Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Informatica

Elaborato finale

Multi-Agent Path Finding

Algoritmi allo Stato dell'Arte a Confronto su Scenari Reali

Supervisore
Prof. Marco Roveri

Co-Supervisore
Dr. Enrico Saccon

Laureando
Alessandro Sartore

Anno accademico 2023/2024

Indice

Sommario **3**

1 Introduzione **4**
1.1 Introduzione ad X^* . . . 4
1.2 Confronto tra X^* e altri algoritmi di Multi-Agent Pathfinding . . . 4

2 Problema **5**
2.1 Analisi degli algoritmi A^* e Dijkstra . . . 5
2.1.1 Algoritmo di Dijkstra . . . 5
2.1.2 Algoritmo A^* . . . 5
2.1.3 Analisi computazionale . . . 5
2.2 Problema classico di MAPF . . . 6
2.3 Tipologia di conflitti . . . 6
2.4 Funzioni obiettivo . . . 7

3 Stato dell'arte **8**
3.1 Categorie di algoritmi MAPF . . . 8
3.1.1 Algoritmi basati sulla riduzione . . . 8
3.1.2 Algoritmi basati sulle regole . . . 8
3.1.3 Algoritmi basati sulla ricerca . . . 9
3.2 Conflict-Based Search (CBS) . . . 9
3.2.1 Elaborazione di un nodo nel CT . . . 10
3.2.2 Risolvere un conflitto . . . 10
3.2.3 Esempio di utilizzo CBS . . . 10
3.3 Intersection Conflict Resolution (ICR) . . . 11
3.3.1 Large Neighborhood Search (LNS) . . . 11
3.3.2 L'algoritmo ICR . . . 11
3.4 Increasing Cost Tree Search (ICTS) . . . 12
3.5 ICTS + ID (Independence Detection) . . . 13

4 L'algoritmo X^* **15**
4.1 Windowed Anytime Multiagent Planning Framework . . . 15
4.1.1 Panoramica del WAMPF . . . 15
4.2 Naïve Windowing A^* . . . 16
4.2.1 Esempio di Finestra singola . . . 17
4.3 Expanding A^*: X^* . . . 18
4.3.1 Gestione e riutilizzo delle informazioni (bookkeeping) di X* per la generazione di piani successivi . . . 18
4.3.2 Implementazione delle sottoprocedure di WAMPF . . . 20
4.4 Implementazione di X^* nel Multi-Agent Open Framework . . . 20

5 Analisi sperimentale **22**
5.1 Implementazione 22
5.1.1 Scenario Industriale 22
5.2 Confronto tra gli algoritmi 26
5.2.1 X* e CBS 26
5.2.2 X* e ICR 27
5.2.3 X* e ICTS 27
5.2.4 X* e ICTS+ID 27

6 Conclusioni e sviluppi futuri **29**

Bibliografia **29**

A Bibliografia delle Immagini **34**
A.1 Esempio Finestra Singola 34
A.2 Processo di Trasformazione per il Riutilizzo dell'Albero di Ricerca in X* 35
A.3 Insieme fuori dalla Finestra 35
A.4 Valore Chiuso 36
A.5 Scenario Reale 36

Sommario

In questo elaborato, il nostro focus è stato rivolto all'analisi e al confronto di diversi algoritmi per la risoluzione del Multi-Agent Pathfinding (MAPF), con particolare attenzione sull'algoritmo X*. Questa ricerca si basa sui sistemi complessi che necessitano di una collaborazione efficiente tra agenti autonomi. Per il corretto funzionamento di tali sistemi, è essenziale che questi agenti siano in grado di coordinare i loro spostamenti in modo efficiente e sicuro.

Contesto e Motivazioni: La pianificazione dei percorsi per più agenti all'interno di uno spazio condiviso, chiamata Multi-Agent Pathfinding (MAPF), rappresenta un campo di studio che si focalizza sullo sviluppo di algoritmi. Questi algoritmi affrontano sfide complesse rispetto alla pianificazione dei percorsi per singoli agenti. Il problema risulta particolarmente complesso a causa della necessità di evitare collisioni e assicurare un movimento scorrevole tra gli agenti, soprattutto in ambienti con restrizioni come ostacoli o aree vietate.

Problema Affrontato: L'obiettivo principale della tesi è valutare le prestazioni dell'algoritmo X* rispetto ad altri algoritmi MAPF noti, come CBS (Conflict-Based Search), ICR (Intersection Conflict Resolution), ICTS (Increasing Cost Tree Search) e ICTS+ID (ICTS + Independence Detection). Il confronto è stato effettuato considerando vari criteri di prestazione tra cui il tempo di esecuzione, la qualità del percorso e la completezza, per diverse tipologie di scenari.

Tecniche Utilizzate e Sviluppate: L'algoritmo X* è stato integrato nel framework MAOF dell'università utilizzando il linguaggio C++. Questa integrazione ha richiesto la dichiarazione di nuove classi e funzioni specifiche per X*, oltre all'adeguamento delle strutture esistenti per supportare il nuovo algoritmo. Per la fase di testing, è stato impiegato un file Excel per registrare e organizzare i dati sperimentali. Python è stato utilizzato per automatizzare l'estrazione delle informazioni dal file Excel e per generare grafici che visualizzano i risultati dei test.

Risultati Raggiunti: Dai risultati sperimentali emerge che ciascun algoritmo presenta punti di forza e di debolezza distinti. Durante i test iniziali, X* e ICR hanno dimostrato di essere molto efficienti, ma quando si è passati ai test più complessi, CBS ha dimostrato una maggiore robustezza. ICTS e ICTS+ID si sono dimostrati efficaci nel risolvere rapidamente i test più complessi, ma hanno mostrato una capacità inferiore nel completare un numero maggiore di test. Il contributo personale del laureando include l'integrazione e l'ottimizzazione dell'algoritmo X* nel framework MAOF e l'analisi comparativa dettagliata delle prestazioni degli algoritmi MAPF.

1 Introduzione

Nella società attuale, sistemi sempre più complessi coinvolgono la collaborazione di diversi agenti autonomi, dai team di robot che lavorano insieme in una fabbrica [14] ai veicoli autonomi che si muovono nelle città [16]. La capacità di questi agenti di coordinare i loro spostamenti in modo efficiente e sicuro è fondamentale. Il Multi-Agent Pathfinding (MAPF) entra in gioco in questo contesto, essendo un settore di ricerca dedicato allo sviluppo di algoritmi che consentano a più agenti di individuare percorsi ottimali all'interno di uno spazio condiviso.

Il MAPF presenta sfide significative se confrontato al Single-Agent Pathfinding (SAPF). Quando più agenti si trovano nello stesso ambiente, diventa importante considerare le interazioni tra loro per evitare collisioni e garantire un movimento fluido. Inoltre, la complessità del problema aumenta ulteriormente in situazioni con vincoli ambientali, come ad esempio ostacoli o zone proibite che gli agenti devono superare [8].

Nonostante le difficoltà incontrate, il MAPF offre numerose applicazioni pratiche. Nel campo della robotica, ad esempio, gli algoritmi MAPF efficienti possono essere impiegati per ottimizzare le attività nei magazzini [25], coordinare squadre di robot soccorritori [20] o gestire droni per la consegna dei pacchi [5]. Nel campo dei trasporti, il sistema di pianificazione multi-agente può essere impiegato per ottimizzare la circolazione del traffico nelle città, ridurre i tempi d'attesa ai semafori e migliorare la sicurezza stradale [4]. Inoltre, il sistema di pianificazione multi-agente è utilizzato anche in simulazioni di folle, videogiochi [23] e situazioni di evacuazione d'emergenza.

1.1 Introduzione ad X^*

Nel campo del Multi-Agent Pathfinding (MAPF) l'algoritmo X*, titolato "*Anytime Multi-Agent Pathfinding for Sparse Domain using Window-Based Iterative Repairs*", è noto per la sua efficacia e capacità di adattamento a scenari complessi. Ma cosa si nasconde dietro questo nome complesso? X* è un algoritmo avanzato, progettato appositamente per il pathfinding simultaneo di più agenti.

Sebbene si basi in parte su A*, questa non è la sua caratteristica principale. La vera forza di X* risiede nel fatto che è un *anytime solver*, questo significa che è in grado di fornire soluzioni di qualità crescente nel tempo, e utilizzare una ricerca locale per risolvere i conflitti tra i percorsi degli agenti. L'approccio iterativo dell'algoritmo consente di migliorare gradualmente le soluzioni, riducendo i conflitti e ottimizzando i percorsi.

Durante lo sviluppo di questa tesi, esploreremo nel dettaglio il significato dietro il nome attribuito all'algoritmo X*.

1.2 Confronto tra X^* e altri algoritmi di Multi-Agent Pathfinding

Questa tesi si concentrerà su X* e verrà confrontato con altri algoritmi allo stato dell'arte MAPF, come CBS (Conflict-Based Search), ICR (Intersection Conflict Resolution), ICTS (Increasing Cost Tree Search) e ICTS+ID (ICTS + Independence Detection). Studieremo come funzionano i principi di ciascun algoritmo, analizzeremo le loro prestazioni in termini di tempo di esecuzione, qualità del percorso e completezza, e valuteremo la loro idoneità rispetto a diversi scenari di MAPF.

L'obiettivo di questa tesi è quello di fornire una valutazione completa di X* rispetto ad altri algoritmi MAPF, identificando i suoi punti di forza e di debolezza e fornendo indicazioni sulla sua scelta in base alle specifiche esigenze di un'applicazione.

2 Problema

Il MAPF è una sfida computazionale complessa che va oltre il semplice calcolo del percorso per un singolo agente. Nel contesto di un ambiente condiviso, in cui si muovono contemporaneamente più agenti, diventa difficile trovare percorsi efficienti e privi di collisione.

2.1 Analisi degli algoritmi A^* e Dijkstra

Per comprendere al meglio le sfide del MAPF, è fondamentale capire il problema del percorso minimo per un singolo agente, noto come Single-Agent Pathfinding (SAPF). Il problema consiste nel trovare il percorso più efficiente, in termini di costo e/o tempo, che un singolo agente può seguire per raggiungere una destinazione partendo da una posizione iniziale all'interno di un ambiente. Esistono diversi algoritmi per la risoluzione del problema del percorso più breve. Di seguito vengono illustrati i due tra i più famosi [22].

2.1.1 Algoritmo di Dijkstra

Il funzionamento di Dijkstra si basa su una logica di "esplorazione graduale" del grafo, e risulta essere abbastanza intuitivo. L'algoritmo tiene traccia della distanza minima conosciuta dal nodo di partenza a tutti gli altri nodi nel grafo. All'inizio, la distanza è impostata a infinito per tutti i nodi tranne che per il nodo di partenza, il cui valore è zero. In seguito, l'algoritmo visita ciascun nodo del grafo in modo sequenziale, scegliendo il nodo con la distanza minore rispetto al nodo di partenza. Nel corso di questa visita, l'algoritmo controlla tutti i nodi adiacenti al nodo attuale e modifica le loro distanze minime nel caso in cui trovi un percorso più breve. Si confronta la somma della distanza dal nodo di partenza al nodo corrente con la distanza tra il nodo corrente e il suo adiacente per aggiornare le distanze minime. Se la somma è minore della distanza minima attualmente conosciuta per il nodo adiacente, allora si aggiorna con questa nuova somma. Il processo di selezione e aggiornamento continua finché ogni nodo del grafo non viene visitato oppure finché si trova un percorso minimo per il nodo di destinazione [7] [53].

2.1.2 Algoritmo A^*

Nel campo dell'intelligenza artificiale, molti problemi si presentano con grafi di stato così estesi da non poter essere memorizzati nella memoria principale o potrebbero addirittura essere infiniti [52]. Di conseguenza, diventa impraticabile utilizzare Dijkstra per trovare i percorsi ottimali. Come alternativa, si utilizza l'algoritmo A^*. La differenza rispetto a Dijkstra è che questo algoritmo non considera solo la distanza da ciascuno stato all'inizio, ma tiene anche conto della distanza stimata dall'obiettivo. Comunque, poiché calcolare il costo effettivo del percorso fino all'obiettivo sarebbe proibitivo dal punto di vista computazionale, si ricorre a stime dei costi tramite funzioni euristiche. Un'euristica $h(x)$ è una strategia o un metodo creato per risolvere un problema in modo rapido ed efficiente. Pur non garantendo sempre la migliore o più precisa soluzione, consente di ottenere un risultato soddisfacente nel minor tempo possibile. Utilizzando conoscenze aggiuntive fornite dalle euristiche, A^* diventa un algoritmo "informato" che guida la ricerca verso la soluzione ottimale. Dijkstra, invece, valuta soltanto la distanza da ciascuno stato rispetto all'inizio, senza considerare informazioni sull'obiettivo [52].

In A* due valori chiave guidano la ricerca del percorso ottimale: *g-value* e *f-value*. **g-value** rappresenta il costo effettivo sostenuto finora per raggiungere un nodo specifico dal nodo iniziale. Viene calcolato come la somma dei costi dei singoli archi percorsi dal nodo iniziale al nodo in questione. Il g-value fornisce una misura concreta della distanza percorsa finora nella ricerca. **f-value** rappresenta il costo totale stimato per raggiungere il nodo obiettivo da un nodo specifico. Si ottiene sommando il costo effettivo sostenuto finora per raggiungere quel nodo (g-value) e una stima euristica del costo rimanente per raggiungere l'obiettivo $h(n)$. In altre parole, $f(n) = g(n) + h(n)$, dove n è il nodo in esame [24].

2.1.3 Analisi computazionale

Di seguito, un confronto tra gli algoritmi di Dijkstra e A^*. Siano V l'insieme dei nodi ed E l'insieme degli archi di un grafo.

Dijkstra. Nel caso peggiore, la complessità temporale dipende sia da quanto risulti essere sparso il

grafo, che dalla struttura dati usata per implementare Q, ovvero la struttura impiegata per gestire i nodi non ancora visitati. Per esempio, in grafi densi si ha che $|E| = O(|V|^2)$ e dal momento che Dijkstra controlla ogni arco due volte, la sua complessità temporale nel caso peggiore è proprio $O(|V|^2)$. Tuttavia, se il grafo è sparso, $|E|$ non è equiparabile con $|V|^2$. Con una coda di Fibonacci come Q, la complessità temporale diventa $O(|E| + |V| \log |V|)$ [52]. La complessità spaziale di Dijkstra è O(V), dove V rappresenta il numero di nodi nel grafo, ciò indica che la memoria richiesta dall'algoritmo aumenta linearmente con il numero di nodi [6].
A^*. La complessità temporale e spaziale di A^* è limitata superiormente dalla dimensione del grafo. Nel caso pessimo, A^* avrebbe come complessità spaziale $O(|V|)$ e come complessità temporale $O(|V|+|E|)$. Questo sarebbe in linea con le complessità di Dijkstra, ma questa è la situazione peggiore possibile. In pratica, gli alberi di ricerca di A^* sono più piccoli di quelli di Dijkstra. Questo perchè le euristiche sono progettate per eliminare (**pruning**) una grande porzione dell'albero che Dijkstra costruirebbe sullo stesso problema. Per queste ragioni, A^* si concentra solamente sui nodi che sono più vicini alla soluzione e dunque trova il percorso migliore più velocemente rispetto a Dijkstra [52].

2.2 Problema classico di MAPF

Descriviamo cosa si intende quando si parla di un problema *classico* di MAPF. L'input con k agenti è una tupla $\langle$G, s, t$\rangle$ suddivisa nel seguente modo [35]:

- $G = (V, E)$ è un grafo non orientato.
- $s : [1, \ldots, k] \to V$ mappa un agente ad un vertice sorgente.
- $t : [1, \ldots, k] \to V$ mappa un agente ad un vertice destinazione.

Il tempo è discreto, e in ogni passo temporale ogni agente è situato in uno dei vertici del grafo e può compiere una singola *azione*. Un'azione è una funzione $a : V \to V$ tale che $a(v) = v'$, ossia se un agente è nel vertice v e compie un'azione a poi sarà nel vertice v' nel prossimo passo temporale. Ogni agente può compiere due azioni: *wait* e *move*. *Wait* significa che l'agente resta nel vertice in cui si trova anche nel successivo passo temporale; formalmente $a(v) = v$. *Move* significa che l'agente si sposta dal vertice corrente v verso un vertice adiacente v' nel grafo, ovvero $(v, v') \in E$; formalmente $a(v) = v'$ [35].
Per una sequenza di azioni $\pi = (a_1, \ldots, a_n)$ e un agente i indichiamo con $\pi_i[x]$ la posizione dell'agente dopo aver eseguito le prime x azioni in π, partendo dalla sorgente dell'agente $s(i)$. Formalmente, $\pi[x] = a_x(a_{x-1}(\ldots a_1(s(i))))$. Una sequenza di azioni è un **single-agent plan** per un agente i se e solo se eseguendo questa serie di azioni da $s(i)$ si arriva in $t(i)$, cioè se e solo se $\pi_i[\pi] = t(i)$. Una **soluzione** è un insieme di k *single-agent plans*, una per ogni agente, tale che tutti gli agenti giungono al loro vertice finale senza conflitti [35].

