NavigAltors

Progetto di Intelligenza Artificiale 2

Università degli Studi di Palermo

Ingegneria Informatica | LM-32

Agostino Messina
Alberto Conti
Federico Princiotta Cariddi

Descrizione del progetto

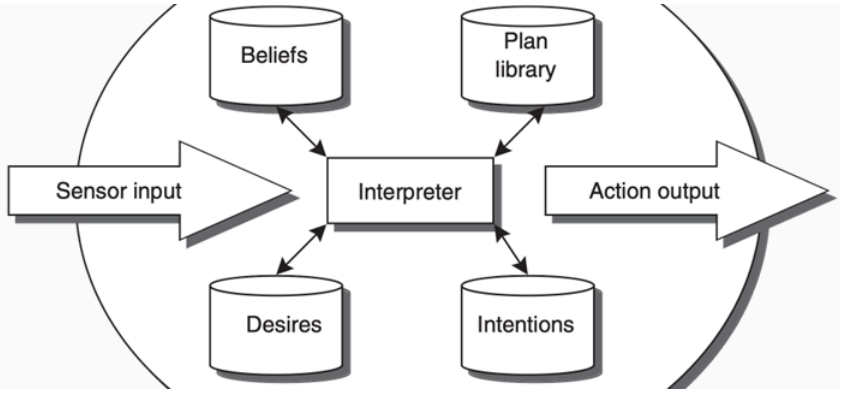
Il progetto NavigAItors ha come scopo la simulazione di una **missione di soccorso** in un contesto di guerra. Viene utilizzato un sistema multi-agente composto da tre unità autonome: un Drone, un Robot di Terra e un Nemico.

Ogni agente segue il modello BDI (Belief-Desire-Intention), che permette loro di prendere decisioni in modo autonomo basandosi sulla conoscenza dell'ambiente circostante.

- **Robot di Terra**: ha come obiettivo primario il salvataggio degli ostaggi.
- **Drone**: supporta la missione identificando la posizione del nemico e trasmettendo le informazioni al robot di terra.
- **Nemico**: cerca di ostacolare il salvataggio degli ostaggi.

Modello BDI e Applicazione nel Progetto NavigAItors

Il modello BDI (Belief-Desire-Intention) è un paradigma teorico che viene utilizzato per la **progettazione di agenti** autonomi in ambienti complessi.



Questo modello si basa su tre componenti principali:

Beliefs

Le credenze rappresentano la **conoscenza** che ogni agente ha dell'ambiente.

Nel progetto NavigAItors, queste credenze includono informazioni come:

- **Posizione degli ostaggi**
- **Ubicazione del nemico**
- **Presenza di ostacoli**
- **Coordinate degli altri agenti**

Ad esempio, il Drone aggiorna continuamente la sua posizione e quella del nemico nella base di conoscenza, mentre il Robot di Terra consulta queste informazioni per pianificare il percorso ottimale schivando gli ostacoli.

Desires

I desideri rappresentano gli **obiettivi** che ciascun agente cerca di raggiungere:

- **Robot di Terra**: Il suo desiderio principale è salvare gli ostaggi. Questo obiettivo guida tutte le sue azioni, come la pianificazione del percorso e l'evitamento del nemico.
- **Drone**: Il suo desiderio è supportare la missione identificando e tracciando il nemico. Questo si traduce in azioni come il pattugliamento dell'area e l'invio di aggiornamenti al Robot di Terra.
- **Nemico**: Il suo desiderio è ostacolare il salvataggio degli ostaggi. Questo si traduce in movimenti strategici per intercettare il Robot di Terra e pattugliare gli ostaggi.

Intentions

Le intenzioni rappresentano le **azioni concrete** che gli agenti eseguono per raggiungere i loro obiettivi. Ad esempio:

- **Robot di Terra**: Utilizza l'algoritmo A* per calcolare il percorso ottimale verso gli ostaggi, evitando ostacoli e il nemico. Se il nemico si avvicina troppo, modifica il percorso per evitare il confronto.
- **Drone**: Segue il nemico e aggiorna la sua posizione in tempo reale nel Knowledge Graph Neo4j. Se il nemico si sposta, il Drone lo traccia e invia aggiornamenti al Robot di Terra.
- **Nemico**: Si muove strategicamente tra gli ostaggi, cercando di intercettare il Robot di Terra. Se un ostaggio viene liberato, si dirige verso quelli ancora in pericolo.

