orientato*, in cui i nodi rappresentano gli aeroporti e gli archi rappresentano i voli diretti, arricchiti da attributi come costo, durata e distanza.

Questo progetto dimostra l'applicabilità dell'Intelligenza Artificiale alla risoluzione di problemi concreti attraverso tecniche classiche di *problem solving* e rappresentazione dello spazio degli stati.

Inoltre, la struttura modulare del codice rende l'app facilmente estendibile per l'introduzione di *ulteriori funzionalità*, come la selezione preferenziale di compagnie aeree, la gestione di orari o la simulazione di scenari alternativi.

Teoria A*

L'algoritmo A* è una tecnica di ricerca informata, utilizzata in Intelligenza Artificiale per trovare il percorso ottimale tra due stati all'interno di uno spazio delle possibilità, rappresentato solitamente come un grafo.

A differenza di algoritmi di ricerca non informata come Breadth-First Search (BFS) o Depth-First Search (DFS), A* utilizza un *criterio combinato* che permette di valutare ogni mossa in base sia al costo effettivo del percorso già compiuto, sia a una stima del costo necessario per raggiungere l'obiettivo.

Questa combinazione lo rende uno degli algoritmi più efficienti per la ricerca del cammino più breve, in particolare quando si dispone di un'euristica ammissibile.

Formula fondamentale:

Ogni nodo n viene valutato secondo la funzione:

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

dove:

• ***g(n)*** è il *costo effettivo* per raggiungere il nodo n, a partire dallo stato iniziale

• ***h(n)*** è la *funzione euristica*, ovvero una stima del costo per andare da n allo stato obiettivo

• ***f(n)*** rappresenta la *stima complessiva del costo totale* del percorso passando per n

L'algoritmo esplora i nodi secondo l'ordine di f(n), scegliendo sempre quello con il valore più basso.

Proprietà di A*:

• ***Completezza*:** se esiste un percorso, A* lo trova

• ***Ottimalità*:** trova il percorso con costo minimo, a patto che h(n) sia ammissibile (cioè non sovrastimi mai il costo reale)

• ***Efficienza*:** evita esplorazioni inutili grazie alla guida offerta dall'euristica

Applicazione nel progetto

Nel nostro caso, i nodi del grafo sono le città, mentre gli archi rappresentano i voli diretti tra due città. Ogni arco è arricchito da attributi come:

• ***costo*** (in euro)

• ***durata*** (in ore)

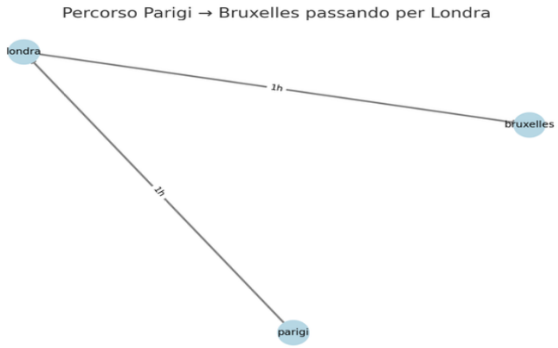
• ***compagnia***

• ***distanza*** (calcolata con *formula Haversine* tra coordinate geografiche)

