Progetto per appello 11/02/2025

Metodi Formali per la Sicurezza A.A 2024/2025

Corso di Laurea Magistrale in Sicurezza Informatica

Università degli Studi di Bari "Aldo Moro"

Studente: *Cellammare Gabriel*

Attività

Si consideri il caso di un drone per la consegna di merci.
Risolvere i seguenti problemi:

a) Adattare la codifica ASP del problema del commesso viaggiatore per poter trovare il percorso migliore di consegna delle merci. Scegliere una opportuna funzione di costo (ad es., distanza in tempo o in spazio fra i punti di consegna, oppure il consumo di batteria, etc) rispetto alla quale fare ottimizzazione.

b) Modellare in LTL/CTL le varie manovre del drone: decollo, atterraggio, hovering (stazionamento su una posizione per, ad es., attendere di poter atterrare o consegnare il pacco), consegna. Si richiede di considerare almeno un caso di non soddisfacimento delle proprietà implementate.

Il problema del commesso viaggiatore

Il problema del **commesso viaggiatore** (*Traveling Salesman Problem*, **TSP**) rappresenta uno dei più **celebri** e studiati problemi di **ottimizzazione combinatoria**[1] nella teoria della complessità **computazionale** e nella ricerca **operativa**. Tale problema fu formalizzato nel 1800 dal matematico irlandese *W. R. Hamilton* e dal matematico britannico *Thomas Kirkman*. **Hamilton rese celebre tale problema attraverso l'Icosian Game,** un puzzle basato sulla ricerca di un ciclo **hamiltoniano**, in cui l'obiettivo del gioco è trovare un percorso attorno a un **dodecaedro** tale che ogni **vertice** sia visitato una volta, nessuno **spigolo** sia visitato più volte e il percorso termini nello stesso **vertice** da cui è partito [1].

Tale problema venne poi affrontato negli anni '30 del **XX secolo**, quando **Karl Menger** – *noto matematico austriaco* - lo presentò a **Vienna**, descrivendolo come il problema di trovare il percorso più **economico** che un venditore dovesse seguire per visitare una **serie** di **città**, tornando infine al punto di **partenza**, minimizzando i **costi** di viaggio tra le città [2]. La natura del problema si presta ad una rappresentazione mediante **grafi**, dove le città sono rappresentate da **vertici** e i collegamenti tra esse da **spigoli** pesati, dove il peso

[1] **L'Ottimizzazione Combinatoria** (*OC*) studia i problemi di **ottimizzazione** in cui l'insieme ammissibile è definito in termini di strutture **combinatorie**, tra le quali svolgono sicuramente un ruolo di rilievo i **grafi.**

rappresenta il **costo** o la **distanza** del viaggio. L'obiettivo diventa quindi quello di determinare un **ciclo hamiltoniano** di costo **minimo** all'interno del **grafo**, ovvero un percorso che visiti ogni **vertice** esattamente **una volta**, ritorni al **punto di partenza** e minimizzi la somma dei pesi degli **spigoli** utilizzati.

Digressione: Fondamenti di Teoria dei Grafi

Un grafo [3] G è definito **formalmente** come una coppia **ordinata** $G = (V, E)$, dove:

1) **Vertici (o nodi)**

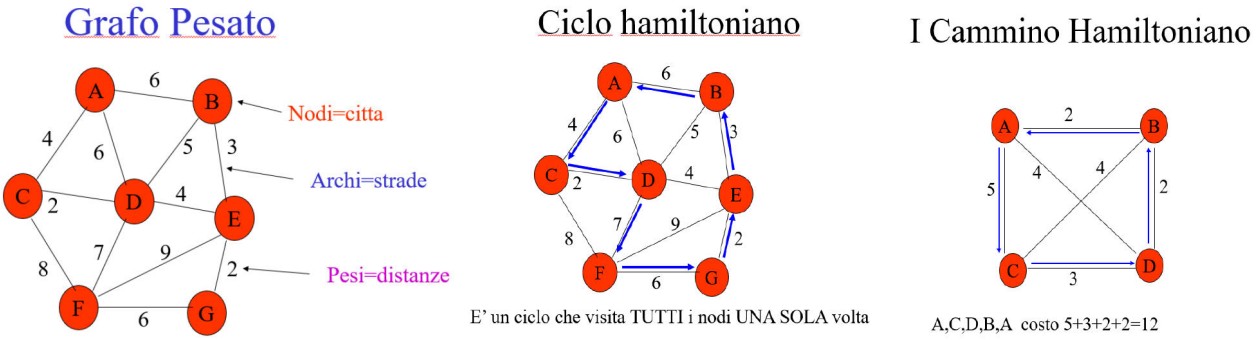
   I vertici, rappresentati dall'insieme V sono gli elementi fondamentali del **grafo**. Un vertice $v \in V$ è un punto che può rappresentare qualsiasi tipologia di **entità**.

2) **Spigoli (o archi)**

   Gli **spigoli**, rappresentati dall'insieme $E \subseteq V \times V$, sono le connessioni tra i **vertici**. Uno spigolo $e \in E$ è una coppia ordinata (u, v) dove $u, v \in V$. È possibile classificare gli **spigoli** in:

   - **Non orientati**: la coppia (u, v) è equivalente alla coppia (v, u)
   - **Orientati**: la coppia (u, v) definisce una direzione da u a v
   - **Pesati**: ad ogni spigolo è associato un valore numerico (*peso*).
     Questo peso può rappresentare vari aspetti come la distanza tra nodi, il costo di attraversamento o altre metriche.

Un **percorso hamiltoniano** in un grafo è un percorso che visita ogni vertice esattamente una volta. Se il percorso ritorna al vertice iniziale, formando un **ciclo**, viene **chiamato ciclo hamiltoniano**. Definito quindi V l'insieme delle città ed E l'insieme degli archi con i relativi **pesi** (*le distanze*), l'obiettivo è determinare il **ciclo hamiltoniano** che minimizzi i **costi** e che permetta al commesso viaggiatore di partire dalla **città *base*** - *il vertice iniziale* - e di farvi ritorno soltanto dopo aver visitato **ogni città** esattamente una **volta** [4].



La sfida legata al TSP

Il problema del TSP è **NP-hard**[2], cioè non esiste un algoritmo noto che possa risolverlo in modo efficiente per tutti gli **input** poiché il numero di soluzioni aumenta all'aumentare del numero di città. La soluzione più semplice (**e più costosa**) è quella di provare semplicemente tutte le **possibilità**. Il problema teorico è che presupponendo di avere **10**

[2] Un problema si definisce **NP-difficile** (NP-hard) se è almeno tanto complesso quanto i problemi più difficili **della classe NP (Non-deterministic Polynomial time).** Nel caso del **TSP Teorico**, questo si traduce in una complessità computazionale fattoriale O(n!), dove n è il numero di città da visitare.

città, ci sono $(10-1!=362.880)$ possibilità, con una precisazione: è necessario considerare solo la **metà**, poiché ogni percorso ha un percorso uguale al **contrario** con la stessa **lunghezza** o lo stesso **costo**, ovvero $\frac{(10-1)!}{2} = 181\,440$. Le applicazioni del **TSP** sono molteplici e spaziano in diversi **settori**. Ad esempio, è utilizzato nella produzione di microchip, dove è necessario ottimizzare il **percorso** di un **laser** per incidere **circuiti**, per la pianificazione dei percorsi per **veicoli**, protocolli di **routing** e **DNA sequencing** [5].