Descrizione Dettagliata degli Agenti

Drone

Pattugliamento

- Il drone esplora la mappa in modo casuale, muovendosi nelle aree non ancora visitate.
- Quando individua il nemico, interrompe la pattuglia e inizia a seguirlo.
- Durante il pattugliamento, aggiorna continuamente la sua KB.

Rilevamento del Nemico

- Il drone utilizza sensori simulati, che verificano la presenza del nemico in un certo raggio d’azione.

Comunicazione con il Robot di Terra

- Il drone comunica con il robot di terra la posizione del nemico, quando individuato.
- Il robot di terra consulta periodicamente queste informazioni per ottimizzare il percorso.

Robot di Terra

Salvataggio degli Ostaggi

- Il robot identifica la posizione degli ostaggi e pianifica il salvataggio in base alla loro posizione e quella del nemico, se conosciuta.

Intercettazione del Nemico

- Se il nemico si avvicina troppo, il robot modifica il percorso per evitare il confronto diretto.

Nemico

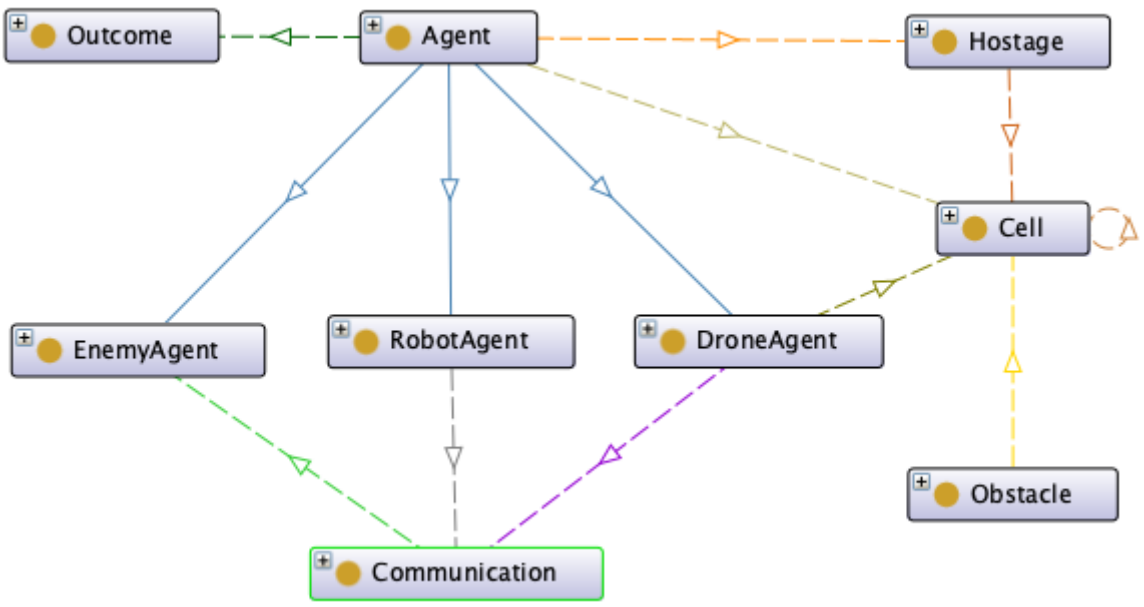
Movimento e Strategia

- Si muove tra gli ostaggi seguendo un percorso per priorità.
- Se un ostaggio è già stato liberato, il nemico lo rimuove dalla conoscenza e si dirige verso quelli ancora in pericolo.

Intercettazione del Robot di Terra

- Se il nemico incontra il robot, va a proteggere l'ostaggio più vicino che presumibilmente il robot sta andando a liberare e se riesce ad avvicinarsi abbastanza può bloccare il robot terminando la simulazione.

