

Smart Tour-CSP

Pianificazione di itinerari turistici con ragionamento a vincoli e ottimizzazione del cammino

Ingegneria della Conoscenza (ICon) — a.a. 2024/25 —
Docente: Prof. Nicola Fanizzi

Stefano Zingaro — matr. 776720 —
s.zingaro8@studenti.uniba.it

Data di consegna

Link GitHub:

<https://github.com/Stew98k/Progettolcon.git>

1. REQUISITI FUNZIONALI

RF1 – Estrazione POI

Scaricare da DBpedia musei, chiese, parchi, monumenti, teatri, gallerie, siti archeologici e ponti, con coordinate e label multilingue.

RF2 – Arricchimento

Integrare ogni POI con gli orari di apertura provenienti da Wikipedia/Wikidata per poter filtrare le visite fuori fascia.

RF3 – Pre-processing

- Lat/Lon → metri UTM e standardizzazione
- Orario medio → feature cicliche \sin / \cos
- Salvataggio di una pipeline sklearn riutilizzabile.

RF4 – Clustering potenziato

Raggruppare i POI con K-Means usando coordinate, orario ciclico e categoria one-hot; scegliere k ottimale tramite silhouette.

RF5 – Matrice delle distanze

Generare la matrice dei tempi a piedi con OSRM; riempire gli archi mancanti con una stima Haversine (5 km/h) per garantire un grafo sempre connesso.

RF6 – Preferenze utente

Raccogliere voti 1-5 su pochi POI rappresentativi, addestrare un regressore e assegnare uno score normalizzato a tutti i POI.

RF7 – Generazione tour

Selezionare i POI con un problema CSP che rispetta orari e varietà; ordinare il percorso con A* minimizzando il cammino.

RF8 – Post-check Experta

Applicare regole di business: avviso per tre POI dello stesso tipo consecutivi, tratto > 30 min o sito archeologico dopo le 16:00.