Soluzioni al TSP

Le soluzioni proposte per il **TSP** si sono evolute nel **tempo**, riflettendo i progressi nella teoria **dell'ottimizzazione** e nelle capacità **computazionali**. Gli approcci risolutivi si dividono principalmente in due categorie: 1) gli *algoritmi esatti*, che garantiscono l'esattezza della **soluzione** ma sono **computazionalmente onerosi**, 2) gli *algoritmi euristici*, che forniscono soluzioni approssimate in tempi **ragionevoli**. Gli algoritmi esatti per il **TSP** si dividono in tre principali approcci metodologici distinti. Il primo è il metodo **Branch and Bound**, che affronta il problema suddividendo **ricorsivamente** lo spazio delle **soluzioni** in sottoproblemi più gestibili ed eliminando i rami che non **possono** portare a soluzioni **ottimali** [6]. Il secondo approccio è il metodo del **Cutting Plane**, sviluppato da *Dantzig, Fulkerson e Johnson nel 1954*, che rappresenta una metodologia completamente diversa: utilizza la programmazione lineare intera, aggiungendo progressivamente **vincoli** (i "tagli") per guidare la soluzione verso valori **interi** [7]. La terza **metodologia** riguarda l'algoritmo di **Held-Karp**, che si basa sulla programmazione **dinamica**. Questo algoritmo memorizza le **soluzioni ottimali** di **sottoproblemi** via via più grandi per costruire la soluzione finale, con una complessità temporale di $O(n^2 2^n)$, e rappresenta uno dei risultati più significativi nell'ambito degli algoritmi **esatti** per il **TSP** [8]. D'altra parte, gli approcci euristici hanno guadagnato particolare rilevanza per la loro **applicabilità** pratica. Le tecniche di ricerca locale, come la ricerca variabile di vicinato (**VNS**) e **l'annealing simulato adattivo**, permettono di esplorare efficacemente lo spazio delle soluzioni [9].

Definizione del contesto

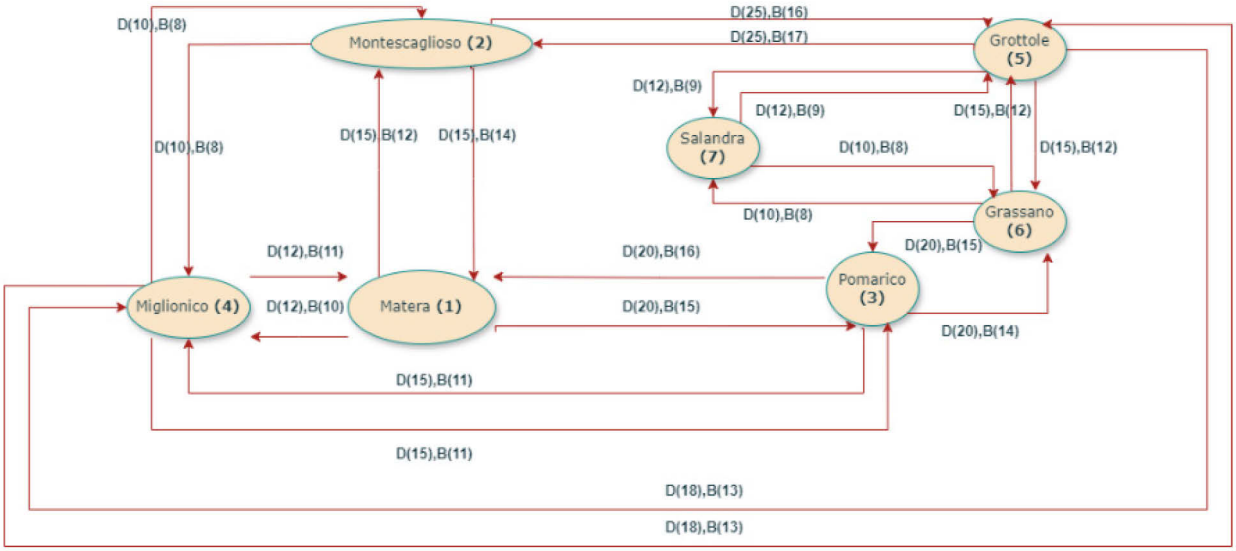
Per una corretta modellazione del problema, è necessario definire il contesto: si ipotizzi che i vertici del grafo corrispondano ad alcune città della provincia di Matera nella **Tabella 1**. Il grafo che rappresenta questi collegamenti non è **completamente**[3] **connesso**[4], il che significa che da ogni **comune** (*vertice*) il drone non può raggiungere direttamente tutti gli altri comuni. Questa aspetto legato alla **connettività** non è **casuale**, ma rispecchia precise considerazioni operative legate alla **morfologia** del territorio, all'efficienza **energetica** del drone e ai vincoli di **navigazione** [10]. Difatti la realtà solitamente si allontana dalla formulazione teorica del **TSP**, a causa di **aspetti** legati **all'ottimizzazione** della **navigazione** e della gestione del **drone**. Nonostante questa connettività parziale, il problema mantiene la sua natura di **ricerca** di un **ciclo hamiltoniano ottimale** minimizzando al contempo i costi complessivi del **viaggio**, considerando distanza percorsa e batteria **utilizzata**. È bene inoltre notare che le **distanze** considerate non riflettono la reale situazione ma bensì astraggono quelli che sono aspetti più **complessi** e non adatti al

[3] Per ***Non completamente connesso*** *si intende che esiste un cammino tra ogni coppia di nodi, ma alcuni nodi non sono direttamente collegati tra loro*.
[4] In teoria dei **grafi**, un grafo G è detto **connesso** se, per ogni coppia di vertici $u, v \in V$, esiste un cammino che collega u a v.

contesto specifico; tale modellazione, inoltre, si basa sul presupposto secondo cui il drone è in grado a livello energetico, di affrontare qualsiasi **ciclo hamiltoniano**. Le distanze ipotizzate variano dai **10 ai 25 chilometri**, mentre il consumo della batteria varia **dall'8% al 17% per tratta**. Per i ritorni verso il **vertice** base, è stato applicato un incremento del consumo di circa l'**1-2%** per compensare possibili situazioni di **vento contrario** dati **dall'altitudine** e dalla conformazione territoriale. Questo approccio permette al sistema di trovare il compromesso **ideale** tra la minimizzazione della distanza percorsa e la conservazione della **batteria**, garantendo che il drone possa completare il suo percorso in sicurezza e con un adeguato margine di batteria residua.

Partenza (Nodo)	Arrivo (Nodo)	Distanza (km)	Consumo Batteria (%)	Consumo al Ritorno (%)
Matera **(1)**	Montescaglioso **(2)**	15	12	14
Matera **(1)**	Pomarico **(3)**	20	15	16
Matera **(1)**	Miglionico **(4)**	12	10	11
Montescaglioso **(2)**	Miglionico **(4)**	10	8	8
Montescaglioso **(2)**	Grottole **(5)**	25	16	17
Pomarico **(3)**	Miglionico **(4)**	15	11	11
Pomarico **(3)**	Grassano **(6)**	20	14	15
Miglionico **(4)**	Grottole **(5)**	18	13	13
Grottole **(5)**	Grassano **(6)**	15	12	12
Grottole **(5)**	Salandra **(7)**	12	9	9
Grassano **(6)**	Salandra **(7)**	10	8	8

Tabella 1: Le rotte navigabili dal drone



Codifica ASP del TSP

Tale problema è stato codificato utilizzando il sistema **Potassco** [11] (*Potsdam Answer Set Solving Collection*), che implementa **ASP** (*Answer Set Programming*). La scelta di **ASP** e **Potassco** è particolarmente adatta per questo tipo di problema **combinatorio** perché permette di esprimere in modo **naturale** e **dichiarativo** sia i vincoli del ciclo hamiltoniano che gli obiettivi di **ottimizzazione**, sfruttando la potenza espressiva di **ASP** e l'efficienza computazionale del solver **clingo** di Potassco; **ASP** non specifica "***come***" trovare la

soluzione, ma piuttosto "**cosa**" costituisce una **soluzione** valida. Nella codifica ASP sono presenti gli **edge**, ovvero tutti i **collegamenti** possibili tra le città, definiti come **fatti**[5].

```
% Definizione degli archi (collegamenti tra città)
edge(1,(2;3;4)). % Collegamenti da Matera
edge(2,(1;4;5)). % Collegamenti da Montescaglioso
```

```
% Distanze in km
distance(1,2,15). distance(2,1,15).
distance(1,3,20). distance(3,1,20).
```

```
% Costi batteria in percentuale
battery_cost(1,2,12). battery_cost(2,1,14).
battery_cost(1,3,15). battery_cost(3,1,16).
```

