

# Relazione Intelligenza Artificiale

*Università degli studi di Brescia*

THOMAS CAUSETTI 731077  
GABRIELE CERESARA 729324  
JACOPO TEDESCHI 722843

*A.A. 2025-2026*

Gennaio-Febbraio 2026

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
<b>2</b>	<b>Scelta degli algoritmi</b>	<b>3</b>
2.1	Preprocessing . . . . .	3
2.2	Large Neighborhood Search (LNS) . . . . .	5
2.2.1	Principio di funzionamento . . . . .	5
2.2.2	Ottimizzazione . . . . .	5
2.3	Lazy Constraints Addition Search for MAPF (LaCAM) . . . . .	6
2.3.1	Principio di funzionamento . . . . .	6
2.3.2	Implementazione di LaCAM2 nel Progetto . . . . .	6
2.4	Priority Inheritance with Backtracking (PIBT) . . . . .	7
2.4.1	Gerarchia delle Euristiche . . . . .	7
2.4.2	Selezione dei Candidati e Tie-Breaking . . . . .	8
2.4.3	Risoluzione dei Conflitti e Ricorsione . . . . .	8
2.4.4	Validazione Post-Pianificazione (Move Check) . . . . .	8
2.5	Rolling Horizon Collision Resolution (RHCR) . . . . .	9
2.5.1	Principio di funzionamento . . . . .	9
2.6	Implementazione e Launcher . . . . .	10
<b>3</b>	<b>Risultati</b>	<b>13</b>
<b>4</b>	<b>Conclusione</b>	<b>15</b>

# Capitolo 1

## Introduzione

Questo progetto si basa sull'implementazione di un sistema di Planning per il problema che è stato presentato nella competizione LORR24 (League of Robot Runners 2024) url: <https://2024.leagueofrobotrunners.org/>. Questa competizione è dedicata al problema del Multi-Agent Path Finding (MAPF) in ambienti dinamici. In questo contesto, più agenti robotici devono navigare in una griglia bidimensionale, evitando collisioni sia di vertice che di spigolo, mentre completano una sequenza di compiti assegnati dinamicamente. Il sistema deve gestire vincoli temporali stringenti, con limiti di tempo per la pianificazione in ogni timestep, e ottimizzare metriche come il tempo di completamento dei compiti e l'efficienza complessiva.

L'approccio implementato in questo progetto è un planner ibrido che combina quattro algoritmi principali:

- **Large Neighborhood Search (LNS)**: ottimizzazione delle traiettorie a lungo termine.
- **Lazy Constraints Addition Search for MAPF (LaCAM)**: Utilizzato come preprocessing per LNS.
- **Priority Inheritance with Backtracking (PIBT)**: risoluzione rapida dei conflitti a breve termine. (Solo come fallback in caso di fallimento di LNS)
- **Rolling Horizon Collision Resolution (RHCR)**: gestione dinamica dei conflitti in tempo reale. (Solo come fallback in caso di fallimento di LNS)

Utilizzando diversi algoritmi in combinazione, l'implementazione mira ad ottenere i vantaggi di ogni metodo, migliorando l'efficacia complessiva del sistema di pianificazione.

Nei capitoli successivi, i 4 diversi algoritmi saranno descritti in dettaglio, insieme alle scelte implementative e alle strategie adottate per affrontare le sfide specifiche del problema MAPF in ambienti dinamici. Nel seguito della relazione, descriveremo l'architettura del sistema, i dettagli implementativi, i risultati sperimentali e le valutazioni ottenute nella competizione.

# Capitolo 2

## Scelta degli algoritmi

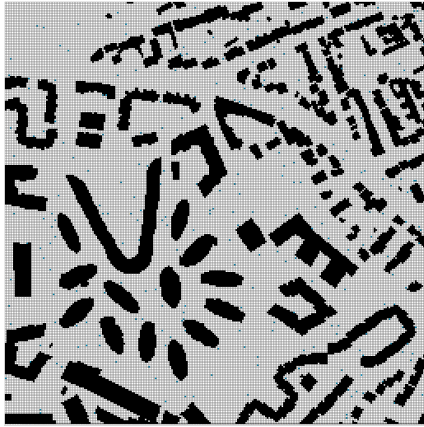
Gli algoritmi scelti per l'implementazione del planner ibrido sono stati selezionati effettuando una valutazione delle varie implementazioni di diversi team della competizione sia nell'anno corrente che in quelli passati. Studiando le varie soluzioni proposte, abbiamo notato come l'approccio ibrido che combina più algoritmi abbia portato a risultati migliori rispetto all'utilizzo di un singolo algoritmo. Infine per implementare il planner abbiamo cercato di capire come implementare gli algoritmi scelti con paper pubblicati in letteratura e con le implementazioni open source disponibili.

### 2.1 Preprocessing

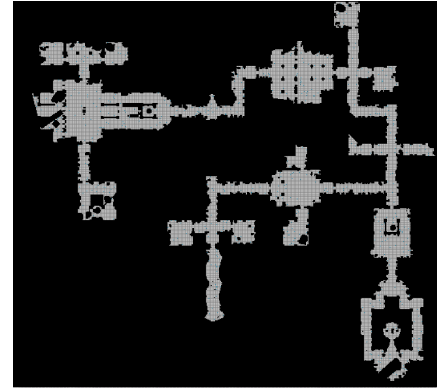
La competizione fornisce un tempo per il preprocessing, che il nostro planner utilizza per precalcolare le euristiche pesate tra tutte le coppie di posizioni sulla mappa. Questo processo sfrutta una ricerca  $A^*$  a più livelli che inizializza rapidamente tutte le euristiche con la distanza di Manhattan, per poi raffinarle progressivamente utilizzando il tempo disponibile. Il calcolo avviene in tre fasi: prima l'inizializzazione con distanza di Manhattan (molto veloce), poi un raffinamento  $A^*$  senza considerare le rotazioni (più veloce), ed infine un raffinamento completo con  $A^*$  che considera anche le orientazioni (più accurato). Il preprocessing è parallelizzato con OpenMP per sfruttare al meglio i core disponibili.