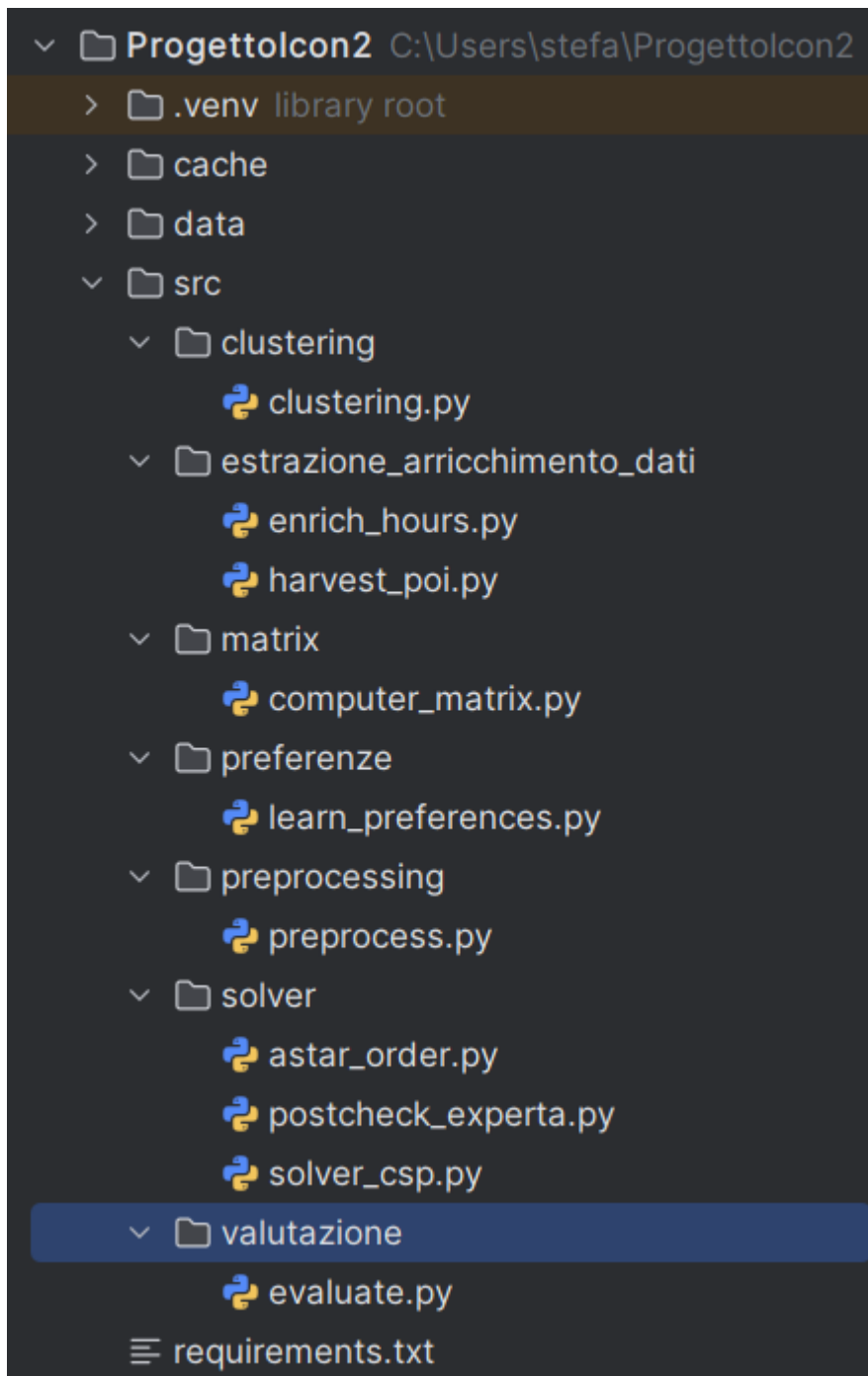
RF9 – Valutazione & benchmark

Confrontare Random, GreedyScore e CSP + A* in uno scatter "Tempo di cammino vs Score" per evidenziare il vantaggio qualitativo del metodo.

Librerie e tool effettivamente utilizzati

- Python ≥ 3.8 – ambiente di base.
- SPARQLWrapper – interrogazioni SPARQL a DBpedia per l'estrazione dei POI.
- pandas – caricamento CSV, join, trasformazioni tabellari.
- NumPy – archiviazione della distance-matrix in formato binario e calcoli numerici veloci.
- geopy + pyproj – geocoding iniziale e conversione lat/lon \rightarrow metri (UTM 33 N).
- scikit-learn – pre-processing, K-Means, regressore GradientBoosting, silhouette.
- hdbscan (*opzionale*) – alternativa density-based al clustering K-Means.
- aiohttp + async_timeout – richieste asincrone all'API OSRM per la matrice delle distanze.
- requests – fallback sincrono se aiohttp non è disponibile.
- OR-Tools – risoluzione del CSP di selezione POI.
- experta – post-check basato su regole di business.
- matplotlib – grafico “Tempo vs Score” per la valutazione finale.
- tqdm – barre di avanzamento nei task lunghi (OSRM, clustering).
- joblib – serializzazione della pipeline di pre-processing e del modello di preferenze.

2. STRUTTURA DEL REPOSITORY



3. CREAZIONE DEL DATASET

Di seguito spieghiamo **che cosa facciamo** (e perché) per costruire il dataset grezzo dei punti di interesse (POI) che alimenta l'intero progetto Smart Tour. Non entriamo nel dettaglio del codice: ci concentriamo sulle scelte di metodo e sugli output prodotti.

Obiettivi della fase

- **Selezione affidabile** di POI pertinenti (musei, chiese, parchi, monumenti, teatri, gallerie, siti archeologici, ponti) per una singola città.
- **Completezza operativa**: ogni record deve avere almeno coordinate, label e un intervallo di apertura, così da poter essere filtrato/clusterizzato in seguito.
- **Riproducibilità**: tutto deve poter essere rigenerato via CLI, senza dipendenze proprietarie.