Cycle invece, rappresenta il percorso **effettivo** che verrà costruito ed è un sottoinsieme di **edge**, che formerà il **ciclo hamiltoniano** finale.

```
% Per ogni nodo, deve essere selezionato esattamente un arco uscente
{ cycle(X,Y) : edge(X,Y) } = 1 :- node(X).
```

Il programma inizia definendo queste regole, di cui la prima stabilisce che per ogni **nodo** X del **grafo** $(node(X))$, deve essere scelto esattamente un **arco** uscente. La notazione $\{\, cycle(X,Y) \,:\, edge(X,Y)\, \}$ è una **choice rule** che genera tutti i possibili archi uscenti da X**,** e = 1 impone che ne venga scelto esattamente **uno**. Il simbolo $:-$ di **implicazione** impone che tutto ciò che sta a destra di questo simbolo debba essere **vero** affinché la regola si **applichi**. La regola esprima quindi che "*Per ogni città **X**, è necessario scegliere esattamente una strada che parte da **X** verso un'altra città **Y**, ma è possibile scegliere solo tra le strade che esistono realmente*", per esempio:

$$\{\, cycle(1,2);\ cycle(1,3);\ cycle(1,4)\, \} \ = \ 1 : -\, node(1).$$

```
% Per ogni nodo, deve essere selezionato esattamente un arco entrante
{ cycle(X,Y) : edge(X,Y) } = 1 :- node(Y).
```

Questa seconda regola è **complementare** alla prima e stabilisce che per ogni nodo Y del grafo, deve essere scelto esattamente un arco **entrante**. **Ogni città Y deve essere raggiunta da esattamente una città X**: ogni città deve avere esattamente una strada in entrata e una in uscita.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% REGOLE DI RAGGIUNGIBILITÀ
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Un nodo è raggiungibile se è direttamente collegato al nodo 1
reached(Y) :- cycle(1,Y).
% Un nodo è raggiungibile se è collegato a un nodo già raggiungibile
reached(Y) :- cycle(X,Y), reached(X).
% Non sono ammesse soluzioni con nodi non raggiungibili
:- node(Y), not reached(Y).
```

Per ciò che concerne le regole di **raggiungibilità**, la prima afferma che **un nodo** Y è **raggiungibile** se esiste un **arco del ciclo** che parte dal nodo **1** e arriva direttamente **a** Y. Nel caso in oggetto, questa regola da sola marca come raggiungibili solo i vicini diretti di **Matera**. La seconda è una regola ricorsiva che estende il concetto di raggiungibilità. Afferma che un nodo Y è raggiungibile se:

1. Esiste un **arco** del ciclo da un qualche nodo X a Y $(cycle(X,Y))$

[5] In **ASP** una **regola senza corpo** $a_0 \leftarrow (n = 0)$ viene chiamato **fatto** e rappresenta un'informazione sempre vera.

2. E quel nodo X è già stato marcato come **raggiungibile** $(reached(X))$

La terza non è una regola, ma bensì una **constraint** (*vincolo*) che elimina le soluzioni indesiderate. Esprime un **vincolo** in cui non è ammessa una soluzione dove esiste un nodo Y che non è stato marcato come **raggiungibile**. Immaginando un **percorso**:

$$1 -> 2 -> 4 -> 5 -> 3 -> 6 -> 7 -> 1$$

Le regole lavorerebbero così:

1. *Prima regola: marca 2 come raggiungibile perché c'è cycle(1,2)*
2. *Seconda regola (primo passo): marca 4 come raggiungibile perché c'è cycle(2,4) e 2 è raggiungibile*
3. *Seconda regola: continua a propagare la raggiungibilità lungo il ciclo*
4. *Terza regola: verifica che alla fine tutti i nodi siano stati marcati come raggiungibili*

Se anche un solo nodo non **fosse raggiungibile**, la terza regola eliminerebbe quella soluzione dal **set** di risposte possibili.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% VISUALIZZAZIONE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#show cycle/2.
#show total_distance/1.
#show total_battery/1.
```

Questa porzione di codice stampa i risultati utilizzando dei numeri per identificare gli argomenti.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% OTTIMIZZAZIONE
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Minimizza il costo totale del percorso
#minimize { C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```

minimize è uno strumento fondamentale in ASP che indica al **solver** di trovare soluzioni che minimizzano una certa **quantità**, in questo caso la somma dei valori specificati C (*distanza e batteria*). X, Y sono le variabili coinvolte nell'ottimizzazione. $cost(X, Y, C)$ e $cycle\ (X, Y)$ sono le condizioni che devono essere vere. La **direttiva** $\#minimize\{\ C, X, Y : cycle(X, Y), cost(X, Y, C)\ \}$. prende questi costi per tutti gli archi selezionati nel ciclo e cerca di **minimizzarne** la somma, trovando così il ciclo hamiltoniano con il costo totale più **basso**.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% CALCOLO COSTI E TOTALI
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calcolo del costo totale come somma di distanza e consumo batteria
cost(X,Y,C) :- distance(X,Y,D), battery_cost(X,Y,B), C = D + B.

% Calcolo della distanza totale
total_distance(TD) :- TD = #sum { D,X,Y : cycle(X,Y), distance(X,Y,D) }.

% Calcolo del consumo totale di batteria
total_battery(TB) :- TB = #sum { B,X,Y : cycle(X,Y), battery_cost(X,Y,B) }.

