DOCUMENTAZIONE DI PROGETTO

Università degli Studi di Bari Aldo Moro

CdLM in Sicurezza Informatica

Sistema di consegna medicinali tramite drone nel Gargano

Corso: Metodi Formali per la Sicurezza

Studente: Giacomo Pagliara

Sommario

1 – Introduzione al problema del commesso viaggiatore

Il problema del commesso viaggiatore (TSP, in inglese, *"travelling salesman problem"*) è un problema di ottimizzazione della programmazione che consiste nel trovare il percorso più breve che collega un dato insieme di punti. Infatti, trovare una soluzione esatta richiede molta potenza di calcolo, perciò vengono utilizzati vari metodi per trovare soluzioni perlopiù approssimative. [1]

Il primo accenno ad un Problema del Commesso Viaggiatore risale ad un manuale tedesco del 1832, dove si presentavano vari esempi pratici di percorsi tra diverse città della Germania e della Svizzera, evidenziando l'importanza pratica di trovare itinerari efficienti. Invece, lo studio formale e matematico dei TSP iniziò durante la prima metà dell'Ottocento con l'introduzione del termine *grafo hamiltoniano*, da parte del matematico Hamilton [2].

Proprio quest'ultimo, insieme al matematico Kirkman, descrissero per la prima volta il problema del commesso viaggiatore attraverso la creazione di un gioco chiamato *"Icosian Game"*.
Il gioco era risolvibile trovando un *ciclo hamiltoniano*, ovvero un percorso che visita tutti i nodi senza sovrapposizioni.

Figura 1 - Icosian Game

Per affrontare questo problema dobbiamo porci un quesito naturale: ***"Un commesso viaggiatore deve guidare fino a N città sulla mappa e poi tornare a casa. Vuole spendere il meno possibile in benzina. Come troviamo il percorso più breve da seguire?"*** Questo è un problema apparentemente semplice. Non a caso, dovrebbe essere facile scrivere un programma per provare ogni possibile percorso e restituire quello più breve. La difficoltà risiede nel fatto che, aumentando il numero di punti N, il numero di percorsi possibili cresce in maniera esponenziale, seguendo il fattoriale N! dei punti. Anche un piccolo numero di punti, come 10, richiederebbe il controllo di 3,6 milioni di possibili percorsi [1].

Figura 2 - Esempio di TLS [1]

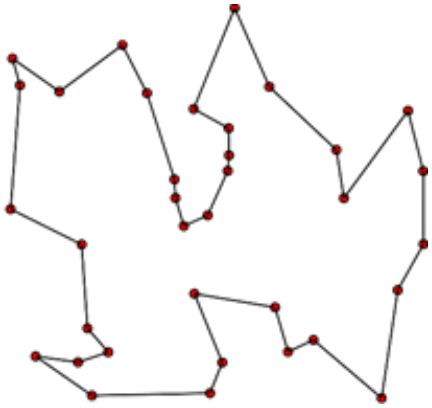


Figura 3 - Esempio di percorso ottimale [11]

1.1 Formalizzazione del TSP

Al problema del TSP è associabile un grafo G= (V, A) in cui V è l'insieme degli *n* nodi (o città) e A è l'insieme degli archi (o strade). Si indica con $c_{i,j}$ il costo dell'arco per andare dal nodo *i* al nodo *j* (ovvero il Cammino minimo). Il TSP è simmetrico se $c_{i,j} = c_{i,j} \forall (\boldsymbol{i}, \boldsymbol{j})$**,** altrimenti si dice asimmetrico. A seconda che il problema sia simmetrico o asimmetrico, esso può essere descritto con modelli matematici differenti. Detta $x_{i,j}$ la generica variabile binaria tale che $x_{i,j}$= 1 se l'arco *(i,j)* appartiene al circuito e $x_{i,j}$= 0 altrimenti. Una possibile formulazione matematica del problema è:

$$z = \min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{i,j} x_{i,j}$$

Quest'ultima formula prende il nome di *funzione obiettivo* che rappresenta la minimizzazione del costo del cammino. Inoltre, i vincoli $(\sum_{i=1}^{n} x_{i,j} = 1 \;\; j = 1, \dots, n)$ e $(\sum_{j=1}^{n} x_{i,j} = 1 \; i = 1, .., n)$ indicano che in ogni nodo i entra ed esce un solo arco, e vengono chiamati vincoli di assegnazione [3].

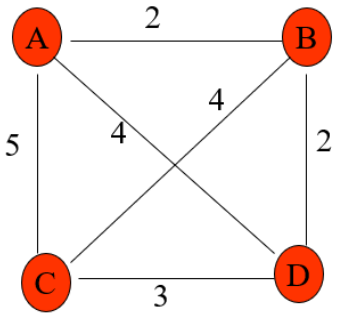


Figura 2 - Esempio di grafo pesato

1.2 Complessità e…, Come Risolverlo?

Come anticipato, diventa difficile calcolare rapidamente il percorso più breve tra due punti o nodi per un normale computer. Per problemi più grandi, anche per i supercomputer potrebbe diventare complicato. I problemi che aumentano in difficoltà in modo più che esponenziale con il numero di elementi sono noti come problemi ***NP-hard***. Molto spesso, quindi, ci si concentra sul trovare la soluzione più rapida che dia risultati sufficientemente buoni. Il gran numero di variabili crea una sfida quando si tratta di trovare il percorso più breve, il che rende le soluzioni approssimative, veloci ed economiche più attraenti. Per rispondere alla domanda posta dal TSP si possono utilizzare vari algoritmi:

- *Algoritmi esatti*: la soluzione viene ricercata in maniera esaustiva, ma molto spesso non è fattibile a causa del lungo tempo di calcolo richiesto.
- *Algoritmi euristici*: permettono di trovare risposte approssimative in meno tempo, esistono vari sottotipi:

- *Metodo della forza bruta*: si calcolano e si confrontano tutti gli itinerari possibili per poi scegliere quello più breve e conveniente. L'utilità di questo metodo è massima in scenari semplici.
- *Sistema branch and bound*: esplora sistematicamente i percorsi possibili, aggiungendo nodi e confrontando le distanze. Se un percorso risulta più lungo del precedente, viene scartato. Questo processo continua fino a trovare il tragitto più breve, rendendo l'algoritmo efficiente nell'ottimizzazione dei percorsi.
- *Metodo del vicino più prossimo*: parte da un nodo casuale, aggiunge il nodo più vicino e ripete il processo fino a visitare tutte le destinazioni, tornando infine al punto di partenza. È semplice ma non sempre fornisce la soluzione ottimale [4].

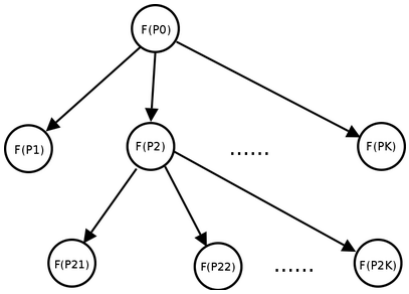


Figura 5 - Metodo Branch and Bound

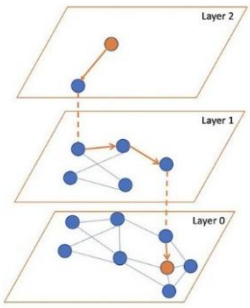


Figura 6 - Metodo del vicino più prossimo

1.3 Applicazioni

Gli approcci al TSP vengono impiegati in tutti i tipi di settori [5] [4]:

- *Robotica e Automazione*: razionalizzare i movimenti di robot e macchine(droni) diminuisce il consumo delle batterie e migliora l'efficienza;
- *Logistica e Trasporti*: per il trasporto delle merci, per il trasporto dei passeggeri, per l'assistenza tecnica;
- *Fabbricazione e pianificazione della produzione*: ottimizza la manutenzione nelle fabbriche, visitando le macchine nel minor tempo possibile;
- *Identificazione del routing di rete*: importanti per aumentare l'efficienza delle reti e dei sistemi distribuiti;
- *Ottimizzazione Hardware*;
- *DNA*: attraverso il sequenziamento di quest'ultimo.

Figura 7 - Sequenziamento DNA

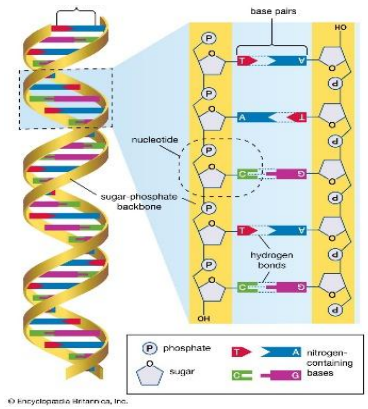
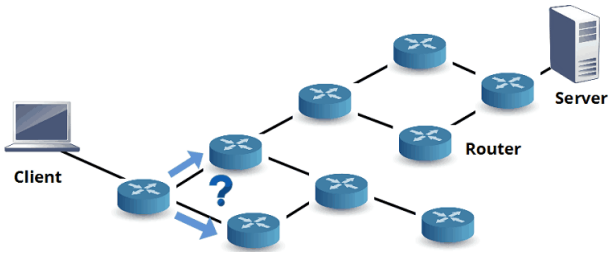


Figura 8 - Routing di rete



2 – Codifica ASP del TSP

2.1 Contesto

Come prima cosa, ho analizzato la codifica ASP (Answer Set Programming) del Problema del Commesso Viaggiatore presente sul sito ***Potassco,*** che mette a disposizione il "The Postdam Answer Set Solving" dell'Università di Postdam (https://potassco.org/clingo/run/), adattandola a un caso di studio reale e socialmente utile: la consegna di farmaci mediante l'uso di un drone, che parta dall'ospedale di San Giovanni Rotondo e voli verso le diverse località del Gargano e delle Isole Tremiti. La scelta è ricaduta su questo particolare scenario per diverse motivazioni: prima di tutto, la necessità di garantire la fornitura di medicinali in maniera rapida in aree geograficamente complesse, in secondo luogo la presenza di zone costiere e isole potrebbero rendere il trasporto tradizionale più lento e come ultimo anche la presenza di varietà di terreni (come volo su terra o volo su mare) e altitudini significative in queste aree.

