Ruggiero Vito e
Rambaudo Aldo

Time-Dependent Shortest Path

Tesi di approfondimento sul problema del Time-Dependent Shortest Path

Indice

1. Shortest Path problem

Il classico Shortest Path (SP) ha svolto un ruolo fondamentale nell'ottimizzazione dei percorsi per decenni. Tuttavia, questo modello presenta delle limitazioni significative quando si tratta di affrontare situazioni del mondo reale in cui il tempo gioca un ruolo cruciale.
Il modello SP suppone che i costi per attraversare gli archi di un grafo siano fissi, ignorando completamente le fluttuazioni temporali che possono verificarsi.

È qui che entra in gioco il Time-Dependent Shortest Path (TDSP), un concetto che rivoluziona l'approccio alla pianificazione dei percorsi.

2. Differenze tra SP e TDSP

Per comprendere a pieno il concetto di Time Dependent Shortest Path (TDSP), è essenziale confrontarlo con il classico problema del Shortest Path (SP).

Costi Variabili nel Tempo

Nel SP classico, i costi associati agli archi del grafo sono considerati fissi; questo significa che il tempo necessario per attraversare un arco rimane costante indipendentemente dal momento in cui si effettua il percorso. Il TDSP invece riconosce le variazioni temporali dei costi.
Ogni arco del grafo temporale è associato a una funzione di costo temporale che rappresenta il tempo di percorrenza in funzione del momento della giornata.

Orario di Partenza

Nel SP classico, non è necessario specificare un orario di partenza, in quanto il calcolo del percorso ottimale si basa esclusivamente sulla struttura statica del grafo.
Invece, nel TDSP, è fondamentale specificare l'orario di partenza desiderato, perché questo determina il punto di partenza per il calcolo del percorso ottimale, tenendo conto delle variazioni dei costi nel tempo a partire da quell'orario.

Applicazioni Pratiche

Il SP classico è spesso utilizzato in scenari in cui i costi degli archi sono statici, come la navigazione su mappe digitali con informazioni fisse sul traffico.
Il TDSP trova applicazioni in situazioni reali in cui il tempo è un fattore critico, come la pianificazione di percorsi ottimali in città congestionate, la gestione della logistica con orari variabili dei trasporti e la risoluzione di emergenze che richiedono percorsi tempestivi e dinamici.

3. Struttura Time Dipendent Shortest Path (TDSP)

1. Modello Concettuale

Il TDSP è un’estensione del classico problema del Shortest Path (SP) che tiene conto delle variazioni dei costi nel tempo. La sua struttura si basa su due componenti principali:

- **Grafo Temporale:** in questo grafo, ogni arco è associato a una funzione di costo temporale che rappresenta il costo (tempo) necessario per attraversare quell'arco in un determinato momento.
  Queste funzioni di costo temporale possono variare durante la giornata in base a fattori come il traffico stradale o gli orari dei mezzi pubblici.
  Un Grafo Temporale può essere definito come un grafo pesato G = (V, E), dove V rappresenta l'insieme dei nodi e E l'insieme degli archi. Ogni arco (u, v) in E è associato a una funzione di costo temporale c(u, v, t), dove u e v sono nodi collegati da quell'arco, e t rappresenta il tempo.

- **Orario di Partenza:** è necessario specificare l'orario di partenza desiderato.
  Questo determina il momento da cui inizierà il calcolo del percorso ottimale. Il TDSP cerca, quindi, un percorso che minimizzi il tempo totale di percorrenza, tenendo conto delle variazioni dei costi in funzione del tempo.

3. Struttura Time Dipendent Shortest Path (TDSP)

2. Rappresentazione del Grafo

La rappresentazione del grafo nel Time Dependent Shortest Path è essenziale per tenere traccia delle variazioni temporali dei costi.
Ciò può essere fatto in diversi modi:

- **Archi Temporali Ponderati:** ogni arco del grafo è associato a un peso temporale che rappresenta il costo (tempo) per attraversare quell'arco in un momento specifico. Questi pesi possono essere pre-calcolati o aggiornati in tempo reale a seconda delle variazioni dei costi.
- **Intervallo di Validità:** ogni arco può avere un intervallo di validità temporale durante il quale il suo costo è applicabile. Ad esempio, un arco potrebbe avere un costo diverso durante le ore di punta rispetto alle ore in cui il traffico è minore.
- **Grafi Dinamici:** in alcuni casi, è possibile utilizzare grafi dinamici che si adattano automaticamente alle variazioni temporali dei costi. Questi grafi possono essere più complessi da gestire, ma consentono una modellazione precisa delle variazioni.