Il calcolo è stato fatto a più livelli imitando un pò l'approccio Anytime, perchè dato che il tempo di preprocessing è limitato, e il software successivo necessita di queste euristiche, nei problemi più complessi, il programma non riesce a completare il calcolo di tutte le euristiche, ma riesce comunque a fornire al software successivo delle euristiche iniziali, raffinate per tutto il tempo disponibile. In questo modo, si garantisce la presenza di un euristica e si rende impossibile il timeout del preprocessing, anche in scenari complessi. Per scenari semplici l'overhead fornito non è notevole, dato che il calcolo delle euristiche di primo e secondo livello è molto veloce.

Durante i test per avere i risultati finali ci si è accorti che per run lunghe nelle mappe `paris_1_256_250.json` e `brc202d_500.json`, il codice aveva performance nettamente inferiori a quanto pronosticato. Dopo un'attenta fase di debugging ci si è trovato il problema, che era più semplice del previsto. Le mappe `paris_1_256_250.json` e `brc202d_500.json` non sono mappe di forma rettangolare come le altre, ma seguono strutture prese dalla vita reale, come la mappa delle vie di Parigi. In questo contesto l'euristica di Manhattan, utilizzata come failsafe, come si evince anche da teoria a risultati pessimi. Questo si è



(a) Paris\_1\_256\_250.json



(b) Brc202d\_500.json

Figura 2.1: Le due mappe menzionate

notato soprattutto utilizzando il visualizzatore PlanViz, che ha permesso di vedere che gli agenti si bloccavano sui bordi della mappa. La soluzione al problema è stata quella di utilizzare per le 2 mappe citate in precedenza BFS (dal goal) come euristica di failsafe invece di Manhattan.