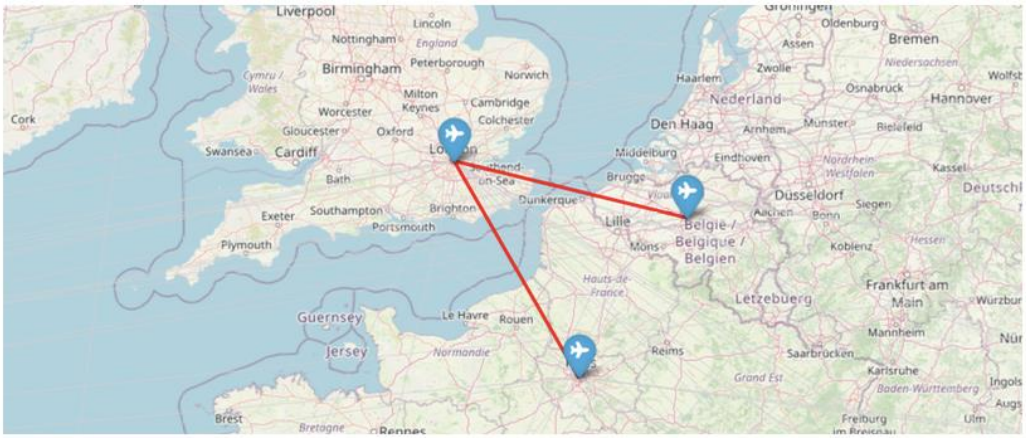
L'algoritmo A* utilizza come euristica ***h(n)*** la distanza geografica diretta (great-circle distance) tra la città corrente e quella di destinazione. Tale scelta garantisce l'ammissibilità, poiché nessun percorso reale sarà più breve della distanza in linea d'aria tra due punti.

Inoltre, per favorire i voli diretti e penalizzare i percorsi con più scali, la funzione di peso ***g(n)*** include una maggiorazione sulla distanza per ogni tratto intermedio.

Idea iniziale:



Esempio effettivo Parigi-Bruxelles:



Euristica del progetto ed implementazione

Uno degli elementi fondamentali dell'algoritmo A* è la funzione euristica, che serve a stimare il costo restante tra un nodo corrente e il nodo obiettivo. In un contesto di *pathfinding geografico*, l'euristica più naturale è la distanza in linea d'area tra due città.

Nel nostro progetto, questa distanza è calcolata tramite la formula di Haversine, un metodo ben noto in geodesia per determinare la distanza tra due punti sulla superficie terrestre, tenendo conto della sfericità della Terra.

La scelta di questa euristica è particolarmente adatta perché:

• È **ammissibile**, ovvero non sovrastima mai la distanza reale di un percorso (nessun volo reale può essere più breve della distanza geografica diretta).

• È **consistente**, cioè la stima rispetta sempre le disuguaglianze triangolari tra i nodi.

```python
# === Distanza geografica (haversine) ===
def distanza_geografica(c1, c2):
    if c1 not in coord or c2 not in coord:
        return 0
    lat1, lon1 = coord[c1]
    lat2, lon2 = coord[c2]
    R = 6371
    phi1, phi2 = math.radians(lat1), math.radians(lat2)
    dphi = math.radians(lat2 - lat1)
    dlambda = math.radians(lon2 - lon1)
    a = math.sin(dphi / 2)**2 + math.cos(phi1) * math.cos(phi2) * math.sin(dlambda / 2)**2
    return R * 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
```

Questa funzione restituisce, basandosi sulla funzione di **Haversine**, la distanza geografica tra due città una volta fornite le coordinate.

```python
def peso_con_scali(u, v, data):
    if u == partenza_corrente and v == arrivo_corrente:
        return data["distanza"]
    return data["distanza"] * 1.10  # penalità per scali
```

Inoltre, per rendere l'algoritmo più umano, viene applicata una penalizzazione per gli scali, in modo da incoraggiare i voli diretti.

Questa combinazione tra **euristica precisa e penalità dinamica** consente al sistema di proporre percorsi non solo brevi in termini geografici, ma anche più realistici dal punto di vista dell'esperienza utente (meno scali, durata più ridotta).