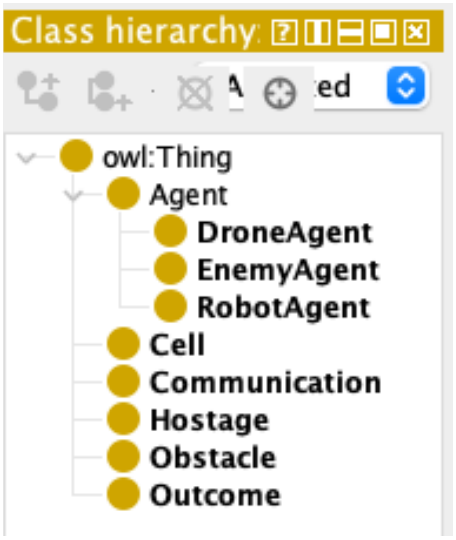
Ontologia



L’ontologia funge da “mappa concettuale” dell’ambiente e fornisce una struttura formale per rappresentare la conoscenza condivisa tra i vari agenti BDI (Drone, Robot di Terra, Nemico).

L’uso di Protégé permette di definire in modo chiaro classi, proprietà, gerarchie e istanze, facilitando la gestione e l’aggiornamento delle informazioni all’interno del Knowledge Graph Neo4j.

Struttura dell’Ontologia



L’ontologia è stata progettata per modellare gli elementi fondamentali della missione di soccorso. In particolare, le **classi principali** sono:

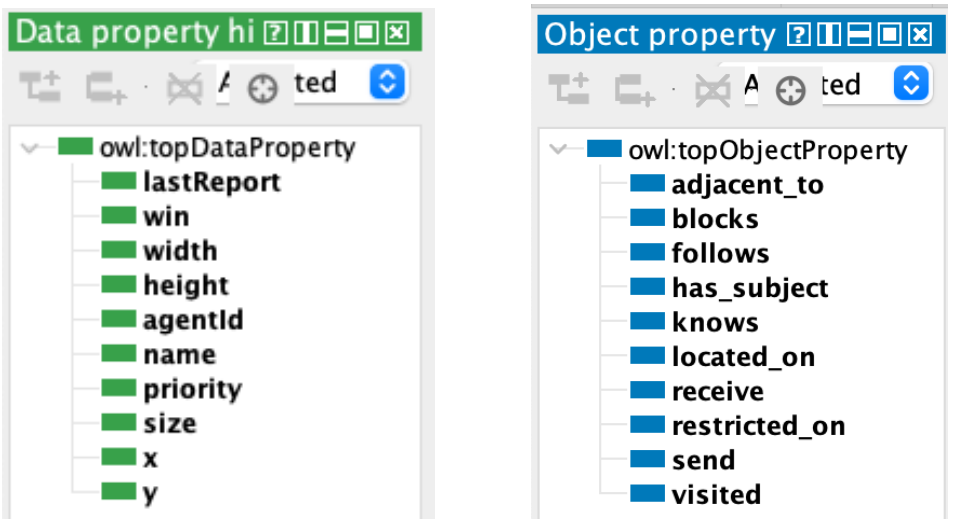
- **Agent**
  Classe generica che rappresenta qualsiasi attore autonomo del sistema. Da essa derivano:
  - **RobotAgent** (il Robot di Terra)
  - **DroneAgent** (il Drone aereo)
  - **EnemyAgent** (il Nemico)
- **Hostage**
  Rappresenta l’ostaggio da salvare, l'obiettivo principale per il Robot di Terra.
- **Cell**
  Identifica un’area all’interno dello scenario di gioco. Può contenere ostacoli, ostaggi o agenti.
- **Obstacle**
  Descrive barriere fisiche che impediscono il movimento degli agenti.
- **Outcome**
  Definisce l’esito della simulazione.

- **Communication**
  Modella lo scambio di informazioni tra gli agenti, fondamentale per coordinare le azioni.

Queste classi riflettono i concetti chiave del dominio: da un lato, gli agenti e i loro obiettivi (Hostage, Outcome), dall'altro, le risorse e le sfide (Cell, Obstacle) che devono essere affrontate o sfruttate nel corso della missione.

Proprietà e Relazioni

Oltre alle classi, l'ontologia include una serie di **object properties** (relazioni) e **data properties** (attributi) per specificare come gli elementi interagiscono tra loro.



Esempi di **object properties** possono essere:

- **located_on**: indica la cella (Cell) in cui si trova un agente.
- **adjacent_to**: descrive la relazione di adiacenza tra celle.
- **blocks**: collega una cella con un ostacolo presente al suo interno.
- **follows**: associa l'agente all'esito finale (successo/fallimento).