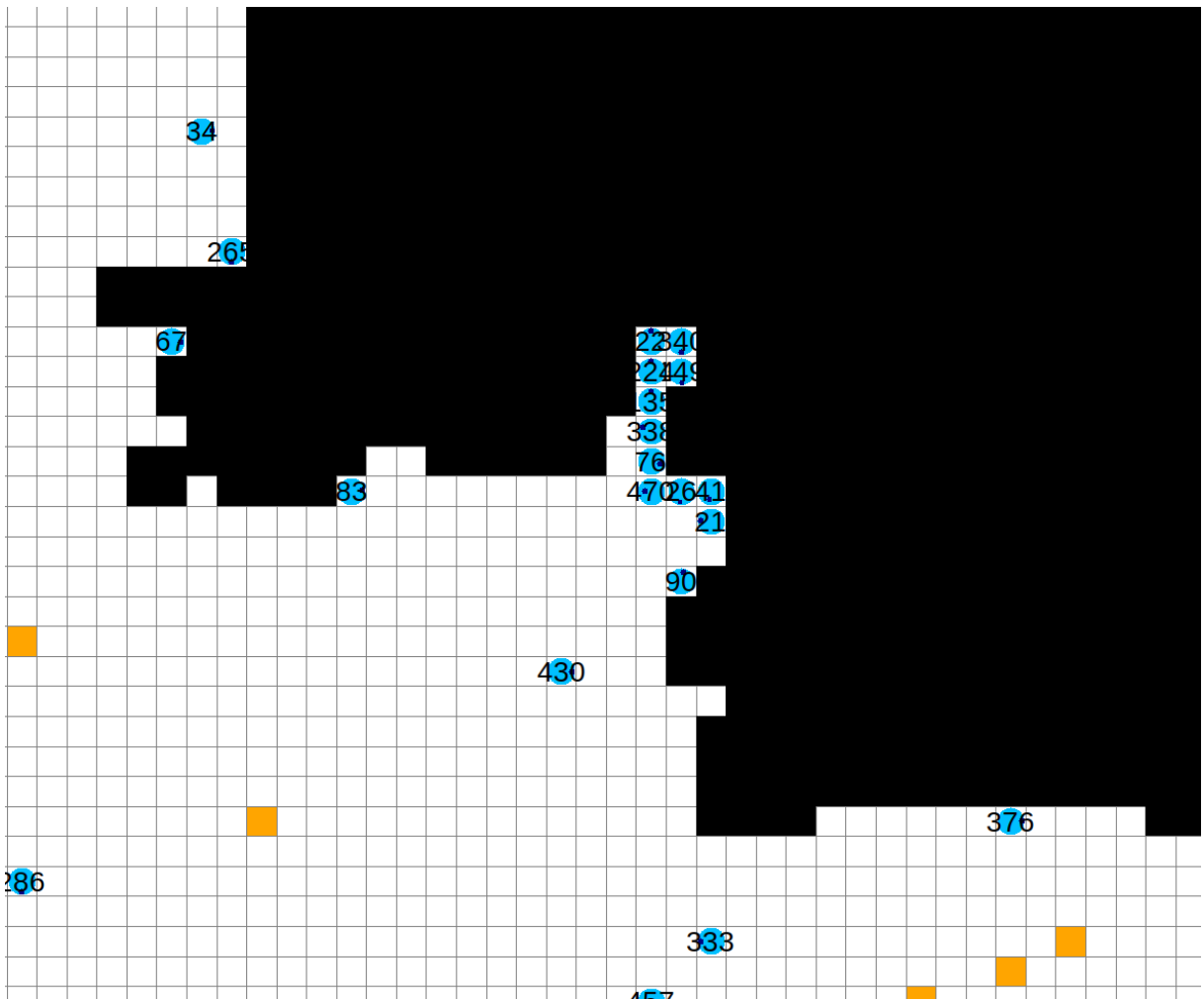


Figura 2.2: Il Problema visto in PlanViz

## 2.2 Large Neighborhood Search (LNS)

### 2.2.1 Principio di funzionamento

Il principio di funzionamento del Large Neighborhood Search (LNS) applicato al Multi-Agent Pathfinding (MAPF) si basa su un approccio metaeuristico che combina distruzione e riparazione iterativa delle soluzioni. L'obiettivo è quello di ottimizzare i percorsi di più agenti in un ambiente condiviso, minimizzando i conflitti e i costi complessivi.

L'algoritmo LNS inizia con una soluzione iniziale, che nel caso di questo progetto viene calcolata utilizzando LaCAM. Successivamente, una parte della soluzione viene distrutta selezionando un sottoinsieme di agenti e rimuovendo i loro percorsi. Questa selezione può essere effettuata utilizzando diverse euristiche, come la scelta casuale, la selezione degli agenti più ritardati o quelli che si trovano in posizioni di conflitto.

Una volta distrutta parzialmente la soluzione, l'algoritmo tenta di ripararla ricalcolando i percorsi per gli agenti selezionati. Questo processo di riparazione utilizza un planner, come il Time-Space A\* con consapevolezza delle orientazioni, che considera i vincoli temporali e spaziali per evitare conflitti. L'algoritmo valuta la nuova soluzione e decide se accettarla in base a criteri di ottimalità o accettazione probabilistica.

L'implementazione del LNS è strutturata in diverse componenti principali:

- **PathT e PathTableWC:** Queste strutture gestiscono l'occupazione spazio-temporale e il rilevamento dei conflitti. Consentono di registrare e annullare i percorsi degli agenti, verificare blocchi e calcolare conflitti futuri.
- **ConstraintTable:** Questa classe gestisce i vincoli rigidi e morbidi, come i blocchi di movimento e le penalità, per garantire che i percorsi rispettino i vincoli definiti.
- **ReservationTable:** Utilizzata per calcolare intervalli di tempo sicuri per gli agenti, considerando i vincoli e i conflitti esistenti.
- **TimeSpaceAStarPlanner:** Un planner avanzato che utilizza una variante del classico A\* per trovare percorsi ottimali minimizzando i conflitti. Tiene conto delle orientazioni degli agenti e dei vincoli temporali.
- **Parallel LNS:** Una versione parallela dell'algoritmo che sfrutta il calcolo multi-thread per migliorare le prestazioni, suddividendo il problema in sottoproblemi gestiti indipendentemente.

L'algoritmo è stato progettato per essere modulare e flessibile, consentendo di adattare le euristiche di distruzione e riparazione in base alle esigenze specifiche del problema. L'uso di strutture dati efficienti, come le heap per le code di priorità e le tabelle hash per il rilevamento dei conflitti, garantisce prestazioni elevate anche in scenari complessi con molti agenti.

### 2.2.2 Ottimizzazione

Per ottimizzare ulteriormente le prestazioni del LNS si è fatto profiling del codice con perf, per identificare eventuali colli di bottiglia. L'heap utilizzata per la coda di priorità è stata modificata da una versione 2 heap iniziale a una 4 heap, che ha portato a un

miglioramento significativo delle prestazioni. Inoltre, sono state implementate versioni parallele del LNS, sfruttando il calcolo multi-thread per gestire sottoproblemi in modo indipendente, riducendo così i tempi di esecuzione complessivi.

Inoltre si è modificato rispetto a una versione iniziale di LNS la gestione del tempo massimo di esecuzione. Dato che ogni timestep ha un limite di tempo di 1 secondo, inizialmente si impostava un limite di 0.9 secondi per l'esecuzione del LNS. Questa versione funzionava bene per istanze più semplici, ma per istanze più complesse, si notava che l'algoritmo andava in timeout per le prime  $n$  iterazioni, stabilizzandosi successivamente. Successivamente, si è capito che per istanze più complesse, il tempo di esecuzione dello scheduler, soprattutto nella fase iniziale, richiedeva più tempo.

Infine il problema è stato risolto modificando il funzionamento del calcolo del tempo massimo, non più basato su un tempo fisso, ma calcolato dinamicamente in base al tempo impiegato dallo scheduler. In questo modo, se lo scheduler utilizza 0.4 secondi, ad esempio, viene utilizzato il tempo rimanente, ovvero 0.6 secondi, come valore massimo per il calcolo del tempo di esecuzione del LNS. Il tempo di esecuzione verrà calcolato utilizzando il cutoff time value presente in config.hpp, che indica quanto tempo, del tempo rimanente, viene utilizzato per esecuzione del LNS. (Il cutoff time è settato a circa 0.9 secondi, dato che si vuole lasciare un margine di tempo per eventuali operazioni di post processing e anche perché il controllo che ferma l'esecuzione viene lanciato solo ogni  $n$  iterazioni altrimenti diventerebbe un collo di bottiglia).