La funzione di **Haversine** viene calcolata grazie alle coordinate, le quali sono state scelte appositamente, basandosi su quelle di *folium* (che verrà approfondito più avanti nella tesina).

```python
# === Coordinate geografiche ===
coord = {
    "doha": (25.3, 51.5), "londra": (51.5, -0.1), "berlino": (52.5, 13.4),
    "tokyo": (35.7, 139.7), "seoul": (37.6, 126.9), "atene": (37.9, 23.7),
    "roma": (41.9, 12.5), "milano": (45.5, 9.2), "budapest": (47.5, 19.0),
    "varsavia": (52.2, 21.0), "newyork": (40.7, -74.0), "helsinki": (60.2, 24.9),
    "copenaghen": (55.7, 12.6), "toronto": (43.7, -79.4), "zurigo": (47.4, 8.5),
    "vienna": (48.2, 16.4), "praga": (50.1, 14.4), "sydney": (-33.9, 151.2),
    "boston": (42.3, -71.1), "cairo": (30.0, 31.2), "oslo": (59.9, 10.7),
    "parigi": (48.8, 2.3), "lisbona": (38.7, -9.1), "dublino": (53.3, -6.3),
    "amsterdam": (52.4, 4.9), "madrid": (40.4, -3.7), "bruxelles": (50.8, 4.4),
    "dubai": (25.2, 55.3), "losangeles": (34.0, -118.2), "stoccolma": (59.3, 18.1),
    "chicago": (41.8781, -87.6298),"mumbai": (19.0760, 72.8777),"beijing": (39.9042, 116.4074),
    "moscow": (55.7558, 37.6176),"singapore": (1.3521, 103.8198),
}
```

Una volta collegato al dataset e ricevuto in input le due città delle quali si vuole analizzare il collegamento aereo, il sistema attua nell'effettivo l'algoritmo A*

```python
# === Algoritmo A* ===
def percorso_astar(G, partenza, arrivo):

    global partenza_corrente, arrivo_corrente
    partenza_corrente = partenza
    arrivo_corrente = arrivo

    try:
        percorso = nx.astar_path(G, partenza, arrivo, heuristic=euristica, weight=peso_con_scali)
        costo = sum(G[u][v]['costo'] for u, v in zip(percorso[:-1], percorso[1:]))
        durata = int(sum(G[u][v]['durata'] for u, v in zip(percorso[:-1], percorso[1:])))
        return percorso, costo, durata
    except nx.NetworkXNoPath:
        return None
```

Le variabili "partenza_corrente" e "arrivo_corrente", vengono utilizzate per la funzione peso_con_scali" (vedi pag 4)

Tutte queste funzioni si basano sul dataset, il quale viene "parsato" da una funzione "carica_voli_da_pl":

```python
# === Caricamento grafo da file .pl ===

def carica_voli_da_pl(file_name):
    G = nx.DiGraph()
    pattern = re.compile(r"flight\((\w+),\s*(\w+),\s*(\d+),\s*(\d+),\s*(\w+)\)\.")

    # Percorso assoluto del file rispetto alla posizione di questo script
    base_dir = os.path.dirname(os.path.abspath(__file__))
    file_path = os.path.join(base_dir, file_name)

    with open(file_path, "r") as f:
        for line in f:
            match = pattern.match(line.strip().lower())
            if match:
                part, arr, costo, durata, comp = match.groups()
                distanza = distanza_geografica(part, arr)
                G.add_edge(part, arr, costo=int(costo), durata=int(durata), compagnia=comp, distanza=float(distanza))
    return G
```

Dataset

Per rappresentare la rete dei voli internazionali, il progetto utilizza un file esterno denominato “voli.pl”, che contiene una collezione di dati espressi in formato logico ispirato a Prolog, pur non essendo direttamente eseguito da un motore Prolog.

```
flight(madrid, amsterdam, 996, 2, ita).
```

Ogni riga segue lo schema: flight(partenza, arrivo, costo, durata, compagnia).

Il file “voli.pl” viene letto e interpretato all'interno del codice Python, trattato come file di testo, non come codice logico eseguibile. Una funzione di parsing personalizzata analizza ogni riga tramite espressioni regolari, estrae i valori e li trasforma in archi di un grafo diretto (networkx.DiGraph).

Questa scelta offre i vantaggi della struttura leggibile di Prolog, mantenendo la flessibilità del linguaggio Python per l’implementazione dell’algoritmo A* e dell’interfaccia utente.

Il dataset comprende diverse decine di città sparse nei cinque continenti, con un focus particolare sull’Europa e collegamenti verso Nord America, Asia, Medio

Oriente e Oceania. Le tratte sono ispirate a rotte realmente esistenti, seppur semplificate in termini di durata e costo per ragioni pratiche.