Le **data properties**, invece, possono rappresentare attributi numerici o testuali, ad esempio la posizione in coordinate (**x**, **y**), la **priority** di un ostaggio, le dimensioni (**width**, **height**) di un ostacolo o l'id (**agentId**) di un agente.

Utilizzo di Protégé

La definizione dell'ontologia avviene in Protégé, strumento che permette di:

1. **Creare e gestire le classi** e le loro gerarchie.

2. **Definire e documentare le proprietà** (object e data) che descrivono le relazioni e gli attributi.
3. **Verificare la consistenza** dell'ontologia tramite reasoner, assicurando che non esistano contraddizioni nella definizione dei concetti.
4. **Esportare** l'ontologia in formati standard (OWL/RDF) per facilitarne l'integrazione con altri sistemi.

Nello sviluppo del progetto, Protégé è stato usato per mantenere una visione d'insieme di tutti i concetti e le relazioni, fornendo una base solida e facilmente **estendibile** qualora si volessero aggiungere nuove tipologie di agenti, ostacoli o regole di comportamento.

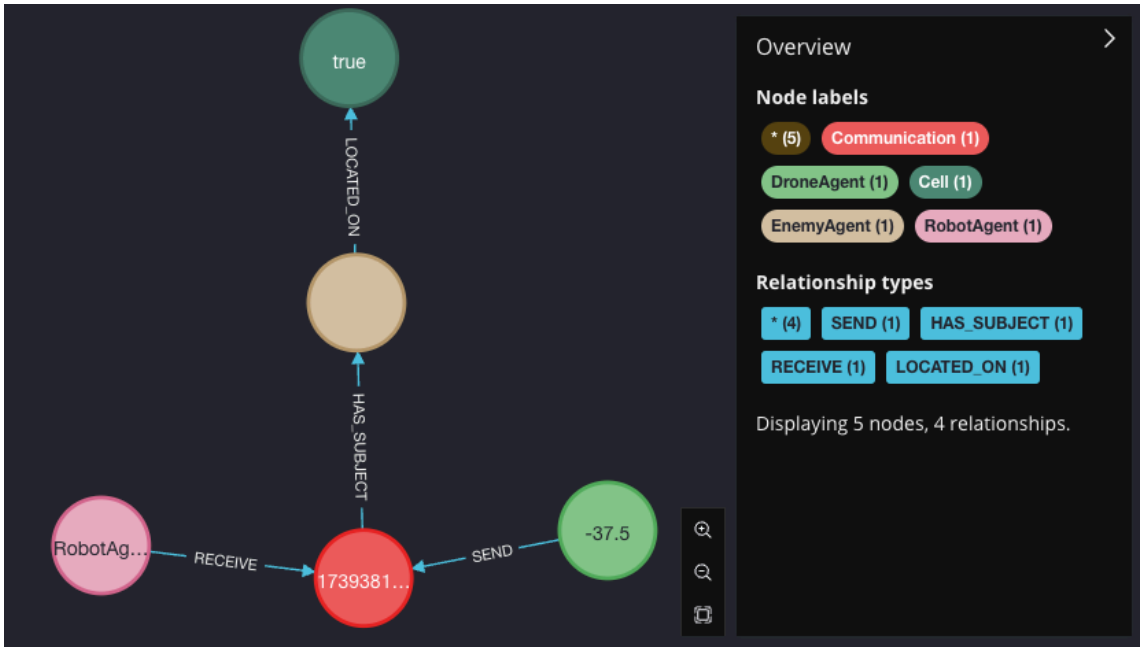
Integrazione con Neo4j e Unity

Una volta definita l'ontologia in Protégé, le classi e le relazioni sono state mappate in **Neo4j** per gestire dinamicamente i dati di runtime:

- **Nodi** di Neo4j corrispondono alle istanze di classi (ad esempio, un singolo DroneAgent o un Hostage specifico).
- **Archi** di Neo4j rappresentano le relazioni tra i nodi (es. visit, located_on, ecc.).

All'interno di **Unity**, gli agenti BDI consultano il grafo per leggere o aggiornare le informazioni (Beliefs). Ad esempio, il Drone modifica le coordinate comunicate del Nemico, mentre il Robot di Terra legge queste stesse coordinate per pianificare il proprio percorso (Desire e Intention).