3. Struttura Time Dipendent Shortest Path (TDSP)

3. Funzione di Costo Temporale

La funzione di costo temporale è responsabile della rappresentazione dei costi variabili nel tempo.
Essa associa un costo temporale a ciascun arco del grafo in base al momento della giornata.
Sia G = (V, E) il grafo pesato, dove V rappresenta l'insieme dei nodi e E rappresenta l'insieme degli archi.
Sia c(u, v, t) la funzione di costo temporale associata all'arco (u, v) nel grafo, dove:

- u rappresenta il nodo di partenza dell'arco;
- v rappresenta il nodo di destinazione dell'arco;
- t rappresenta il tempo, che è una variabile continua.

- **Funzioni Lineari**

c(u, v, t) = a(u, v) * t + b(u, v); dove a(u, v) rappresenta la pendenza della funzione lineare e b(u, v) rappresenta il termine di intercetta.
Un approccio comune consiste nell'utilizzare funzioni di costo lineari, dove il costo varia in modo proporzionale al tempo

- **Funzioni non Lineari**

c(u, v, t) = f(u, v, t); dove f(u, v, t) è una funzione non lineare che modella variazioni complesse dei costi nel tempoIn alcuni casi, le variazioni dei costi potrebbero non essere uniformi nel tempo, ad esempio a causa di congestioni stradali improvvise. Per modellare questo tipo di variazioni più complesse, possono essere utilizzate le funzioni di costo non lineari, in cui il costo per attraversare un arco varia in modo non lineare rispetto al tempo.

- **Funzioni a scaglioni**

c(u, v, t) = c_i(u, v) per t in [t_i, t_{i+1}); dove c_i(u, v) rappresenta il costo dell'arco (u, v) durante l'intervallo di tempo [t_i, t_{i+1}). Le funzioni di costo a scaglioni suddividono il tempo in intervalli discreti e assegnano un costo fisso a ciascun intervallo. I costi rimangono costanti all'interno di ogni intervallo e possono cambiare bruscamente quando si passa da un intervallo all'altro, come il traffico nei pressi di diversi punti di interesse di una città che cambia in base alla fascia oraria.

4. Algoritmi di risoluzione del TDSP

1. Dijkstra Modificato
2. Algoritmo A star
3. Contraction Hierarchies

4.1 Dijkstra

Dijkstra con Heap Binario Modificato è una variante di Dijkstra e tiene conto della variazione dei pesi degli archi nel tempo, questa variazione può rappresentare situazioni reali come il traffico stradale che cambia nel corso della giornata.

- **Inizializzazione:** si parte impostando tutti i costi dei nodi a "infinito" tranne il costo del nodo di partenza, che impostiamo a zero. Inoltre, creiamo un heap binario e aggiungiamo il nodo di partenza con un costo di zero all'heap binario.
- **Heap Binario Modificato:** l'heap binario viene utilizzato per mantenere i nodi in ordine di costo crescente. In ogni iterazione, estraiamo il nodo con il costo minimo dall'heap binario. Questo nodo rappresenta il candidato per il percorso più breve fino a quel momento. Successivamente, esaminiamo tutti i nodi adiacenti al nodo estratto e calcoliamo il costo dei loro relativi percorsi. Se il costo calcolato è inferiore al costo attualmente registrato per il nodo adiacente, aggiorniamo il costo e inseriamo il nodo adiacente nell'heap binario. Ripetiamo questa procedura finché non abbiamo estratto il nodo di destinazione dall'heap binario o finché l'heap binario non è vuoto.
- **Tempo-Dipendente:** i costi associati ai nodi e agli archi possono variare nel tempo. Pertanto, quando calcoliamo il costo per raggiungere un nodo adiacente, dobbiamo tener conto dell'orario corrente o della variabile temporale associata.
- **Ricalcolo dei percorsi:** a causa delle variazioni dei pesi degli archi nel tempo, l'algoritmo potrebbe dover ricalcolare i percorsi più brevi in modo dinamico.

4.2 A star

L'algoritmo A* modificato per il problema del TDSP è un algoritmo di ricerca euristica utilizzato per trovare il percorso più breve da un punto di partenza a un punto di destinazione in un grafo, tenendo conto delle variazioni temporali nei costi dei nodi e degli archi.