Questo garantisce una **rete sufficientemente estesa** per:

- Simulare scenari realistici con **più soluzioni** possibili
- Verificare la capacità dell’algoritmo di **gestire scali**
- Testare l’efficacia dell’euristica anche su **distanze molto ampie**

La struttura è volutamente leggera affinché sia semplice il *parsing*, il *debugging*

e *la modifica*. Ogni riga è autosufficiente e può essere aggiunta/rimossa senza

influenzare le altre.

Le città elencate nel file sono collegate a **coordinate geografiche** presenti nel modulo Python

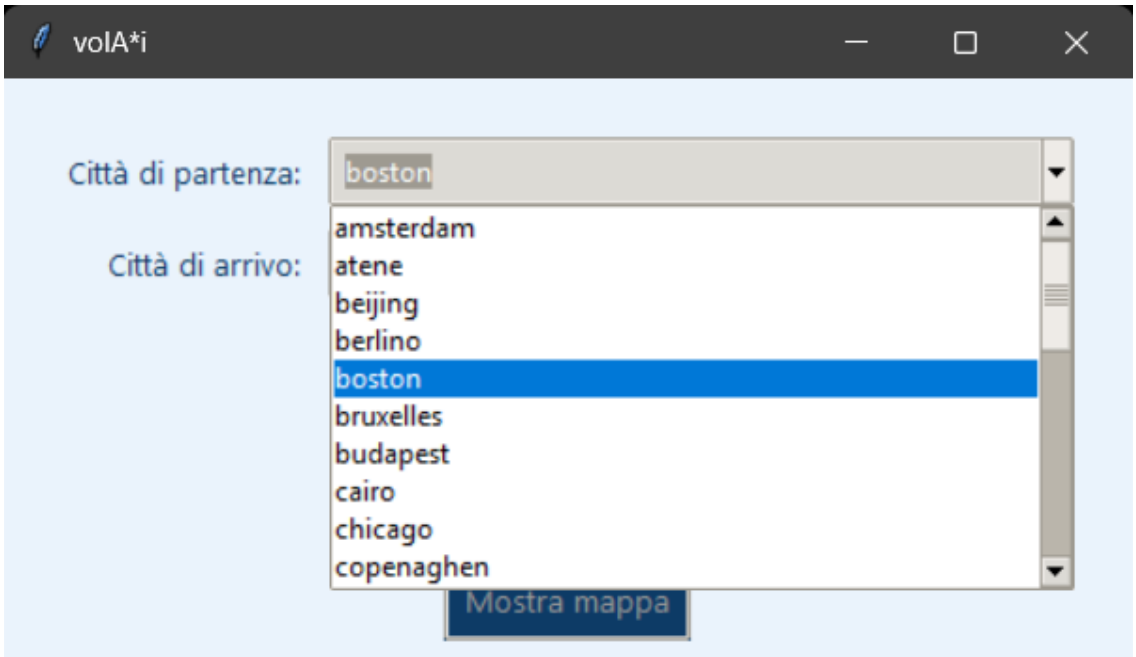
Il dataset conta 500 voli, circa 100 rotte (a/r) tra le città più importanti, è stato generato automaticamente restando però coerente con le durate dei voli reali.