2.3 Tipologia di conflitti

Negli algoritmi MAPF l'obiettivo principale è quello di determinare un insieme di percorsi, uno per ciascun agente, in modo che tutti possano raggiungere le proprie destinazioni senza collisioni. Gli algoritmi MAPF si basano sul concetto di collisione, che varia a seconda delle caratteristiche dell'ambiente in cui gli agenti si trovano. Conoscendo la tipologia di collisione, l'algoritmo è in grado di determinare la soluzione. Considerando π_i e π_j una coppia di single-agent plans, definiamo quelli che sono i conflitti più comuni [54]:

- **Conflitto di vertici**: c'è un conflitto di vertici tra i percorsi π_i e π_j quando i due agenti occupano la stessa posizione nello stesso momento. Formalmente, un conflitto di vertici avviene quando $\pi_i[x] = \pi_j[x]$.
- **Conflitto nell'arco**: un conflitto nell'arco si verifica ogni volta che due agenti attraversano lo stesso arco nella stessa direzione, nello stesso momento, cioè quando $\pi_i[x] = \pi_j[x]$ e $\pi_i[x + 1] = \pi_j[x + 1]$. Se il conflitto di vertici non è consentito, allora il conflitto nell'arco non può esistere.

- **Conflitto di scambio**: un conflitto di scambio è il caso in cui due agenti scambiano la loro posizione, passando sullo stesso arco nello stesso momento, in due direzioni diverse. È espresso come $\pi_i[x+1] = \pi_j[x]$ e $\pi_j[x+1] = \pi_i[x]$.

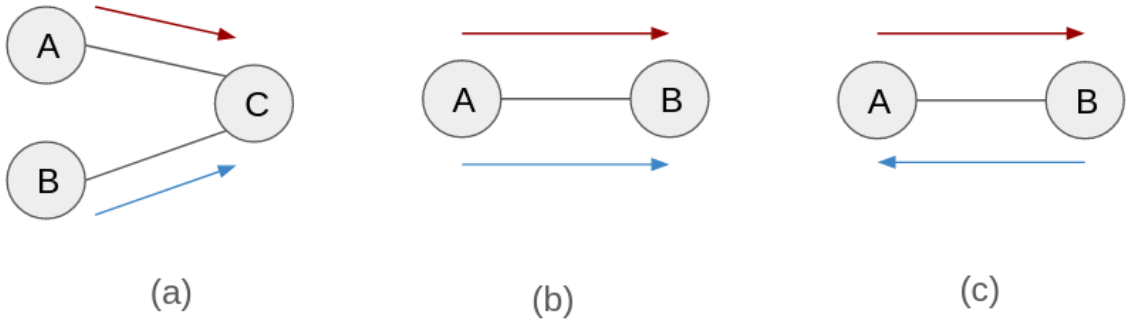

Figura 2.1: Tipologia di conflitti. (a) Conflitto di Vertice, (b) Conflitto nell'Arco, (c) Conflitto di Scambio

È da sottolineare che l'elenco di definizioni di conflitto sopra riportato non è esaustivo per tutti i tipi possibili. Considerando le definizioni formali di questi conflitti, è evidente che esiste una relazione di dominanza tra di essi: se i conflitti dei vertici sono vietati, allora anche i conflitti negli archi devono essere vietati. Allo stesso modo se si permettono i conflitti negli archi significa che i conflitti di vertici sono altrettanto permessi.

È necessario specificare i tipi di conflitti ammessi in una soluzione per definire correttamente un problema MAPF classico. La restrizione meno vincolante è quella di proibire solo i conflitti nell'arco. Tuttavia, rispetto a quanto sappiamo, tutti i lavori precedenti sul MAPF classico proibiscono anche i conflitti tra i vertici [35].

2.4 Funzioni obiettivo

Una soluzione è detta *ottimale* se ha il **costo** minore tra tutte le soluzioni possibili. Il costo $C(\pi_i)$ del percorso π_i è dato dal numero di azioni eseguito in π_i fino al raggiungimento della destinazione t_i senza contare eventuali azioni di attesa successive. Si noti che eseguire un *wait* in t_i conta nel conteggio del costo se l'agente dovesse cambiare stato in qualsiasi momento futuro [41].
Esistono due metriche comuni per valutare la qualità di un percorso $\prod$:

- **Sum of individual costs (SIC)**. La somma dei costi individuali è la somma dei costi di tutti i cammini. Formalmente $C_{SIC}(\prod) = \sum_i(C(\pi_i))$.
- **Makespan (MKS)**. Makespan è il massimo costo tra tutti i cammini. Formalmente $C_{MKS}(\prod) = max_i C(\pi_i)$.

Risolvere in modo ottimale un problema MAPF è noto essere un problema NP-hard [50]. La figura 2.2 è un esempio di percorso su cui è possibile utilizzare SIC e MKS per trovare una soluzione ottima. Data un'istanza di problema MAPF con due agenti A_1 e A_2 calcoliamo le soluzioni ottimali. Per SIC si ha che $\pi_1 = (s1, A, B, t1)$ e $\pi_2 = (s2, C, D, E, F, H, t2)$, i quali riportano $C_{SIC}(\prod) = 9$ e $C_{MKS}(\prod) = 6$. La soluzione ottimale per MKS invece è data da $\pi_1 = (s1, C, D, E, F, t1)$ e $\pi_2 = (s2, A, B, t1, G, t2)$, i quali riportano $C_{SIC}(\prod) = 10$ e $C_{MKS}(\prod) = 5$. Questo esempio illustra come ottimizzare una funzione di costo possa aumentare l'altra [41].

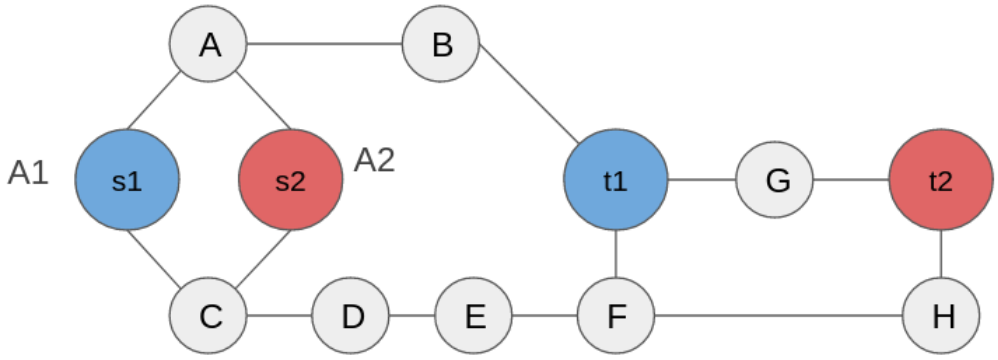

Figura 2.2: Esempio di percorso su cui è applicabile SIC e MKS

3 Stato dell'arte

Dopo aver presentato il problema nel capitolo precedente, di seguito ci si concentra sugli algoritmi principali utilizzati per trovare soluzioni efficienti. Il capitolo comincia con una breve introduzione su come i diversi algoritmi di MAPF sono suddivisi. Si parla delle sfide principali e delle categorie di algoritmi utilizzati per affrontarle. Successivamente, il focus viene spostato su alcuni algoritmi specifici che rappresentano lo stato dell'arte nel campo. Viene fornita una descrizione del funzionamento di ciascun algoritmo, evidenziandone i punti di forza e debolezza. Inoltre, si parlerà delle prestazioni di questi algoritmi e verranno confrontate le loro caratteristiche.

3.1 Categorie di algoritmi MAPF

I più moderni algoritmi MAPF possono essere categorizzati nei seguenti modi: basati sulla riduzione, basati sulle regole, basati sulla ricerca. Di seguito, analizziamo i loro metodi ed evidenziamo le loro caratteristiche in termini di completezza (completo per tutte le istanze del problema MAPF, completo per le istanze del problema MAPF su grafi con proprietà speciali, o incompleto) e ottimalità (ottimale, subottimale limitato o subottimale rispetto a diversi obiettivi). Un algoritmo MAPF è completo per una classe di istanze del problema MAPF se assicura la restituzione di una soluzione per ogni istanza del problema che è risolvibile o decide correttamente che l'istanza data non è risolvibile in tempo finito [19].

3.1.1 Algoritmi basati sulla riduzione

Gli algoritmi MAPF basati sulla riduzione riducono il MAPF ad altri problemi combinatori ben studiati, come *Boolean Satisfiability*[36], *Integer Linear Programming*[51], e *Answer Set Programming*[11]. Gli algoritmi che si basano sulla riduzione sono capaci di risolvere in modo ottimale il problema del MAPF con la funzione obiettivo *makespan* [19]. Tuttavia, è possibile adattarli in modo ottimale per affrontare efficacemente il problema con altre funzioni obiettivo [51, 11, 38]. In questo caso, è possibile configurarli in modo che operino in maniera non ottimale, ma assicurando che la soluzione finale sia entro un certo fattore di subottimalità rispetto alla soluzione ottimale fornita dall'utente [39]. Infine, è possibile implementare questi algoritmi per ottenere soluzioni subottimali senza garanzie sulla qualità della soluzione [37, 47, 12]. Funzionano bene per le istanze di problemi MAPF su grafi di piccole dimensioni con robot densamente posizionati [19].

3.1.2 Algoritmi basati sulle regole

Un algoritmo basato sulle regole è un tipo di algoritmo che utilizza un insieme predefinito di regole per prendere decisioni e risolvere problemi. Queste regole sono spesso basate sulle conoscenze o sull'esperienza di un esperto in un dominio specifico. Fondamentalmente, i criteri degli algoritmi funzionano valutando lo stato attuale di un problema in relazione alle regole stabilite [19] . Quando le condizioni di una regola sono soddisfatte, viene eseguita l'azione corrispondente. Il processo continua in modo iterativo finché il problema non viene risolto o finché una condizione di terminazione è soddisfatta [24]. Spesso garantiscono la completezza solo per una classe limitata di istanze del problema MAPF

e non forniscono alcuna garanzia sulla qualità della soluzione (ottimalità). Un insieme di algoritmi basati sulle regole, come *Push and Swap*[18] e le sue estensioni [26], possono calcolare una soluzione per 100 agenti in poco tempo di esecuzione, ma non sono in grado di fornire alcuna garanzia teorica sulla completezza. Uno dei loro discendenti, *Push and Rotate*[48], è completo per le istanze del problema MAPF su grafi con almeno due vertici non occupati da robot. Invece, *SplitAndGroup*[49] è completo per le istanze del problema MAPF su grafi ben connessi simili a griglie, si esegue in tempo polinomiale e minimizza il *makespan* su tali grafi [19].

3.1.3 Algoritmi basati sulla ricerca

Gli algoritmi MAPF basati sulla ricerca risolvono i problemi con tecniche di ricerca basate sull'euristica. La principale sfida computazionale nel risolvere ottimalmente il MAPF con un algoritmo di ricerca è che il numero di stati possibili di un'istanza del problema può crescere in modo esponenziale con il numero di robot [19]. Un esempio importante di algoritmo subottimale basato sulla ricerca è *Hierarchical Cooperative A** (HCA*)[30]. In HCA*, gli agenti vengono pianificati uno alla volta secondo un ordine predefinito. Una volta trovato un percorso per il primo agente, tale percorso (cioè tempi e posizioni) viene riservato in una tabella di prenotazione globale. Qualsiasi agente che cerca un percorso non può occupare posizioni specifiche nei tempi specifici riservati dagli agenti precedenti. Gli algoritmi basati sulla ricerca di solito non sono estremamente veloci, ma le soluzioni restituite sono di alta qualità (tipicamente quasi ottimali). È importante notare che HCA* non garantisce la completezza [10].

3.2 Conflict-Based Search (CBS)

Un algoritmo che risolve in modo ottimale i problemi del MAPF è **Conflict-Based Search (CBS)**. Ricordiamo che il numero di possibili stati coperti da A* nel MAPF è esponenziale in k (il numero di agenti). Al contrario, in un problema di ricerca del percorso di un singolo agente, questo numero di stati è solo lineare rispetto alla dimensione del grafo. Per risolvere il MAPF, CBS adotta un approccio che scompone il problema in numerosi sotto problemi di ricerca del percorso di singoli agenti soggetti a dei **vincoli**. Ognuno di questi sotto problemi può essere risolto in tempo proporzionale alla dimensione del grafo e alla lunghezza della soluzione, anche se il loro numero potrebbe crescere esponenzialmente [27].

Un *vincolo* per un agente a_i è rappresentato da una tupla $\langle a_i, v, t\rangle$ in cui all'agente a_i è proibito occupare il vertice v al tempo t. Un percorso *consistente* per l'agente a_i è un percorso che rispetta tutti i vincoli di a_i, mentre una *soluzione consistente* è composta solo da percorsi consistenti. Una volta individuato un percorso consistente per ogni agente, tali percorsi vengono validati rispettando i percorsi degli altri agenti, simulando il loro movimento lungo i percorsi pianificati. Se tutti gli agenti raggiungono la loro destinazione senza alcun conflitto, la soluzione viene restituita [27].

Tuttavia, se durante la validazione viene identificato un conflitto tra due o più agenti, la procedura si interrompe e il conflitto viene risolto mediante l'aggiunta di ulteriori vincoli. Se si verifica un conflitto, $\langle a_i, a_j, v, t\rangle$, ovvero gli agenti a_i e a_j collidono nel vertice v al tempo t, sappiamo che in qualsiasi soluzione valida, al massimo uno degli agenti in conflitto può occupare il vertice v al tempo t. Pertano, almeno uno dei vincoli, $\langle a_i, v, t\rangle$ o $\langle a_j, v, t\rangle$ deve essere soddisfatto. Di conseguenza, CBS suddivide la ricerca in due rami, creando un nuovo ramo per ciascun vincolo, con uno dei vincoli impostato come radice di ciascun nuovo ramo [40].

CBS lavora su due livelli. Ad **alto livello**, vengono generati vincoli per ogni agente. A **basso livello**, vengono trovati percorsi per ogni agente in modo che siano consistenti con i relativi vincoli. Se questi percorsi collidono, vengono risolti aggiungendo nuovi vincoli e invocando nuovamente il basso livello [2] .

Alto livello: in questo livello, CBS ricerca l'*albero dei vincoli (CT)*. Il CT è un albero binario in cui ogni nodo N contiene [2]:

1. **Un insieme di vincoli** ($N.costraints$), imposti su ciascun agente.
2. **Una soluzione** ($N.solution$). Un percorso per ciascun agente che rispetta $N.costraints$.
3. **Il costo totale** ($N.cost$). Il costo della soluzione attuale.

La radice di CT contiene un insieme vuoto di vincoli. Un successore di un nodo nel CT eredita i vincoli dal genitore e ad ogni agente associa un nuovo vincolo. *N.solution* viene calcolato dal basso livello. Un nodo N in CT è un nodo obiettivo quando *N.solution* è valida, cioè l'insieme di percorsi per tutti gli agenti non presenta conflitti. L'alto livello di CBS esegue una ricerca in ampiezza (BFS) sul CT dove i nodi sono ordinati per il loro costo [2].

3.2.1 Elaborazione di un nodo nel CT

Dato l'elenco dei vincoli per un nodo N del CT, viene chiamata la ricerca a basso livello. Questa ricerca restituisce il percorso più breve per ciascun agente a_i, in modo che sia consistente con tutti i vincoli associati ad a_i nel nodo N [27]. Una volta trovato un percorso valido per ogni agente, questi percorsi vengono **validati** rispetto agli altri agenti. La validazione viene eseguita iterando su tutti i passi temporali e confrontando le posizioni riservate da tutti gli agenti. Se nessun agente ha pianificato di essere nella stessa posizione contemporaneamente, questo nodo N di CT viene dichiarato come nodo obiettivo e la soluzione corrente ($N.solution$) che contiene questo insieme di percorsi, viene restituita. Se invece durante la *validazione* un conflitto $C = (a_i, a_j, v, t)$ viene trovato tra due o più agenti a_i e a_j, la *validazione* si interrompe e il nodo viene dichiarato come nodo **non-obiettivo** [27].