- **Inizializzazione:** vengono inizializzate le distanze da un nodo iniziale a tutti gli altri nodi a "infinito", ad eccezione del nodo iniziale stesso, il cui valore di distanza è impostato a zero.

- **Esecuzione Principale:** l'algoritmo continua finché ci sono nodi da esplorare. In ogni iterazione:
  - viene estratto il nodo con la distanza minima dalla coda con priorità, diventando il nodo attuale;
  - si verifica se il nodo attuale è uguale al nodo finale; se sì, l'algoritmo è completato;
  - altrimenti, l'algoritmo esamina i nodi vicini al nodo attuale.

- **Aggiornamento dei Costi:** per ciascun nodo vicino, l'algoritmo calcola un punteggio F che tiene conto della distanza percorsa fino a quel momento e di una stima euristica del costo rimanente per raggiungere il nodo finale. Se il nuovo punteggio F è migliore (più basso) di quello precedentemente calcolato, vengono aggiornate le distanze e i punteggi F. Il nodo vicino viene quindi inserito nella coda con priorità.

- **Costruzione del Percorso Ottimo:** se il nodo finale viene raggiunto, l'algoritmo ricostruisce il percorso ottimo risalendo dai nodi finali ai nodi precedenti, utilizzando le informazioni sui predecessori memorizzate durante l'esecuzione.

4.3 Contraction Hierarchies

Le Contraction Hierarchies Time-Dependent (TD-CH) sono una variante delle Contraction Hierarchies (CH) che consentono di calcolare i percorsi più brevi nei grafi temporali, dove i costi degli archi possono variare nel tempo. Sono utilizzate per ottimizzare le query di percorsi ottimali tenendo conto delle condizioni temporali, come il traffico stradale variabile nel corso della giornata.

- **Preprocessing:** in questa fase iniziale, il grafo temporale viene preprocessato per creare una struttura gerarchica. Questo coinvolge la contrazione di alcuni nodi in modo da creare un grafo gerarchico in cui le informazioni temporali sono organizzate in livelli di gerarchia. I nodi "meno importanti" vengono contratti insieme per formare nodi più grandi e semplificare il grafo.

- **Query Time:** quando viene effettuata una query per trovare il percorso ottimo tra due punti nel grafo temporale, l'algoritmo viene attivato. Questo coinvolge una ricerca a due fasi: ricerca in avanti e ricerca all'indietro.
  -Ricerca in Avanti: Si inizia dal nodo di partenza e si esplorano i nodi gerarchici verso il nodo di destinazione, tenendo conto delle informazioni temporali.
  -Ricerca all'Indietro: Si inizia dal nodo di destinazione e si esplorano i nodi gerarchici verso il nodo di partenza, considerando le variazioni temporali.

- **Combining:** durante la fase di ricerca, l'algoritmo esegue la ricerca sia in avanti che all'indietro contemporaneamente. Durante questa fase, i risultati delle due ricerche vengono combinati per determinare il percorso ottimo completo, tenendo conto delle informazioni temporali

- **Restitution:** una volta ottenuto il percorso ottimo completo, l'algoritmo restituisce il percorso all'utente, fornendo il percorso e le informazioni temporali associate, come il tempo di arrivo previsto in base alle condizioni temporali considerate nel grafo. L'utente può quindi utilizzare queste informazioni per la navigazione ottimizzata nel tempo dipendente.

5. Applicazioni Pratiche al problema del Time Dependent Shortest Path

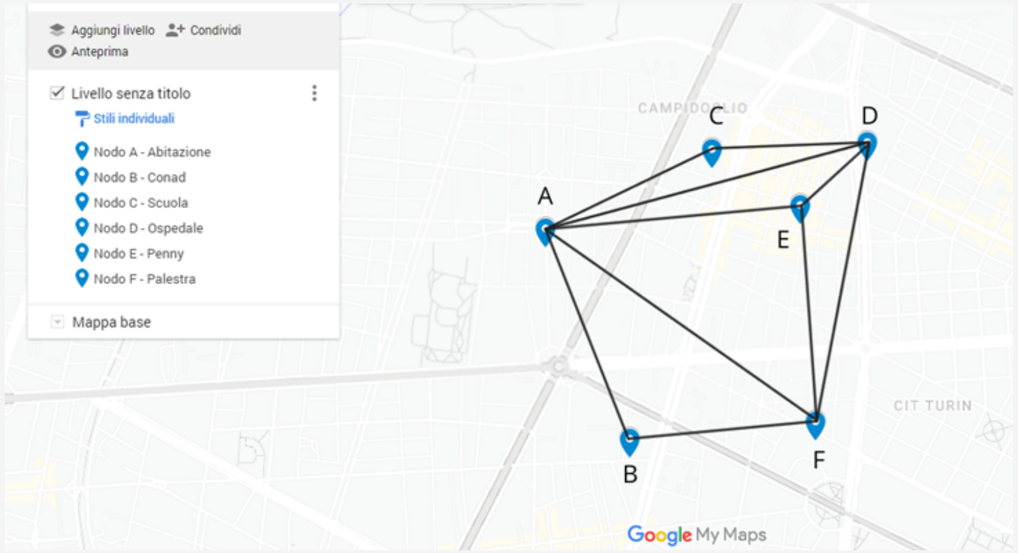
1. Rappresentazione del Grafo
2. Applicativo Python