Gui

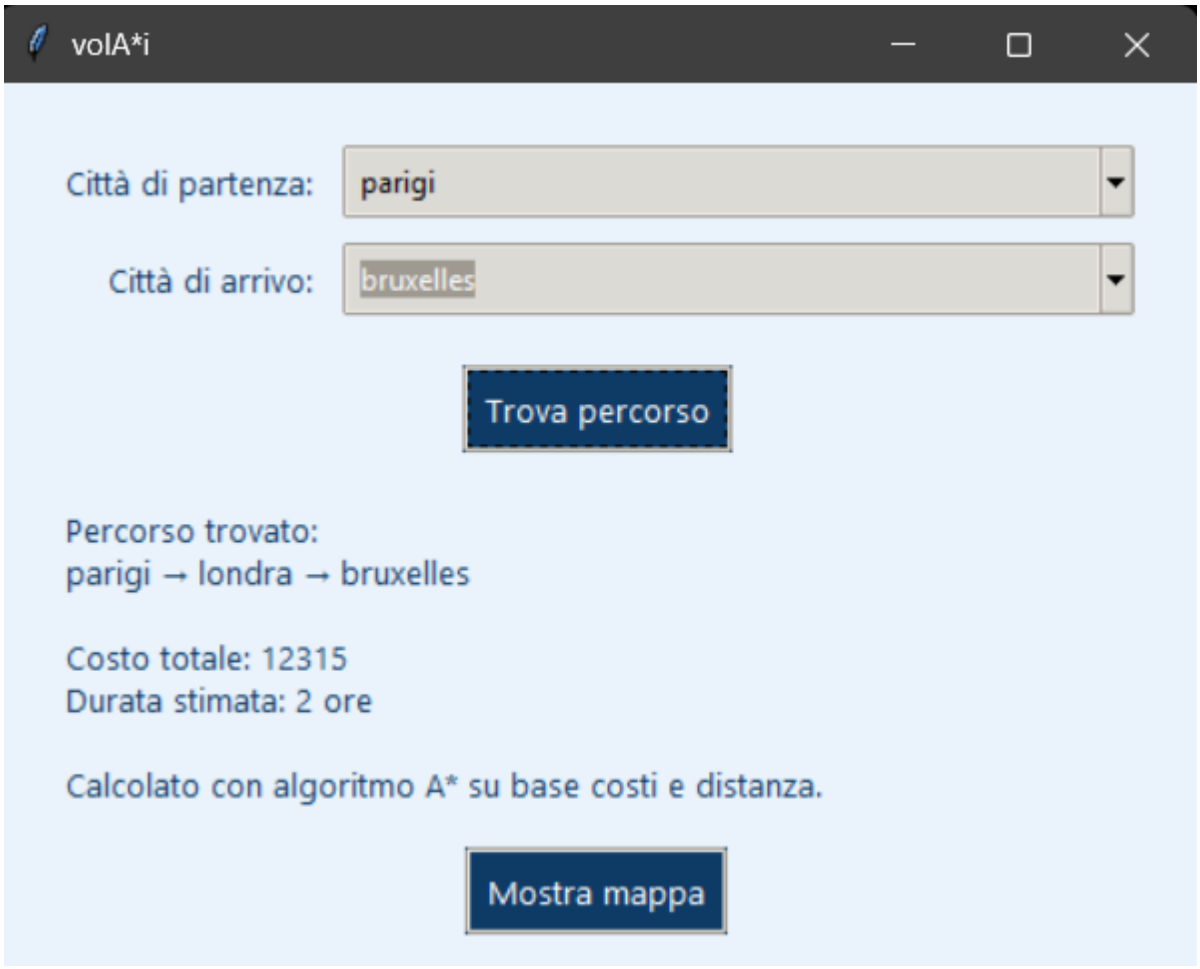
Per rendere l'applicazione accessibile anche a utenti non tecnici, il progetto include una interfaccia grafica (GUI) sviluppata interamente in Python utilizzando il modulo tkinter. L'interfaccia consente di interagire con l'algoritmo A* in modo semplice e intuitivo, attraverso una finestra grafica chiara, moderna e autoesplicativa.

Come si presenta:

Menù a tendina:



Esempio pratico:



Spiegazione

L’interfaccia si apre in una finestra compatta, con sfondo chiaro e uno stile coerente che richiama i toni del blu.

All’interno della finestra l’utente trova due menù a tendina: uno per selezionare la città di partenza, l’altro per scegliere la destinazione.

Una volta impostati i due aeroporti, è sufficiente cliccare su **“Trova percorso”** per avviare il calcolo del tragitto più efficiente tra le due città.