2.2 Struttura generale e sfide affrontate

La struttura generale della codifica creata comprende sette punti strategici:

- **San Giovanni Rotondo** (nodo *start, altitudine: 500 m):* sede dell'ospedale e punto di partenza e di ritorno del drone:
- Località costiere: come **Vieste** (nodo *1, altitudine: 40 m*), **Peschici** (nodo *2, altitudine: 90 m*) e **Rodi Garganico** (nodo *3, altitudine: 40 m*);
- Isole Tremiti: come **l'Isola di San Domino** (nodo *4, altitudine: 70 m*) e **l'Isola di San Nicola** (nodo *5, altitudine: 70 m*);
- Entroterra: **Monte Sant'Angelo** (nodo *6, altitudine: 700 m*).

Le sfide che ho affrontato riguardano la presenza di variazioni geografiche per la presenza di tratti terreni lungo la costa, tratti marittimi verso le isole e le significative variazioni di altitudine. Inoltre, ho tenuto conto della necessità di dover gestire alcuni vincoli operativi come il consumo energetico sui tratti marittimi, l'impatto dei dislivelli sulle prestazioni del drone e infine l'ottimizzazione del percorso andando a considerare queste variabili, che analizzeremo.

Per quanto riguarda la funzione di costo, quest'ultima è data dal prodotto di tre fattori fondamentali:

- La **distanza fisica**: rappresenta la distanza in linea d'aria tra le località in km, calcolata utilizzando il sito https://www.freemaptools.com/how-far-is-it-between.htm , il quale utilizza le coordinate geografiche reali;
- **Fattore Terreno sorvolato**: terreno terrestre (con valore 1) e terreno marittimo (con valore 1.5, poiché c'è un maggior consumo energetico sorvolando il mare, il drone, necessità di avere maggiore stabilizzazione e ovviamente impatta anche l'assenza di punti di atterraggio di emergenza). Ho deciso di utilizzare questi due valori per avere una differenza chiara e impattante;
- **Fattore altitudine**: si basa sulle altitudini reali delle varie località.

Questi fattori permettono di modellare il consumo energetico del drone durante il suo percorso. Si assume che il drone possa completare l'intero ciclo con una singola carica e che il carico dei medicinali sia entro i limiti consentiti.

2.3 Formalizzazione del problema

Analizziamo il codice [6] [7]:

- **Generazione dei cicli possibili**

```
{ cycle(X,Y) : edge(X,Y) } = 1 :- node(X).
{ cycle(X,Y) : edge(X,Y) } = 1 :- node(Y).
```

  Vengono generati tutti i possibili cicli nel grafo.
  - "{ cycle(X,Y) : edge(X,Y) } = 1", per ogni nodo X, sceglie esattamente una relazione "cycle(X,Y)" tra quelle possibili tali che esista un arco "edge(X,Y)." Lo stesso discorso vale per il secondo predicato. Ci assicuriamo quindi che da ogni nodo esca esattamente un arco e in ogni nodo arrivi esattamente un arco. Si genera un ciclo Hamiltoniano in cui ogni nodo viene visitato una volta e si ha un arco in entrata e uno in uscita per ciascun nodo.
- **Definizione della raggiungibilità**

```
% Define
reached(Y) :- cycle(start,Y).
reached(Y) :- cycle(X,Y), reached(X).
```

  - Definisce quali nodi sono raggiungibili nel percorso. Il primo predicato "reached(Y) :- cycle(start,Y)": afferma che se esiste un arco selezionato "cycle(start,Y)" che parte dal nodo di partenza *start* e arriva a Y, allora Y è raggiungibile. Il secondo predicato "reached(Y) :- cycle(X,Y), reached(X)." afferma che se esiste un arco selezionato "cycle(X,Y)" e se X è già raggiungibile, allora anche Y diventa raggiungibile. Si definisce una definizione ricorsiva che "propaga" la raggiungibilità lungo il percorso.
- **Vincoli per garantire un ciclo valido**

```
% Test
:- node(Y), not reached(Y).
:- not reached(start).
```

  :- rappresenta un vincolo, infatti:
  - Il primo vincolo ": - node(Y), not reached(Y)." afferma che per ogni nodo Y, se non risulta raggiungibile (cioè, se "not reached(Y)." È vero), la soluzione viene scartata. Questo vincolo garantisce che tutti i nodi siano effettivamente raggiunti partendo da *start*.
  - Il secondo vincolo: ":- not reached(start)." Impone che il nodo *start* debba essere raggiungibile, assicura che il percorso ritorni al punto di partenza per chiudere il ciclo.
- **Istruzione di Display**

```
#show cycle/2.
```

- Questa direttiva indica al solver di mostrare gli atomi relativi al predicato "cycle/2" nell'output finale. In questo modo, l'output si focalizzerà sul percorso cioè, sugli archi selezionati che fanno parte del ciclo.

➢ **Definizione dei Nodi e degli Archi**

```
% Nodes
node(start).        % San Giovanni Rotondo - SGR
node(1..6).        % Altri nodi da visitare

% Edges
edge(start, (1;2;3;4;5;6)).   % Da SGR posso andare ovunque
edge(1, (2;3;4;5;6)).         % Da Vieste agli altri (non SGR)
[...]
edge((1;2;3;4;5;6), start).   % Da tutti posso tornare a SGR
```

- Nei nodi, *start* rappresenta il punto di partenza (San Giovanni Rotondo) e poi vengono definiti i nodi numerati da 1 a 6, che abbiamo definito già prima.
- Gli *edge* definiscono i possibili collegamenti da *start* verso tutti gli altri nodi, i collegamenti tra i nodi intermedi e la possibilità di ritorno a *start* da ogni nodo.