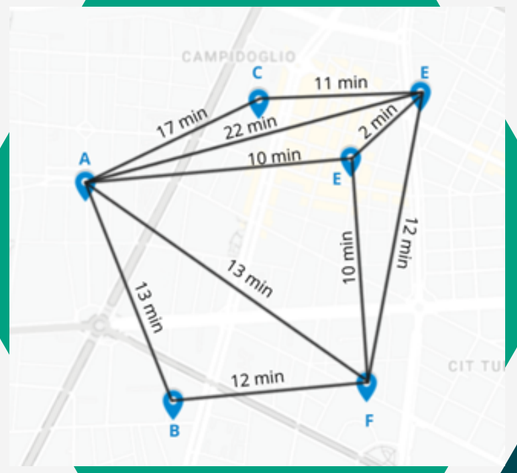
5.1 Rappresentazione del Grafo

Per testare gli algoritmi abbiamo intrapreso un approccio basato su dati reali, utilizzando la città di Torino (precisamente zona Campidoglio/Parella) come scenario di studio. Abbiamo creato un grafo di base utilizzando nodi che rappresentano punti di interesse reali nella città. Per ogni punto abbiamo acquisito dati di **distanza** e **tempistiche di percorrenza**.
Ecco i sei nodi principali che abbiamo considerato nel nostro studio, insieme alla loro corrispondente ubicazione:

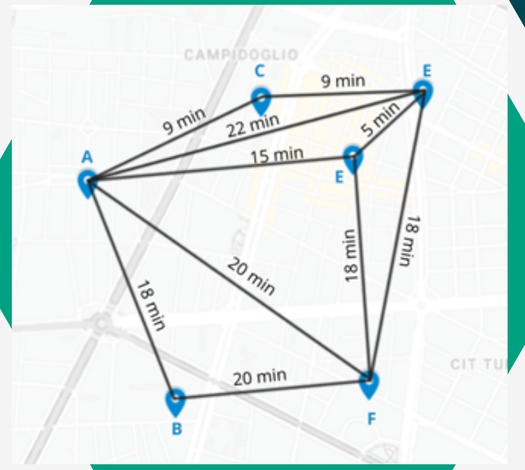
- Nodo A (Casa): Via Domodossola, 35
- Nodo B (Conad City): Via Bardonecchia 5/C
- Nodo C (Scuola Pubblica): Via Balme, 42
- Nodo D (Ospedale Maria Vittoria): Via Luigi Cibrario, 72
- Nodo E (Penny Market): Piazza Risorgimento, 20
- Nodo F (Palestra Freestyle SSD): Via Enrico Cialdini, 13

Versioni del Grafo

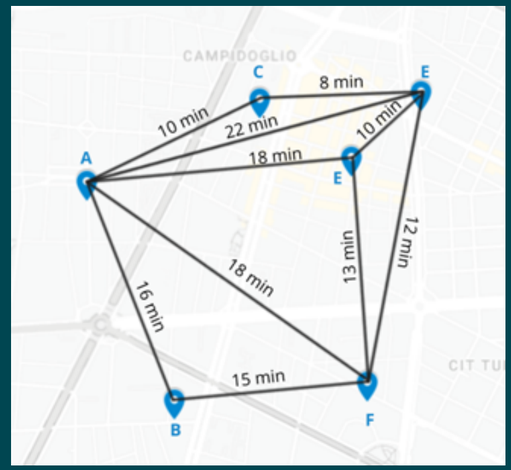




ore 08:00



ore 18:00



ore 12:00

KM/time	Nodo A	Nodo B	Nodo C	Nodo D	Nodo E	Nodo F
Nodo A	/	900m/13min	1km/13min	1,5km/22min	1km/10min	1,3km/18min
Nodo B	900m/13min	/	Non diretto	Non diretto	Non diretto	900m/12min
Nodo C	1km/13min	Non diretto	/	600m/9min	Non diretto	Non diretto
Nodo D	1,5km/22min	Non diretto	600m/9min	/	600m/9min	1,2km/12min
Nodo E	1km/10min	Non diretto	Non diretto	600m/9min	/	900m/13min
Nodo F	1,3km/18min	Non diretto	900m/12min	1,2km/12min	900m/13min	/