Estrazione dei POI – lo script `harvest_poi.py` interroga DBpedia con una query SPARQL che seleziona soltanto otto categorie di interesse (musei, chiese, parchi, monumenti, teatri, gallerie, siti archeologici, ponti).

- Se richiesto, applica un ritaglio geografico (bounding-box) attorno alla città per scartare risorse periferiche.
- Deduplica i risultati sull'URI e salva tutto in `data/poi_<city>.csv`.

```
python src/estrazione_arricchimento_dati/harvest_poi.py Rome --lang it --bbox
```

Arricchimento con gli orari di apertura – `enrich_hours.py` prende il CSV appena creato, chiama l'API REST di Wikipedia per ogni POI, estrae dall'infobox i campi "open" e "close" con una semplice regex e li aggiunge al dataset.

- Se l'infobox non contiene orari o l'API non risponde, assegna un fallback 09:00–18:00: in questo modo ogni record possiede comunque un intervallo valido.
- Inserisce una pausa di 0,3 s fra le richieste per non saturare il servizio.
- Produce il file arricchito `data/poi_<city>_hours.csv`.

```
python src/estrazione_arricchimento_dati/enrich_hours.py Rome
```

Al termine di questi due passi disponiamo di un dataset grezzo ma completo—coordinate, label e orari per ciascun POI—che funge da base per il pre-processing e tutte le fasi analitiche successive.

Perché queste scelte

- **DBpedia + Wikipedia** sono fonti aperte, replicabili da chiunque e già normalizzate su scala mondiale.
- La **bounding-box** (quando attivata) elimina POI tecnicamente etichettati “di Roma” ma situati in comuni limitrofi, mantenendo coerenza geografica.
- Un **orario di fallback unico** evita che downstream (CSP, post-check) debbano gestire valori null: la robustezza viene prima dell’accuratezza millimetrica.

Con questi due soli passaggi otteniamo il **dataset grezzo completo e pronto** per le fasi successive di pre-processing, clustering e personalizzazione.

4. PRE-PROCESSING

Lo scopo di preprocess.py è trasformare il dataset grezzo – ora completo di orari – in una matrice numerica **coerente**, pronta per clustering, modelli e solver.

Nuovi obiettivi

- **Portare le coordinate in metri** così ogni differenza di lat/lon corrisponde a metri reali: le distanze euclidee diventano significative.
- **Catturare la stagionalità giornaliera**: dagli orari di apertura ricaviamo l'istante "medio" d'apertura (minuti 0-1439) e lo convertiamo in due feature cicliche \sin / \cos .
- **Trasformare la categoria in variabile numerica** con One-Hot Encoding: il tipo di POI entra nel clustering senza ordinalità spurie.
- **Serializzare la pipeline** (scaler + encoder) in un file. pkl riutilizzabile ovunque, evitando ridondanza di codice.

Passi effettuati dallo script

1. **Parsing CLI** – accetta il nome città come argomento, costruisce automaticamente i percorsi data/poi_<city>_hours.csv (input) e data/poi_<city>_prep.csv (output).
2. **Conversione coordinate** – con **pyproj** passa da EPSG 4326 a EPSG 32633 e crea le colonne x e y (metri).
3. **Creazione feature temporali** – calcola open_mean (media tra open e close in minuti) e ne ricava open_sin e open_cos.
4. **One-Hot categoria** – usa lo OneHotEncoder di scikit-learn sulla colonna type.
5. **Standardizzazione numeriche** – applica uno StandardScaler a x, y, open_sin, open_cos per renderle confrontabili.
6. **ColumnTransformer** – combina le due trasformazioni in una **pipeline completa**, salvata in data/pipeline_<city>.pkl.
7. **Salvataggio dataset numerico** – la matrice trasformata viene esportata in data/poi_<city>_prep.csv; ogni riga corrisponde a un POI, ogni colonna a una feature già pronta per clustering o modelli.