Approfondimento sulla Comunicazione



La classe **Communication** gioca un ruolo cruciale per il coordinamento tra gli agenti. In particolare:

- **Scopo**: registrare il “messaggio” o l’informazione scambiata (la posizione del Nemico) e tenere traccia di **chi** l’ha inviata e **chi** l’ha ricevuta.
- **Proprietà rilevanti**:
    - **send**: relativamente all’agente che invia l’informazione (DroneAgent).
    - **receive**: riferito all’agente destinatario del messaggio.
    - **has_subject**: descrive il contenuto della comunicazione (coordinate del Nemico).
    - **lastReport**: data property per indicare quando il messaggio è stato inviato.

Principali Operazioni

InitializeGrid

Descrizione

Questa operazione inizializza la griglia di gioco su Neo4j e definisce la disposizione degli ostacoli. In particolare:

- Rimuove eventuali nodi Obstacle già esistenti.
- Per ogni oggetto in Unity con tag “Obstacle”, crea un nodo Obstacle e lo collega alle celle coperte dall’ostacolo tramite la relazione BLOCKS.
- Esegue lo “snap” delle posizioni in modo che corrispondano alle coordinate della griglia.

Pseudocodice

```
FUNCTION InitializeGrid(areaDiameter):
    CONNECT to Neo4j
    // Pulizia ostacoli esistenti
    MATCH (o:Obstacle) DETACH DELETE o

    // Per ogni oggetto Unity con tag "Obstacle":
    FOR each obstacle in scene:
        GET position (x, z)
        SNAP position to nearest cell center
        MERGE obstacle node in Neo4j
        SET attributes: width, height, name, etc.
        // Collegamento con le celle
        FOR each cell coperta dall'ostacolo:
            MERGE (obstacle)-[:BLOCKS]->(cell)
```

ReportEnemyPosition

Descrizione

Permette al Drone di segnalare in tempo reale la posizione del Nemico nel knowledge graph, aggiornando così le credenze del Robot. Vengono utilizzati i nodi di tipo Communication per rappresentare il messaggio e collegarlo ai nodi DroneAgent, RobotAgent ed EnemyAgent.

Pseudocodice

```
FUNCTION ReportEnemyPosition(droneId, enemyPos):
    CONNECT to Neo4j
    // 1) Trova (d:DroneAgent {id: $droneId})
    // 2) Trova (comm:Communication {id: $commId})
    // 3) Setta comm.lastReport timestamp
    // 4) Collega DroneAgent a Communication (SEND)
    // 5) Collega Communication a RobotAgent (RECEIVE)
    // 6) Collega Communication all’EnemyAgent (HAS_SUBJECT)
    // 7) Aggiorna la cella su cui si trova l’enemy (LOCATED_ON)
```

GetEnemyPosition

Descrizione

Questa operazione permette al RobotAgent di recuperare la posizione corrente dell'EnemyAgent dal Knowledge Graph, basandosi sulle comunicazioni ricevute dal DroneAgent.

Pseudocodice

```
FUNCTION GetEnemyPosition():
    CONNECT to Neo4j

    // 1) Trova (r: RobotAgent) che riceve (RECEIVE) un Communication (comm)
    //    il quale è inviato (SENDS) da un DroneAgent
    MATCH (r: RobotAgent)-[:RECEIVE]->(comm:
Communication)<-[:SENDS]-(:DroneAgent)

    // 2) comm ha come messaggio (HAS_SUBJECT) l'EnemyAgent (e)
    MATCH (comm)-[:HAS_SUBJECT]->(e: EnemyAgent)

    // 3) L'EnemyAgent e si trova su una cella (c:Cell) tramite LOCATED_ON
    MATCH (e)-[:LOCATED_ON]-(c:Cell)

    // 4) Restituisce la posizione della cella
    RETURN c.x AS x, c.z AS z
```

FreeHostage

Descrizione

Questa operazione "libera" effettivamente un ostaggio rimuovendo la relazione RESTRICTED_ON che lo collega a una determinata cella. In tal modo, si aggiorna lo stato dell'ostaggio nel database, segnalando che non è più vincolato a quella posizione (quindi è stato soccorso con successo).