5.2 Applicativo Python

L'applicativo Python sviluppato è caratterizzato da tre Fasi di sviluppo
1. **Creazione della Struttura per il Grafo TDSP**

Abbiamo iniziato con la creazione di una struttura dati che rappresenta in modo accurato il grafo TDSP. Questa struttura tiene conto di variabili chiave come nodi, archi e dati temporali associati agli archi.
Ogni arco è associato a informazioni temporali che riflettono le variazioni nei tempi di percorrenza in diverse fasce orarie.

```
#  Implementazione del grafo

grafo = {
    'A': {
        'B': {
            '08:00': 13,
            '12:00': 16,
            '18:00': 18,
            },
        'C': {
            '08:00': 17,
            '12:00': 10,
            '18:00': 9,
            },
        'D': {
            '08:00': 22,
            '12:00': 22,
            '18:00': 22,
            },
        'E': {
            '08:00': 10,
            '12:00': 18,
            '18:00': 15,
            },
        'F': {
            '08:00': 13,
            '12:00': 18,
            '18:00': 20,
            }
        },
    'B': {
        'A': {
            '08:00': 13,
            '12:00': 16,
            '18:00': 18,
            },
        'F': {
            '08:00': 12,
```

Applicativo Python

L'applicativo Python sviluppato è caratterizzato da tre Fasi di sviluppo
2. **Interazione con l'Utente**

Abbiamo implementato una funzione interattiva che consente all'utente di specificare tre elementi fondamentali, ovvero il luogo (nodo) di partenza, il luogo (nodo) di destinazione, la fascia oraria in cui desidera pianificare il percorso.

```
# Interazione con l'utente

nodo_iniziale = input("Inserisci il nodo di partenza: ").strip().upper()
nodo_finale = input("Inserisci il nodo di destinazione: ").strip().upper()
orario_desiderato = input("Seleziona l'orario (8:00, 12:00 o 18:00): ").strip()
tempo_percorrenza, percorso = dijkstra_temporale(grafo, nodo_iniziale, nodo_finale, orario_desiderato)

if tempo_percorrenza is not None:
    print(f"Il tempo di percorrenza da {nodo_iniziale} a {nodo_finale} alle {orario_desiderato} è di {tempo_percorrenza} minuti.")
    print(f"Percorso: {' -> '.join(percorso)}")
else:
    print(f"Non è stato trovato un percorso da {nodo_iniziale} a {nodo_finale} alle {orario_desiderato}.")
```