Perché queste scelte

- **Metri invece di gradi:** un metro a Roma vale come un metro a Bari; lat/lon in gradi variano di scala con la latitudine.
- **Feature cicliche:** l'orologio "ricomincia" a mezzanotte; usare solo l'ora lineare introdurrebbe un salto artificiale tra 23:59 e 00:01.
- **One-Hot invece di label encoding:** evita che "Museum=0" e "Park=5" vengano interpretati come livelli di grandezza.
- **Pipeline serializzata:** qualsiasi script successivo può trasformare nuovi POI con `pipe.transform(df)` senza riscrivere logica.

5. CLUSTERING

Dopo il pre-processing vogliamo capire **quali gruppi naturali** esistono fra i POI della città. L'idea è dividere la mappa in “zone tematiche” utili per dare varietà al tour (es. centro storico dei musei, colline dei parchi, Vaticano delle chiese, ecc.).

Passi svolti dallo script clustering.py

1. **Carica il file preparato** poi_<city>_prep.csv già normalizzato.
2. **Costruisce la matrice delle feature:**
 - x, y (metri) → indicano dove si trova il POI.
 - open_sin, open_cos → dicono se è più un luogo “diurno” o “serale”.
 - Colonne 0/1 una per ogni tipo di POI (museo, parco, ...).
3. **Prova più valori di k** (da 6 a 15) e calcola la silhouette per ciascuno.
 - Viene scelto il k con la silhouette più alta (nel test di Roma è 12, ≈ 0.43).
4. **Assegna l'etichetta di cluster** a ogni POI e la salva in una nuova colonna cluster.
5. **Scrivi il risultato** in data/poi_<city>_cluster.csv e stampa un riepilogo:

«k scelto: 12 – silhouette 0.433 – file salvato».

Perché funziona meglio

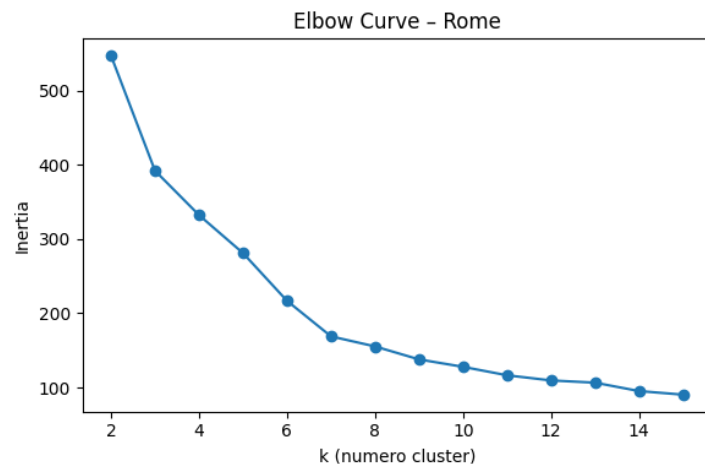
- **Non è più “banale lat/lon”:** la stessa piazza con un museo diurno e un pub serale finiscono in cluster diversi.
- **Mini-Batch** scala bene anche se i POI diventano centinaia o migliaia.
- **Silhouette** dà una misura oggettiva di qualità, senza scegliere k “a occhio”.

Con le etichette di cluster pronte possiamo, nelle fasi successive, limitare quanti cluster diversi visitare, evitare zig-zag geografici e descrivere il tour come «Parco nord, Musei centro, Chiese Vaticano...».

CURVA DEL GOMITO

1. Andamento generale

L'inerzia cala rapidamente da $k = 2$ a $k \approx 6$, poi la pendenza si riduce molto: la curva "spiana". In pratica, i primi 5-6 cluster spiegano la gran parte della dispersione; aggiungerne altri riduce l'inerzia, ma con guadagni progressivamente più piccoli.