3.2.2 Risolvere un conflitto

Dato un nodo **non-obiettivo** N di CT la cui soluzione $N.solution$ contiene un conflitto $C_n = (a_i, a_j, v, t)$, sappiamo che in qualsiasi soluzione valida, al massimo uno degli agenti in conflitto può occupare il vertice v al tempo t [2]. Di conseguenza, almeno uno dei due vincoli (a_i, v, t) o (a_j, v, t) deve essere aggiunto all'insieme dei vincoli $N.constraints$. Per garantire l'ottimalità, entrambe le possibilità vengono esaminate e il nodo N viene diviso in due figli. Entrambi i figli ereditano l'insieme dei vincoli da N. Il figlio sinistro risolve i conflitti aggiungendo il vincolo (a_i, v, t) e il figlio destro aggiungendo il vincolo (a_j, v, t) [2].

Da notare che non è necessario memorizzare tutti i vincoli cumulativi per un dato nodo N di CT. È sufficiente salvare solo l'ultimo vincolo aggiunto e ricostruire gli altri risalendo l'albero da N fino alla radice tramite i suoi predecessori. Analogamente, a parte per il nodo radice, la ricerca effettuata dal basso livello, dovrebbe essere eseguita solo per l'agente associato al nuovo vincolo appena aggiunto. I percorsi degli altri agenti rimangono invariati poichè non vengono aggiunti nuovi vincoli per loro [27].

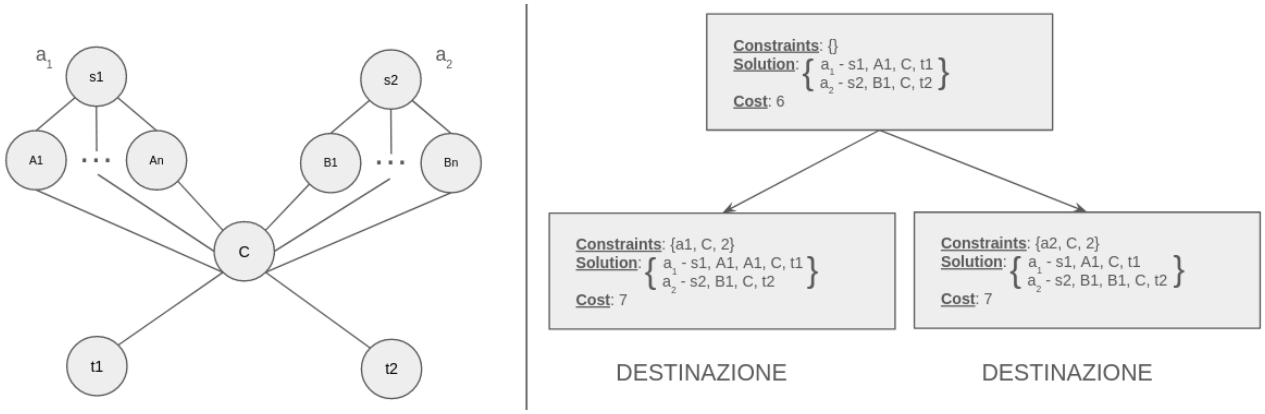

Figura 3.1: Esempio MAPF | esempio CT

3.2.3 Esempio di utilizzo CBS

Utilizzando l'esempio fornito in Figura 3.1, mostriamo come funziona l'algoritmo CBS. [2] Inizialmente, la radice del CT contiene un insieme vuoto di vincoli. Il primo step è eseguito dal basso livello, il quale ritorna una soluzione ottimale per ciascun agente. Per a_1 abbiamo che $\langle s1, A1, C, t1\rangle$, mentre per a_2 abbiamo che $\langle s2, B1, C, t2\rangle$. Quindi il costo totale di questo nodo è 6 e tutte queste informazioni sono salvate al suo interno. Durante la convalida delle soluzioni dei due agenti, viene trovato un conflitto quando entrambi arrivano al vertice C al tempo 2. Questo crea il conflitto $\langle a_1, a_2, C, 2\rangle$, e di conseguenza, la radice viene dichiarata come non-obiettivo e vengono generati due figli per risolvere il conflitto. Il figlio sinistro aggiunge il vincolo $\langle a_1, C, 2\rangle$ mentre il figlio destro aggiunge il vincolo

$\langle a_2, C, 2\rangle$. Viene dunque invocata la ricerca a basso livello per il figlio sinistro, in modo tale da trovare un percorso ottimale che soddisfi anche il nuovo vincolo. Per questo a_1 deve attendere un'unità di tempo o in $s1$ o in $A1$. Il percorso $\langle s1, A1, A1, C, 2\rangle$ viene restitutito per a_1. Il percorso per a_2, $\langle s2, B1, C, 2\rangle$, rimane invariato. Il costo totale del figlio sinistro ora è 7, dove il costo è calcolato seguendo la somma dei costi individuali (SIC). Analogamente, il figlio destro viene generato, anch'esso con costo 7. Alla fine il figlio sinistro viene scelto per l'espansione e i percorsi sottostanti vengono convalidati. Dal momento che non ci sono conflitti, il figlio sinistro viene dichiarato come un nodo obiettivo e la sua soluzione viene restituita. È stato dimostrato che CBS è sia ottimale che completo ([28]) [2].

3.3 Intersection Conflict Resolution (ICR)

Proseguiamo ora con l'introduzione di un altro algoritmo noto come *Intersection Conflict Resolution* (ICR). Questo metodo per la risoluzione dei problemi di MAPF rappresenta una variante del *Large Neighborhood Search* (LNS).

3.3.1 Large Neighborhood Search (LNS)

LNS è una meta euristica popolare per trovare ottime soluzioni a problemi di ottimizzazione complessi. Partendo da una soluzione data, eliminiamo una parte della soluzione, chiamata *neighborhood*, e consideriamo la parte rimanente come fissa. Il risultato è una forma più semplice del problema originale da risolvere. Possiamo utilizzare qualsiasi approccio desideriamo per risolvere il problema ridotto, supponendo che possa tenere conto delle informazioni fisse. Se la nuova soluzione trovata è migliore della soluzione corrente, la sostituiamo con la nuova soluzione [15].

Anche se LNS è ampiamente usato per risolvere differenti problemi di ottimizzazione [13, 3, 31], non siamo a conoscenza di precedenti approcci LNS per MAPF. Data un'istanza di MAPF, chiamiamo innanzitutto un algoritmo MAPF per trovare una soluzione iniziale P. In questa fase è possibile utilizzare qualsiasi algoritmo, anche se non ottimale. Successivamente, in ogni iterazione, selezioniamo un sottoinsieme di agenti $A_s \in A$, rimuoviamo loro i percorsi $P_s^- = \{p_i \in P | a_i \in A_s\}$ da P e ricalcoliamo dei nuovi percorsi usando un algoritmo MAPF [15]. Questo restituisce un insieme di percorsi P_s^+, uno per ogni agente in A_s, che non collidono tra loro e con i percorsi in P. La maggior parte degli algoritmi ottimali, subottimali con limite garantito e basati su priorità, possono essere adattati a questa variante modificata trattando i percorsi in P come ostacoli mobili. Confrontiamo quindi l'insieme di percorsi (vecchio) P_s^- con l'insieme dei percorsi (nuovo) P_s^+ e aggiungiamo a P quello con la somma dei costi minore. Ripetiamo questa procedura fino a raggiungere il timeout. Il risultante algoritmo viene chiamato MAPF-LNS [15].

3.3.2 L'algoritmo ICR

Come detto in precedenza, ICR è una variante di LNS. L'algoritmo inizia risolvendo prima i problemi di SAPF, ovvero identificando i percorsi ottimali per ciascun agente come se fossero soli sul grafo. Successivamente, per ogni conflitto tra gli agenti, l'algoritmo esegue le seguenti operazioni:

1. Estrae una porzione del grafo corrispondente all'area del conflitto.
2. Considera solo gli agenti che si trovano all'interno di questo sottografo.
3. Risolve il nuovo problema MAPF sul sottografo utilizzando *constraint programming* [46].
4. Unisce la soluzione ottenuta con quella precedente.

Se non si trova una soluzione nel sottografo, esso viene espanso iterativamente finché non si trova una soluzione accettabile o fino a quando l'intero grafo viene preso in considerazione. A quel punto, si risolve direttamente il problema utilizzando *constraint programming*.

L'estrazione del sottografo inizia dal nodo in conflitto e si espande iterativamente ai nodi vicini utilizzando la ricerca in ampiezza (BFS) fino a raggiungere un'intersezione. Un'intersezione è un nodo con più di due vicini (escluso se stesso). Una volta raggiunta l'intersezione, vengono aggiunti anche questi vicini al sottografo. L'algoritmo considera inizialmente una sola intersezione e, se non riesce a

trovare una soluzione all'interno del sottografo, lo espande includendo un'altra intersezione. Questo processo continua fino a quando non viene trovata una soluzione.

Per quanto riguarda gli agenti che l'algoritmo dovrebbe considerare, possiamo ignorare quelli che non attraversano il sottografo. Essi infatti non sono coinvolti attivamente nel conflitto (cioè, non sono nessuno dei due agenti che lo causano), né potrebbero esserne parte in futuro poiché non si muovono in quello spazio. Nella pratica, prendiamo in considerazione solo gli agenti che sono attivamente coinvolti nel conflitto o che stanno attraversando la zona del conflitto. Una volta identificati gli agenti, dobbiamo modificare correttamente i loro obiettivi: il nodo attraverso il quale entrano nel sottografo sarà la loro posizione iniziale, mentre l'ultimo nodo attraversato per uscire sarà la loro posizione finale. Infine, i nuovi obiettivi saranno tutti quelli precedentemente raggiunti all'interno del sottografo durante quel periodo di tempo.

Una volta estratto correttamente il sottografo e individuati gli agenti coinvolti, si applica *constraint programming* al nuovo problema MAPF. La soluzione al problema è considerare il primo agente che entra nel sottografo come istante iniziale della simulazione e far attendere tutti gli agenti che entrano successivamente. Se con *constraint programming* non si riesce a trovare una soluzione locale, il processo viene riavviato aumentando di uno il numero di intersezioni considerate. Viceversa, se il risolutore ha trovato una soluzione, l'algoritmo deve integrare la soluzione locale con quelle precedenti:

1. Conserva la parte del percorso che ha condotto l'agente ad entrare nel sottografo.
2. Sostituisce tutti i nodi all'interno del sottografo con la nuova soluzione.
3. Conserva la parte restante del vecchio percorso che non era all'interno dell'area di conflitto.

Infine, controlliamo nuovamente la presenza di conflitti. Se ci sono, riavviamo l'intero processo. In assenza di conflitti, la soluzione viene restituita. Pur non garantendo soluzioni ottimali, consente di risolvere rapidamente possibili conflitti.

3.4 Increasing Cost Tree Search (ICTS)

Presentiamo ora un nuovo approccio per risolvere in modo ottimale le istanze del problema MAPF. Questo metodo si basa sulla ricerca in un albero chiamato *Increasing Cost Tree* (ICT) utilizzando un algoritmo di ricerca corrispondente, chiamato *Increasing Cost Tree Search* (ICTS). L'ICTS è un algoritmo di ricerca su due livelli [27].

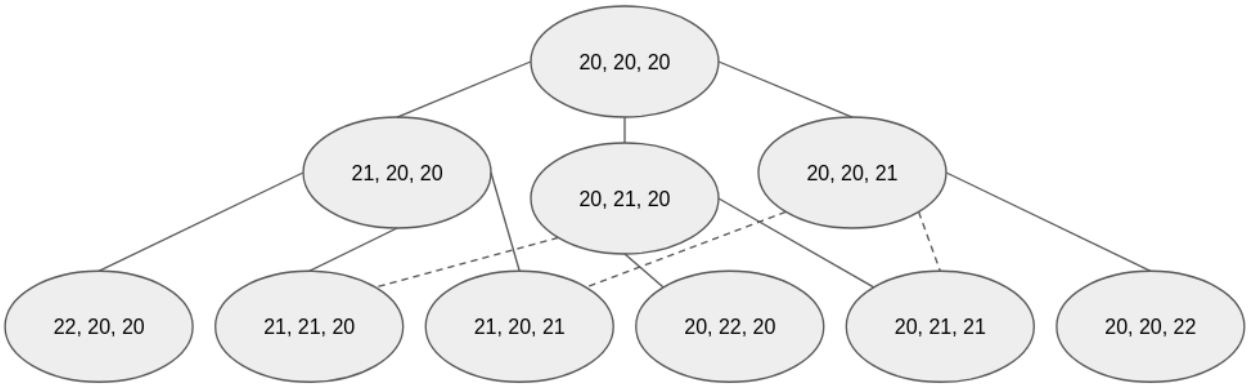

Figura 3.2: Increasing Cost Tree con 3 agenti

Alto livello: Ad alto livello, l'ICTS effettua una ricerca nell'ICT. Ogni nodo nell'ICT è composto da un vettore k-esimo $[C_1, \ldots, C_k]$ che rappresenta tutte le possibili soluzioni in cui il costo del percorso individuale dell'agente a_i è esattamente C_i. La radice dell'ICT è $[opt_1, \ldots, opt_k]$, dove opt_i è il costo ottimale del percorso individuale per l'agente a_i, ovvero la lunghezza del percorso più breve da s_i a t_i ignorando gli altri agenti. Un figlio nell'ICT viene generato aumentando di uno (o di un costo unitario) il limite di costo per uno degli agenti [27]. Un nodo dell'ICT $[C_1, \ldots, C_k]$ è un nodo obiettivo se esiste

una soluzione completa e senza conflitti tale che il costo del percorso individuale per a_i sia esattamente C_i. La Figura 3.2 illustra un ICT con 3 agenti, tutti con costi di percorso individuali ottimali di 20. Il costo totale di un nodo è $C_1 + \ldots + C_k$. Per la radice, questo corrisponde esattamente all'euristica SIC dello stato iniziale, ovvero $SIC(start) = opt_1 + opt_2 + \ldots + opt_k$. Usiamo Δ per indicare la profondità del nodo obiettivo ICT con il costo più basso. La dimensione dell'albero ICT è esponenziale rispetto a Δ. Poiché tutti i nodi alla stessa altezza hanno lo stesso costo totale, una ricerca in ampiezza dell'ICT troverà la soluzione ottimale [27].

Basso livello: Il basso livello funge da test per gli obiettivi dell'alto livello. Per ogni nodo del ICT $[C_1, \ldots, C_k]$ visitato dall'alto livello, viene invocato il basso livello [40]. Il suo compito è trovare una soluzione completa e senza conflitti, tale che il costo del percorso individuale dell'agente a_i sia esattamente C_i. Per ciascun agente a_i, l'ICTS memorizza tutti i percorsi per singolo agente con costo C_i in una speciale struttura dati compatta chiamata *Multi-value Decision Diagram* (MDD) [32]. Il basso livello esplora il prodotto cartesiano degli MDD per trovare un insieme di k percorsi non in conflitto per i diversi agenti. Se esiste un tale insieme di percorsi non in conflitto, il basso livello restituisce $True$ e la ricerca si interrompe. Altrimenti, viene restituito $False$ e l'alto livello procede al nodo di alto livello successivo (con una combinazione di costi diversa). L'ICTS implementa anche diverse regole di pruning per migliorare la ricerca [40].

Regole di pruning: Vengono introdotte speciali tecniche di pruning per i nodi di alto livello. Queste tecniche cercano una sotto-soluzione per i agenti, con $i < k$. Se esiste un sottogruppo per il quale non esiste una soluzione valida, allora non può esistere una soluzione valida per tutti i k agenti. Di conseguenza, il nodo di alto livello può essere dichiarato come non-obiettivo senza dover cercare una soluzione nello spazio dei percorsi dei k agenti [27].

3.5 ICTS + ID (Independence Detection)

Standley [33] ha adottato l'approccio A* e introdotto un importante miglioramento per risolvere i problemi MAPF: *Independence Detection* (ID). Questo miglioramento, presentato da Standley, rappresenta ad oggi il risolutore basato su A* più efficace per risolvere in modo ottimale i problemi MAPF [29]. Questo approccio basato su A* è stato usato in concomitanza con l'algoritmo ICTS, creando una versione migliore chiamata ICTS+ID.

Il framework *Independence Detection* viene eseguito a livello base, ovvero è l'utente che lo chiama. A partire dal livello ID, viene invocato un risolutore $A*$ per gruppi specifici di agenti. ID funziona nel seguente modo. Due gruppi di agenti sono **indipendenti** se esiste una soluzione ottimale per ciascun gruppo tale che le due soluzioni non entrino in conflitto [29] . L'idea di base di ID è quella di dividere gli agenti in gruppi indipendenti e risolvere separatamente questi gruppi. Ogni gruppo viene risolto utilizzando l'algoritmo A*, che restituisce una soluzione ottimale per quel gruppo di agenti. Le soluzioni dei diversi gruppi vengono poi eseguite simultaneamente fino a quando non si verifica un conflitto tra due o più gruppi. In caso di conflitto, ID tenta di risolverlo cercando di ripianificare uno dei gruppi per evitare il percorso dell'altro. Se il conflitto non viene risolto, i gruppi conflittuali vengono fusi in un unico gruppo e risolti utilizzando A*. Questo processo viene ripetuto finché non vengono trovate soluzioni senza conflitti per tutti i gruppi [29].