## 2.3 Lazy Constraints Addition Search for MAPF (LaCAM)

### 2.3.1 Principio di funzionamento

Il principio di funzionamento di LaCAM (Lazy Constraints Addition Search for MAPF) si basa sull'idea di risolvere il problema del Multi-Agent Path Finding (MAPF) in modo incrementale, aggiungendo vincoli in modo "lazy" solo quando necessario. L'approccio è progettato per gestire efficacemente i conflitti tra agenti, evitando di dover considerare tutte le possibili combinazioni di traiettorie fin dall'inizio. Nel caso di questo progetto, LaCAM è stato utilizzato come fase di preprocessing per ottimizzare le traiettorie a lungo termine prima di applicare l'algoritmo LNS (Large Neighborhood Search) per ulteriori ottimizzazioni.

### 2.3.2 Implementazione di LaCAM2 nel Progetto

L'implementazione di LaCAM nel progetto rappresenta un adattamento sostanziale dell'algoritmo originale per gestire le specifiche sfide del competizione MAPF (MultiAgent Path Finding).

L'architettura Generale è un codice strutturato su tre livelli principali. Al livello più basso troviamo la rappresentazione del grafo e delle configurazioni, implementata nei file lacam2\_core.cpp e lacam2\_core.hpp. Questo livello gestisce la struttura dati fondamentale: un grafo dove ogni vertice rappresenta una posizione valida nell'ambiente e mantiene riferimenti ai suoi vicini, permettendo una navigazione efficiente dello spazio. La classe

Config memorizza la configurazione completa di tutti gli agenti in un dato istante, includendo non solo le posizioni ma anche gli orientamenti e lo stato di completamento dei task.

Il livello intermedio, contenuto in `lacam2_planner.cpp`, implementa il cuore dell'algoritmo di ricerca. Qui troviamo la struttura a due livelli caratteristica di LaCAM: i nodi di alto livello (HNode) rappresentano configurazioni complete di tutti gli agenti, mentre i nodi di basso livello (LNode) esplorano le possibili mosse individuali all'interno di una configurazione. Questa struttura permette di esplorare lo spazio delle soluzioni in modo efficiente, evitando di generare esplicitamente tutte le possibili combinazioni di mosse.

Il codice usa l'algoritmo PIBT (Priority Inheritance with Backtracking) come meccanismo di risoluzione dei conflitti a basso livello. Quando generiamo una nuova configurazione, non tutti gli agenti hanno una mossa pianificata dalla ricerca di basso livello. Per questi agenti, PIBT trova una mossa valida considerando le priorità degli agenti e propagando i vincoli. Da notare che questo PIBT non è lo stesso codice fornito dalla competizione, ma un'implementazione parziale necessaria a lacam.

Questa implementazione rappresenta quindi un adattamento completo di LaCAM alle specificità della competizione, bilanciando fedeltà all'algoritmo originale e necessità pratiche di integrazione con il sistema di valutazione.

## 2.4 Priority Inheritance with Backtracking (PIBT)

La gestione della navigazione reattiva e la risoluzione dei conflitti immediati sono affidate all'algoritmo *Priority Inheritance with Backtracking* (PIBT). È fondamentale precisare che l'implementazione descritta in questa sezione corrisponde al solver di *default* fornito dagli organizzatori della competizione League of Robot Runners 2024. Nel contesto del nostro progetto, questo modulo svolge un duplice ruolo: funge da *benchmark* per valutare le prestazioni delle nostre ottimizzazioni (LNS e LaCAM) e agisce come meccanismo di *fallback* (rete di sicurezza) per garantire la validità dei movimenti quando la pianificazione globale non è disponibile o incompleta.

L'algoritmo opera in modo decentralizzato: a ogni timestep, ogni agente calcola la propria mossa basandosi sul proprio stato locale e interagendo con i vicini immediati. L'implementazione specifica fornita (denominata nel codice sorgente `DefaultPlanner`) adotta una variante "Causal PIBT", adattata per gestire i vincoli cinematici del modello MAPF-T (che include l'orientamento).

### 2.4.1 Gerarchia delle Euristiche

Per selezionare la mossa migliore tra i candidati possibili (muoversi nelle 4 direzioni cardinali o attendere), l'implementazione di default utilizza una funzione di valutazione gerarchica che assegna un costo a ciascuna cella target. L'ordine di priorità rilevato nel codice è il seguente:

1. **Distanza dal Percorso Guida (Trajectory Distance):** Se il planner di alto livello (LNS) ha prodotto una traiettoria valida, PIBT utilizza la distanza da tale percorso come euristica primaria. Questo permette all'algoritmo reattivo di allinearsi con la strategia globale.

2. **Tabelle Euristiche Pre-calcolate:** In assenza di una traiettoria guida, il sistema consulta tabelle di lookup (calcolate durante il preprocessing tramite BFS inversa) che forniscono la distanza reale dagli obiettivi statici, tenendo conto della topologia della mappa.
3. **Distanza di Manhattan:** Come ultima risorsa (fallback), se non sono disponibili né traiettorie né tabelle, viene utilizzata la semplice distanza di Manhattan.

### 2.4.2 Selezione dei Candidati e Tie-Breaking

Una volta generati i successori validi (escludendo muri e ostacoli statici), questi vengono ordinati in base al valore euristico crescente. Tuttavia, in ambienti a griglia, è comune che più celle abbiano lo stesso valore euristico. L'implementazione di default gestisce questi casi con una strategia di *tie-breaking* a due livelli, cruciale per la fluidità del movimento:

- **Inerzia e Forward Bias:** A parità di euristica, viene data priorità assoluta alla mossa che segue l'orientamento attuale dell'agente. Se un agente è orientato verso Est e la cella a Est ha lo stesso costo euristico di quella a Nord, l'agente sceglierà Est. Questo riduce drasticamente le rotazioni, che sono costose in termini di tempo.
- **Randomizzazione:** Se persiste la parità (es. due celle laterali con stesso costo), viene utilizzato un valore casuale (`rand()`) assegnato a ogni candidato per rompere la simmetria e prevenire *livelock* (cicli infiniti tra agenti).