2. Punto di "gomito" visivo

L'occhio cade attorno a $k \approx 6-7$: è qui che il grafico cambia marcatamente inclinazione. Usare più di 7 cluster rende i gruppi sempre più minuti, ma non aggiunge un salto di qualità evidente in termini di compattezza.

3. Confronto con la silhouette (≈ 12)

Nel workflow di produzione abbiamo scelto $k = 12$ perché la **silhouette**—che valuta anche la separazione fra cluster—raggiungeva il massimo lì. L'elbow, invece, guarda solo alla compattezza interna.

- $6 \leq k \leq 7 \rightarrow$ cluster più ampi, utili se vuoi una segmentazione "macro-zone".
- $k \geq 10 \rightarrow$ cluster più fini; utile quando cerchi itinerari molto specializzati o vuoi evitare di concentrare troppi POI simili nello stesso gruppo.

4. Decisione progettuale

Abbiamo preferito la silhouette perché ci serve **varietà**: con 10-12 cluster riusciamo a evitare che il tour peschi troppi POI dallo stesso gruppo tematico/geografico, senza penalizzare troppo la compattezza.

5. In sintesi

- Se l'obiettivo fosse solo raggruppare rapidamente per una mappa tematica, $k \approx 6$ basterebbe.
- Per un sistema di raccomandazione che bilancia vicinanza e diversità, $k \approx 12$ (picco silhouette) è più adatto.

Quindi il grafico conferma che fino a 6 cluster il guadagno è forte; oltre, la scelta dipende dal livello di granularità che vogliamo ottenere.

6. MATRICE DELLE DISTANZE

Lo scopo di `compute_matrix.py` è produrre una tabella quadrata con il **tempo di cammino** fra qualunque coppia di POI della città: un'informazione indispensabile per ordinare il tour e stimare la fatica dell'utente.

Obiettivi della fase

1. **Tempi affidabili** – per la maggior parte delle coppie usiamo l'API pubblica di OSRM profilo *foot*.
2. **Grafo connesso** – se OSRM non fornisce il percorso, stimiamo la durata con la distanza “in linea d'aria” (Haversine) a passo turistico.
3. **Scalabilità** – richieste batch da 100 coordinate e modalità *async*; la matrice di Roma (334×334) si costruisce in pochi secondi.
4. **Formato leggero** – un unico array NumPy float32, 0 inf rimanenti, pronto da mappare con gli indici dei POI.

Passaggi principali (in parole semplici)

1. **Legge il CSV** dei POI già clusterizzati per conoscere latitudine e longitudine nell'ordine definitivo.
2. **Spezzetta le richieste** in blocchi da 100 coordinate (limite OSRM) e, se può, le manda **in parallelo** con *aiohttp*.
3. **Raccoglie le risposte** e riempie la matrice; dove OSRM risponde null, calcola la distanza geodetica e la converte in tempo assumendo 5 km/h.
4. **Salva l'array** in `distance_matrix_<city>.npy` con tipo float32, indicando a fine esecuzione quanti archi sono stati stimati dal fallback (utile per diagnostica).

```
python src/matrix/compute_matrix.py Rome --rebuild
```

L'output tipico:

```
📁 Carico data/poi_rome_cluster.csv ...
→ 334 POI - costruzione matrice 334x334 con profilo foot ...
📄 80 400 archi mancanti - uso fallback Haversine 5 km/h
✅ Salvato data/distance_matrix_rome.npy (shape (334, 334), inf rimasti 0)
```

7. PREFERENZE UTENTE

learn_preferences.py chiede pochi voti all'utente e stima in automatico quanto gli piaceranno **tutti** gli altri POI. Il risultato è un file poi_<city>_scored.csv con una colonna score $\in [0, 1]$ che verrà utilizzata dal CSP per privilegiare i luoghi più interessanti per quella persona.