Si garantisce che il percorso possa iniziare da *start,* si possano visitare tutti i nodi e si possa ritornare a *start* da qualsiasi punto.

➢ **Definizione delle Distanze**

```
% Distance
distance(start,1,55). distance(1,start,55).    % SGR-Vieste
distance(start,2,40). distance(2,start,40).    % SGR-Peschici
[...]
distance(4,5,3). distance(5,4,3).         % San Domino-San Nicola
```

- "distance(X,Y,D)": X nodo di partenza, Y nodo di arrivo, D distanza in km tra i nodi. Per ogni coppia di nodi connessi, viene definita la distanza in linea d'aria reale in km. Per quanto riguarda la distanza, il grafo è bidirezionale e la distanza è la stessa in entrambe le direzioni.

➢ **Fattore Terreno**

```
% Terrain factor
terrain(start,1,1). terrain(1,start,1).   % terra
terrain(start,2,1). terrain(2,start,1).   % terra
[...]
terrain(4,5,15). terrain(5,4,15).         % mare
```

- "terrain(X,Y,T)": X nodo di partenza, Y nodo di arrivo, T tipo di terreno attraversato. Si definisce il fattore *terreno* per ogni arco, che modifica il costo in base alla natura del percorso. Infatti, sono due i possibili valori per T: 1 per tratti terrestri (1.0) e 15 per tratti marittimi (che rappresenta 1.5 dopo la normalizzazione, infatti nella formula finale questo fattore verrà diviso per 10). Non ho messo da subito 1.5 perché clingo genera delle eccezioni e non può essere utilizzato un valore non intero nel fattore.

➢ **Fattore Altitudine e calcolo del relativo costo**

```
% Altitude
altitude(start,500).  % San Giovanni Rotondo
altitude(1,40).       % Vieste
[...]
altitude(6,700).      % Monte Sant'Angelo

% Altitude cost
alt_cost(X,Y,C) :- altitude(X,AX), altitude(Y,AY),
                   C = |AX-AY|/100.
```

- "altitude(N,A)": N identificativo del nodo, A altitudine reale in metri sul livello del mare. Il predicato "alt_cost(X,Y,C)" calcola la differenza in valore assoluto di altitudine tra i nodi X e Y e la normalizza dividendo per 100. In questo modo si ottiene un valore C che rappresenta l'impatto della variazione di altitudine sul costo complessivo del volo. *Esempio: 500m di dislivello = 5% di costo extra.*

- **Calcolo del costo finale e ottimizzazione**

```
% Final cost
cost(X,Y,C) :- distance(X,Y,D), terrain(X,Y,T), alt_cost(X,Y,A),
               C = (D * T / 10) * (1 + A).
% Optimize
#minimize { C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```

- "cost(X,Y,C)" definisce il costo C per il tratto da X a Y dove: X è il nodo di partenza, Y nodo di arrivo e C il costo totale calcolato. La funzione recupera: "distance(X,Y,D)" la distanza D in km tra X e Y, "terrain(X,Y,T)" il fattore terreno T, "alt_cost(X,Y,A)" il fattore altitudine A calcolato dai dislivelli. Per la formula "C = (D * T / 10) * (1 + A)", **la prima parte** "(D * T / 10)" moltiplica la distanza D per il fattore terreno T, si divide per 10 per normalizzare il fattore terreno in modo da ottenere il costo base del tratto. Ad esempio: "Se T=1 (terra): D * 1/10 = D/10" oppure "Se T=15 (mare): D * 15/10 = D * 1.5". **La seconda parte**, invece, "(1 + A)", aggiunge un costo percentuale basato sul dislivello, il numero 1 è necessario per garantire che la funzione di costo non sia mai zero e per mantenere il costo originale quando non c'è dislivello, l'addizione infatti permette di interpretare A come percentuale aggiuntiva trasformando il dislivello in un incremento proporzionale e mantenendo una relazione lineare tra dislivello e aumento di costo. **Esempi pratici**: *Nessun dislivello: A = 0 (stessa altitudine), (1 + 0) = 1, nessun costo aggiuntivo. Piccolo dislivello: A = 0.5 (50m di differenza), (1 + 0.5) = 1.5, aumento del 50% del costo. Grande dislivello: A = 4.6 (460m di differenza), (1 + 4.6) = 5.6 aumento del 460% del costo.*
  Quindi, il costo finale C per l'arco "(X,Y)" è dato dal prodotto di queste due parti. La formula tiene conto sia della lunghezza dell'arco, modificata dal tipo di terreno, sia delle variazioni di altitudine che influiscono sul consumo del drone. ***Esempio pratico*** *per il tratto SGR → Vieste: D = 55 km (distanza), T = 1 (terreno terrestre), A = 4.6 (|500-40|/100). C = (55 * 1/10) * (1 + 4.6) = 5.5 * 5.6 = 30.8.*

- Mentre la direttiva di minimizzazione "#minimize { C,X,Y : cycle(X,Y), cost(X,Y,C) }" comunica al solver ASP di minimizzare la somma dei costi C di tutti gli archi selezionati nel percorso definiti dal predicato "cycle/2". Il solver cercherà il ciclo Hamiltoniano che minimizza il costo totale calcolato con la funzione di costo.

2.4 Risultati