Dato che il tempo di esecuzione dell'algoritmo è dominato dal tempo necessario per pianificare i percorsi per il gruppo più grande [33], le prestazioni possono essere notevolmente migliorate evitando fusioni non necessarie, ovvero usando *conflict avoidance* (CA). L'algoritmo ID con CA può talvolta evitare di unire due gruppi in conflitto trovando un nuovo percorso ottimale per uno dei gruppi. Per garantire l'ottimalità, per un gruppo i nuovi percorsi devono avere lo stesso costo totale dei percorsi iniziali [34].

Poiché la complessità di un problema MAPF in generale è esponenziale nel numero di agenti, il tempo di esecuzione per risolvere un problema MAPF con ID è dominato dal tempo di esecuzione per risolvere il problema indipendente più grande [29] . ID può identificare che una soluzione a un problema MAPF con k agenti può essere composta da soluzioni di diversi sottoproblemi indipendenti. Indichiamo con k' la dimensione del sottoproblema più grande ($k' \leq k$). Poiché il problema è esponenziale nel numero di agenti, ID fornisce un'accelerazione esponenziale in $k - k'$ [29] .

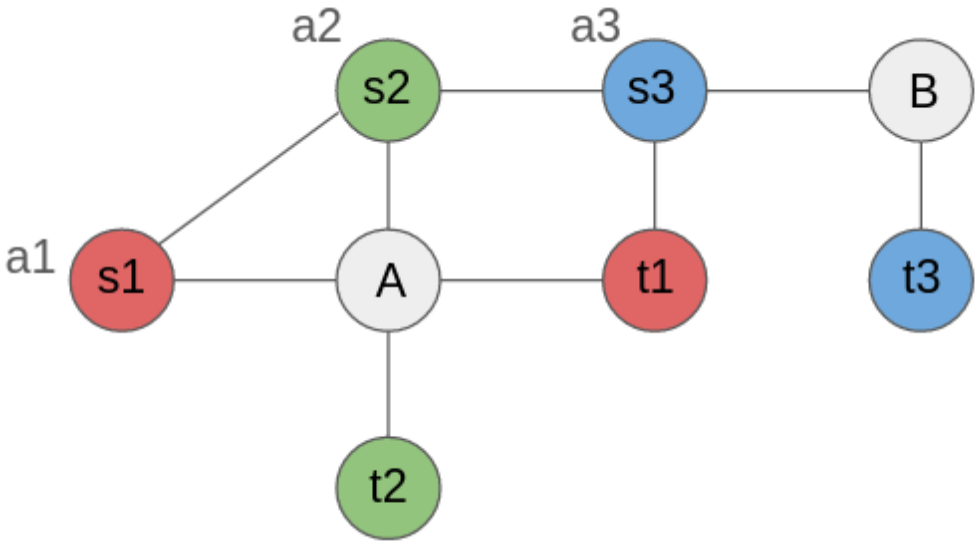

Figura 3.3: Problema MAPF con 3 agenti su cui applichiamo A* + ID

In Figura 3.3 abbiamo un grafo di esempio in cui applichiamo A* + ID. ID funziona nel seguente modo. Per ogni agente viene individuato un percorso di costo 2 ottimale. Il percorso $(s1, A, t1)$ per l'agente a_1, il percorso $(s2, A, t2)$ per l'agente a_2, il percorso $(s3, B, t3)$ per l'agente a_3. Quando si tenta di eseguire i percorsi degli agenti a_1 e a_2, si verifica un conflitto nel vertice A. Non c'è modo di risolvere questo conflitto e gli agenti a_1 e a_2 vengono quindi uniti in un unico gruppo. Viene chiamato A* su questo gruppo e viene restituita una soluzione di costo 5 $(2 + 2 + 1$ per l'unione). Questa soluzione viene ora eseguita simultaneamente con la soluzione dell'agente a_3 . Non si riscontrano conflitti e l'algoritmo si interrompe. Il gruppo più grande invocato da A* era di dimensione 2. Senza ID, A* dovrebbe risolvere un problema con tre agenti, mentre in questo caso risultano 2 agenti complessivi. È importante notare che il framework ID può essere implementato su qualsiasi risolutore MAPF ottimale, cioè un risolutore che garantisce di restituire soluzioni ottimali. Pertanto, ID può essere considerato come un framework generale che utilizza un risolutore MAPF [29].

4 L'algoritmo X^*

Procederemo ora a discutere l'algoritmo principale di questa tesi: X*. Prima di approfondire i dettagli di X*, è utile introdurre il concetto degli *Anytime Path Planner* (APP), poiché X* si basa su questi principi. Gli APP sono algoritmi di pianificazione in grado di sviluppare rapidamente una soluzione iniziale al problema e, se gli viene concesso più tempo di calcolo, migliorano iterativamente la qualità del percorso. Gli algoritmi *Anytime* sono desiderabili in molti ambiti in quanto consentono di effettuare compromessi tra la qualità della soluzione e il tempo di pianificazione [44, 43, 42]. Un modo semplice per costruire un *Anytime Planner* è quello di eseguire un algoritmo di pianificazione standard con parametri che scambiano l'ottimalità della soluzione con un miglioramento del tempo di esecuzione, e poi rieseguire iterativamente l'algoritmo con limiti più rigidi se rimane del tempo di calcolo [55]. Sebbene questa prima generazione di percorsi sia spesso veloce, le iterazioni successive diventano sempre più lente a causa della mancanza di riutilizzo delle informazioni. Gli *Anytime Planner* che invece riutilizzano le informazioni delle ricerche precedenti sono generalmente più veloci nel generare i percorsi successivi ([17, 21, 1]) [45].

4.1 Windowed Anytime Multiagent Planning Framework

La crescita esponenziale della dimensione dello spazio degli stati con l'aumento del numero di agenti, rende necessario l'impiego di approcci basati su sottospazi per accelerare la ricerca. Questi approcci suddividono il problema MAPF in sottoproblemi più gestibili che coinvolgono un numero inferiore di agenti. Una scoperta fondamentale è che i sottospazi non solo possono restringere la ricerca a un gruppo selezionato di agenti, ma possono anche confinare la ricerca a un gruppo selezionato di stati, rendendo il processo di soluzione più efficiente [45].

Presentiamo una struttura chiamata **finestra** che incapsula un sottoinsieme di agenti e un sottoinsieme connesso di stati. Una finestra viene posizionata intorno a una collisione nel percorso globale al fine di produrre una riparazione eseguendo una ricerca all'interno di quest'area delimitata. L'inizio della ricerca di riparazione in w_k, denotato s_k, è il primo stato sul percorso globale nella finestra e l'obiettivo della ricerca di riparazione in w_k, denotato g_k, è l'ultimo stato sul percorso globale nella finestra. Ogni finestra w_k ha una finestra successiva w_{k+1} che condivide lo stesso insieme di agenti ma ha un insieme più ampio di stati. Questo consente di far crescere iterativamente una finestra sostituendola con la sua successiva che considera un dominio più ampio nella sua riparazione. Due finestre possono essere unite per formare una finestra più grande che incorpora entrambe le finestre più piccole tramite l'operatore $\cup$. Ad esempio, w e w' possono essere unite per formare una finestra più grande $w'' := w \cup w'$; w'' deve avere un insieme di agenti $\alpha'' = \alpha \cup \alpha'$ e tutti gli stati in w e w' devono far parte degli stati congiunti di w''. Infine, due finestre possono essere sovrapposte tramite l'operatore $\cap$. Ad esempio, $w \cap w'$ è vero se e solo se i loro insiemi di agenti α e α' si sovrappongono e condividono uno o più stati di agenti [45].

Sebbene una riparazione basata su finestre (***window-based***) non garantisca che il percorso globale risultante dopo la riparazione sia ottimale, una riparazione in una finestra successiva w_{k+1} garantisce che il percorso globale riparato avrà al massimo lo stesso costo del percorso globale riparato da w_k e spesso costerà meno. Quindi, far crescere ripetutamente il sottospazio e generare riparazioni migliora in modo monotono la qualità del percorso globale. Inoltre, se una finestra w_k è sufficientemente grande da far sì che s_k e g_k siano l'inizio globale s e l'obiettivo globale g per i suoi agenti α e w_k non ostacola la ricerca da s_k a g_k, cioè non limita l'esplorazione della ricerca con le restrizioni dello stato w_k, allora i percorsi congiunti per gli agenti α in w_k sono congiuntamente ottimali e w_k può essere scartato. Se non esistono più finestre, allora il percorso congiunto è una soluzione ottimale. Questa intuizione è alla base del framework MAPF *anytime* chiamato **Windowed Anytime Multiagent Planning Framework** (WAMPF) [45].

4.1.1 Panoramica del WAMPF

Poichè WAMPF è un framework che può essere usato per pianificatori MAPF *anytime*, richiede alcune definizioni e sottoprocedure specifiche. Ogni pianificatore che implementa WAMPF deve definire cosa sia una "finestra" in relazione allo spazio degli stati. Una finestra w_k deve [45]:

- Includere un sottoinsieme connesso di stati per un gruppo di agenti.
- Avere un punto di partenza s_k e un punto di arrivo g_k sul percorso globale.
- Avere una finestra successiva w_{k+1} con più stati e lo stesso gruppo di agenti.
- Potersi unire con un'altra finestra usando l'operatore $\cup$, creando così una nuova finestra che incorpora agenti e stati di entrambe le finestre.
- Potersi sovrapporre con un'altra finestra usando l'operatore $\cap$, restituendo un valore booleano per indicare la sovrapposizione.

Una volta definite le proprietà della finestra, WAMPF utilizza diverse sottoprocedure per gestire i percorsi e le collisioni tra agenti. Le sottoprocedure chiave includono [45]:
FirstCollisionWindow(π): identifica la prima collisione tra agenti lungo un percorso π iniziando dal primo stato π_0. Se trova collisioni, restituisce una finestra che le racchiude; se non ci sono collisioni, restituisce un insieme vuoto.
PlanIn(w_k, π): il percorso π ha un insieme di agenti associato α e la finestra w_k ha un insieme di agenti associato α', dove $\alpha' \subseteq \alpha$. Questa sottoprocedura genera una riparazione senza collisioni in w_k pianificando un percorso ottimale da s_k a g_k, rispettando i tempi di ingresso degli agenti in s_k. La riparazione viene inserita in sostituzione al sottoinsieme di π, rispettando i tempi relativi degli agenti coinvolti nelle finestre successive, e π viene restituito.
GrowAndReplanIn(w_k, π): il percorso π ha un insieme di agenti associato α e la finestra w_k ha un insieme di agenti associato α', dove $\alpha' \subseteq \alpha$. Questa sottoprocedura estende w_k sostituendola con la sua finestra successiva, w_{k+1}, e genera una riparazione in w_{k+1} pianificando un percorso ottimale da s_{k+1} a g_{k+1}, inserendola come sostituzione al sottoinsieme rilevante di π, rispettando i tempi relativi degli agenti coinvolti nelle finestre successive, e poi restituendo (w_{k+1}, π). GrowAndReplanIn(w_{k+1}, π) viene invocata solo quando PlanIn(w_{k+1}, π) o GrowAndReplanIn(w_k, π) sono state precedentemente invocate e si garantisce che w_{k+1} non si sovrapponga con nessun'altra finestra esistente.
ShouldQuit(w_k, π): decide se scartare una finestra w_k. Una finestra può essere scartata solo se non limita la ricerca di riparazioni e copre completamente il percorso globale per gli agenti associati [45].

Con l'aiuto di queste sottoprocedure, WAMPF è in grado di mantenere la sua flessibilità e capacità di migliorare progressivamente i percorsi degli agenti, assicurando contemporaneamente l'ottimalità globale delle soluzioni [45].

4.2 Naïve Windowing A^*

Per illustrare un esempio pratico di un pianificatore basato su WAMPF, introduciamo *Naïve Windowing A** (NWA*). Questa versione elementare di WAMPF è specifica per griglie a quattro connessioni (su, giù, destra, sinistra) con costi unitari e utilizza l'algoritmo A* come risolutore per le finestre. NWA* non riutilizza le informazioni di ricerca quando la finestra viene ampliata. Presentiamo le definizioni/sottoprocedure necessarie per WAMPF [45]:
Definizione di finestra: La finestra è formulata come un prisma rettangolare, caratterizzato dal suo angolo inferiore sinistro e superiore destro. Nuove finestre vengono inizializzate intorno a uno stato di collisione selezionando tutti gli stati che hanno una distanza L_∞ dallo stato di collisione, più o meno un parametro. Un esempio di tale finestra è mostrato nella Figura 4.2, dove quest'ultima, disegnata come un rettangolo tratteggiato, è nello spazio congiunto di a e b e creata tramite una norma L_∞ di 1. Una finestra viene ingrandita spostando i suoi angoli più lontani dal centro di un numero fisso di passi. Un esempio di crescita della finestra è mostrato nella transizione dalla Figura 4.2 alla Figura 4.3, dove la finestra viene ingrandita aumentando il raggio di uno stato.
FirstCollisionWindow(π): questa sottoprocedura cerca collisioni lungo il percorso globale π, iniziando con π_0 e terminando con lo stato $\pi_{|\pi|-1}$. Se viene rilevata una collisione, viene inizializzata una finestra attorno allo stato di collisione con gli agenti coinvolti nella collisione; altrimenti, viene restituito $\emptyset$.
PlanIn(w_k, π): il percorso globale π dato ha un insieme di agenti associato α e la finestra data w_k ha un insieme di agenti associato α', dove $\alpha' \subseteq \alpha$. s_k e g_k sono calcolati da $\phi(\pi, \alpha')$ (dove per $\phi(\pi, \alpha')$

si intende una funzione filtro che estrae il percorso associato al sottoinsieme di agenti α' dal percorso globale π); s_k è il primo stato su $\phi(\pi, \alpha')$ in w e g_k è l'ultimo stato su $\phi(\pi, \alpha')$ in w. A* viene eseguito nello spazio di w_k da s_k a g_k, con gli stati che non appartengono a w scartati . Il risultato della riparazione π' sostituisce la sezione del percorso in $\phi(\pi, \alpha')$ da s_k a g_k. È importante notare che, se π non è già una soluzione valida, il costo di π potrebbe rimanere lo stesso o aumentare dopo l'inserimento di π'; se π è già una soluzione valida, allora π' avrà un costo uguale o ridotto rispetto alla regione di $\phi(\pi, \alpha')$ da s_k a g_k, poiché π sarà già stata riparata da una finestra w_{k-1}, e quindi la finestra più grande w_k potrebbe trovare una riparazione π' per la stessa regione di $\phi(\pi, \alpha')$ che costa meno. Nel caso in cui π' costi meno, deve essere adattata per garantire che tutti gli agenti lascino g_k allo stesso tempo di prima dell'inserimento di π' in π. Inoltre, se la ricerca A* non restituisce un percorso valido, w_k viene espansa per formare w_{k+1} e viene restituito il risultato di PlanIn(w_{k+1}, π).
GrowAndReplanIn(w_k, π): questa sottoprocedura espande w_k sostituendola con la sua successiva, w_{k+1}, e poi restituisce il risultato di PlanIn(w_{k+1}, π).
ShouldQuit(w_k, π): il percorso globale π ha un insieme di agenti associato α e la finestra w_k ha un insieme di agenti associato α'. Questa sottoprocedura restituisce vero se e solo se s_k e g_k sono rispettivamente $\phi(\pi, \alpha')_0$ e $\phi(\pi, \alpha')_{|\pi|-1}$, e w_k non ha ostacolato la ricerca durante l'ultima invocazione di PlanIn(w_k, π), cioè i vicini non sono stati scartati durante nessuna delle espansioni di A* a causa delle restrizioni dello spazio degli stati di w_k [45].

4.2.1 Esempio di Finestra singola

L'esempio di finestra singola mostrato in Figura 4.5 mostra la creazione, crescita, ripianificazione e terminazione delle finestre utilizzando la definizione di finestra di NWA*. L'esempio dimostra una singola collisione tra due agenti risolta tramite una ricerca nella finestra; questa finestra viene poi ripetutamente ampliata e nuovamente ricercata fino a includere una ricerca senza ostacoli da s a g [45].