Il risultato viene mostrato direttamente nella finestra: il percorso viene elencato in ordine, con le città attraversate, seguito dal **costo totale** del viaggio e dalla **durata stimata** in ore.

Se il sistema trova una soluzione valida, si attiva anche un secondo pulsante, **“Mostra mappa”**, che permette di visualizzare il percorso su una **mappa geografica interattiva**, generata in HTML e aperta automaticamente nel browser.

Dal punto di vista tecnico, la visualizzazione si basa sulla libreria folium, che consente di tracciare linee tra coordinate geografiche e posizionare marker sulle città.

Le tratte percorse effettivamente sono evidenziate in rosso, mentre tutte le connessioni disponibili tra gli aeroporti vengono disegnate in grigio chiaro sullo sfondo, per contestualizzare il percorso scelto.

L’obiettivo della GUI non è solo estetico: essa serve a **rendere trasparente il funzionamento dell’algoritmo**, facilitando la comprensione dei percorsi selezionati e rendendo l’esperienza più coinvolgente anche per utenti non esperti di Intelligenza Artificiale.

In questo modo, la GUI si affianca al motore logico come strumento di **comunicazione e verifica**, rendendo più accessibile un processo decisionale complesso.

Folium

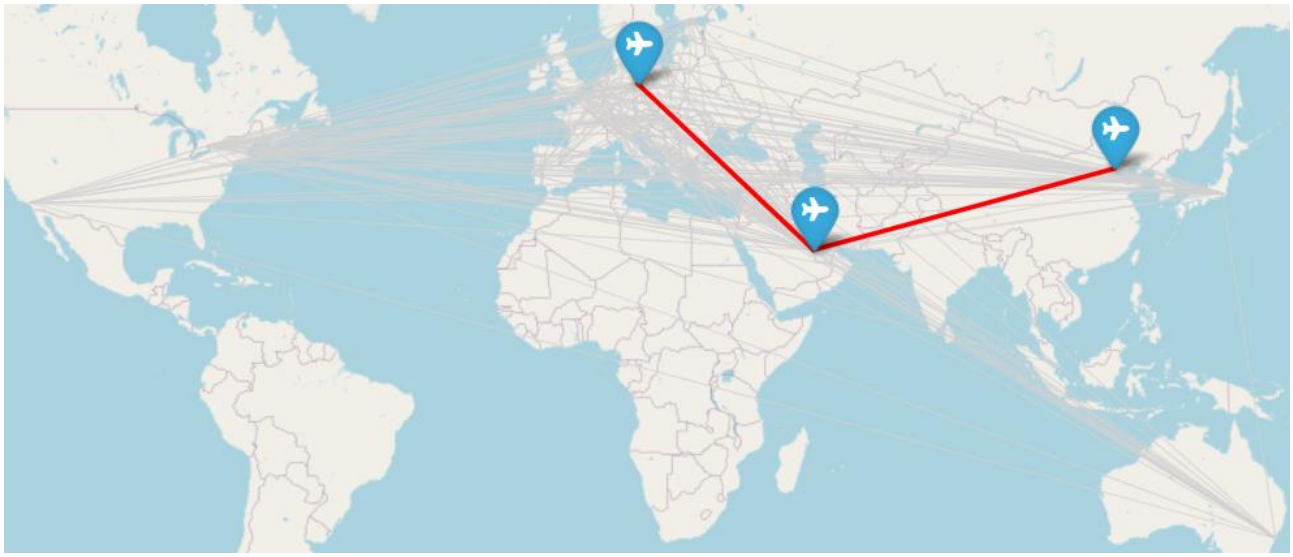
Folium è una libreria Python che permette di creare mappe interattive basate su **Leaflet.js**, un framework JavaScript per la visualizzazione geografica. Viene spesso usata per visualizzare **dati spaziali** in modo dinamico e personalizzabile.