```
clingo version 5.7.0
Reading from stdin
Solving...
Answer: 1
cycle(start,3) cycle(6,5) cycle(5,2) cycle(4,6) cycle(3,4) cycle(2,1) cycle(1,start)
Optimization: 1679
Answer: 2
cycle(start,3) cycle(6,start) cycle(5,2) cycle(4,5) cycle(3,4) cycle(2,1) cycle(1,6)
Optimization: 215
Answer: 3
cycle(start,3) cycle(6,start) cycle(5,4) cycle(4,2) cycle(3,5) cycle(2,1) cycle(1,6)
Optimization: 213
OPTIMUM FOUND

Models       : 3
  Optimum    : yes
Optimization : 213
Calls        : 1
Time         : 0.047s (Solving: 0.01s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

Analizzando i risultati ottenuti il percorso hamiltoniano ottimizzato è il seguente**:**
San Giovanni Rotondo(start) – Rodi Garganico(3) – San Nicola(5) – San Domino(4) – Peschici(2) – Vieste(1) – Monte Sant'Angelo(6) – start.

3 – Modellazione in LTL/CTL

Dopo aver adattato la codifica ASP del TSP, si passa alla fase di modellazione in LTL/CTL delle varie manovre del drone. Definiamo prima di tutto gli stati del drone e la relativa struttura di Kripke. Di seguito sono stati definiti i vari stati del drone:

- **Decollo**: stato iniziale del sistema, il drone inizia il decollo.
- **Navigazione**: il drone segue il percorso pianificato, seguendo la rotta ottimizzata calcolata nella prima parte.
- **Hovering**: stato di stazionamento stabile, rappresenta il punto di decisione per le prossime azioni.
- **Consegna**: Il drone rilascia il carico medico arrivando al completamento della missione in quel determinato punto.
- **Posizionamento**: Il drone si allinea in maniera precisa, per prepararsi alla fase finale.
- **Atterraggio**: Il drone atterra e completa una fase del percorso.

È doveroso fare delle precisazioni in merito agli stati di *Consegna* e *Posizionamento.* Ho ipotizzato che per la *Consegna* il drone non atterra, ma mantiene **una quota costante** di sicurezza, in cui c'è la verifica delle condizioni per il rilascio sicuro e il medicinale viene rilasciato

tramite una tecnica chiamata *"autonomia in volo stazionario [8]"* dove si utilizza un meccanismo a verricello, essenzialmente un piccolo argano motorizzato, per calare il contenitore dei farmaci. Mentre per il *Posizionamento* il drone gestisce i vari dislivelli significativi in modo tale da adattarsi all'altitudine per prepararsi all'atterraggio e quindi c'è una **riduzione graduale della quota**.

3.1 Struttura di Kripke

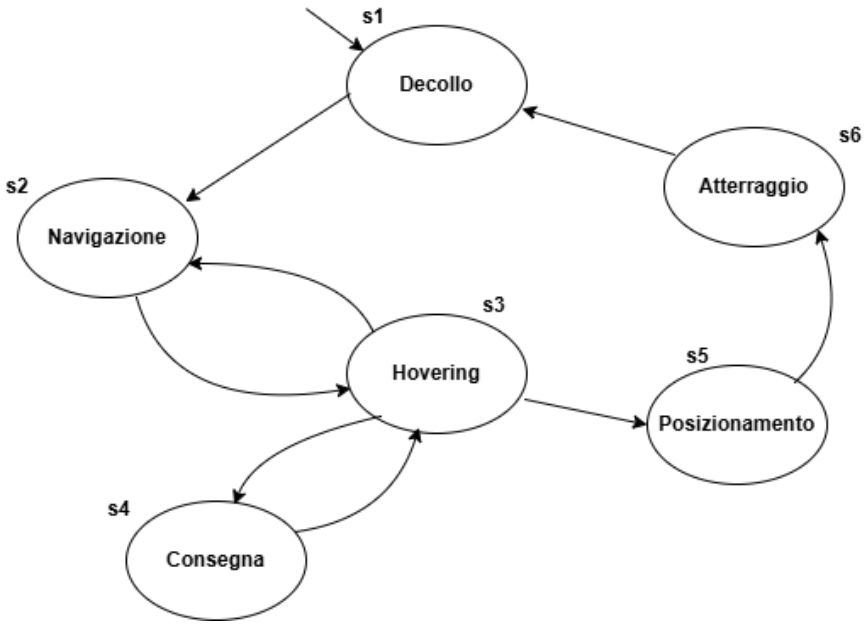
La logica temporale viene interpretata mediante la **struttura di Kripke**, cioè un sistema a transizioni di stato che è in grado di rappresentare il comportamento di un sistema [9]. Matematicamente la struttura di Kripke si definisce come M = (S, I, R, L) in cui:

- S, insieme di stati;
- I, insieme di stati iniziali t.c. $I \subseteq S$;
- $R \subseteq SxS$, insieme relazione sugli stati di transizione, cui per ogni $s \in S$, esiste s' t.c. $s \rightarrow s'$, con tale relazione denotata come $(s, s') \in R$;
- L, funzione di etichettatura che mappa S all'insieme potenza di $\sum$ (insieme di proposizione atomiche);

Questo sistema di transizione può essere più facilmente illustrato come un grafo diretto in cui: ogni nodo è uno stato; gli stati di transizione sono frecce direzionali; l'etichettatura di ogni stato è marcata su ogni nodo.