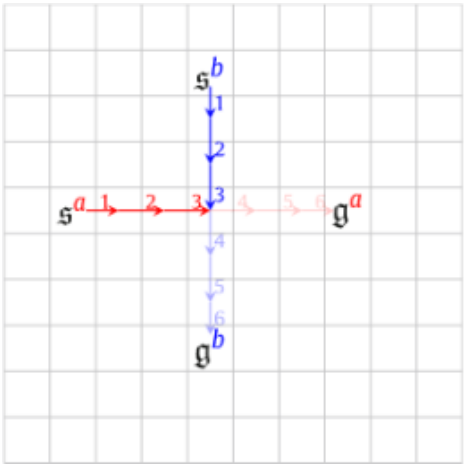

Figura 4.1: Vengono generati percorsi individuali per ogni agente da s a g in modo da formare un percorso globale. Una collisione agente-agente accade al tempo $t = 3$ tra l'agente a e b.

Figura 4.2: La collisione tra a e b viene riparata facendo aspettare l'agente a al tempo $t = 2$ all'interno della finestra w^{ab}. Il percorso globale è ora valido, ma non è detto che sia ottimale.

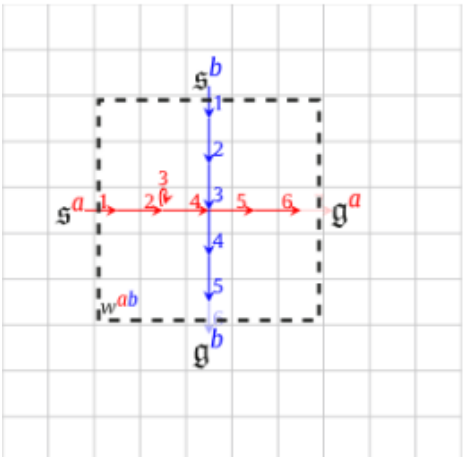

Figura 4.3: w^{ab} viene ingrandita e viene generata una nuova riparazione per a e b. La finestra non racchiude ancora la ricerca da s^{ab} e g^{ab}, quindi il percorso globale riparato non è ancora garantito essere ottimale.

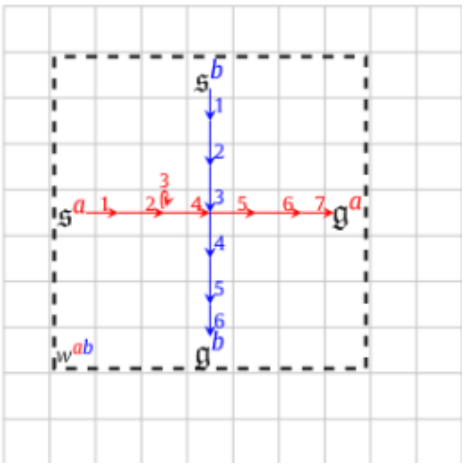

Figura 4.4: L'albero w^{ab} viene ampliato e una nuova riparazione viene generata. La ricerca per la riparazione va da s a g e non è ostacolata da w^{ab}, ciò permette di rimuovere w^{ab} e di considerare il percorso globale come ottimo.

Figura 4.5: Esempio di finestra singola WAMPF usando la definizione di finestra di NWA* [45]

4.3 Expanding A^*: X^*

Expanding A* (X*) è un algoritmo di pianificazione efficiente basato su WAMPF. X* è quasi identico a NWA*, differendo solo per l'implementazione di una gestione aggiuntiva delle informazioni che permette di riutilizzare le informazioni (*bookkeeping*) delle ricerche di riparazione precedenti quando si risolve una riparazione successiva. Grazie a questo riutilizzo, X* è significativamente più efficiente di NWA* nella generazione di piani successivi [45].

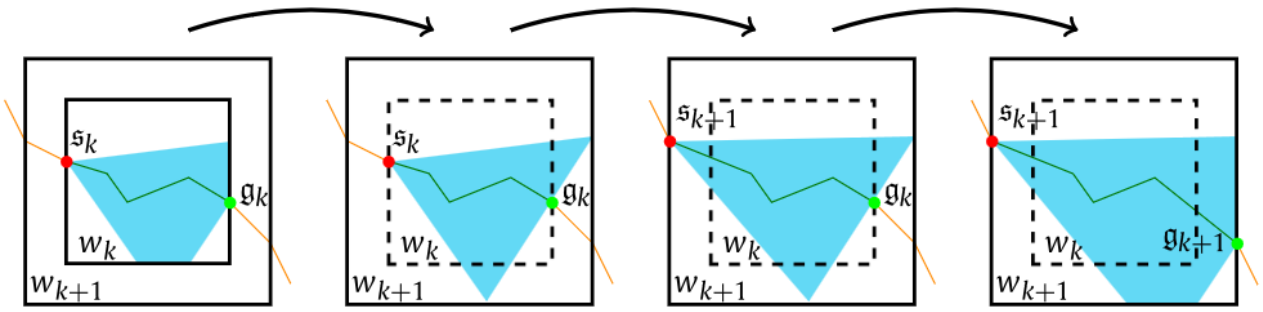

Figura 4.6: Le tre fasi di trasformazione impiegate da X* per consentire il riutilizzo dell'albero di ricerca. w_k e w_{k+1} rappresentano rispettivamente la finestra k-esima e $k+1$-esima. s_k e s_{k+1} rappresentano l'inizio della riparazione per w_k e w_{k+1}, rispettivamente. g_k e g_{k+1} rappresentano l'obiettivo della riparazione per w_k e w_{k+1}, rispettivamente. Da sinistra verso destra, la prima immagine (configurazione iniziale) mostra l'albero di ricerca iniziale. La seconda (Fase 1) espande la finestra senza spostare l'inizio o l'obiettivo. La terza (Fase 2) sposta l'inizio mantenendo lo stesso obiettivo. L'ultima (Fase 3) sposta l'obiettivo [45].

4.3.1 Gestione e riutilizzo delle informazioni (bookkeeping) di X* per la generazione di piani successivi

Il processo interno dell'algoritmo X* tiene traccia delle informazioni durante la ricerca di una soluzione (o riparazione) in una specifica porzione del grafo w_k. Questo permette di riutilizzare i risultati della ricerca per trovare soluzioni in porzioni di grafo più grandi w_{k+1}, evitando di ripetere calcoli inutili [45]. La Figura 4.6 illustra il funzionamento di questa tecnica.

Configurazione iniziale: La prima immagine della Figura 4.6 rappresenta un albero di ricerca da

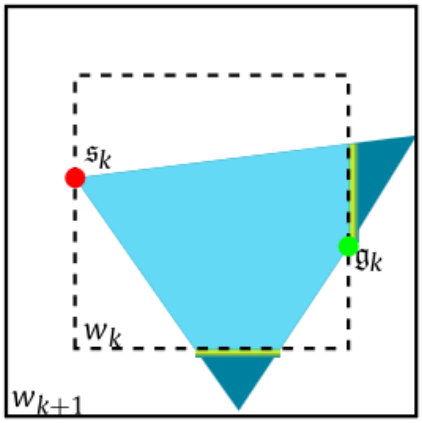

Figura 4.7: Esempio del primo bookkeeping. La parte in giallo mostra gli stati salvati nell'Insieme fuori dalla Finestra durante la ricerca in w_k, dopo l'espansione della parte in azzurro. Gli stati gialli formano la frontiera per l'espansione degli stati quando la finestra viene ingrandita a w_{k+1} e la nuova ricerca comprende l'area blu scuro [45].

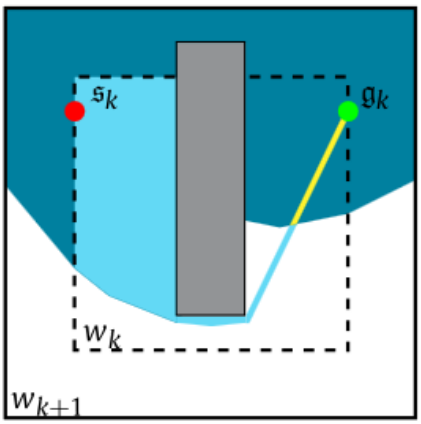

Figura 4.8: Secondo esempio di bookkeeping. L'oggetto grigio è un ostacolo nello spazio di ricerca. La regione azzurra indica l'area espansa durante la ricerca iniziale di w_k a g_k. La regione blu scuro indica l'area espansa durante w_{k+1} sulla base del f-value dell'espansione di g in w_k. La regione gialla indica gli stati riespansi con un g-value inferiore [45].

s_k a g_k, ristretto in w_k. L'albero di ricerca viene generato usando A*.
Fase 1: La prima fase, detta Espansione della Finestra (seconda immagine della Figura 4.6), mostra un albero di ricerca modificato come se la ricerca fosse stata eseguita da s_k a g_k in una finestra meno restrittiva w_{k+1}. Per passare da un albero di ricerca A* con una finestra piccola w_k a una più grande w_{k+1}, è necessario espandere tutti gli stati che sarebbero stati esplorati nella ricerca in w_{k+1} ma sono bloccati da w_k. Questi stati, rappresentati in blu scuro in Figura 4.7, devono essere raggiunti tramite uno stato non in w_k il cui predecessore diretto è in w_k; l'insieme di questi stati è rappresentato in giallo. Per questo motivo, introduciamo un nuovo elemento di *bookkeeping*: **Insieme fuori dalla Finestra**. Per ogni stato $s \in w_k$ che è stato espanso, teniamo traccia dei vicini (insieme N) di s che sono stati scartati a causa delle restrizioni di w_k, cioè $N(s) \setminus w_k$, inserendoli nell'Insieme fuori dalla Finestra. Questo *bookkeeping* ci consente di aggiungere questi stati all'insieme aperto O (ricordiamo che nell'insieme aperto sono presenti i nodi non ancora esaminati) di A*, inizializzando così la ricerca in w_{k+1}, rappresentata in giallo in Figura 4.7. Inoltre, questo *bookkeeping* fornisce un modo conveniente per tracciare se la ricerca è stata ostacolata quando si calcola ShouldQuit; se l'Insieme fuori dalla Finestra è vuoto dopo una riparazione in w_k, allora la ricerca non è stata ostacolata [45].

Quando la finestra viene espansa, dobbiamo anche considerare la possibilità di nuovi percorsi più brevi verso stati già visitati. Un esempio di ciò è mostrato nella Figura 4.8, dove l'ostacolo grigio costringe la ricerca iniziale in w_k a passare sotto l'ostacolo per raggiungere g_k. Tuttavia, espandendo la ricerca in w_{k+1}, l'algoritmo identifica un percorso che passa sopra l'ostacolo. Questo non solo permette di raggiungere g_k più rapidamente, ma consente anche di esplorare più velocemente gli stati successivi rappresentati in giallo. Pertanto, dobbiamo consentire che gli stati che sono stati visitati nella ricerca di w_k siano riespansi nella ricerca di w_{k+1} se la ricerca in w_{k+1} assegna a questi stati un *g-value* inferiore. Questo motiva il nostro secondo *bookkeeping*: **Valore Chiuso**. Per facilitare questa riespansione, durante la ricerca iniziale di A* teniamo traccia anche del *g-value* con cui uno stato viene inserito nell'Insieme Chiuso C, chiamato "Valore Chiuso dello stato" [45].

È importante notare che tutti gli stati in C alla fine della ricerca in w_k non possono essere raggiunti con un costo inferiore rispetto al loro Valore Chiuso tramite alcun percorso che rimanga interamente all'interno di w_k [45]. Poiché la ricerca in w_k è ottimale, qualsiasi percorso a costo inferiore verso uno stato in C deve uscire da w_k, attraversare una parte di w_{k+1} e rientrare in w_k, proprio come il percorso sopra l'ostacolo grigio nella Figura 4.8. Pertanto, l'aggiunta degli stati nell'insieme aperto O dal nostro Insieme fuori dalla Finestra (primo bookkeeping) garantisce che tutti questi percorsi

possano essere considerati. Inoltre, gli stati possono essere riespansi se il loro *g-value* registrato nel Valore Chiuso (secondo bookkeeping) è superiore al loro *g-value* mentre si trovano in O. Con questa modifica, possiamo eseguire A* fino a quando il *g-value* minimo in O è maggiore del *g-value* di g_k. Questo aggiornerà tutti gli stati in C e O per avere il *g-value* ottimale per una ricerca in w_{k+1} e quindi produrrà l'albero di ricerca mostrato nella Fase 1 della Figura 4.6 [45].
Fase 2: La seconda fase, detta Spostamento dell'Inizio (terza immagine della Figura 4.6), mostra come l'albero di ricerca viene trasformato quando l'inizio viene spostato da s_k (come visto nella Fase 1) a s_{k+1}. Per spostare l'inizio all'indietro, dobbiamo incorporare l'albero di ricerca radicato in s_k nell'albero di ricerca radicato in s_{k+1}. Per raggiungere un qualsiasi stato nell'albero esistente partendo da s_{k+1}, ad esempio s' (che si trova nello spazio azzurro in figura 4.6), il costo del percorso minimo è al massimo pari al costo per viaggiare da s_{k+1} a s più il costo per viaggiare da s_k a s'. Questo perché il percorso da s_{k+1} a s_k è parte del percorso globale π, che è garantito essere privo di collisioni in questa regione e quindi rappresenta un limite superiore valido. Il percorso da s_k a s' è fornito dai *g-value* dell'albero di ricerca esistente e quindi rappresenta il costo ottimale da s_k a s'. Pertanto, se aumentiamo il *g-value* e il Valore Chiuso di ogni stato (secondo bookkeeping) del costo del percorso da s_{k+1} a s_k, ed espandiamo ogni stato lungo il percorso da s_{k+1} a s_k, possiamo eseguire A* fino a quando il *g-value* minimo in O è maggiore del *g-value* di g_k, sfruttando il secondo bookkeeping per riespandere gli stati nell'albero radicato in s_k secondo necessità, come fatto nella Fase 1 [45].
Fase 3: La terza fase, detta Spostamento dell'Obiettivo (quarta immagine della Figura 4.6), mostra come l'albero di ricerca radicato in s_{k+1} viene trasformato quando l'obiettivo viene spostato da g_k a g_{k+1} (come visto nella Fase 2). Per fare questo, gli stati nell'insieme aperto O devono semplicemente aggiornare i loro *g-value* con i nuovi valori calcolati rispetto a g_{k+1}. Dopo questo aggiornamento, l'algoritmo A* può continuare a funzionare normalmente fino a quando g_{k+1} viene espanso [45].

4.3.2 Implementazione delle sottoprocedure di WAMPF

Tre delle cinque implementazioni chiave di X* sono identiche a NWA* (Sezione 4.2); tuttavia, le altre due implementazioni sfruttano le garanzie fornite da WAMPF per quanto riguarda l'ordine delle chiamate PlanIn e GrowAndReplanIn sulle finestre successive, al fine di migliorare l'efficienza. Inoltre, affinché queste tecniche di riutilizzo funzionino, è necessario assumere che l'euristica sia coerente, cioè che valga la disuguaglianza triangolare [45]. Questa disuguaglianza afferma che per ogni nodo n, ogni suo successore n', e il nodo obiettivo t, deve valere la seguente relazione [9]:

$$h(n) \leq c(n, n') + h(n')$$

- $h(n)$ è il valore euristico del nodo n, cioè la stima del costo dal nodo n al nodo obiettivo t.
- $c(n, n')$ è il costo reale per muoversi dal nodo n al nodo successore n'.
- $h(n')$ è il valore euristico del nodo n'.

PlanIn(w_k, π): Questa sottoprocedura è implementata quasi identicamente a PlanIn di NWA* nella Sezione 4.2, ma con l'implementazione dei due bookkeeping della Sezione 4.3.1.
GrowAndReplanIn(w_k, π): Come definito in precedenza, GrowAndReplanIn verrà chiamato solo su una finestra in cui GrowAndReplanIn o PlanIn sono stati precedentemente chiamati. Pertanto, questa sottoprocedura sfrutta l'albero di ricerca X* prodotto dalla ricerca precedente di w_k per supportare la ricerca attuale di w_{k+1} tramite la trasformazione mostrata nella Figura 4.6 [45].

4.4 Implementazione di X^* nel Multi-Agent Open Framework

Per integrare l'algoritmo X* nel framework dell'università, chiamato MAOF (Multi-Agent Open Framework), è stato necessario un approccio accurato per adattarlo all'architettura esistente del sistema. MAOF è un framework che permette di testare efficacemente gli algoritmi allo stato dell'arte di Multi-Agent Path Finding grazie alla sua struttura modulare e flessibile. La struttura di MAOF è progettata per offrire un elevato livello di flessibilità e la possibilità di scambiare facilmente diversi algoritmi di MAPF. Per mantenere la formalità della struttura, viene utilizzato un diagramma chiaro che definisce

i moduli e le loro interazioni. Grazie a questo approccio, è possibile integrare nuovi algoritmi senza dover apportare modifiche rilevanti al codice esistente.