% Verifica completezza del ciclo
cycle_complete :- N = #count { X : node(X) },
                  N = #count { X : reached(X) }.
:- not cycle_complete.
```

In questa porzione di codice, la **prima** regola definisce come calcolare il costo totale per andare da un nodo X a un nodo Y. Il costo C è la **somma** della **distanza** D e del consumo

di **batteria** B. Questa regola usa l'aggregato[6] $\#sum$ per calcolare la distanza totale del percorso. Nella **seconda** regola, per ogni **arco** che fa parte del **ciclo** $(cycle(X,Y))$, prende la sua distanza $(distance(X,Y,D))$ e **somma** tutti questi valori. La **terza** regola è simile alla regola precedente, ma per il **consumo** di batteria. Difatti, $C = D + B$ non è un **atomo** ma è un'operazione **aritmetica** che viene utilizzata all'interno di una **regola** per assegnare un valore a C. L'ultima **regola** e l'ultimo **vincolo** invece, verificano che il ciclo sia **completo**, ovvero che tutti i **nodi** siano stati raggiunti nel percorso **trovato**. La **regola** confronta il numero totale di nodi N con il numero di **nodi** raggiunti $\#count\{X : reached(X)\}$. Se coincidono, significa che ogni nodo è stato visitato e il **ciclo hamiltoniano** è valido. Il **vincolo** esclude tutte le soluzioni in cui **cycle_complete non è vero**, ovvero quelle in cui esistono **nodi non raggiunti**.

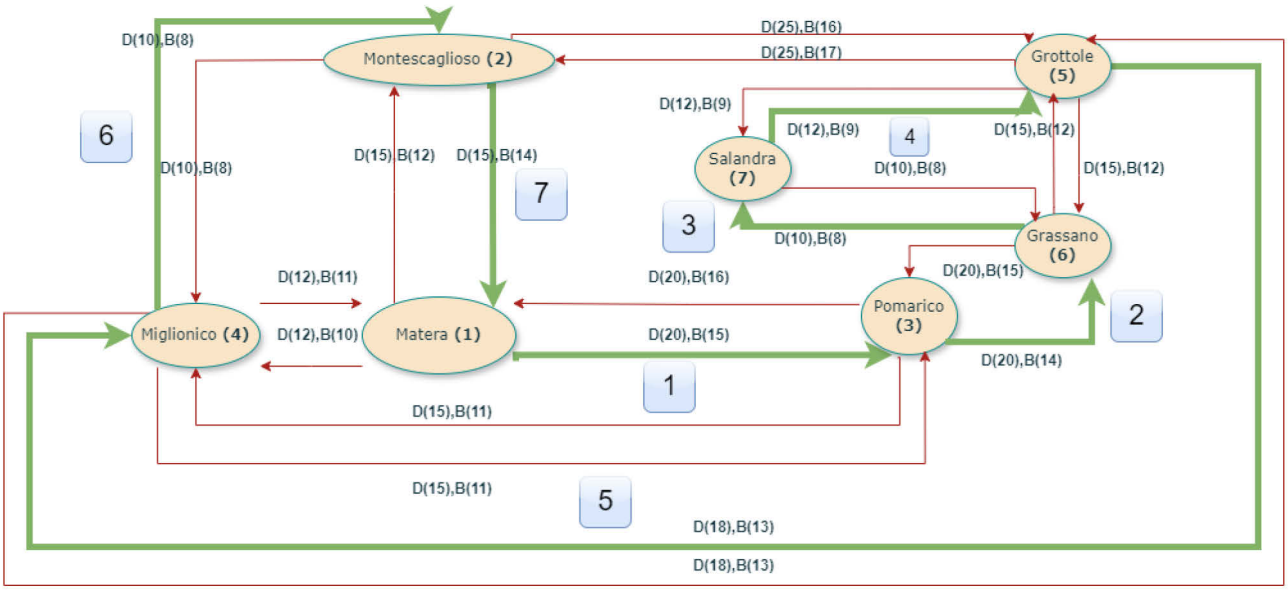
Risultati della codifica ASP del TSP: Ciclo Hamiltoniano ottimizzato