### 2.4.3 Risoluzione dei Conflitti e Ricorsione

Il cuore dell'algoritmo è la gestione dei conflitti tramite ereditarietà della priorità. Quando un agente *A* tenta di selezionare il suo miglior candidato:

1. Se la cella target è libera, *A* la occupa.
2. Se la cella è occupata da un agente *B* che ha già pianificato la mossa per questo turno, *A* scarta il candidato (poiché è bloccato).
3. Se la cella è occupata da un agente *B* che **non ha ancora pianificato**, *A* "spinge" *B*. L'algoritmo viene chiamato ricorsivamente su *B*, passandogli *A* come "agente a priorità superiore" (*higher\_id*). Questo impedisce a *B* di muoversi verso la posizione attuale di *A* (prevenendo scambi diretti o *swap conflicts*) e lo forza a cercare una cella libera.

Se la chiamata ricorsiva fallisce (cioè *B* non può muoversi), *A* esegue il **backtracking**, scartando la scelta ottima e provando il secondo miglior candidato, fino all'azione di attesa garantita.

### 2.4.4 Validazione Post-Pianificazione (Move Check)

Una particolarità dell'implementazione di default per il modello MAPF-T è la fase di validazione successiva alla pianificazione, gestita dalla funzione `moveCheck`. Poiché le rotazioni non sono istantanee, si possono creare situazioni di stallo nei "treni" di agenti. Se un agente leader decide di ruotare (azione che non libera la cella corrente), ma l'agente che lo segue ha pianificato di muoversi in avanti (`Action::FW`) occupando quella cella, si

verificherebbe una collisione. La funzione `moveCheck` risolve questo problema propagando lo stato di attesa all'indietro: se un agente non libera la cella (perché ruota o attende), forza l'agente che intendeva occuparla a convertire la sua azione in *Wait*.

## 2.5 Rolling Horizon Collision Resolution (RHCR)

### 2.5.1 Principio di funzionamento

RHCR (Rolling-Horizon Collision Resolution) è un algoritmo utilizzato per la risoluzione di problemi di lifelong MAPF (Multi-Agent Path Finding), una particolare famiglia di problemi MAPF in cui, oltre a calcolare per ciascun agente un piano che consenta di raggiungere una posizione obiettivo a partire da una posizione iniziale evitando collisioni, si introduce l'ulteriore complessità di dover gestire obiettivi non noti a priori, assegnati dinamicamente agli agenti.

RHCR consente di risolvere questa istanza di problemi scomponendola in una sequenza di problemi di *Windowed MAPF* e ricalcolando tutti i piani ogni  $h$  intervalli temporali. Il *Windowed MAPF* si differenzia da un classico problema MAPF per due motivi principali:

- a ciascun agente può essere assegnata una lista di goal all'interno della stessa risoluzione;
- le collisioni devono essere risolte solo per i successivi  $w$  intervalli temporali.

Questa variante viene utilizzata in quanto, a causa della natura dinamica degli obiettivi, consente di mantenere gli agenti costantemente impegnati, minimizzandone il tempo di inattività; inoltre, risolvere le collisioni su intervalli temporali troppo lunghi risulterebbe inefficace, poiché a ogni nuova assegnazione di un obiettivo sarebbe necessario ricalcolare l'intero piano.

#### Definizione del problema

L'input dell'algoritmo è un grafo  $G = (V, E)$ , dove  $V$  rappresenta l'insieme delle posizioni occupabili dagli agenti ed  $E$  l'insieme delle connessioni tra posizioni adiacenti. È inoltre dato un insieme di  $m$  agenti  $a_1, a_2, \dots, a_m$ , ciascuno con una propria posizione iniziale.

Le posizioni obiettivo non sono note a priori, ma vengono assegnate dinamicamente agli agenti da uno scheduler esterno all'algoritmo. Il tempo è discretizzato in intervalli temporali. A ogni intervallo, ciascun agente può muoversi verso una posizione adiacente rispetto a quella corrente oppure rimanere in attesa; entrambe le azioni hanno durata di un'unità di tempo.

Una *collisione* può verificarsi in due casi:

- due agenti occupano la stessa posizione nello stesso istante temporale;
- due agenti attraversano lo stesso arco in direzioni opposte nello stesso istante temporale.

L'obiettivo è generare un piano privo di collisioni che consenta di muovere tutti gli agenti verso le rispettive posizioni obiettivo massimizzando il *throughput*, definito come il numero medio di posizioni obiettivo raggiunte per intervallo temporale. Il piano è quindi rappresentato da un insieme di percorsi senza collisioni, uno per ciascun agente.

## Descrizione dell'algoritmo

RHCR utilizza due parametri definiti dall'utente: l'orizzonte temporale  $w$ , per il quale viene garantita l'assenza di collisioni, e il periodo  $h$  (espresso in unità temporali) dopo il quale i piani di tutti gli agenti vengono ricalcolati. È importante notare che  $w$  deve essere maggiore o uguale a  $h$ , al fine di evitare la possibilità di collisioni.