La struttura di Kripke M = (S, I, R, L) del nostro sistema è definita come:

- S = {s1, s2, s3, s4, s5, s6}
- I = {s1}
- R = {(s1, s2), (s2, s3), (s3, s2), (s3, s4), (s3, s5), (s5, s6), (s6, s1), (s4, s3)}
- L(s1) = {Decollo}
- L (s2) = {Navigazione}
- L (s3) = {Hovering}
- L (s4) = {Consegna}
- L (s5) = {Posizionamento}
- L (s6) = {Atterraggio}



3.2 Proprietà LTL (Linear Temporal Logic)

LTL ragiona su un singolo cammino di esecuzione alla volta, vede il tempo come un percorso lineare che si estende verso il futuro. Le proprietà LTL definite per il nostro sistema sono:

- **G (decollo -> X navigazione)** : Dopo il decollo, il prossimo stato deve essere navigazione.
- **G (navigazione -> X hovering)** : Dopo la navigazione, il prossimo stato deve essere hovering.
- **G (hovering -> X (navigazione ∨ consegna ∨ posizionamento))** : Da hovering, il prossimo stato può essere navigazione, consegna o posizionamento.
- **G (consegna -> X hovering)** : Dopo la consegna si torna sempre in Hovering.
- **G (posizionamento -> X atterraggio)** : Dopo il posizionamento deve seguire l'atterraggio
- **G (hovering -> (X consegna -> X X hovering))** : Se da hovering si va in consegna, dopo si tornerà in hovering.
- **G (atterraggio -> F decollo)** : Dopo l'atterraggio, prima o poi si tornerà in decollo.

3.3 Proprietà CTL (Computation Tree Logic)

CTL è una logica temporale di ramificazione, modella il tempo come una struttura ad albero in cui il futuro è non deterministico. Le proprietà CTL definite per il nostro sistema sono:

- **AG (EF decollo)** : Da ogni stato è sempre possibile raggiungere, ad un certo punto, il decollo.
- **AG (decollo -> EF consegna)** : Dal decollo è sempre possibile, in futuro, raggiungere la consegna.
- **AG ¬(navigazione ∧ hovering) :** Non si può essere contemporaneamente in navigazione e hovering.
- **AG (navigazione -> EF hovering)** : Dalla navigazione si può sempre raggiungere, a un certo punto, hovering.
- **AG (hovering -> EF (posizionamento ∧ EX atterraggio))** : Da hovering esiste un percorso che lo porta, prima a entrare in posizionamento e poi raggiunge l'atterraggio.
- **AG (hovering -> E (hovering U (consegna ∨ posizionamento)))** : Da hovering esiste un percorso dove si rimane in hovering fino a raggiungere consegna o posizionamento.

3.4 – Codifica NuSMV

L'implementazione in NuSMV del modello implementato è strutturata in questo modo [10]:

- **Il modulo principale** "`MODULE main`" definisce il sistema che ho modellato. Per la **dichiarazione delle Variabili (VAR),** nella sezione `VAR` vengono dichiarate "`state`" ovvero la variabile che rappresenta gli stati del sistema e può assumere uno dei valori (s1 = decollo, s2= navigazione, ecc.). Oltre allo stato, ho definito anche delle variabili booleane *(decollo, navigazione, hovering, consegna, posizionamento, atterraggio)* che indicano, per ogni stato, quale proprietà è attiva. Ad esempio, quando lo stato è *s1,* la variabile *decollo* è vera. Queste variabili permettono di seguire esplicitamente in quale fase si trova il drone e di verificare le proprietà associate.

```
MODULE main
VAR
    state : {s1, s2, s3, s4, s5, s6};
    decollo : boolean;
    navigazione : boolean;
    hovering : boolean;
    consegna : boolean;
    posizionamento : boolean;
    atterraggio : boolean;
```

- **Assegnazioni iniziali (ASSIGN):** "`init(state) := s1;`" indica che il sistema parte dallo stato s1 (Decollo). Le variabili booleane sono inizializzate in modo che solo decollo sia TRUE, mentre tutte le altre sono FALSE. Questo riflette il fatto che, all'inizio, il drone si trova in fase di decollo. "`next(state)`" definisce tutte le possibili transizioni tra gli stati, utilizzo "`case`" per specificare il prossimo stato. Il valore {s2, s4, s5} indica una scelta non deterministica tra gli stati.

```
ASSIGN
    init(state) := s1;
    init(decollo) := TRUE;
    init(navigazione) := FALSE;
    init(hovering) := FALSE;
    init(consegna) := FALSE;
    init(posizionamento) := FALSE;
    init(atterraggio) := FALSE;

    next(state) :=
        case
            state = s1 : s2;  -- da decollo a navigazione
            state = s2 : s3;  -- da navigazione a hovering
            state = s3 : {s2, s4, s5};  -- da hovering a navigazione, consegna o posizionamento
            state = s4 : s3;  -- da consegna a hovering
            state = s5 : s6;  -- da posizionamento a atterraggio
            state = s6 : s1;  -- da atterraggio a decollo
        esac;
```