```
clingo version 5.7.0
Reading from stdin
Solving...
Answer: 1
cycle(1,4) cycle(4,3) cycle(3,6) cycle(2,1) cycle(6,7) cycle(5,2) cycle(7,5) total_battery(83) total_distance(109)
Optimization: 192
Answer: 2
cycle(1,3) cycle(4,1) cycle(3,6) cycle(2,4) cycle(6,7) cycle(5,2) cycle(7,5) total_battery(82) total_distance(109)
Optimization: 191
Answer: 3
cycle(1,3) cycle(4,2) cycle(3,6) cycle(2,1) cycle(6,7) cycle(5,4) cycle(7,5) total_battery(81) total_distance(105)
Optimization: 186
OPTIMUM FOUND
```

Tra le tre soluzioni proposte, Il **ciclo hamiltoniano ottimale** trovato è: $Matera(\mathbf{1}) \rightarrow Pomarico(\mathbf{3}) \rightarrow Grassano(\mathbf{6}) \rightarrow Salandra(\mathbf{7}) \rightarrow Grottole(\mathbf{5}) \rightarrow Miglionico(\mathbf{4}) \rightarrow Montescaglioso(\mathbf{2}) \rightarrow Matera(\mathbf{1})$.
Questo ciclo ha un costo totale di **186 unità**, che è la somma di:

$$\boldsymbol{C = Distanza\ totale + Consumo\ totale\ batteria}$$

$$C = 105 + 81 = 186$$



[6] Le funzioni di aggregazione in **ASP** sono funzioni **speciali** che permettono di calcolare un **valore unico** a partire da un **insieme** di valori. La loro sintassi generale è molto **specifica** e **strutturata**:
$Nome\_Predicato(Risultato) :- Risultato = \#funzione\_aggregazione\{Termini : Condizioni\}$.

Modellazione delle manovre di un drone per la consegna di merci

La modellazione **formale** del comportamento dei **droni** rappresenta un campo di ricerca di particolare rilevanza nell'ambito **dell'ingegneria** dei sistemi **cyber-fisici**. La formulazione **matematica** di questi sistemi si inserisce nel più ampio contesto dei **metodi formali**, dove l'obiettivo principale consiste nella **verifica rigorosa** delle proprietà di **sicurezza** che garantiscano il **rispetto** di specifiche **temporali**. Il paradigma fondamentale si basa sulla **rappresentazione** del drone come **sistema** attraverso una macchina a **stati finiti,** dove le transizioni tra stati operativi devono soddisfare precise **proprietà temporali** espresse mediante **LTL e CTL**. Questa formalizzazione permette di catturare le operazioni del drone, dalla fase di **decollo** fino **all'atterraggio o alla consegna** [12]. Durante la definizione delle manovre classiche, è stata tenuta in considerazione anche **l'altitudine**, necessaria per determinare il corretto **atterraggio** del drone o la **consegna** della merce. Le attuali sperimentazioni di **consegna di merci** attraverso **droni** considerano diverse opzioni in base alla tipologia di **merce**, alla **tecnologia** del **drone** utilizzato e alle **specifiche** fornite dall'azienda di **consegna**: alcuni adottano una strategia di consegna ***in aria***, in una situazione di ***hovering***[7], mentre altri per prediligere **sicurezza**, preferiscono un atterraggio **sicuro**. Nel modello formalizzato la consegna della merce avverrà dopo che il **drone** avrà correttamente effettuato **l'hovering** preceduto da un corretto posizionamento **all'altitudine target**. La scelta di focalizzarsi principalmente **sull'altitudine nella modellazione tralasciando le coordinate del piano orizzontale** permette di concentrarsi sugli aspetti più **critici** del sistema, riducendo la **complessità** del modello senza perdere le proprietà essenziali da **verificare**. Tutte le operazioni fondamentali del **drone** (*decollo, atterraggio, hovering, consegna*) dipendono primariamente dalla corretta gestione **dell'altitudine**. Quando parliamo di **altitudine**, ci riferiamo specificamente all'altezza del **drone** rispetto a un punto di riferimento, in questo caso il **suolo**. Tuttavia, è importante comprendere che esistono diversi tipi di **altitudine** [13]:

1) **Altitudine Assoluta** (*MSL - Mean Sea Level*):

   È l'altezza del drone rispetto al livello medio del mare. Questa misura è importante per la navigazione aerea generale, ma non è la più rilevante per le operazioni di un drone;

2) **Altitudine Relativa** (*AGL - Above Ground Level*):

   È l'altezza del drone rispetto al terreno direttamente sotto di esso: difatti, tale misura risulta più pertinente per le operazioni considerate, perché è fondamentale per le manovre di decollo e atterraggio ed è cruciale per mantenere una distanza sicura dagli ostacoli.

Un'altezza troppo elevata durante l'atterraggio – seppur *preceduta dallo stato di hovering* - comporterebbe una perdita di **stabilità**, influenzata anche dalla presenza **maggiore** di agenti **atmosferici** rispetto a quote più **basse**. Allo stesso modo, il **rilascio** della **merce** a

[7] **Hovering** è una particolare tipologia di volo che si verifica quando il drone staziona in aria a velocità nulla e quota costante. In questo caso si parla, appunto, di **volo stazionario** o **volo puntiforme**, in quanto la rotta del drone nello spazio indica un semplice **punto**. Tale manovra è utilizzata per stabilizzare il drone e **prepararlo** a manovre successive.

partire da una **quota elevata**, **comporterebbe** danni concreti al prodotto. In un **atterraggio** inizializzato invece ad **altitudini** troppo **basse**, il drone potrebbe non stabilizzarsi correttamente a causa del flusso d'aria generato dalle eliche che può creare una **zona di turbolenza instabile**. Nell'atterraggio, il drone deve gestire la **transizione** da una condizione di volo a una di **hovering** vicino al **suolo**, richiedendo aggiustamenti più **delicati**. Il fenomeno del flusso d'aria influenza anche la fase di **decollo**, ma in questo caso una bassa **altitudine** non costituisce **pericolo** poiché il drone parte da una posizione **stabile** (al suolo) e aumenta gradualmente la **potenza**, permettendo un controllo più preciso della **transizione** anche grazie al successivo hovering.

Alcune sperimentazioni in **Italia** [14] confermano il modello del **rilascio della merce** "*in aria*" e suggeriscono **un'altitudine** di crociera che varia dai **55** ai **115 metri** dal punto più basso del suolo, rispettando quindi il limite imposto dall'Italia di **150 metri** [15]. In tale modellazione, la fase di **hovering**, in cui il drone si stabilizzerà per **rilasciare** la merce o effettuare **l'atterraggio**, inizierà a circa **4-5** metri sopra il livello del **suolo**, il che consente un tempo **sufficiente** per adattarsi a eventuali ostacoli **imprevisti** o cambiamenti nelle condizioni del **vento** durante la discesa. A questa altitudine, i droni possono utilizzare **sensori** di bordo per valutare la sicurezza e la **stabilità** dell'area di rilascio, il che è fondamentale per le operazioni di **consegna** di merce in ambienti urbani. La ricerca ha dimostrato inoltre che i **droni** operanti a quote più basse sperimentano meno **complicazioni** legate al **vento** e possono mantenere un controllo **migliore**, riducendo così il rischio di incidenti durante **l'atterraggio** [16].

Definizione del contesto

In base alla precedente **analisi**, è necessario definire un **contesto** di riferimento per una corretta **modellazione**. È bene sottolineare che tale **modellazione** non prende in considerazione nessun **percorso** specifico da seguire – *a differenza del punto* a*) dell'attività* – ma **astrae** il comportamento del drone in un contesto più **ampio** e **generale**, dove il drone seguirà un percorso lineare – *Partenza dalla base, Consegna 1, Consegna n, Ritorno alla base* - in cui potrà variare la sua **altitudine**, non considerando in questo caso aspetti legati al consumo energetico e di ricarica. Durante il **volo** – *in seguito definito come navigazione verso punti di rilascio* – il drone potrà raggiungere diverse **altitudini** rispetto a quella **target** a causa della presenza di ostacoli di natura antropica e naturale, conformazione **morfologiche** del territorio e altri **fattori** non considerati appositamente dato il livello di **astrazione** di riferimento. Inoltre, l'hovering è **essenziale** per il controllo della traiettoria e della stabilità [17], per questo motivo è necessario che avvenga in determinati **istanti** di tempo, ovvero:

1) Dopo il **decollo**, per preparare correttamente il drone alla normale **crociera**
2) Prima **dell'atterraggio**, per finalizzare le manovre necessarie
3) Prima del rilascio della **merce** da **consegnare** per un corretto assestamento

Difatti durante la **crociera**, i droni sono **progettati** per volare in modo più efficiente, riducendo il consumo **energetico** rispetto alla modalità di **hovering**. Ad esempio, un design ibrido di droni riduce il consumo di energia del **64%** durante il volo in crociera rispetto all'hovering, dimostrando **l'efficienza** del volo in **crociera** per le **consegne** e specificando come **l'hovering** sia necessario soltanto in situazioni **critiche** [18]. Inoltre, il **rilascio** della merce data l'analisi precedente, dovrà avvenire ad **un'altitudine** target per evitare di **destabilizzare** il drone durante il rilascio o danneggiare la **merce** stessa. Per tale motivo,

l'altitudine di **rilascio merce** e quella di **hovering** – *da ora specificata come* ***altitudine target*** – corrisponderà, poiché il drone dovrà mantenere una posizione **stabile** e **controllata** durante l'intera operazione di **rilascio**: cambiare altitudine tra la fase di hovering e quella di **rilascio** introdurrebbe instabilità non **necessarie** nel sistema, compromettendo la precisione dell'operazione. Complessivamente, richiederebbe una nuova fase di **valutazione** e **stabilizzazione**, aumentando inutilmente i tempi operativi e il consumo **energetico**. Considerando che l'hovering consuma **significativamente** più energia rispetto al volo di crociera è fondamentale **minimizzare** il tempo speso in questa fase mantenendo la stessa altitudine target per entrambe le **operazioni** [18]. Il drone quindi si occuperà del **rilascio** della merce attraverso un corretto **svolgimento** di tutte quante le **manovre** necessarie - *precedute per quelle più critiche dalla fase di hovering* - e sarà in grado di variare **altitudine** in maniera **graduale** durante la normale crociera attraverso un numero prefissato di **punti di consegna** – *non specializzati in tale modellazione* - per adattarsi a qualsiasi **situazione** attraverso un percorso **linearmente** definito e illustrato in **Figura 1, che terminerà con il ritorno alla base di partenza.**

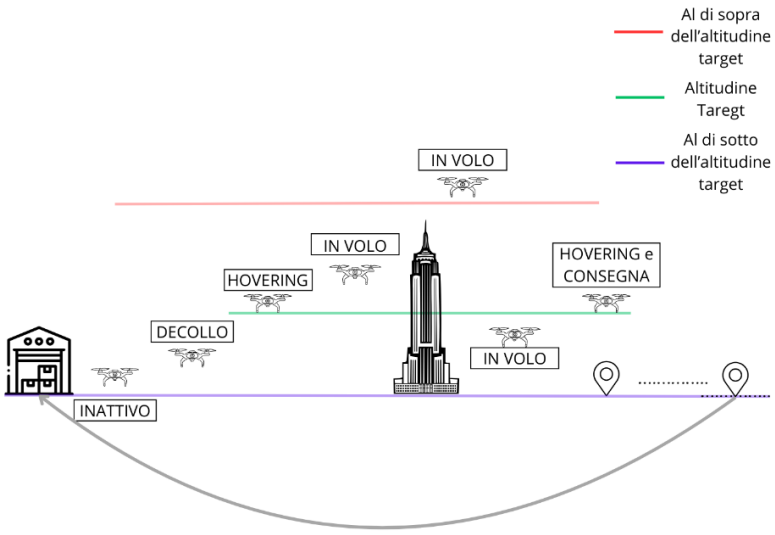


Figura 1: *Consegna della merce*

Dato il principio di **astrazione**, non sono stati considerati casi specifici di atterraggi **d'emergenza,** seppure la modellazione - *definita in seguito* - permetta al drone di atterrare in qualsiasi momento. Per lo stesso principio, le caratteristiche del **drone, le situazioni in cui non è possibile rilasciare la merce**, così come valori numerici **specifici** dell'altitudine o di percorsi in zone **specifiche**, non sono oggetti del caso di studio, considerando però la relazione tra **l'altitudine** corrente e quella **target**. Questo permette di:

- mantenere il modello **semplice** e **verificabile**
- applicare lo stesso **modello** a diverse situazioni **operative**
- concentrarsi sulle **transizioni** di stato piuttosto che sui valori precisi

Per modellare correttamente il comportamento del drone, si considerino le seguenti **proposizioni atomiche** fondamentali:

Sono state introdotte delle ***etichette*** per rendere più immediata l'interpretazione del modello

Proposizione	Etichetta	Descrizione

i	**INATTIVO**	*Il drone non è in volo e posizionato sul suolo*
d	**DECOLLO**	*Il drone è in fase di decollo*
h	**HOVERING**	*Il drone è in fase di hovering*
r	**RILASCIO**	*il drone ha rilasciato la merce*
v	**IN VOLO**	*il drone naviga in volo per raggiungere i punti di consegna*
a	**ATTERRAGIO**	*il drone è in fase di atterraggio*
t	**ALTITUDINE TARGET**	*il drone ha raggiunto l'altitudine target*
s	**SOTTO ALTITUDINE TARGET**	*il drone è sotto l'altitudine target*
u	**SOPRA ALTITUDINE TARGET**	*Il drone è sopra l'altitudine target*

Struttura di Kripke

Nella logica temporale viene utilizzata la struttura di **Kripke** per la corretta modellazione del comportamento del sistema, introducendo **stati finiti** e transizioni che rappresentano i cambiamenti nelle **manovre** [19]. Ogni stato rappresenta una **configurazione** specifica del **drone**, caratterizzata principalmente dalla sua **altitudine** e dalla **manovra** in esecuzione.

Dato l'insieme $\sum = \{i, d, h, r, v, a, t, s, u\}$ rappresentativo delle **proposizioni atomiche** e il rispettivo Insieme delle Parti $\mathbf{2^{\Sigma}}$, per ogni singolo stato $\boldsymbol{s}$, $\boldsymbol{L(s)}$ è l'insieme di tutte le proposizioni atomiche che vengono **valutate** come **vere** nello stato. Una struttura di **Kripke** è definita come $\boldsymbol{M} = (\boldsymbol{S}, \boldsymbol{I}, \boldsymbol{R}, \boldsymbol{L})$ dove:

- $\boldsymbol{S}$ è l'insieme di tutti gli stati del sistema, definito come $S = \{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8,\}$
- $\boldsymbol{I}$ è l'insieme che contiene **lo stato iniziale**, definito come. $\left(I \subseteq S \mid I = \{S_1,\}\right)$
- $\boldsymbol{R} \subseteq S \times S$ è l'insieme di relazioni di transizione di stato. Per ogni $s \in S$, c'è s' tale che $s \rightarrow s'$ e la relazione è indicata come $(s, s') \in R$

$$\boldsymbol{R} = \{(S_1, S_2), (S_2, S_4), (S_3, S_1), (S_4, S_3), (S_4, S_5), (S_4, S_6), (S_4, S_7), (S_4, S_8), (S_5, S_4), (S_5, S_6), (S_5, S_7), (S_6, S_5), (S_7, S_5), (S_8, S_4)\}$$

- $\boldsymbol{L}$ è una funzione di etichettatura che mappa $\boldsymbol{S}$ all'insieme $\mathbf{2^{\Sigma}}$, dove $\boldsymbol{\Sigma}$ è l'insieme delle proposizioni atomiche.
  - $\boldsymbol{L}(S_1) = \{i, s\},\ \boldsymbol{L}(S_2) = \{d, s\},\ \boldsymbol{L}(S_3) = \{a, s\},\ \boldsymbol{L}(S_4) = \{h, t\},\ \boldsymbol{L}(S_5) = \{v, t\},\ \boldsymbol{L}(S_6) = \{v, u\},\ \boldsymbol{L}(S_7) = \{v, s\},\ \boldsymbol{L}(S_8) = \{r, t\}$

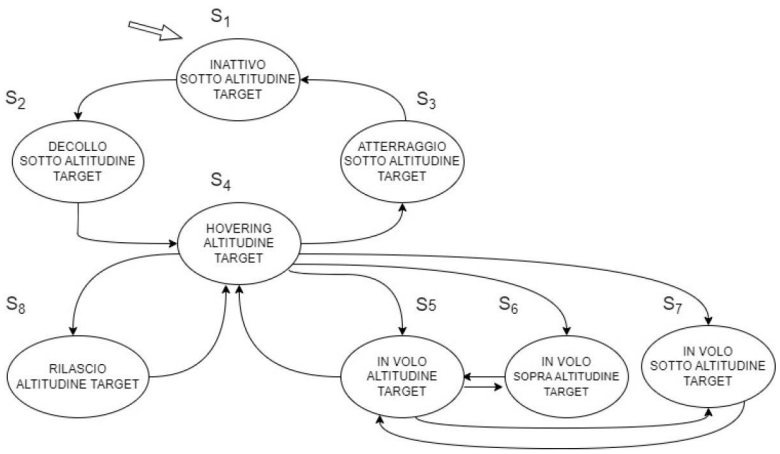


Fig. 2 Struttura di Kripke

Descrizione

Il drone quindi, posizionato sul **suolo** e **inattivo** procederà con il **decollo** mantenendo in entrambi i casi un'altitudine relativa **inferiore** rispetto a quella **target**. Successivamente si posizionerà **all'altitudine** target per effettuare **l'hovering** e in seguito potrà iniziare il volo verso i punti **target** per la **consegna**, intervallando diverse **altitudini** in base a quelli che sono i fattori **influenzanti**. Nel momento in cui si troverà nel punto di **consegna**, si posizionerà **all'altitudine** target ed effettuerà **l'hovering** prima di procedere al **rilascio** della **merce**. Successivamente continuerà il **volo** verso le **rotte** predeterminate e infine raggiungerà la base di partenza, portandosi dapprima **all'altitudine target** ed effettuando **hovering**, per poi effettuare **l'atterraggio**. Come descritto nella fase di analisi, non sono state **specificate** situazioni di pericolo o **emergenza**, ma da come si può dedurre dalla struttura il drone potrebbe **decollare**, effettuare l'hovering e atterrare nuovamente, senza **consegnare** nessuna merce. La possibilità di atterrare dopo il decollo potrebbe essere necessaria per una corretta gestione delle **emergenze**, cancellazione della **missione**, malfunzionamenti rilevati dopo il **decollo** o ancora condizioni meteorologiche **avverse**.

Definizione delle proprietà tramite LTL e CTL

Le logiche temporali **LTL** (*Linear Temporal Logic*) e **CTL** (*Computation Tree Logic*) rappresentano strumenti **formali** fondamentali, consentendo di **specificare** e verificare rigorosamente **proprietà** del modello in diversi **istanti temporali**, garantendo la correttezza del **sistema** rispetto a requisiti critici di **sicurezza** e **funzionalità**. **LTL** adotta una visione lineare del tempo, dove ogni **istante** ha un unico futuro possibile. Nel contesto dei droni, **LTL** permette di specificare sequenze obbligatorie di **operazioni**, come la necessità di stabilizzazione in **hovering** dopo il rilascio della **merce**. **CTL**, d'altra parte, considera una struttura **ramificata** del tempo, dove ogni istante può evolvere in molteplici futuri **alternativi**. Questa caratteristica rende **CTL** particolarmente efficace nella descrizione di sistemi non deterministici, dove è necessario ragionare sulla presenza o assenza di determinate situazioni [19]. Nel caso dei droni, **CTL** consente di esprimere proprietà relative alle diverse **opzioni** di volo disponibili, come la possibilità di scegliere tra multiple **altitudini** o la presenza di percorsi **alternativi** per raggiungere gli obiettivi. Questa **complementarità** risulta essenziale nella progettazione di tali sistemi, poiché le formule temporali definite per il modello del drone rappresentano un insieme completo di **proprietà** che ne caratterizzano il **comportamento**, bilanciando requisiti di **sicurezza** e **flessibilità** operativa.

Proprietà temporali LTL

$$G(rilascio \rightarrow X\ hovering)$$

Tale proprietà, assicura che dopo ogni operazione di **rilascio** il drone si stabilizzi in **hovering**, un requisito fondamentale per il mantenimento del **controllo**.

$$G(inattivo \rightarrow X\ decollo)$$

Stabilisce che dallo stato **inattivo** il sistema deve necessariamente procedere con il **decollo**, garantendo l'inizializzazione corretta della sequenza **operativa**.

$$G((volo \wedge altitudine = sopra\_target) \rightarrow X(volo \wedge altitudine = target))$$

Il drone che si trova in volo ad un'altitudine **superiore** a quella target deve necessariamente ritornare **all'altitudine target** prima di poter effettuare altre **manovre**. Questa restrizione riflette la necessità di un controllo graduale delle variazioni di **altitudine**.

$$G((volo \wedge altitudine = sotto\_target) \rightarrow X(solo \wedge altitudine = target)$$

Analogamente alla precedente, questa proprietà garantisce che da **un'altitudine inferiore** a quella **target**, il drone deve riportarsi all'altitudine di **riferimento**. Questo vincolo assicura una gestione **simmetrica** e controllata delle variazioni di **quota**.

$$G\big(hovering \wedge altitudine = target \rightarrow X\ (atterraggio \vee volo \vee rilascio)\big)$$

La proprietà delinea le possibili **evoluzioni** del sistema dallo stato di **hovering**, assicurando transizioni controllate.

$$G((volo \wedge altitudine = sopra\_target) \rightarrow (\neg atterraggio\ U\ altitudine = target))$$

La proprietà garantisce che il drone non possa effettuare un **atterraggio** direttamente da un'altitudine superiore al target, ma debba necessariamente passare per **l'altitudine target**.

$$G(decollo \rightarrow (altitudine = sotto\_target \wedge F(altitudine = target)))$$

Il decollo avviene sempre da una **posizione** sotto il target e procede **gradualmente.**

$$F(G(hovering))$$

Questa proprietà – ***correttamente non soddisfatta dal modello*** - afferma che esiste un'esecuzione in cui, da un certo punto in poi, il drone rimane bloccato indefinitamente in hovering.

Proprietà CTL

$$AG(volo \wedge altitudine = target \rightarrow EX(hovering \vee (volo \wedge \neg(altitudine = target))))$$

Questa proprietà esprime la flessibilità operativa del drone quando si trova all'altitudine **target**: può scegliere se stabilizzarsi in **hovering** o modificare la propria **altitudine**. Questa capacità decisionale è fondamentale per **l'adattabilità** del sistema a diverse condizioni **operative**.

$$AG(volo \rightarrow E\ (volo\ U\ hovering)$$

In questo caso si garantisce l'esistenza di **percorsi** che mantengano il **volo** fino al raggiungimento **dell'hovering.**

$$AG\big(hovering \rightarrow EF(atterraggio) \wedge EF(volo)\big)$$

La proprietà stabilisce la possibilità di completare la **missione** in modi diversi.

$$AG(volo \rightarrow EF(altitudine = sopra\_target) \wedge EF(altitudine = sotto\_target))$$

Durante il volo, è sempre possibile raggiungere sia **altitudini** sopra che **sotto** il target, catturando la libertà di **navigazione verticale**.

$$AG(volo \rightarrow EF(atterraggio))$$

La proprietà garantisce che da ogni **stato** di **volo** sia sempre possibile raggiungere uno stato di atterraggio **sicuro**.

$$AG(hovering \rightarrow AF\ rilascio)$$

Tale proprietà afferma che ogni stato di **hovering** deve necessariamente portare a un **rilascio**: il requisito non può essere soddisfatto dal sistema perché **contraddirebbe** la sua natura **intrinsecamente** non **deterministica**. Il modello prevede esplicitamente che dallo stato di hovering sia possibile sia procedere con il **rilascio** che con **l'atterraggio**, riflettendo scenari operativi come l'interruzione della missione per condizioni avverse o emergenze. La presenza di questa flessibilità operativa è fondamentale per la **sicurezza** e **l'adattabilità** del sistema a condizioni reali, rendendo impossibile garantire che ogni **hovering** culmini necessariamente in un **rilascio**. Questa combinazione di proprietà **LTL** e **CTL** fornisce una caratterizzazione formale completa del comportamento del drone, bilanciando requisiti **deterministici** di sicurezza con la necessaria **flessibilità** operativa.

Implementazione attraverso il tool di model checking NuSMV

NuSMV è uno strumento di model checking **simbolico**, sviluppato[8] come evoluzione del model checker originale **SMV** (*Symbolic Model Verifier*). Il model checking è una tecnica di **verifica formale** che permette di determinare se un **sistema** soddisfa determinate **proprietà** attraverso un'esplorazione **sistematica** di tutti i suoi possibili **stati**. **NuSMV** implementa questa verifica in modo "*simbolico*", il che significa che non esplora gli stati uno