A ogni ricalcolo dei piani, eseguito a un generico istante temporale  $t$ , l'algoritmo:

- aggiorna la posizione iniziale di ciascun agente con la sua posizione reale;
- aggiorna la lista degli obiettivi di ogni agente sulla base della lista di goal corrente, calcolando  $d$ , ovvero il numero minimo di intervalli temporali necessari all'agente per raggiungere tutti i propri obiettivi. In altre parole,  $d$  rappresenta la somma delle distanze, a partire dalla posizione iniziale, per raggiungere tutti i goal assegnati. Se  $d < h$ , l'agente potrebbe completare i propri obiettivi prima del successivo ricalcolo dei piani; in tal caso, RHCR assegna nuovi goal fino a quando  $d$  risulta maggiore di  $h$ , in modo da garantire che l'agente rimanga impegnato per i successivi  $h$  intervalli temporali.

Una volta determinate le posizioni iniziali e gli obiettivi di tutti gli agenti, l'algoritmo calcola un piano che consenta di visitare tutti i goal nell'ordine corretto, garantendo l'assenza di collisioni per i primi  $w$  intervalli temporali. Successivamente, gli agenti eseguono il piano calcolato per i successivi  $h$  intervalli temporali; tutti i goal raggiunti durante questi  $h$  step vengono rimossi dalla lista degli obiettivi.

Completata questa fase, l'algoritmo ripete l'intera procedura per gestire i nuovi obiettivi che, nel frattempo, vengono assegnati agli agenti dallo scheduler.

In conclusione, affinché RHCR possa essere applicato efficacemente a problemi di lifelong MAPF, è necessaria una modifica al problema MAPF classico, introducendo la variante di Windowed MAPF. Tale estensione consente di limitare la risoluzione delle collisioni a una finestra temporale di ampiezza finita e di gestire sequenze dinamiche di obiettivi assegnati agli agenti. Questa formulazione risulta fondamentale per affrontare la natura dinamica del problema e per rendere computazionalmente sostenibile il ricalcolo periodico dei piani richiesto da RHCR.

## 2.6 Implementazione e Launcher

L'implementazione del planner è stata realizzata in C++, per garantire prestazioni migliori rispetto a codice Python.

Per semplificare l'esecuzione del codice, è stato sviluppato un launcher in Python che si occupa di gestire l'esecuzione del planner. Il launcher consente di installare le dipendenze necessarie e il visualizzatore PlanViz. Inoltre, permette di compilare ed eseguire il codice direttamente da GUI. Dall'interfaccia grafica è anche possibile selezionare esecuzioni precedenti e visualizzare il plan su PlanViz.