Nel contesto di questo progetto, Folium è stata utilizzata per:

- Creare una **mappa** centrata sulla città di partenza
- Tracciare **le rotte disponibili** tra gli aeroporti (in grigio chiaro)
- Evidenziare **il percorso effettivamente calcolato** dall'algoritmo A* (in rosso)
- Inserire **marker con icone** sulle città attraversate

Il risultato è una **pagina HTML** generata automaticamente che si apre nel browser dell'utente e offre una visualizzazione chiara e interattiva del tragitto selezionato.

Esempio Berlino- Beijing;



Risultati e test del sistema

Per verificare il corretto funzionamento dell'algoritmo e dell'interfaccia, sono stati effettuati diversi test su coppie di città selezionate all'interno del dataset.

L'obiettivo era valutare se il sistema fosse in grado di:

- Trovare il percorso corretto in presenza di più tratte intermedie
- Penalizzare correttamente i percorsi con scali rispetto ai voli diretti
- Calcolare coerentemente costo e durata
- Mostrare il percorso in modo chiaro, sia testualmente che graficamente

Esempio 1: Parigi - Bruxelles

Nel primo test è stata calcolata la rotta da Parigi a Bruxelles. Il sistema ha identificato come percorso ottimale:

Parigi → Londra → Bruxelles,

con durata complessiva di 2 ore e costo aggregato

Questo risultato è coerente con la struttura del grafo, che non prevede voli diretti tra Parigi e Bruxelles, obbligando a un passaggio intermedio a Londra.

La penalizzazione per lo scalo è stata correttamente gestita, ma non ha impedito all'algoritmo di selezionare questa opzione come la più breve tra le disponibili.

Esempio 2: Roma - Tokyo

Nel secondo test è stato chiesto al sistema di calcolare un percorso da Roma a Tokyo. L'algoritmo ha proposto una soluzione con scali multipli, scegliendo rotte disponibili tra Europa e Asia, e valutando il compromesso tra costo, distanza e numero di tratte.

Visualizzazione

In entrambi i casi, è stato possibile visualizzare il percorso sulla mappa: le città selezionate sono state evidenziate con marker, e il tragitto effettivo è stato disegnato in rosso, sovrapposto a tutte le possibili connessioni (in grigio chiaro).

Questo ha reso l'output più leggibile e immediato, facilitando la comprensione del risultato.

Conclusioni

Questo progetto ha rappresentato un'applicazione concreta dei concetti fondamentali dell'Intelligenza Artificiale, in particolare nel contesto della **ricerca del percorso ottimale** su una rete di stati. L'algoritmo A* , scelto come nucleo del sistema, si è dimostrato efficace, flessibile e adatto a gestire situazioni reali come quelle simulate dalla rete di voli implementata.

La struttura modulare del progetto: separando dati, logica algoritmica, interfaccia e visualizzazione, ha reso possibile uno sviluppo ordinato e

facilmente estendibile. Anche l'uso del formato .pl ispirato a Prolog, pur senza eseguire inferenza logica, ha permesso di mantenere **leggibilità e coerenza con l'approccio dichiarativo** proprio dell'IA classica.

Dal punto di vista tecnico, l'algoritmo ha fornito risultati corretti e coerenti in tutti i casi testati, dimostrando la validità dell'euristica geografica e la funzionalità della penalizzazione per gli scali. L'interfaccia grafica ha semplificato l'uso del sistema e la visualizzazione su mappa ha aggiunto un elemento esplicativo utile per l'interpretazione del risultato.

A livello personale, il progetto ha rappresentato una **sintesi tra teoria e pratica**, offrendo l'occasione di applicare concretamente algoritmi studiati a lezione, e di integrare logiche tipiche dell'IA con strumenti moderni di programmazione e visualizzazione. Inoltre, ha lasciato spazio alla sperimentazione, aprendo la possibilità per **sviluppi futuri** e miglioramenti, sia tecnici che concettuali.

In conclusione, il sistema realizzato non è solo un'applicazione funzionante, ma anche una dimostrazione di come i principi dell'Intelligenza Artificiale possano essere tradotti in soluzioni pratiche, didattiche e, con le opportune estensioni, potenzialmente anche applicative.