Pseudocodice

```
FUNCTION FreeHostage(pos):
    CONNECT to Neo4j
    // 1) Trova l'ostaggio associato alla cella (c:Cell) nelle coordinate pos.x, pos.z
    MATCH (h:Hostage)-[r:RESTRICTED_ON]->(c:Cell {x: pos.x, z: pos.z})
    // 2) Rimuove la relazione r
    DELETE r
```

Selezione dell'Ostaggio da Salvare

Descrizione

Il Robot decide quale ostaggio salvare in base alla conoscenza (o meno) della posizione del Nemico. Se la posizione è sconosciuta, preferisce l'ostaggio più vicino al Drone (considerato un'area "sicura"). Se invece la posizione del Nemico è nota, cerca l'ostaggio più lontano dal Nemico, in modo da minimizzare il rischio di scontro.

Pseudocodice

```
FUNCTION ChooseHostagesToRescue():
    IF enemyPosition is unknown:
        // Nemico sconosciuto: ostaggio più vicino al Drone
        RETURN ChooseClosestHostageToDrone()
    ELSE:
        // Nemico noto: ostaggio più "sicuro" (lontano dal Nemico)
        RETURN ChooseSafeHostage()

FUNCTION ChooseClosestHostageToDrone():
    IF hostagesToRescue is empty:
        RETURN currentPosition // se non ci sono ostaggi, resta fermo

    chosen = null
    minDist = +∞
    FOR each hostage IN hostagesToRescue:
        d = distance(dronePosition, hostage)
        IF d < minDist:
            minDist = d
            chosen = hostage
    RETURN chosen

FUNCTION ChooseSafeHostage():
    IF hostagesToRescue is empty:
        RETURN currentPosition

    chosen = null
    bestScore = -∞
    FOR each hostage IN hostagesToRescue:
        // Più è distante dal Nemico, più è "sicuro"
        riskScore = distance(enemyPosition, hostage)
        IF riskScore > bestScore:
            bestScore = riskScore
            chosen = hostage
```

```
    RETURN chosen
```

Verifica della presenza del Nemico sul percorso

Descrizione

Durante l'esecuzione della missione di salvataggio, il *RobotAgent* deve costantemente controllare che il percorso calcolato non venga intercettato dal *Nemico*. In caso contrario, il Robot effettua un ricalcolo del tragitto e cambia obiettivo, per evitare il confronto diretto. L'algoritmo:

1. **Calcola il percorso** verso l'ostaggio selezionato.
2. **Itera** sui nodi di questo percorso e **misura la distanza** tra ogni nodo e la posizione del *Nemico*.
3. Se almeno un nodo risulta troppo vicino (sotto una soglia di "sicurezza"), il Robot considera il percorso "minacciato" e avvia un **ricalcolo**.

Pseudocodice

```
FUNCTION IsEnemyThreateningPath(enemyPosition, currentPath, threatDistance):
    IF enemyPosition is null:
        RETURN false

    FOR each node IN currentPath:
        IF distance(enemyPosition, node) < threatDistance:
            RETURN true
    RETURN false

FUNCTION RescueCoroutine():
    WHILE not isGameOver:
        IF currentPath is empty OR rescuePoint is zero:
            // Calcola nuovo percorso
            currentPath = FindShortestPath(robotId, currentPosition, rescuePoint)

        IF IsEnemyThreateningPath(enemyPosition, currentPath, threatDistance):
            currentState = RobotState.Recalculating
            // Reset rescuePoint
            rescuePoint = Vector3.zero
            CONTINUE  // riparte a ricalcolare

        // Se il nemico non minaccia il percorso, prosegui
        MoveAlongPath(currentPath)
        ...
        // se arrivi vicino all'ostaggio, lo liberi (FreeHostage)
```

Conclusioni

Il progetto **NavigAItors** dimostra l'efficacia dell'implementazione di un sistema multi-agente per simulare una missione di soccorso in contesto bellico, utilizzando il modello BDI per permettere agli agenti di prendere **decisioni** autonome basate sulla **conoscenza** dell'ambiente. L'integrazione di strumenti avanzati come Protégé per la definizione dell'ontologia e Neo4j per il knowledge graph ha permesso di creare una soluzione integrata e funzionale che risponde a complesse esigenze operative.

Un aspetto fondamentale del progetto è stata la **collaborazione** di gruppo. Questo approccio collaborativo ha non solo facilitato la divisione del lavoro e l'integrazione delle diverse parti del progetto, ma ha anche arricchito l'esperienza di **apprendimento** di ciascun membro del team, permettendo uno scambio di idee e soluzioni che ha significativamente contribuito al **successo** del progetto.