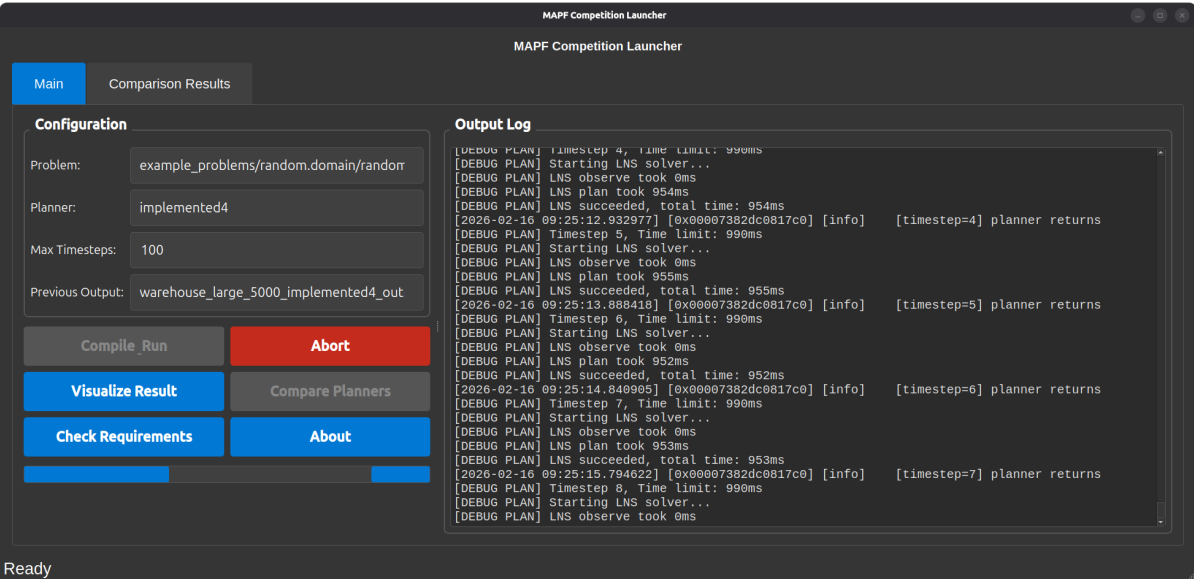


Figura 2.3: Homepage del Launcher

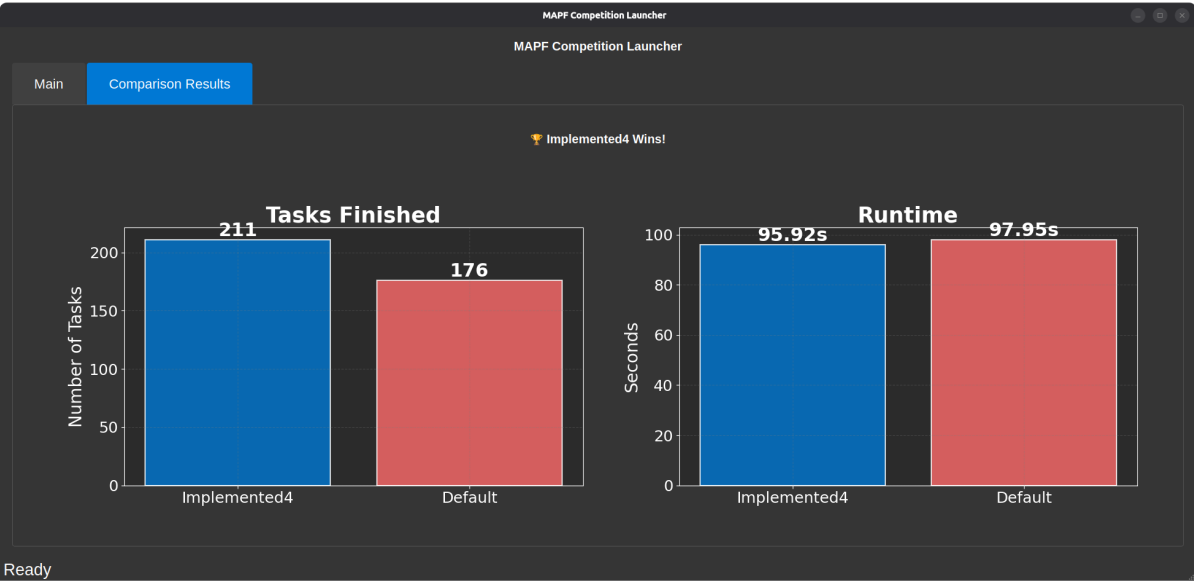


Figura 2.4: Pagina Grafici

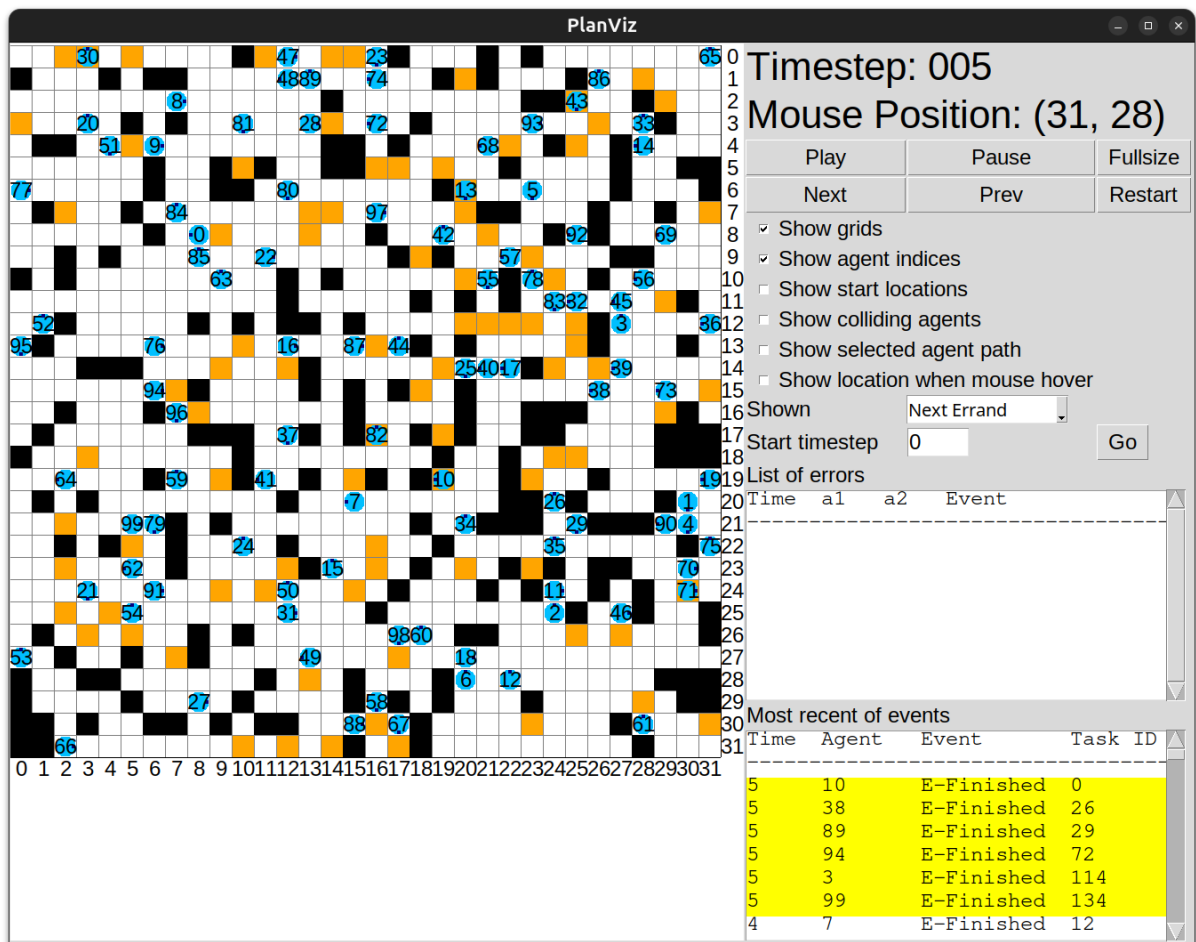


Figura 2.5: PlanViz

Un altro dettaglio da far notare è l'aggiunta del file di configurazione `config.hpp`, che permette di configurare parametri diversi per ogni istanza.

# Capitolo 3

## Risultati

Dato che la competizione è terminata non è stato possibile confrontare i risultati ottenuti con quelli degli altri team, perchè anche se pubblicati, le nostre macchine non possono avere le stesse prestazioni dei server forniti dalla competizione. Quindi, un confronto diretto risulterebbe poco significativo. I risultati sono stati confrontati con il solver che viene fornito dalla competizione, che è un planner basato su PIBT e RHCR, ed è implementato molto bene.