- **Aggiornamento delle variabili booleane:** Per ogni proprietà (decollo, navigazione, ecc.) ho definito che essa diventi TRUE nel passo successivo, se il prossimo stato corrisponde a quello associato. Ad esempio: **next(decollo):** è TRUE se il prossimo stato (next(state)) è s1, altrimenti è FALSE. E così per le altre variabili (navigazione, hovering, ecc.), verificando che il prossimo stato corrisponda a quello in cui la proprietà è attiva. Questa parte del codice sincronizza il valore della variabile booleana con lo stato del sistema.

```
next(decollo) :=
    case
        next(state) = s1 : TRUE;
        TRUE : FALSE;
    esac;

next(navigazione) :=
    case
        next(state) = s2 : TRUE;
        TRUE : FALSE;
    esac;
```

- **Verifica delle proprietà:** Le proprietà sono specificate usando le keyword LTLSPEC e CTLSPEC che poi vengono verificate automaticamente dal model checker. Il sistema conferma se ogni proprietà è soddisfatta o fornisce un controesempio.

```
-- Proprietà CTL
CTLSPEC AG(EF decollo)
-- Proprietà LTL
LTLSPEC G(decollo -> X navigazione)
```

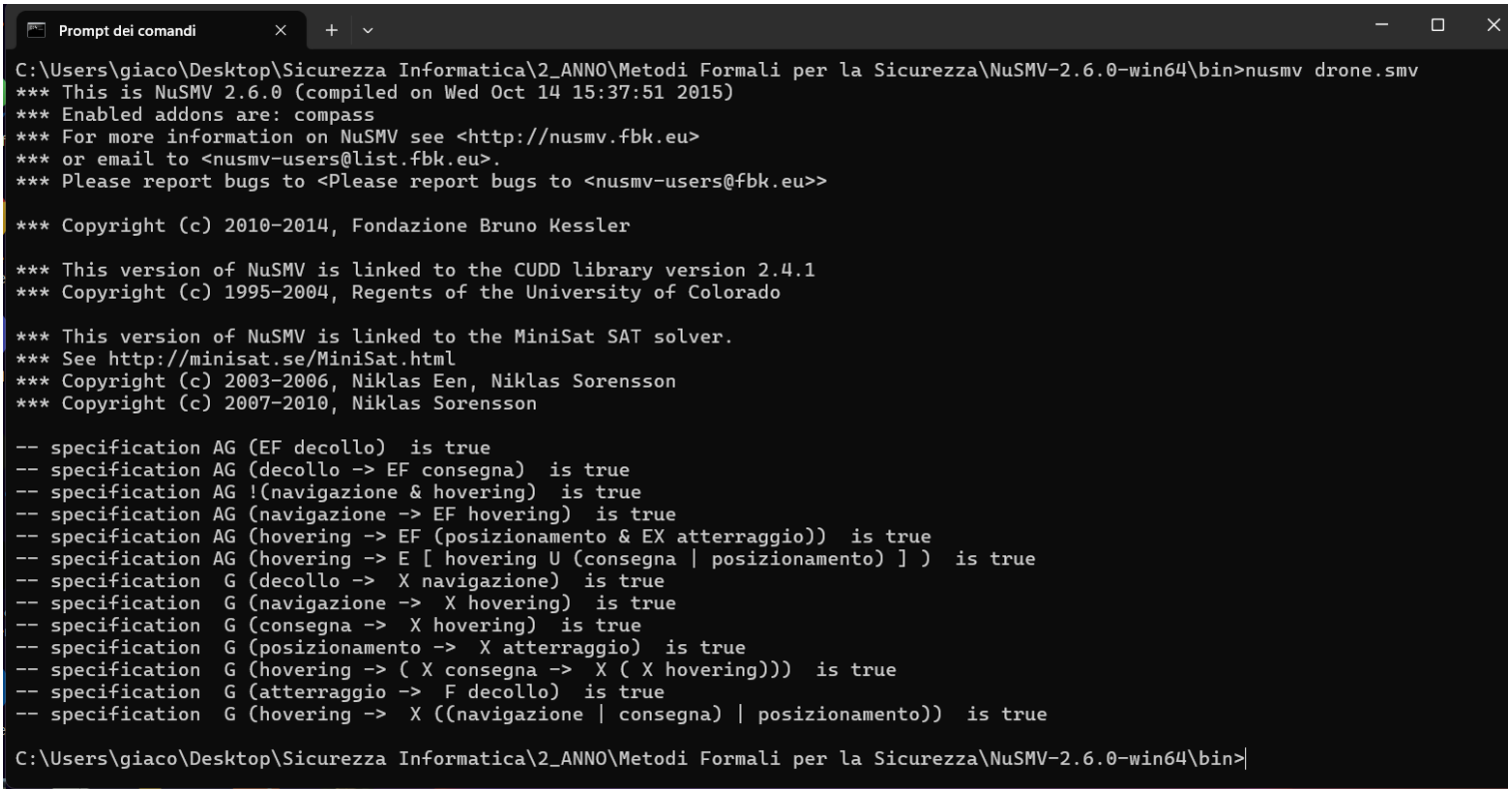
3.5 Casi di non soddisfacimento

Per i casi di non soddisfacimento ho considerato una proprietà LTL e una CTL.