Per integrare efficacemente X* nel MAOF, il primo passo è stato dichiarare il nuovo nome "XSTAR" all'interno del file **MAPFSolver.hpp**. Questa operazione ha permesso di aggiungere il tipo XSTAR nello stesso file. Successivamente, è stato possibile integrare il risolutore nel file **iMAPF.cpp**. In questo file, se l'algoritmo selezionato è X*, viene invocato il risolutore appropriato. Successivamente, è stato dichiarato il file **XSTAR.hpp** per la definizione delle funzioni, e il file **Window.hpp** per le definizioni e implementazioni delle finestre e delle operazioni eseguibili, come l'unione o l'intersezione di due finestre, discusse precedentemente. Infine, il file **XSTAR.cpp** contiene l'implementazione delle sottoprocedure descritte in 4.3.2.

Per utilizzare X*, si può modificare il file **main.cpp** inserendo

```
std::string mapfSolver = parser.addArg("M", "string", &mapfSolver, "XSTAR", "mapfSolver");
```

che imposta X* come default per l'esecuzione, oppure passando la flag `-M XSTAR` da linea di comando. Questo comando invoca il risolutore di X* che, dopo aver calcolato i percorsi per ogni agente tramite A* (già implementato nel framework), avvia l'uso di X*.

X* inizia chiamando la funzione **recXSTAR**, la funzione principale che termina solo quando non ci sono più collisioni, indicando che è stata trovata una soluzione. All'interno di questa funzione, la sottoprocedura *FirstCollisionWindow* identifica la prima collisione, sfruttando la classe per le collisioni già fornita da MAOF (**Conflict.hpp**), e crea la prima finestra che racchiude la collisione e gli agenti coinvolti. Successivamente, la sottoprocedura *PlanIn* viene chiamata per eseguire l'unione delle finestre e poi avviare la pianificazione. In questa fase si cerca di risolvere la collisione identificata. Se la finestra è troppo piccola per risolvere il conflitto, interviene la sottoprocedura *GrowAndReplanIn*, espressa nel codice come **window→Grow()**. Questo processo continua finché la collisione non viene risolta e, se non vengono trovate altre collisioni, si esce dalla funzione **recXSTAR** grazie alla sottoprocedura *ShouldQuit()*.

Riassumendo, l'integrazione di X* nel MAOF costituisce un importante progresso per potenziare la capacità del framework nell'affrontare test su algoritmi di MAPF. Grazie alla sua struttura, il framework ha reso l'implementazione più agevole, consentendo un confronto efficace tra X* e gli altri algoritmi.

5 Analisi sperimentale

In questa sezione, presentiamo un'analisi comparativa delle prestazioni di vari algoritmi risolutivi su un insieme di test. L'obiettivo principale è valutare l'efficienza dei diversi approcci. I dati utilizzati per questa analisi derivano da un insieme di test, ciascuno dei quali rappresenta un problema che gli algoritmi devono risolvere. Per ogni test, abbiamo misurato il tempo di esecuzione, il percorso più breve, la somma dei costi individuali (SIC) e lo spazio occupato in memoria dei cinque algoritmi discussi nelle sezioni precedenti: CBS, ICR, ICTS, ICTS+ID, XSTAR. I risultati sono stati registrati e riportati in un formato CSV per facilitarne l'analisi.

5.1 Implementazione

Tutti gli algoritmi sono stati implementati in C++ utilizzando le librerie standard. Il codice sorgente con l'implementazione di tutti gli algoritmi è disponibile nel framework dell'università[1]. I test sono stati eseguiti su una macchina dotata di un processore Intel Core i7-8550U con frequenza base di 1.8 GHz. Questo processore è un quad-core (4 core fisici) con Hyper-Threading abilitato, permettendo a ciascun core fisico di gestire 2 thread. La macchina è inoltre equipaggiata con 16 GB di SDRAM DDR4 e utilizza il sistema operativo Linux. Il timeout per ogni test è stato fissato a 10 minuti di runtime utilizzando il comando *Timeout 10m* di Linux, mentre il programma *Runlim*[2] è stato impiegato per raccogliere varie informazioni di runtime, tra cui lo spazio occupato in memoria.

5.1.1 Scenario Industriale

Per gli esperimenti abbiamo considerato un magazzino reale, frutto di una collaborazione con un'azienda che opera nel campo dei magazzini assistiti da robot. L'intero magazzino e la sua rappresentazione grafica sono mostrati nella Figura 5.1. Il grafo topologico ottenuto dalla mappa consiste di 406 nodi con archi non orientati. Per gli esperimenti abbiamo suddiviso il magazzino in sottoproblemi come segue [25]:

- WH1 che corrisponde al rettangolo dorato nell'angolo in alto a destra della Figura 5.1.
- WH2 che corrisponde al rettangolo blu nell'angolo in basso a sinistra della Figura 5.1.
- WH2_1 che corrisponde al rettangolo rosso nell'angolo in basso a sinistra della Figura 5.1.
- WH2_2 che corrisponde al rettangolo verde nell'angolo in basso a sinistra della Figura 5.1.
- WH2_1_1 che corrisponde alle prime 4 righe del rettangolo rosso nell'angolo in basso a sinistra della Figura 5.1.
- WH2_1_2 che corrisponde alle ultime 4 righe del rettangolo rosso nell'angolo in basso a sinistra della Figura 5.1.
- WH2_2_1 che corrisponde alle prime 4 righe del rettangolo verde nell'angolo in basso a sinistra della Figura 5.1.
- WH2_2_2 che corrisponde alle ultime 4 righe del rettangolo verde nell'angolo in basso a sinistra della Figura 5.1.
- WH12 che corrisponde a tutto il grafo considerato in Figura 5.1.

Per ogni scenario, sono stati eseguiti 12 test, abbiamo considerato problemi con un numero crescente di agenti scelti tra $\{2, 4, 8\}$ e un numero crescente di obiettivi scelti tra $\{0, 1, 2, 4, 8, 10\}$. Gli obiettivi sono stati generati per somigliare agli obiettivi tipici delle attività logistiche svolte nel magazzino considerato [25].

A seguire, sono presentati i risultati dei test tramite tabelle (5.1 5.2 5.3 5.4 5.5) che offrono un riassunto per ciascuna sezione del magazzino e per ogni algoritmo. I tempi vengono misurati

[1] https://gitlab.com/chaff800/MAOF/-/tree/dev
[2] https://github.com/arminbiere/runlim

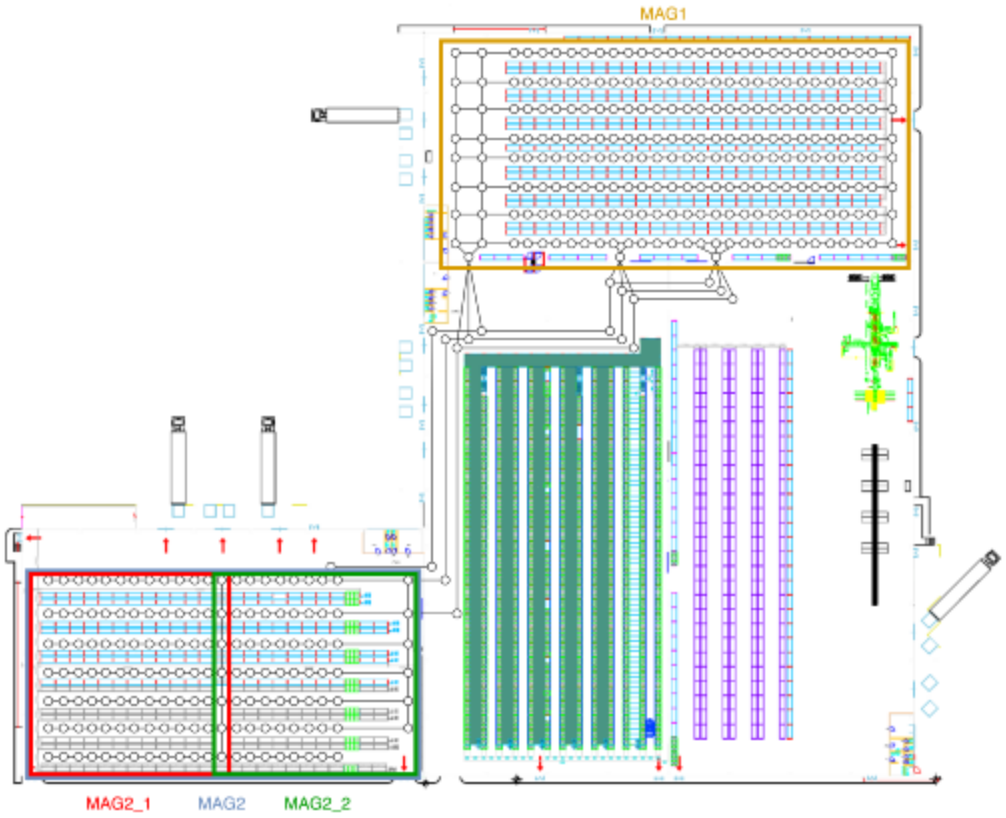

Figura 5.1: Lo schema del magazzino considerato per i test [25]

in millisecondi (ms), l'esecuzione media non tiene conto dei timeout, mentre lo spazio occupato in memoria viene espresso in megabyte (MB). Lo spazio viene calcolato solo per i test superati e i test andati in timeout. La colonna "TO" rappresenta il conteggio dei timeout totali, che si verificano quando un test supera i 10 minuti di esecuzione. La colonna "ERR" indica il numero di errori, che possono includere: l'algoritmo A* non riesce a trovare un percorso valido, si verifica una segmentation fault, la soluzione non è valida.

Test (Totali)	**Test Superati**	**TO**	**ERR**	**Media esecuzione(ms)**	**Media spazio(MB)**
WH1 (12)	8	4	0	24973,4735	17,83
WH2 (12)	7	5	0	78630,6007	32,53
WH2_1 (12)	5	7	0	2042,901851	62,78
WH2_1_1 (12)	4	8	0	6901,525368	142,99
WH2_1_2 (12)	4	8	0	19233,4616	217,03
WH2_2 (12)	9	3	0	26101,20401	30,83
WH2_2_1 (12)	7	5	0	241,3188844	130,24
WH2_2_2 (12)	5	7	0	1218,357144	216,53
WH12 (12)	8	4	0	1462,615556	18,85

Tabella 5.1: Risultati dei test per CBS

Dopo aver presentato le tabelle dettagliate per ogni suddivisione del magazzino e per ciascun algoritmo, segue la Tabella 5.6, che riassume le analisi condotte. Vengono presentate diverse metriche per valutare le prestazioni degli algoritmi:

- **Efficienza** si riferisce al rapporto tra il numero di soluzioni valide e il numero di soluzioni non valide. Questo rapporto offre un'indicazione della frequenza con cui l'algoritmo riesce a trovare una soluzione rispetto a quando fallisce. Un valore più alto di efficienza suggerisce una maggiore affidabilità dell'algoritmo.

Test (Totali)	Test Superati	TO	ERR	Media esecuzione(ms)	Media spazio(MB)
WH1 (12)	6	1	5	18404,19697	15,02
WH2 (12)	4	3	5	5,421845	112,79
WH2_1 (12)	2	5	5	11,670186	84,14
WH2_1_1 (12)	4	8	0	7060,985388	144,01
WH2_1_2 (12)	4	8	0	19658,24685	217,06
WH2_2 (12)	5	3	4	1419,799904	84,04
WH2_2_1 (12)	6	5	1	328,3103252	130,61
WH2_2_2 (12)	5	7	0	1222,499388	216,13
WH12 (12)	6	0	6	7,118678333	15,05

Tabella 5.2: Risultati dei test per ICR

Test (Totali)	Test Superati	TO	ERR	Media esecuzione(ms)	Media spazio(MB)
WH1 (12)	7	5	0	1653,523629	63,24
WH2 (12)	6	6	0	3179,96565	79,23
WH2_1 (12)	6	5	1	83542,39877	63,92
WH2_1_1 (12)	5	5	2	151526,8428	64,98
WH2_1_2 (12)	4	7	1	1932,749875	61,64
WH2_2 (12)	8	1	3	75536,69442	34,88
WH2_2_1 (12)	5	3	4	60,403464	24,94
WH2_2_2 (12)	4	5	3	1710,058563	59,97
WH12 (12)	8	4	0	2527,709313	43,00

Tabella 5.3: Risultati dei test per ICTS

- **Media Spazio Occupato** è calcolata dividendo lo spazio totale occupato (in megabyte) per la somma del numero di soluzioni valide e del numero di timeout. Questa metrica fornisce una misura dell'efficienza della memoria dell'algoritmo in relazione al numero di esecuzioni che non hanno generato errori.

- **Tempo Totale Soluzioni** rappresenta la somma complessiva dei tempi di esecuzione sia per le soluzioni valide che per i timeout, espressa in secondi. Questa metrica permette di valutare il carico temporale totale richiesto dall'algoritmo durante i test.

- **Media Tempi** è ottenuta dividendo il Tempo Totale Soluzioni per la somma del numero di soluzioni valide e dei timeout. Questa metrica fornisce una stima del tempo medio di esecuzione per ciascun test, considerando sia le esecuzioni di successo che i timeout.

- **Tempo Totale w/o Timeout** è il tempo totale di esecuzione considerando solo le soluzioni valide, escludendo quindi i timeout. Questa metrica è utile per valutare le prestazioni temporali dell'algoritmo in condizioni ideali, quando riesce a trovare una soluzione.

- **Media Tempi w/o Timeout** rappresenta la media dei tempi di esecuzione per le sole soluzioni valide, ottenuta dividendo il Tempo Totale w/o Timeout per il numero di soluzioni valide. Questa metrica fornisce un'idea più precisa del tempo di esecuzione richiesto dall'algoritmo quando è in grado di trovare una soluzione entro il tempo limite.

Queste metriche forniscono una panoramica completa delle prestazioni degli algoritmi testati, evidenziando sia l'efficienza nella risoluzione dei problemi che l'uso delle risorse di sistema, come il tempo di esecuzione e lo spazio di memoria.

La Figura 5.2 presenta un istogramma dei cinque algoritmi considerati. Come si può osservare, l'algoritmo X* risulta essere uno dei più esigenti in termini di spazio di memoria, occupando circa 13,000 MB. A seguire, gli algoritmi ICR e CBS, che pur registrando numerosi test falliti nel caso di ICR, utilizzano circa 10,000 MB ciascuno. Gli algoritmi ICTS e ICTS+ID, invece, sono i meno

Test (Totali)	Test Superati	TO	ERR	Media esecuzione(ms)	Media spazio(MB)
WH1 (12)	8	4	0	4810,253518	60,78
WH2 (12)	6	6	0	5870,376087	74,93
WH2_1 (12)	5	6	1	1290,502258	61,26
WH2_1_1 (12)	4	6	2	28164,83126	60,51
WH2_1_2 (12)	4	7	1	3963,547075	69,78
WH2_2 (12)	7	2	3	4440,743528	32,33
WH2_2_1 (12)	5	3	4	100,4533874	31,25
WH2_2_2 (12)	4	5	3	3414,322665	51,38
WH12 (12)	8	4	0	4507,006269	47,95

Tabella 5.4: Risultati dei test per ICTS+ID

Test (Totali)	Test Superati	TO	ERR	Media esecuzione(ms)	Media spazio(MB)
WH1 (12)	7	5	0	39547,25183	45,35
WH2 (12)	6	6	0	18384,46217	42,12
WH2_1 (12)	5	7	0	2061,422538	70,98
WH2_1_1 (12)	4	8	0	7366,782974	289,79
WH2_1_2 (12)	4	8	0	19766,32653	184,88
WH2_2 (12)	9	3	0	28290,80411	24,59
WH2_2_1 (12)	7	5	0	402,8651453	21,83
WH2_2_2 (12)	5	7	0	1241,699039	389,90
WH12 (12)	8	4	0	2662,521238	21,18

Tabella 5.5: Risultati dei test per X*

	CBS	ICR	ICTS	ICTS+ID	XSTAR
N° Soluzioni valide	57	42	53	51	55
N° Timeout	51	40	41	43	53
N° Segmentation Fault	0	0	12	12	0
N° A* non valido	0	26	0	0	0
N° Soluzione non valida	0	0	2	2	0
Totale Errori	51	66	55	57	53
Efficienza	1.12	0.64	0.96	0.89	1.04
Spazio Totale Occupato (MB)	10,435.30	10,610.60	5,310.10	5,254.00	13,087.40
Media Spazio Occupato (MB)	96.62	129.40	56.49	55.89	121.18
Tempo Totale Soluzioni (s)	31,719.35	24,232.57	26,528.93	26,089.97	32,590.92
Media Tempi (s)	293.70	295.52	282.22	277.55	301.77
Tempo Totale w/o Timeout (s)	1,119.35	232.57	1,928.93	289.97	790.92
Media Tempi w/o Timeout (s)	19.64	5.54	36.39	5.69	14.38

Tabella 5.6: Riassunto delle analisi condotte per ogni algoritmo.

dispendiosi in termini di memoria, con un utilizzo complessivo di circa 5,000 MB ciascuno. Ulteriori dettagli sono riportati nella Tabella 5.6, nella riga denominata *Spazio Totale Occupato*. Invece la Figura 5.3 fornisce un riassunto dettagliato dei risultati ottenuti, evidenziando la distribuzione degli errori per ciascun algoritmo. Per ogni algoritmo, vengono mostrati il numero di test completati con successo, i test che sono andati in timeout, quelli che hanno generato un segmentation fault, gli errori dovuti a un percorso non trovato da A*, e le soluzioni non valide. Confrontando questi dati con quelli riportati nella Tabella 5.6, si può osservare che CBS risulta essere l'algoritmo più efficiente in termini di soluzioni valide trovate, seguito da X*, ICTS, ICTS+ID e infine ICR.