Il nostro planner è risultato essere in grado di risolvere tutti i problemi con un numero di task terminati maggiore rispetto al solver fornito, come visibile nel seguente grafico, che mostra il numero di task completati per ogni problema, confrontando il nostro planner con quello fornito dalla competizione.

Di seguito sono presentati i risultati, per ogni mappa sono state fatte 2 run e calcolata la media. Per esempio per ottenere i dati delle run da 500 sec sono stati eseguiti in sequenza,  $5 \text{ mappe} \times 2 \text{ run} \times 500 \text{ sec} \times 2 \text{ algoritmi} = 10000 \text{ sec} \approx 2,78 \text{ ore}$  (con algoritmi si intende quello implementato per questo progetto e quello di default)

Nei grafici c'è il confronto tra il planner implementato (Implemented4) e il solver di default fornito dalla competizione, con a destra i margini di vantaggio percentuali del nostro planner in ogni problema.

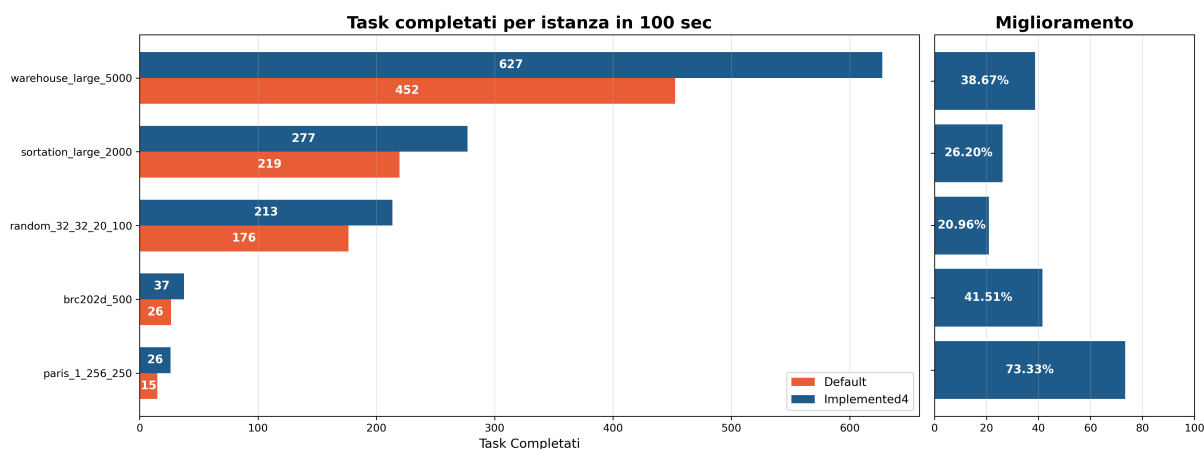


Figura 3.1: Confronto del numero di task completati in 100 secondi

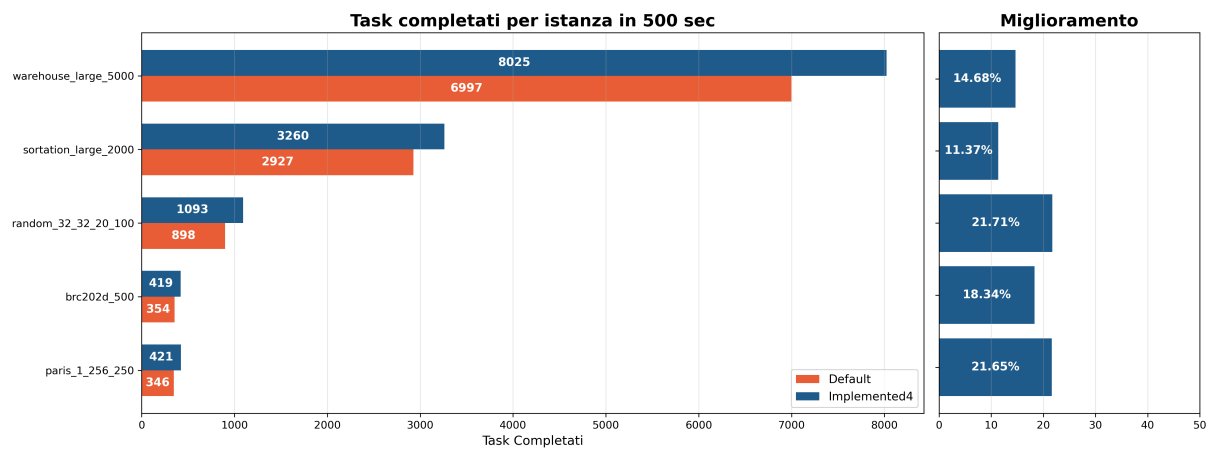


Figura 3.2: Confronto del numero di task completati in 500 secondi

# Capitolo 4

## Conclusione

I risultati ottenuti dal nostro progetto mostrano dei risultati promettenti grazie all'utilizzo di algoritmi molto efficienti per il task in questione.

Dato che la competizione si è svolta precedentemente allo sviluppo del nostro progetto non abbiamo potuto confrontare direttamente i nostri risultati con quelli degli altri concorrenti. Abbiamo rispettato completamente le regole della competizione e il codice è dunque perfettamente compatibile con la modalità di esecuzione da parte della macchina virtuale utilizzata durante la competizione. Dato il considerevole miglioramento sulle performance del nostro algoritmo rispetto a quello di default che è molto ben implementato (anche a livello teorico), riteniamo che avremmo potuto ottenere un buon piazzamento anche alla competizione ufficiale.

Questo progetto ci ha dato l'opportunità di mettere in pratica le nostre competenze nel settore, affiancando alla teoria una concreta implementazione software.