Proprietà LTL considerata: G (decollo -> X navigazione). Immaginiamo di modificare la transizione dallo stato s1 (decollo) in modo che non sia sempre garantito il passaggio a s2 (navigazione). Quindi al passo 0 il sistema parte da s1 (decollo è TRUE). Al passo 1 il sistema sceglie la transizione s1 -> s1 (rimane in decollo) anziché andare in s2 (navigazione). La proprietà è violata perché nel passo 0, essendo in s1 il sistema sa che nel passo successivo (passo 1) deve essere in navigazione. In questo controesempio, nel passo 1 il sistema resta in s1 (decollo) e non passa a navigazione. Perciò, la proprietà iniziale non viene rispettata in quel passaggio, e di conseguenza la proprietà globale (G(…)) fallisce.

Proprietà CTL considerata: AG (hovering -> E (hovering U (consegna V posizionamento))). Immaginiamo di avere un'esecuzione in cui, a partire dallo stato s1 (decollo), il sistema arriva in s3 (hovering) ma poi segue un percorso in cui non raggiunge mai né s4 (consegna) né s5 (posizionamento). Ad esempio, il sistema potrebbe evolvere in questo modo: al passo 0 in s1 (decollo è TRUE), al passo 1 da s1 passa a s2 (navigazione), al passo2 da s2 passa a s3 (hovering), al passo 3 da s3 invece di prendere la transizione verso s4 (consegna) o s5 (posizionamento), il sistema passa direttamente a s6 (atterraggio). Per mostrare il controesempio a partire da s1 (decollo), il percorso seguito è **s1 (decollo) → s2 (navigazione) → s3 (hovering) → s6 (atterraggio) …** in nessun punto del percorso una volta raggiunto hovering, esiste la possibilità di rimanere in hovering fino a raggiungere consegna o posizionamento. La proprietà è violata perché deve essere sempre possibile, da ogni stato in cui il sistema è in hovering, trovare un percorso in cui si rimane in hovering fino a raggiungere consegna o posizionamento. Nel controesempio, una volta in hovering, il sistema va in atterraggio rendendo impossibile soddisfare questa condizione, violando la proprietà.

3.6 Risultati ottenuti



Qui invece mostro i due casi di non soddisfacimento:

```
-- specification  G (decollo ->  X navigazione)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 1.1 <-
    state = s1
    decollo = TRUE
    navigazione = FALSE
    hovering = FALSE
    consegna = FALSE
    posizionamento = FALSE
    atterraggio = FALSE
  -> State: 1.2 <-
```

```
-- specification AG (hovering -> E [ hovering U (consegna | posizionamento) ] )  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 3.1 <-
    state = s1
    decollo = TRUE
    navigazione = FALSE
    hovering = FALSE
    consegna = FALSE
    posizionamento = FALSE
    atterraggio = FALSE
  -> State: 3.2 <-
    state = s2
    decollo = FALSE
    navigazione = TRUE
  -> State: 3.3 <-
    state = s3
    navigazione = FALSE
    hovering = TRUE
```

4 – Conclusioni

Con questo progetto è stato possibile analizzare formalmente il comportamento di un drone per la consegna di farmaci nel territorio del Gargano e delle Isole Tremiti. Con ASP sono riuscito a ottimizzare il percorso considerando i tre fattori fondamentali e soprattutto significativi per questo territorio. Le modellazioni in LTL e CTL mi hanno permesso di verificare formalmente la correttezza delle transizioni tra i vari stati del drone. È importante notare che il modello sviluppato rappresenta una semplificazione del problema reale

Bibliografia

[1] TechTarget and Gavin Wright, “What is traveling salesman problem (TSP)?” [Online]. Available: https://www.techtarget.com/whatis/definition/traveling-salesman-problem

[2] S. Elshater, “TSP - Il problema del Commesso Viaggiatore - Teoria dei grafi,” 2016. doi: 10.13140/RG.2.2.17201.02409.

[3] Britannica, “traveling salesman problem mathematics.” [Online]. Available: https://www.britannica.com/science/combinatorics

[4] Mecalux, “Traveling Salesman Problem e la sua applicazione nella logistica.” [Online]. Available: https://www.mecalux.it/blog/traveling-salesman-problem

[5] S. Elshater and E. Zurich, “TSP-Il problema del Commesso Viaggiatore-Teoria dei grafi”, doi: 10.13140/RG.2.2.17201.02409.

[6] “Potassco guide version 2.2.0”, Accessed: Feb. 08, 2025. [Online]. Available: https://github.com/potassco/guide/releases/

[7] V. Lifschitz, “Programming with CLINGO.”

[8] eliteCONSULTING, “30 cose da sapere su DJI FlyCart 30,” 2023. [Online]. Available: https://www.eliteconsulting.it/30-cose-da-sapere-su-dji-flycart-30/

[9] J. Wang and W. Tepfenhart, “Formal Methods in Computer Science.”

[10] R. Cavada *et al.*, “NuSMV 2.6 User Manual.” [Online]. Available: http://nusmv.fbk.eu.

[11] Math is in the air Blog divulgativo sulla matematica applicata and Martina Gallato, “Breve storia del commesso viaggiatore.”