```
MODULE main
-- Definizione delle variabili del sistema
VAR
    -- Stato corrente del sistema secondo la struttura di Kripke
    state : {s1, s2, s3, s4, s5, s6, s7, s8};
    -- Variabili booleane che rappresentano le azioni del drone
    inattivo : boolean;      -- Drone fermo al suolo
    decollo : boolean;       -- Fase di decollo
    hovering : boolean;      -- Fase di stazionamento
    volo : boolean;          -- Fase di volo
    atterraggio : boolean;   -- Fase di atterraggio
    rilascio : boolean;      -- Fase di rilascio merce
    -- Altitudine relativa rispetto al target
    altitudine : {sotto_target, target, sopra_target};
```

```
-- Gestione dell'altitudine in base allo stato
next(altitudine) := case
    next(state) = s6 : sopra_target;    -- Volo ad alta quota
    next(state) in {s4, s5, s8} : target;  -- Hovering, volo normale e rilascio
    next(state) in {s1, s2, s3, s7} : sotto_target;  -- Stati vicini al suolo
    TRUE : altitudine;
esac;
```

impossibile utilizza chiamate (*BDD*) per grandi

amente [19]. All'interno del codice sviluppato, si trova il modulo **e** su cui si sviluppa l'intero sistema di **gestione** del drone. All'interno, troviamo la sezione **VAR** che dichiara tutte le variabili di stato del **sistema**, in cui ogni variabile riflette una delle **proposizioni** utilizzate che potranno assumere un valore **boolean** (*True o false*). Le tre proposizioni circa **l'altitudine**, sono state rappresentate tramite una variabile **enumerativa**, che descrive i tre possibili stati di altitudine. Nella sezione **ASSIGN**, le istruzioni **init** stabiliscono lo stato **iniziale** del sistema, posizionando il drone in uno stato **inattivo** al suolo. In secondo luogo, la funzione ***next(state)*** definisce tutte le possibili **transizioni** tra stati; infine, le funzioni **next** per le altre variabili mantengono la **coerenza** del sistema, aggiornando appropriatamente tutti i **parametri** operativi in base allo stato corrente. Inizialmente, il model **checker** legge le dichiarazioni **VAR** per costruire lo **spazio** degli stati possibili e inizializza il **sistema** secondo le istruzioni **init** nella sezione **ASSIGN**, posizionando il drone nello stato S_1 (**inattivo**) con le appropriate variabili **boolean** e di **altitudine**.

[8] Si tratta di un prodotto **open-source** sviluppato congiuntamente da *ITC-IRST, Trento, Italia, Carnegie Mellon University, Università di Genova e Università di Trento.*

```
ASSIGN
   -- Inizializzazione delle variabili di stato
   -- Il drone parte da fermo al suolo
   init(inattivo) := TRUE;
   init(decollo) := FALSE;
   init(hovering) := FALSE;
```

```
next(state) := case
   state = s1 : s2;                -- Da inattivo a decollo
   state = s2 : s4;                -- Da decollo a hovering
   state = s3 : s1;                -- Da atterraggio a inattivo
   state = s4 : {s3,s5,s6,s7,s8};  -- Da hovering a volo/atterraggio/rilascio
```

A questo punto, il sistema inizia a calcolare le possibili evoluzioni utilizzando le definizioni **next** nella sezione **ASSIGN**. Per ogni stato corrente, il **model checker** determina quali **transizioni** sono possibili consultando la struttura case di **next(state)** e aggiorna conseguentemente tutte le variabili **ausiliarie** attraverso le loro rispettive definizioni **next**. Ad esempio, quando il sistema passa da S_1 a S_2 automaticamente aggiorna le variabili **booleane** per riflettere che il drone è passato dallo stato **inattivo** allo stato di **decollo**.

La sezione delle **SPECIFICATION**, posta alla fine del modello, contiene le proprietà **temporali** - *descritte in precedenza* - che il sistema deve soddisfare, espresse sia in logica **LTL** che **CTL**. Da come si **evince**, le proprietà definite vengono correttamente **rispettate**, così come **proprietà** che descrivono comportamenti **inattesi** vengono violate correttamente, descrivendo anche un controesempio - *omesso per ragioni di spazio* -.