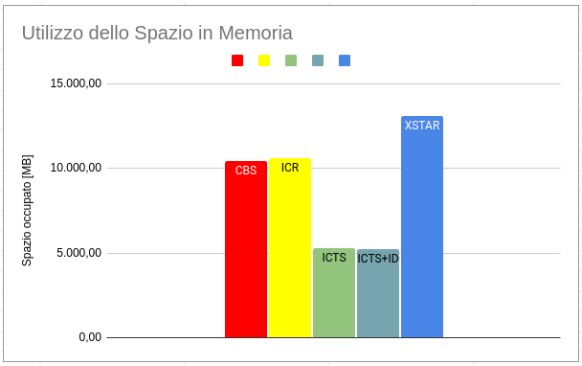

Figura 5.2: Utilizzo dello spazio in memoria (MB)

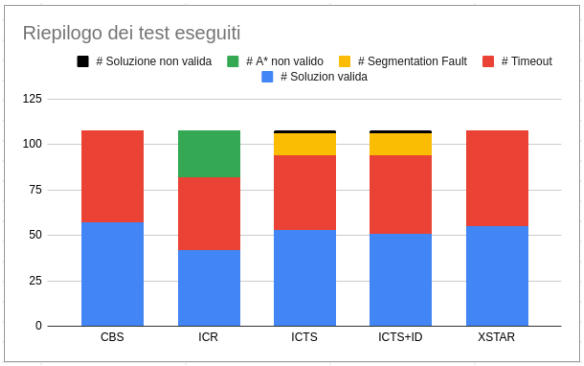

Figura 5.3: Riepilogo dei test condotti nel magazzino

5.2 Confronto tra gli algoritmi

In questa sezione, analizziamo le prestazioni dei vari algoritmi utilizzando un grafico di sopravvivenza (Figura 5.4, 5.5) e successivamente esaminiamo il confronto diretto di ciascun algoritmo con X* tramite grafici di dispersione.

La Figura 5.4 presenta il grafico di sopravvivenza per i primi 25 test che hanno riportato una soluzione. I test sono ordinati in base ai tempi di esecuzione crescenti, con il tempo cumulativo di esecuzione in millisecondi (ms) riportato sulle ordinate e il numero di test sulle ascisse. Dal grafico, emerge chiaramente che l'algoritmo ICTS mostra le peggiori prestazioni, distinguendosi per i tempi di esecuzione più elevati a parità di test. Seguono ICTS+ID, X* e CBS, che presentano tempi di esecuzione simili tra loro. In contrasto, l'algoritmo ICR si avvicina maggiormente alla linea del *Virtual Best*, che rappresenta i risultati migliori ottenuti per ciascun test.

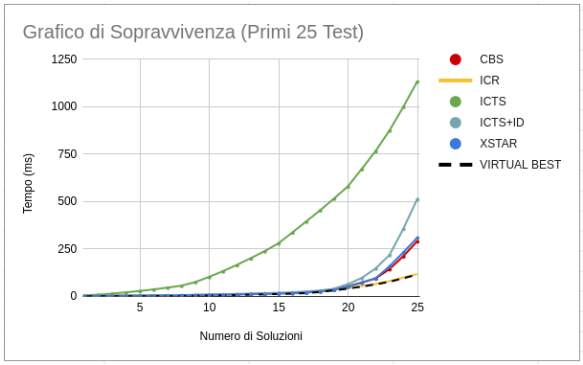

Figura 5.4: Grafico di sopravvivenza (Primi 25 test)

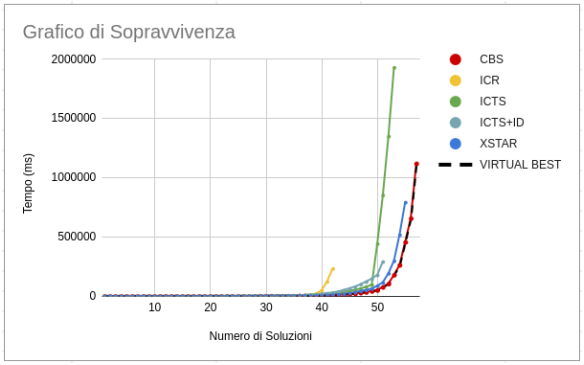

Figura 5.5: Grafico di sopravvivenza complessivo

Invece, la Figura 5.5 presenta il grafico di sopravvivenza complessivo. Come evidenziato nella Figura 5.2 e nella Tabella 5.6, l'algoritmo ICR è tra quelli che falliscono più rapidamente, riuscendo a completare solo 42 test. Sebbene ICTS riesca a completare più test rispetto a ICTS+ID, impiega significativamente più tempo per farlo. Gli algoritmi X* e CBS sono in testa, con X* che completa 55 test avvicinandosi notevolmente alla linea del *Virtual Best*. Tuttavia, CBS ottiene i migliori risultati, portando a termine 57 test e seguendo chiaramente la linea del *Virtual Best*.

Nella prossima sezione, confronteremo gli algoritmi utilizzati con X* tramite grafici di dispersione. Sulle ascisse sarà riportato il numero del test, mentre sulle ordinate il tempo di esecuzione in millisecondi. La linea del *Virtual Best* è calcolata considerando esclusivamente i due algoritmi presi in esame. I test sono stati ordinati in modo crescente in base al tempo di esecuzione per facilitarne la visualizzazione.

5.2.1 X* e CBS

Il grafico a dispersione in Figura 5.6 confronta le prestazioni di due algoritmi di ricerca, XSTAR e CBS, in termini di tempo di esecuzione e numero di test superati. Ogni punto del grafico rappresenta un singolo test. I punti blu rappresentano i risultati di XSTAR, mentre i punti rossi rappresentano i

risultati di CBS. La linea tratteggiata nera rappresenta il *Virtual Best*. Come mostrato dal grafico, nei primi test l'algoritmo X* risulta più veloce rispetto a CBS. Tuttavia, con l'aumentare della complessità dei test, CBS dimostra una performance superiore. Si osserva inoltre che X* inizia a fallire prima, raggiungendo la soglia del timeout (600.000 ms), mentre CBS riesce a risolvere ancora alcuni test. Questo andamento evidenzia come CBS sia più robusto nei confronti di test complessi, riuscendo a mantenere prestazioni accettabili anche quando X* non riesce più a completare i test entro il tempo limite.

5.2.2 X* e ICR

Il grafico a dispersione in Figura 5.7 confronta le prestazioni di due algoritmi di ricerca, XSTAR e ICR, in termini di tempo di esecuzione e numero di test superati. Nei primi venti test, X* e ICR mostrano prestazioni comparabili. Tra il ventesimo e il trentesimo test, emerge un evidente vantaggio di ICR, che purtroppo viene meno oltre il trentesimo test, come evidenziato dal grafico di sopravvivenza in Figura 5.5, che mostra come ICR fallisca prima. A partire dal test 43, è X* a seguire la linea del virtual best, continuando a fornire soluzioni anche quando ICR non è più in grado di farlo. Questa analisi evidenzia la variabilità delle prestazioni degli algoritmi a seconda della complessità dei test. Mentre ICR mostra inizialmente un vantaggio in termini di tempo di esecuzione, X* dimostra una maggiore robustezza nel lungo termine, riuscendo a completare un numero maggiore di test complessi.

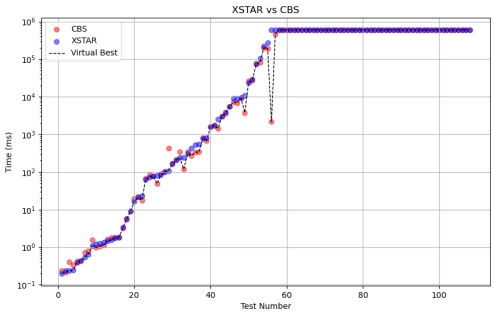

Figura 5.6: Confronto tra X* e CBS attraverso l'uso di un grafico a dispersione.

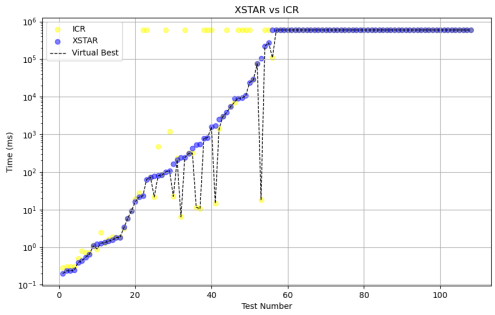

Figura 5.7: Confronto tra X* e ICR attraverso l'uso di un grafico a dispersione.

5.2.3 X* e ICTS

Il grafico a dispersione in Figura 5.8 confronta le prestazioni di due algoritmi di ricerca, XSTAR e ICTS, in termini di tempo di esecuzione e numero di test superati. Nei primi venti test, X* risulta significativamente superiore rispetto a ICTS, con una differenza di prestazioni ben evidente. Dal ventesimo test in poi, le prestazioni dei due algoritmi diventano più simili. Nei test più complessi, ICTS riesce a trovare soluzioni in tempi inferiori rispetto a X*. Tuttavia, è importante notare che ICTS termina i suoi test prima di X*, poiché quest'ultimo riesce a risolvere due test in più. In sintesi, questa analisi mostra come X* abbia un vantaggio iniziale netto rispetto a ICTS, ma con l'aumentare della complessità dei test, ICTS dimostra una maggiore efficienza nel trovare soluzioni più velocemente. Nonostante ciò, la capacità di X* di risolvere un numero maggiore di test suggerisce una maggiore robustezza complessiva.

5.2.4 X* e ICTS+ID

Il grafico a dispersione in Figura 5.9 confronta le prestazioni di due algoritmi di ricerca, XSTAR e CBS, in termini di tempo di esecuzione e numero di test superati. Nei primi venti test, ICTS+ID e X* mostrano prestazioni molto simili, con X* che segue più da vicino la linea del virtual best. Tra il ventesimo e il quarantesimo test, salvo alcune eccezioni, X* dimostra una maggiore efficienza rispetto a ICTS+ID. Tuttavia, nei test più complessi, ICTS+ID riesce a trovare soluzioni più rapidamente rispetto a X*. Come ICTS, anche ICTS+ID termina i test prima di X*, completando quattro test in meno.

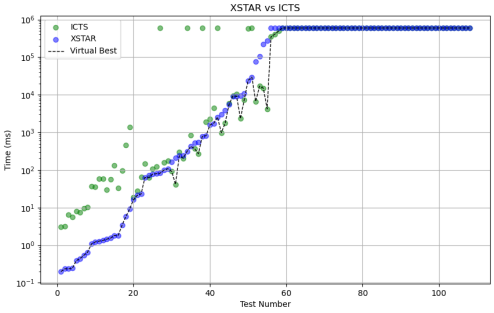

Figura 5.8: Confronto tra X* e ICTS attraverso l'uso di un grafico a dispersione.

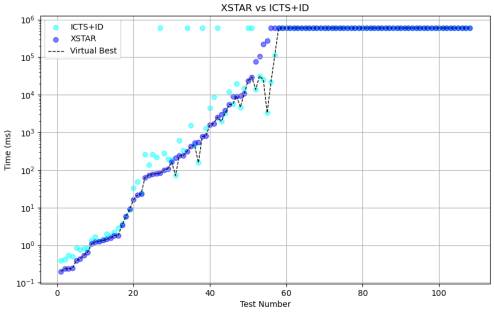

Figura 5.9: Confronto tra X* e ICTS+ID attraverso l'uso di un grafico a dispersione.

6 Conclusioni e sviluppi futuri

In sintesi, dalle analisi dei grafici a dispersione e dei risultati sperimentali emerge chiaramente che ci sono differenze di prestazioni tra gli algoritmi di ricerca X*, CBS, ICR, ICTS e ICTS+ID. Durante i test iniziali, sia X* che ICR hanno dimostrato di essere estremamente efficienti, mantenendo un alto livello di prestazioni e seguendo da vicino la linea del virtual best. In ogni caso, man mano che i test diventavano più complessi, CBS ha dimostrato di essere in grado di risolvere problemi impegnativi entro i tempi previsti con maggiore efficacia. Nonostante ICTS abbia inizialmente mostrato una prestazione inferiore, è riuscito a superare X* nei test più impegnativi, anche se ha completato meno test nel complesso. ICTS+ID, simile a ICTS, si è dimostrato efficiente nel risolvere rapidamente anche i test più complessi. Tuttavia, ha mostrato una minore robustezza rispetto a X* nell'insieme delle situazioni. Sebbene l'utilizzo di Independence Detection abbia permesso di individuare soluzioni più rapidamente rispetto a non adottarlo, questo approccio ha comportato un calo nel numero di test superati.

È importante notare che molti dei test condotti erano troppo semplici per evidenziare differenze significative tra gli algoritmi. In particolare, le prestazioni di X* e CBS risultavano spesso quasi identiche, con curve dei grafici a dispersione che in molti punti erano quasi sovrapponibili. Questo suggerisce che, in scenari meno complessi, entrambi gli algoritmi sarebbero in grado di fornire soluzioni rapide ed efficienti, rendendo difficile determinare quale dei due sia effettivamente superiore.

I risultati indicano che non esiste un chiaro vincitore tra gli algoritmi analizzati: ciascuno presenta vantaggi e svantaggi specifici. CBS si distingue per la sua robustezza nei confronti di test complessi, mentre X* offre ottime prestazioni nei test meno impegnativi. ICTS e ICTS+ID si sono dimostrati efficienti nel risolvere rapidamente test complessi, ma a discapito della capacità di completare un numero maggiore di test.

Questa ricerca apre la strada a future indagini che potrebbero includere lo sviluppo di nuove euristiche per risolvere problemi difficili che emergono in scenari reali, nonché la progettazione di nuovi algoritmi che combinino i punti di forza di ciascun approccio. Inoltre, sarà interessante esplorare metodi che integrino la pianificazione delle missioni nel problema MAPF, considerando variabili aggiuntive. Questi miglioramenti potrebbero portare a soluzioni più efficaci e ottimizzate per la gestione delle risorse e l'allocazione dei compiti in ambienti complessi e dinamici.

Bibliografia

[1] Sandip Aine, P. P. Chakrabarti, and Rajeev Kumar. Awa*-a window constrained anytime heuristic search algorithm. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, page 2250–2255, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[2] M. Barer, Guni Sharon, Roni Stern, and A. Felner. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. *Frontiers in Artificial Intelligence and Applications*, 263:961–962, 01 2014.

[3] Gustav Björdal, Pierre Flener, Justin Pearson, Peter Stuckey, and Guido Tack. Solving satisfaction problems using large-neighbourhood search. pages 55–71, 09 2020.

[4] Shailesh Chandra. Safety-based path finding in urban areas for older drivers and bicyclists. *Transportation Research Part C: Emerging Technologies*, 48:143–157, 2014.

[5] Shushman Choudhury, Kiril Solovey, Mykel Kochenderfer, and Marco Pavone. Coordinated multi-agent pathfinding for drones and trucks over road networks, 2021.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[7] E.W. DIJKSTRA. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[8] Lucía Díaz-Vilariño, Pawel Boguslawski, Kourosh Khoshelham, and Henrique Lorenzo. Obstacle-aware indoor pathfinding using point clouds. *ISPRS International Journal of Geo-Information*, 8(5), 2019.

[9] Stefan Edelkamp and Stefan Schrödl. *Heuristic Search - Theory and Applications.* 01 2012.

[10] Ariel Felner, Roni Stern, Solomon Shimony, Eli Boyarski, Meir Goldenberg, Guni Sharon, Nathan Sturtevant, Glenn Wagner, and Pavel Surynek. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. *Proceedings of the International Symposium on Combinatorial Search*, 8:29–37, 09 2021.