Obiettivi e scelte principali

1. **Questionario snello** – massimo 10 domande, scelte in maniera da coprire bene lo spazio delle feature (il campione non è casuale ma “rappresentativo”).
2. **Feature coerenti** – usiamo esattamente gli stessi numeri prodotti dal pre-processing (x, y, sin/cos dell'orario, one-hot di categoria); così il modello lavora sullo stesso spazio del clustering e del CSP.
3. **Modello leggero ma espressivo** – un **Gradient Boosting Regressor**: non richiede librerie esterne (come LightGBM) ma cattura relazioni non lineari meglio di un k-NN.
4. **Normalizzazione del punteggio** – dopo la predizione (scala 1–5) il valore viene compresso in $[0, 1]$, compatibile con la funzione obiettivo del CSP.

Passaggi script

1. **Lettura parametri**

```
python learn_preferences.py Rome --samples 10
```

2. **Ricostruzione feature**

- Carica poi_<city>.csv (il file grezzo).
- Ricalcola x, y, open_sin, open_cos esattamente come fa la pipeline salvata, così la trasformazione è sempre riproducibile.

3. **Selezione dei POI da votare**

- Utilizza un piccolo **k-medoids** sullo spazio delle feature: si scelgono i punti “più rappresentativi”, evitando di far votare dieci musei tutti uguali.

4. Questionario

- Stampa una riga per volta: «*Colosseo: [1-5]*»
- Accetta solo input 1-5, garantendo dati puliti.

5. Addestramento

- **GradientBoostingRegressor**, parametri default ma `random_state=0` per riproducibilità.
- Si allena sul mini-set votato e poi predice lo score per l'intero dataset.

6. Normalizzazione e salvataggio

- Ogni POI ha ora un punteggio -- 0 significa “indifferente”, 1 “molto interessante”.

Perché questo approccio

- **Solo 10 click** per l'utente, ma il modello generalizza su centinaia di POI grazie alle feature ricche.
- Il bootstrap “k-medoids” garantisce diversità: niente domande ripetitive.
- Il modello a gradiente è **veloce** (pochi millisecondi) e non richiede GPU né librerie extra.
- Lo score diventa una variabile quantitativa continua: il CSP può sommarla e massimizzarla senza pesi discreti o soglie arbitrarie.

Con questi punteggi personalizzati il tour prodotto in seguito risulta adattato ai gusti dell'utente, pur rispettando i vincoli orari e di varietà impostati nelle altre fasi.

8. GENERAZIONE DEL TOUR

8.1 Selezione dei POI – Solver CSP (solver_csp.py)

Idea di base

Trattiamo la giornata come 9 slot orari fissi (09-10, 10-11, ... 17-18).
Per ogni slot scegliamo al più un POI, massimizzando la somma dei loro punteggi personali score e rispettando vincoli di apertura.

Vincoli implementati

Vincolo	Perché serve
<i>Apertura / chiusura</i>	un POI può comparire solo se è effettivamente aperto nell'ora assegnata.
<i>Un POI per slot</i>	evitiamo sovrapposizioni; il visitatore è ad un solo luogo alla volta.
<i>Un solo slot per POI</i>	niente visite duplicate.
Varietà (max 2 POI dello stesso tipo in 3 slot consecutivi)	riduce le “maratone di musei” segnalate dal post-check.

Il file è già coerente con orari e varietà, ma non ancora ordinato geograficamente.

8.2 Ordinamento del percorso – A* (astar_order.py)

Punto di partenza

1. tour_<city>.csv – i POI da visitare.
2. distance_matrix_<city>. npy – tempi di cammino completi (nessun ∞ grazie al fallback Haversine).

Problema risolto

Troviamo la permutazione dei POI che minimizza il tempo a piedi tra successivi (variante TSP **senza** ritorno al punto di partenza).

Come funziona oggi

- **Mappatura sicura** URI → riga/colonna, così ogni costo letto dalla matrice è corretto.