```
-- specification  F ( G hovering)  is false
```

```
-- specification  G (rilascio ->  X hovering)  is true
-- specification  G (inattivo ->  X decollo)  is true
-- specification  G ((volo & altitudine = sopra_target) ->  X (volo & altitudine = target))  is true
-- specification  G ((volo & altitudine = sotto_target) ->  X (volo & altitudine = target))  is true
-- specification  G (decollo -> (altitudine = sotto_target &  F altitudine = target))  is true
-- specification  G ((volo & altitudine = sopra_target) -> (!atterraggio U altitudine = target))  is true
-- specification  G ((hovering & altitudine = target) ->  X ((atterraggio | volo) | rilascio))  is true
```

```
-- specification AG (volo -> E [ volo U hovering ] )  is true
-- specification AG (volo -> (EF altitudine = sopra_target & EF altitudine = sotto_target))  is true
-- specification AG (volo -> EF atterraggio)  is true
-- specification AG (hovering -> AF rilascio)  is false
```

```
-- specification AG ((volo & altitudine = target) -> EX (hovering | (volo & altitudine != target)))  is true
-- specification AG (hovering -> (EF atterraggio & EF volo))  is true
```

Per una simulazione in tempo **reale** del **modello**, si rimanda alla presentazione allegata con la documentazione, in cui è stato prodotto un video **demo** che mostra l'esecuzione del sistema attraverso diversi comandi messi a disposizione da **NuSMV**. È bene sottolineare, che le **proprietà** fornite catturano alcuni degli aspetti più **importanti** del comportamento del drone, ampiamente descritti nella fase di **analisi**.

Bibliografia

[1] Attigui, Mohamed. *Operations Research: The Traveling Salesman Problem (TSP)*. 16 Apr. 2021, www.researchgate.net/publication/350924023_Operations_Research_The_Traveling_Salesman_Problem_TSP.

[2] Hoffman, Karla L., et al. "Traveling Salesman Problem." *Encyclopedia of Operations Research and Management Science*, 2013, pp. 1573–1578, link.springer.com/referenceworkentry/10.1007%2F978-1-4419-1153-7_1068, https://doi.org/10.1007/978-1-4419-1153-7_1068

[3] “Teoria Dei Grafi.” *“Graph - Encyclopedia of Mathematics.” Encyclopediaofmath.org, 2023, encyclopediaofmath.org/wiki/Graph*

[4] Held, Michael, and Richard M. Karp. “The Traveling-Salesman Problem and Minimum Spanning Trees.” *Operations Research*, vol. 18, no. 6, Dec. 1970, pp. 1138–1162, https://doi.org/10.1287/opre.18.6.1138. Accessed 29 Mar. 2022.

[5] Scholz, Jan. *GENETIC ALGORITHMS and the TRAVELING SALESMAN PROBLEM a HISTORICAL REVIEW*.

[6] Little, John D. C., et al. “An Algorithm for the Traveling Salesman Problem.” *Operations Research*, vol. 11, no. 6, Dec. 1963, pp. 972–989, https://doi.org/10.1287/opre.11.6.972. Accessed 10 Apr. 2020.

[7] Applegate, David, et al. “Implementing the Dantzig-Fulkerson-Johnson Algorithm for Large Traveling Salesman Problems.” *Mathematical Programming*, vol. 97, no. 1, July 2003, pp. 91–153, www.math.uwaterloo.ca/~bico/papers/dfj_mathprog.pdf, https://doi.org/10.1007/s10107-003-0440-4.

[8]Helbig Hansen, Keld, and Jakob Krarup. “Improvements of the Held—Karp Algorithm for the Symmetric Traveling-Salesman Problem.” *Mathematical Programming*, vol. 7, no. 1, Dec. 1974, pp. 87–96, https://doi.org/10.1007/bf01585505.

[9] Huang, Ting, et al. “A Niching Memetic Algorithm for Multi-Solution Traveling Salesman Problem.” *IEEE Transactions on Evolutionary Computation*, 2019, pp. 1–1, https://doi.org/10.1109/tevc.2019.2936440.

[10] Jeong, Ho Young, et al. “Truck-Drone Hybrid Delivery Routing: Payload-Energy Dependency and No-Fly Zones.” *International Journal of Production Economics*, vol. 214, Aug. 2019, pp. 220–233, www.sciencedirect.com/science/article/abs/pii/S0925527319300106, https://doi.org/10.1016/j.ijpe.2019.01.010.

[11] potassco. “Release Potassco Guide Version 2.2.0 · Potassco/Guide.” *GitHub*, 15 Jan. 2019, github.com/potassco/guide/releases/tag/v2.2.0.

[12] Tichakorn Wongpiromsarn, et al. “Formal Methods for Autonomous Systems.” Foundations and Trends® in Systems and Control, vol. 10, no. 3-4, 1 Jan. 2023, pp. 180–407, https://doi.org/10.1561/2600000029

[13] Drone Launch Academy. “AGL vs MSL: Meanings & Calculations (with Sample Questions).” Drone Launch Academy, 11 Nov. 2019, dronelaunchacademy.com/resources/the-difference-between-msl-and-agl/

[14] Rossano D'Antonio. “In Arrivo La Consegna Amazon Con Drone in Basso Molise E Vicino Abruzzo.” Monteneronotizie.net, 8 Oct. 2024, www.monteneronotizie.net/notizie/attualita/4542/in-arrivo-la-consegna-amazon-con-drone-in-basso-molise-e-vicino-abruzzo.

[15] Ente Nazionale per l'aviazione civile. “Sottocategorie A1, A2, A3 - Ente Nazionale per l’Aviazione Civile.” Ente Nazionale per l’Aviazione Civile, 15 May 2024, www.enac.gov.it/sicurezza-aerea/droni/categoria-aperta-open-category/sottocategorie-a1-a2-a3/

[16] Bahabry, Ahmed, et al. “Low-Altitude Navigation for Multi-Rotor Drones in Urban Areas.” IEEE Access, vol. 7, 2019, pp. 87716–87731, https://doi.org/10.1109/access.2019.2925531.

[17] Chipade, Vishnu S., et al. “Systematic Design Methodology for Development and Flight Testing of a Variable Pitch Quadrotor Biplane VTOL UAV for Payload Delivery.” https://doi.org/10.1016/j.mechatronics.2018.08.008.

[18] Alsawy, Assem, et al. “An Image Processing Approach for Real-Time Safety Assessment of Autonomous Drone Delivery.” Drones, vol. 8, no. 1, 1 Jan. 2024, p. 21, www.mdpi.com/2504-446X/8/1/21, https://doi.org/10.3390/drones8010021.

[19] Wang, Jiacun. Formal Methods in Computer Science. CRC Press, 21 June 2019.