[11] Rodrigo N. Gómez, Carlos Hernández, and Jorge A. Baier. Solving sum-of-costs multi-agent pathfinding with answer-set programming. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(06):9867–9874, Apr. 2020.

[12] Shuai D. Han and Jingjin Yu. Integer programming as a general solution methodology for path-based optimization in robotics: Principles, best practices, and applications. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2019*, IEEE International Conference on Intelligent Robots and Systems, pages 1890–1897, United States, November 2019. Institute of Electrical and Electronics Engineers Inc. Publisher Copyright: © 2019 IEEE.; 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2019 ; Conference date: 03-11-2019 Through 08-11-2019.

[13] Khoi Hoang, Ferdinando Fioretto, William Yeoh, Enrico Pontelli, and Roie Zivan. A large neighboring search schema for multi-agent optimization: 24th international conference, cp 2018, lille, france, august 27-31, 2018, proceedings. pages 688–706, 08 2018.

[14] Ivan Kuric, Vladimir Bulej, Milan Saga, and Peter Pokorny. Development of simulation software for mobile robot path planning within multilayer map system based on metric and topological maps. *International Journal of Advanced Robotic Systems*, 14(6):1729881417743029, 2017.

[15] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter Stuckey, and Sven Koenig. Anytime multi-agent path finding via large neighborhood search. pages 4127–4135, 08 2021.

[16] Jiaoyang Li, The Anh Hoang, Eugene Lin, Hai L. Vu, and Sven Koenig. Intersection coordination with priority-based search for autonomous vehicles. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(10):11578–11585, Jun. 2023.

[17] Maxim Likhachev, Geoffrey J Gordon, and Sebastian Thrun. Ara* : Anytime a* with provable bounds on sub-optimality. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems*, volume 16. MIT Press, 2003.

[18] Ryan Luna and Kostas Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. pages 294–300, 01 2011.

[19] Hang Ma. Graph-based multi-robot path finding and planning. *Current Robotics Reports*, 3(3):77–84, Sep 2022.

[20] Ashish Macwan, Julio Vilela, Goldie Nejat, and Beno Benhabib. A multirobot path-planning strategy for autonomous wilderness search and rescue. *IEEE Transactions on Cybernetics*, 45(9):1784–1797, 2015.

[21] Ramkumar Natarajan, Muhammad Saleem, Sandip Aine, Maxim Likhachev, and Howie Choset. A-mha*: Anytime multi-heuristic a*. *Proceedings of the International Symposium on Combinatorial Search*, 10:192–193, 09 2021.

[22] Dian Rachmawati and Lysander Gustin. Analysis of dijkstra's algorithm and a* algorithm in shortest path problem. *Journal of Physics: Conference Series*, 1566(1):012061, jun 2020.

[23] Abdul Rafiq, Tuty Asmawaty Abdul Kadir, and Siti Normaziah Ihsan. Pathfinding algorithms in game development. *IOP Conference Series: Materials Science and Engineering*, 769(1):012021, feb 2020.

[24] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[25] Enrico Saccon, Luigi Palopoli, and Marco Roveri. Comparing multi-agent path finding algorithms in a real industrial scenario. In Agostino Dovier, Angelo Montanari, and Andrea Orlandini, editors, *AIxIA 2022 – Advances in Artificial Intelligence*, pages 184–197, Cham, 2023. Springer International Publishing.

[26] Qandeel Sajid, Ryan Luna, and Kostas Bekris. Multi-agent pathfinding with simultaneous execution of single-agent primitives. *Proceedings of the 5th Annual Symposium on Combinatorial Search, SoCS 2012*, 3:88–96, 01 2012.

[27] Guni Sharon, Roni Stern, Ariel Felner, and Nathan Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66, 02 2015.

[28] Guni Sharon, Roni Stern, Ariel Felner, and Nathan Sturtevant. Conflict-based search for optimal multi-agent path finding. *Proceedings of the AAAI Conference on Artificial Intelligence*, 26(1):563–569, Sep. 2021.

[29] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195:470–495, 2013.

[30] David Silver. Cooperative pathfinding. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 1(1):117–122, Sep. 2021.

[31] Jialin Song, Ravi Lanka, Yisong Yue, and Bistra Dilkina. A general large neighborhood search framework for solving integer linear programs, 2020.

[32] Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*, 1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers, pages 92–95. Publ by IEEE, 1990. 1990 IEEE International Conference on Computer-Aided Design - ICCAD-90 ; Conference date: 11-11-1990 Through 15-11-1990.

[33] Trevor Standley. Finding optimal solutions to cooperative pathfinding problems. volume 1, 01 2010.

[34] Trevor Standley and Richard Korf. Complete algorithms for cooperative pathfinding problems. pages 668–673, 01 2011.

[35] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Bart´ak. Multi-agent pathfinding: Definitions, variants, and benchmark. *Proceedings of the Twelfth International Symposium on Combinatorial Search (SoCS 2019)*, 2021.

[36] Pavel Surynek. Towards optimal cooperative path planning in hard setups through satisfiability solving. pages 564–576, 2012.

[37] Pavel Surynek. Reduced time-expansion graphs and goal decomposition for solving cooperative path finding sub-optimally. page 1916–1922, 2015.

[38] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Efficient sat approach to multi-agent path finding under the sum of costs objective. page 810–818, 2016.

[39] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Modifying optimal sat-based approach to multi-agent path-finding problem to suboptimal variants. *Proceedings of the International Symposium on Combinatorial Search*, 8(1):169–170, September 2021.

[40] Pavel Surynek, Roni Stern, Eli Boyarski, and Ariel Felner. Migrating techniques from search-based multi-agent path finding solvers to sat-based approach. *Journal of Artificial Intelligence Research*, 73:553–618, 02 2022.

[41] Jin Svancara, Dor Atzmon, Klaus Strauch, Roland Kaminski, and Torsten Schaub. Which objective function is solved faster in multi-agent pathfinding? it depends. *Proceedings of the 16th International Conference on Agents and Artificial Intelligence (ICAART 2024)*, 2024.

[42] Justin Svegliato, Prakhar Sharma, and Shlomo Zilberstein. A model-free approach to meta-level control of anytime algorithms. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11436–11442, 2020.

[43] Justin Svegliato, Kyle Hollins Wray, and Shlomo Zilberstein. Meta-level control of anytime algorithms with online performance prediction. *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*.

[44] Justin Svegliato and Shlomo Zilberstein. Adaptive metareasoning for bounded rational agents. 2018.

[45] Kyle Vedder and Joydeep Biswas. X*: Anytime multi-agent path finding for sparse domains using window-based iterative repairs. *Artificial Intelligence*, 291:103417, February 2021.

[46] Mark Wallace and Contact Wallace. Constraint programming. 02 1998.

[47] Jiangxing Wang, Jiaoyang Li, Hang Ma, Sven Koenig, and T. K. Satish Kumar. A new constraint satisfaction perspective on multi-agent path finding. In *Proceedings of 18th International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS '19)*, pages 2253 – 2255, May 2019.

[48] Boris Wilde, Adriaan Mors, and Cees Witteveen. Push and rotate: Cooperative multi-agent path planning. volume 1, pages 87–94, 05 2013.

[49] Jingjin Yu. Constant factor optimal multi-robot path planning in well-connected environments. *Autonomous Robots*, 44, 03 2020.

[50] Jingjin Yu and Steven M. LaValle. Planning optimal paths for multiple robots on graphs. pages 3612–3617, 2013.

[51] Jingjin Yu and Steven M. LaValle. Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics. *IEEE Transactions on Robotics*, 32(5):1163–1177, October 2016.

[52] Dijkstra vs. a* – pathfinding. https://www.baeldung.com/cs/dijkstra-vs-a-pathfinding. ultimo accesso 11/05/2024.

[53] Pseudocodice algoritmo dijkstra. https://it.wikipedia.org/wiki/Algoritmo_di_Dijkstra.

[54] Types of collisions. https://en.wikipedia.org/wiki/Multi-agent_pathfinding.

[55] R. Zhou and Eric A. Hansen. Multiple sequence alignment using anytime a*. In *AAAI/IAAI*, 2002.

Bibliografia delle Immagini

A.1 Esempio Finestra Singola

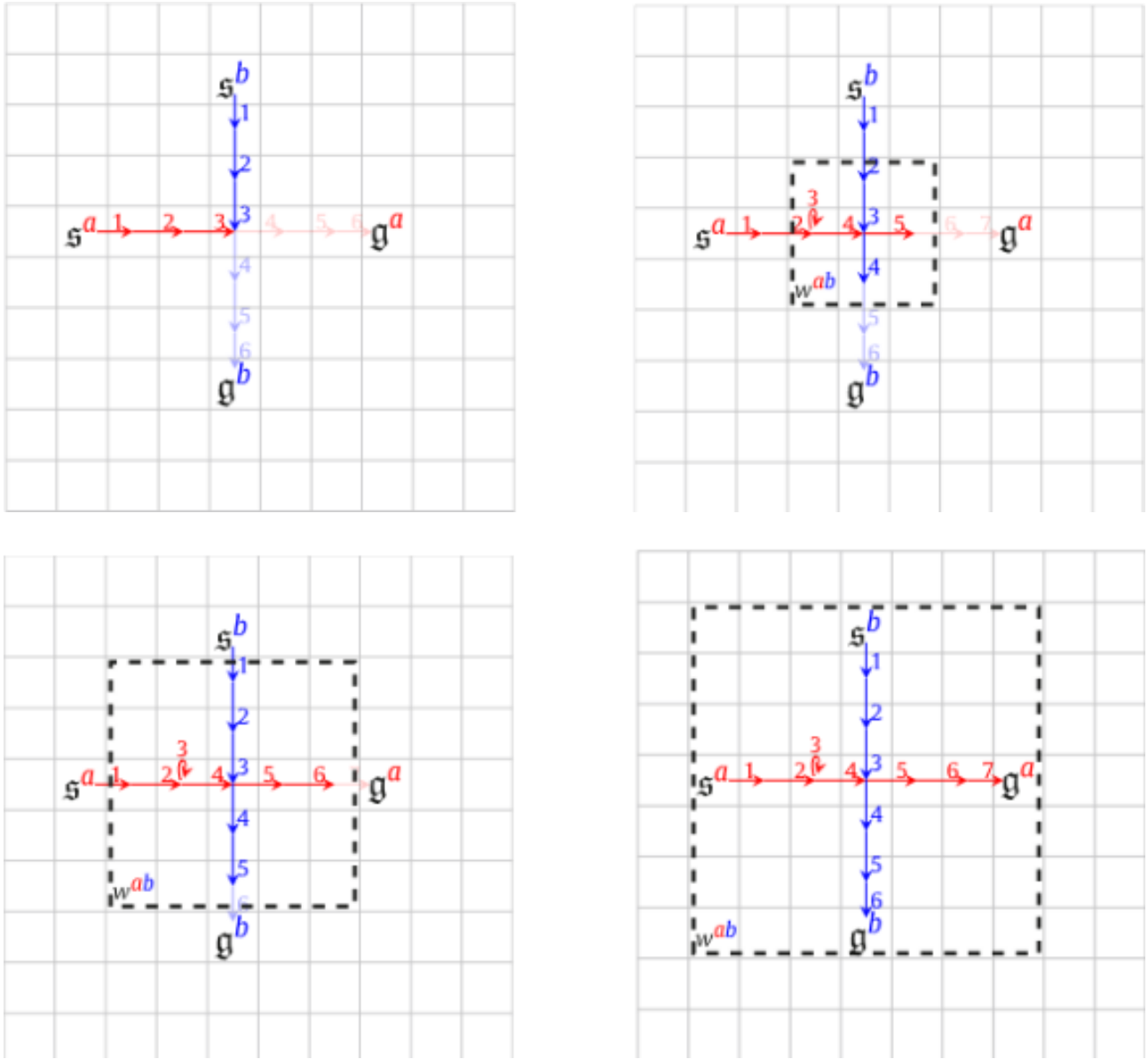

Figura A.1: Presa da [45]

A.2 Processo di Trasformazione per il Riutilizzo dell'Albero di Ricerca in X*

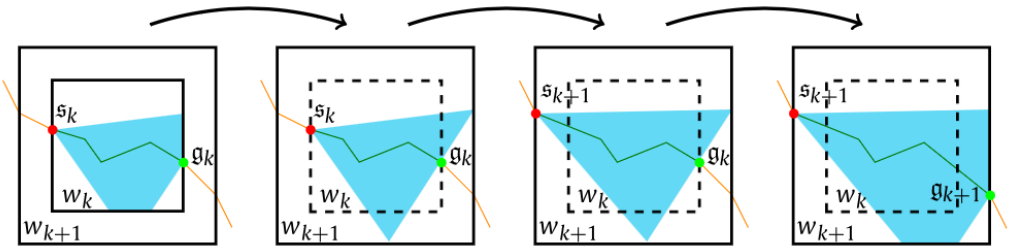

Figura A.2: Presa da [45]

A.3 Insieme fuori dalla Finestra

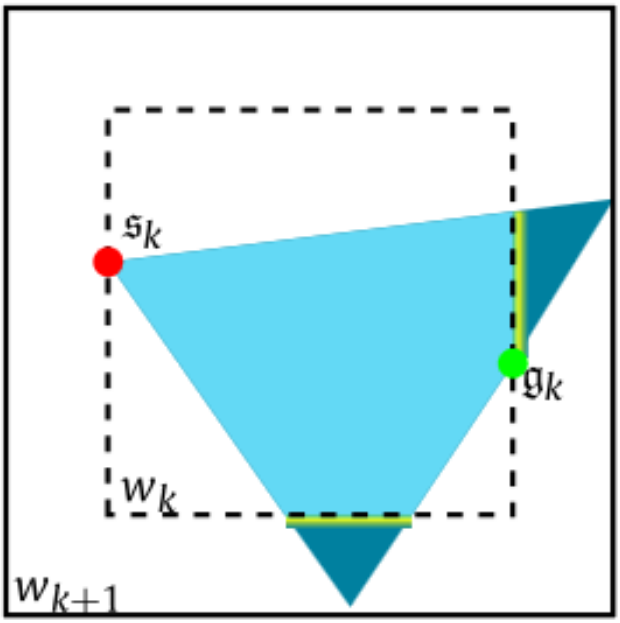

Figura A.3: Presa da [45]

A.4 Valore Chiuso

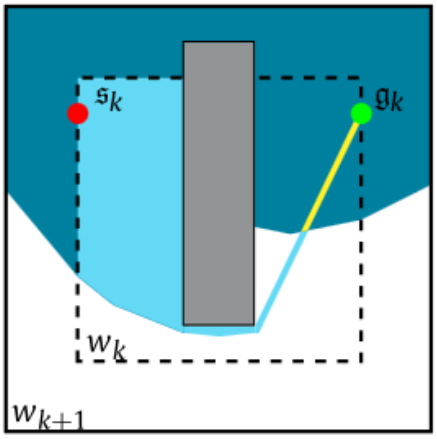

Figura A.4: Presa da [45]

A.5 Scenario Reale

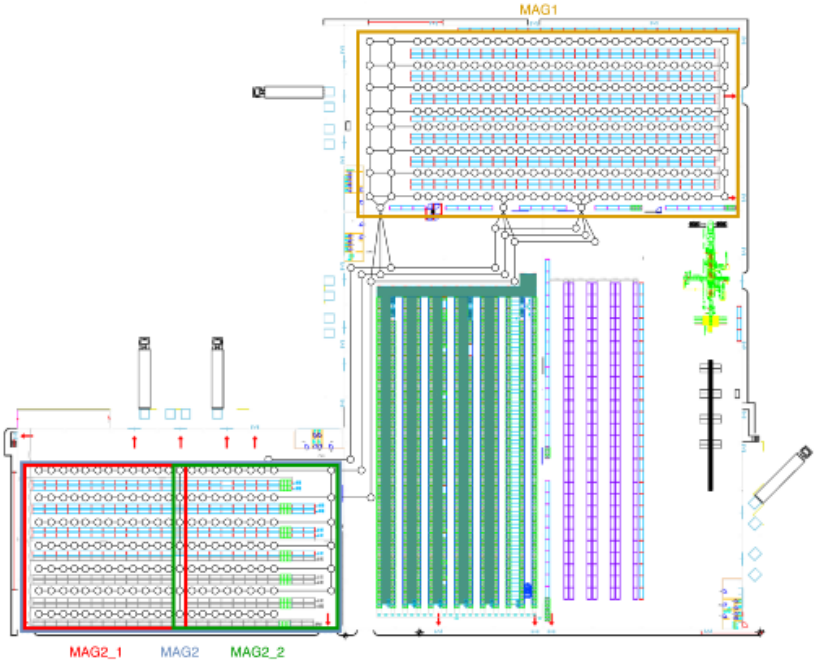

Figura A.5: Presa da [25]