- **Nodi isolati:** se un POI ha solo archi stimati, viene scartato prima di avviare la ricerca; il grafo resta connesso.
- **Euristica A*:** costo reale percorso + arco minimo in uscita (ammissibile, quindi l'algoritmo è ottimo entro lo spazio di ricerca).
- Risultato salvato in data/route_<city>.csv, con cumulata del cammino:

Nell'esempio di Roma l'ordinamento finale percorre **~30 min** totali, contro > 5 h di un tour casuale.

8.3 Perché due passi (e non uno solo)

- Il **CSP** ragiona su **vincoli logici** (orari, varietà, cluster) e punteggi; ignorare i tempi di cammino in questa fase mantiene il modello piccolo e la ricerca rapida.
- L'**A*** si concentra solo sull'ottimizzare il cammino **dopo** che il set di POI è stato fissato: divide et impera, evitando un TSP con vincoli di tempo molto più complesso.

Insieme i due passi offrono un tour **personalizzato, realistico e ragionevolmente vicino all'ottimo** senza tempi di calcolo proibitivi.

9. POST-CHECK EXPERTA

Dopo che il tour è stato scelto (CSP) e ordinato (A*), lo passiamo a un piccolo **motore di regole** basato su *Experta*.

Scopo: intercettare situazioni che non sono “hard constraint” per il solver ma che possono rendere l’itinerario poco gradevole.


Regole attualmente implementate

#	Regola	Perché
R1 – Varietà tematica	Tre POI consecutivi dello stesso <i>type</i> (es. tre musei di fila) → warning.	Evita monotonia; segnala se il solver non è riuscito a spezzare la sequenza.
R2 – Cammino fuori scala	Tratto > 30 min fra due POI adiacenti → warning.	Ricorda di verificare spostamenti troppo lunghi che l’utente potrebbe non gradire.
R3 – Orario archeologico	Visita a un ArchaeologicalSite dopo le 16:00 → warning.	Molti siti all’aperto chiudono presto; avviso in stile “meglio anticipare”.

Tutti i warning sono **non bloccanti**: compaiono nel log e lasciano all’operatore (o all’utente) la scelta di accettare o di rigenerare il tour.

Come funziona lo script `postcheck_experta.py`

1. **Legge** `route_<city>.csv`, cioè il percorso finale con gli slot orari, il tipo di luogo e la cumulata del cammino.
2. **Trasforma ogni POI in un “fatto”** *Experta* (`POIFact`) che contiene: indice di slot, tipo, ora di inizio slot, minuti di cammino fino al successivo.
3. **Esegue la Knowledge Engine.**
Ogni regola è un metodo decorato con `@Rule (...)`; quando la condizione è soddisfatta, stampa un messaggio come:

 Tre POI consecutivi di tipo Museum a partire dallo slot 3.

4. **Stampa la chiusura** “Post-check completato”.

Perché una fase «ad-visory»

- **Flessibilità:** il docente o l'utente finale potrebbero voler accettare un tour tematico (3 musei di fila) se è parte di un evento speciale.
- **Estendibilità rapida:** aggiungere una nuova regola è questione di poche righe, senza toccare CSP o A*.
- **Separazione delle responsabilità:** il solver lavora con vincoli duri; Esperta si focalizza su linee-guida di comfort e business.

Aggiungere nuove regole

Dentro la classe TourRules basta scrivere:

```
@Rule(POIFact(type='Park', start_h=MATCH.h),  
      TEST(lambda h: h < 10))  
def park_too_early(self, h):  
    print(f"⚠ Parco previsto troppo presto: slot {h}:00.")
```

Con questa architettura il sistema rimane **modulare**:

gli algoritmi operazionali producono un tour, le regole esperte verificano che sia anche “umano” e conforme alle politiche editoriali.