Applicativo Python

L'applicativo Python sviluppato è caratterizzato da tre Fasi di sviluppo

3. **Implementazione degli Algoritmi A* e Dijkstra**

Abbiamo integrato gli algoritmi A* e Dijkstra modificato nell'applicativo per il calcolo dei percorsi minimi. Questi algoritmi sono stati adattati per considerare i dati temporali e calcolare i percorsi ottimali in base all'orario specificato dall'utente.

```python
#  Implementazione di Dijkstra

import heapq  # Per utilizzare il modulo heapq per l'heap binario

def dijkstra_temporale(grafo, nodo_iniziale, nodo_finale, orario_desiderato):
    # Inizializza le distanze con infinito per tutti i nodi tranne il nodo iniziale
    distanze = {nodo: float('inf') for nodo in grafo}
    distanze[nodo_iniziale] = 0

    # Crea un dizionario per tenere traccia dei predecessori per ciascun nodo
    predecessori = {}

    # Crea un heap binario iniziale con il nodo iniziale
    heap = [(0, nodo_iniziale)]

    while heap:
        # Estrai il nodo con la distanza minima dalla coda
        (distanza_attuale, nodo_attuale) = heapq.heappop(heap)

        # Se abbiamo raggiunto il nodo finale, costruisci il percorso e restituiscilo
        if nodo_attuale == nodo_finale:
            percorso = []
            while nodo_attuale is not None:
                percorso.insert(0, nodo_attuale)
                nodo_attuale = predecessori.get(nodo_attuale)
            return distanza_attuale, percorso

        # Se la distanza attuale è maggiore di quella già calcolata, passa al prossimo nodo
        if distanza_attuale > distanze[nodo_attuale]:
            continue

        # Itera sugli archi uscenti dal nodo attuale
        for vicino, pesi_temporali in grafo[nodo_attuale].items():
            tempo_percorrenza = pesi_temporali.get(orario_desiderato)
            if tempo_percorrenza is not None:
                distanza_alternativa = distanza_attuale + tempo_percorrenza
                # Se la distanza alternativa è più breve, aggiornala
                if distanza_alternativa < distanze[vicino]:
                    distanze[vicino] = distanza alternativa
```

```python
#  Implementazione di A*

coordinate_nodi = {
    'A': (0, 0),
    'B': (1, 2),
    'C': (3, 1),
    'D': (2, 3),
    'E': (4, 2),
    'F': (5, 0)
}

import heapq
import math

def distanza_euclidea(node1, node2):
    # Calcola la distanza euclidea tra due nodi.
    x1, y1 = coordinate_nodi[node1]
    x2, y2 = coordinate_nodi[node2]
    return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

def astar_temporale(grafo, nodo_iniziale, nodo_finale, orario_desiderato):
    # Inizializza le distanze con infinito per tutti i nodi tranne il nodo iniziale
    distanze = {nodo: float('inf') for nodo in grafo}
    distanze[nodo_iniziale] = 0
    # Inizializza una coda aperta con il nodo iniziale
    heap = [(0, nodo_iniziale)]
    # Crea un dizionario per tenere traccia dei predecessori per ciascun nodo
    predecessori = {}

    while heap:
        # Estrae il nodo con il punteggio F minimo dalla coda aperta
        _, nodo_attuale = heapq.heappop(heap)
        # Se abbiamo raggiunto il nodo finale, costruisce il percorso e lo restituisce
        if nodo_attuale == nodo_finale:
            percorso = []
            while nodo_attuale is not None:
                percorso.insert(0, nodo_attuale)
```

Risultati della Sperimentazione

Percorso	ore 08:00	ore 12:00	ore 18:00
A->D Casa -> Ospedale Maria Vittoria	A->E->D Casa -> Penny -> Ospedale Maria Vittoria	A->C->D Casa -> Scuola-> Ospedale Maria Vittoria	A->C->D Casa -> Scuola-> Ospedale Maria Vittoria
B->D Conad -> Ospedale Maria Vittoria	B->F->D Conad -> Palestra Freestyle SSD-> Ospedale Maria Vittoria	B->F->D Conad -> Palestra Freestyle SSD-> Ospedale Maria Vittoria	B->A->C->D Conad -> Casa -> Scuola-> Ospedale Maria Vittoria
E->B Conad -> Penny	E->F->B Conad -> Palestra Freestyle SSD -> Penny	E->F->B Conad -> Palestra Freestyle SSD-> Penny	E->A->B Conad -> Casa ->Penny
C->F Scuola -> Palestra Freestyle SSD	C->D->F Scuola -> Ospedale Maria Vittoria -> Palestra Freestyle SSD	C->D->F Scuola -> Ospedale Maria Vittoria -> Palestra Freestyle SSD	C->D->F Scuola -> Ospedale Maria Vittoria -> Palestra Freestyle SSD

Conclusione

Questo studio ci ha permesso di concentrarci su problemi che vengono affrontati quotidianamente:

- le condizioni metereologiche imprevedibili
- l'aumento dei mezzi di trasporto
- La crescente aspettativa di servizi efficienti e tempestivi

Tali situazioni di caos e incertezza possono avere un impatto significativo sul benessere psicologico dei cittadini. L'ansia, lo stress e la frustrazione possono emergere quando le persone si sentono impotenti nel gestire il proprio tempo o quando hanno la sensazione che il tempo scorra troppo rapidamente.
Quando le persone perdono la cognizione del tempo, possono sentirsi intrappolate in un ciclo frenetico di attività senza riuscire a gestire efficacemente i propri impegni.

I risultati dimostrano chiaramente la capacità dell’applicativo di adattare i percorsi in base all’orario specificato dall’utente. Le variazioni temporali vengono gestite in modo efficace, consentendo di identificare i percorsi ottimali in diverse fasce orarie.

Grazie per l’attenzione!