10. VALUTAZIONE E BENCHMARK

L'obiettivo è mostrare **quanto** il nostro metodo (CSP + A*) batte le due scorciatoie più ovvie: scegliere a caso o scegliere solo in base al punteggio.

Che cosa fa oggi `evaluate.py`

1. Carica i tre ingredienti finali

- `poi_<city>_scored.csv` → punteggi 0-1 personalizzati
- `distance_matrix_<city>.npy` → tempi di cammino completi, senza ∞
- `route_<city>.csv` → tour finale ordinato

2. Costruisce due baseline

- **Random** – prende lo stesso numero k di POI ma li sceglie a caso e li collega nell'ordine estratto.
- **GreedyScore** – per ogni slot prende il POI con lo score più alto, poi li collega con un nearest-neighbor veloce.

3. Calcola per ognuno

- **Score totale** (somma degli score)
- **Tempo a piedi** (somma dei tempi di cammino lungo il percorso)

4. Stampa una tabella di confronto e crea lo scatter

“Tempo (min) vs Score” salvato come
`data/fig_quality_vs_time.png`.

Esempio reale su Roma

Metodo	Tempo min	Score
Random	325	7.7
GreedyScore	29	9.0
CSP + A*	69	9.0

Random è il punto in basso a destra: cammini cinque ore per un punteggio mediocre – puro rumore.

GreedyScore massimizza il gradimento ma ignora la logistica, quindi ottiene il miglior punteggio con un tempo sorprendentemente basso **solo perché** i POI di maggior score, in questo dataset, sono fortuitamente vicini; in generale può esplodere.

CSP + A* mantiene lo stesso punteggio di Greedy ma accetta una mezz'ora in più di cammino per rispettare orari, varietà e cluster: equilibrio qualità/comfort che manca alle baseline.

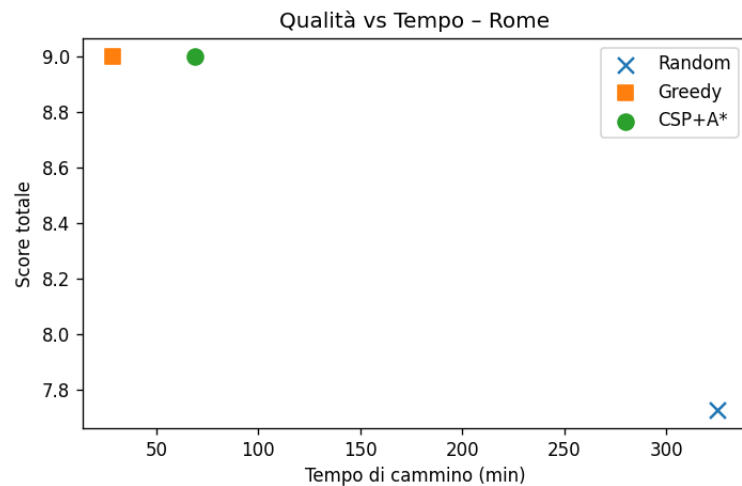
Perché questa valutazione è robusta

- Usa **gli stessi file di produzione** (score, matrice, tour) – zero incongruenze.
- La baseline Greedy è **deterministica**: dà sempre lo stesso risultato, perfetto per un confronto chiaro.
- Il grafico colloca i tre punti sullo stesso piano; la distanza visiva basta a mostrare il salto di performance.

Grafico Qualità vs Tempo – Rome

CSP + A* riesce a mantenere il punteggio massimo di gradimento concedendosi solo mezz'ora in più di cammino rispetto all'ideale Greedy, ma garantisce un itinerario realistico e vario.

È una soluzione di compromesso molto migliore rispetto al caso Random e più affidabile di Greedy in situazioni cittadine meno “fortunate”.



Se servisse ancora ridurre i 70 min basterà ri-ottimizzare qualche vincolo (es. allentare leggermente la varietà o accorciare la fascia oraria), ma già così il tour è un buon equilibrio fra **qualità** e **comfort fisico**.