Kandidatarbete 2026

Planering och styrning av en flotta autonoma mobila robotar med centraliserad rörelsespårning

Julia von Brömsen, Elsa Edofsson, Tim Leffler, Oscar Lindblom,
Altaf Rajabi, Debora Strandberg

CHALMERS

Institutionen för Elektroteknik E2
Chalmers Tekniska Högskola
Göteborg 2026

Handledare: Knut Åkesson, Institutionen för Elektroteknik E2
Examinator: Martin Fabian, Institutionen för Elektroteknik E2

Abstract

Autonomous mobile robots are increasingly utilized in logistical workflows and material flow management. The process of automating heavy transportations leads to increased efficiency of material handling while simultaneously decreasing manual labour. This study investigates the coordination of a scalable fleet of autonomous mobile robots (AMR) as they transition from logistical patterns to dynamic scenarios. External real time data about the location of a moving target is obtained from a drone based surveillance system whereas the fleet is traced in real time by a motion capture system with high precision localization. The realisation of the system is achieved through a strategy of efficient path planning and collision avoidance. An A* path planning strategy A* is implemeted which utilizes euclidean distance heuristics to minimize the cost between start and goal positions. This determines the optimal trajectory from inital, to designated position. As the fleet scales, the increasing number of agents leads to collisions and conflicts such as deadlocks. This issue is addressed through a multi agent path finding (MAPF) strategy evaluating a local collision resolver A* (LCRA*). LCRA* is based on the model lifelong priority based search (LPBS) and the benefit of using LCRA* is that it significantly improves computational performance as it does not require global time optimization to handle collisions locally. This results in a smooth transition between a logistical pattern mode or an intruder mode in order to collectively encircle the moving target.

Keywords: Autonomous mobile robots, motion capture, multi agent path finding, search algorithms, A*.

Sammanfattning

Autonoma fordon har i växande utsträckning implementerats inom logistik och i lagermiljöer. Genom att automatisera tunga transporter och reducera manuella moment kan systemen öka effektiviteten i processer för materialhantering. Studien undersöker hur en skalbar flotta autonoma mobila robotar (AMR) kan koordineras vid övergången från rutinmässiga logistiska rörelsemönster till dynamiska scenarier. Genom ett drönarbaserat avsökningssystem erhålls extern information om det okända målet för att identifiera och spåra målet i realtid. Positioneringen av flottans enheter i realtid sker med hög noggrannhet via ett motion capture-system. Realisering av systemet sker genom strategier för effektiv ruttplanering och kollisionshantering. För att möjliggöra koordinering av flottan tillämpas ruttplanering för ett skalbart antal agenter med A*-algoritmen, som använder euklidisk heuristik för att identifiera den mest optimala vägen från startposition till slutposition. När ett flertal agenter används ökar komplexiteten vilket leder till kollisioner och konflikttyper som dödlägen. Dessa konflikttyper hanteras med hjälp av Multi-Agent Path Finding (MAPF) genom en modell av Lifelong priority based search (LPBS) kallad lokalt konfliktlösande A* (LCRA*). Fördelen med att LCRA* hanterar kollisioner lokalt under körning medför att systemet inte kräver global tidskoordinering vilket markant förbättrar beräkningsprestanda. Denna funktionalitet möjliggör för systemet att låta flottan dynamiskt skifta mellan olika lägen som att följa ett logistiskt rörelsemönster eller ett förföljarläge för att kollektivt rama in det rörliga målet.

Akronymer

Nedan följer en sammanställning av de akronymer och tekniska förkortningar som har använts genomgående i projektet. Samtliga akronymer presenteras i alfabetisk ordning.

AMR	Autonomous Mobile Robot
BEV	Bird's Eye View
BFS	Breadth First Search
CBS	Conflict Based Search
DMAPF	Dynamic Multi Agent Path Finding
DoF	Degrees of Freedom
ICBS	Incremental Conflict Based Search
IMU	Inertia Measuring Unit
LCRA*	Locally Collision Resolving A*
LPBS	Lifelong Priority Based Search
MAPF	Multi Agent Path Finding
MAPFR	Multi Agent Path Finding with Real-valued times
ML	Machine Learning
MoCap	Motion Capture system
PBS	Priority Based Search
PE	Pursuit-Evasion
RL	Reinforcement Learning
ROS	Robot Operating System
SAPF	Single Agent Path Finding
SDK	Software Development Kit
SIMC	Skogestad Internal Model Control
SoC	Sum of Cost
TCP	Transmission Control Protocol
TOF	Time Of Flight
UDP	User Datagram Protocol
QTM	Qualisys Track Manager

Innehåll

Akronymer **v**

1 Inledning **2**
1.1 Bakgrund . . . 2
1.2 Syfte . . . 3
1.3 Problemformulering . . . 3
1.4 Avgränsningar . . . 3

2 Ansats **5**
2.1 Positionering . . . 5
2.2 Fysisk testplattform . . . 6
2.3 Ruttplanering . . . 6
2.4 Kollisionshantering . . . 6
2.5 Inspärrning . . . 7

3 Positionering **8**
3.1 Motion Capture . . . 8
3.2 Robot Operating System 2 . . . 9
3.3 Nätverkskommunikation . . . 9
3.4 Implementation . . . 9
3.5 Utvärdering . . . 11

4 Fysisk testplattform **12**
4.1 Styrning av AMR . . . 13
4.2 Kollisionsundvikning . . . 15
4.3 Utvärdering av körförmåga . . . 16

5 Ruttplanering **18**
5.1 Sökalgoritmer . . . 18
5.1.1 Dijkstras algoritm . . . 18
5.1.2 A* . . . 18
5.2 Heuristik för ruttplanering . . . 19
5.2.1 Manhattan . . . 19
5.2.2 Diagonal . . . 20
5.2.3 Euklidisk . . . 20
5.3 Definition av miljö och modell . . . 21
5.3.1 Modellering av lagermiljön . . . 21
5.4 Implementation . . . 23
5.4.1 Implementation av sökalgoritm . . . 23
5.4.2 Heuristikdesign och teknisk motivering . . . 24
5.5 Resultat . . . 25

6 Kollisionshantering **27**
6.1 Planeringsparadigm: val av modell . . . 27
6.1.1 Multi Agent Path Finding – definition och antaganden . . . 27
6.1.2 Lifelong och One-Shot . . . 28
6.1.3 Dynamisk Multi Agent Path Finding . . . 28
6.1.4 Tidsrepresentationer och implikationer . . . 29
6.2 Konflikthanterande algoritmer: översikt . . . 29
6.2.1 Priority Based Search . . . 29
6.2.2 Lifelong Priority Based Search – förbättringar av PBS . . . 30
6.2.3 Locally Collision Resolving A* – vår metod . . . 30
6.3 Konflikter i MAPF: kategorier och betydelse . . . 31
6.3.1 Kollisionstyper med exempel . . . 31
6.3.2 Dödlägen och relaterade problem i MAPF . . . 33
6.4 Implementation . . . 33
6.5 Resultat och analys . . . 35

7 Inspärrning av inkräktare **36**
7.1 Introduktion . . . 36
7.2 Praktisk grund och designval . . . 36
7.2.1 Situation och antaganden . . . 36
7.2.2 Testning och utvärderingsgrund . . . 37
7.2.3 Implementering och metodval . . . 37
7.3 Resultat . . . 39
7.3.1 Blockeringsmetoden, Blockering av noder nära inkräktare . . . 39
7.3.2 Avgränsningsmetoden, minska inkräktarens rörelseområde . . . 41
7.3.3 Hybridlösning, kombination . . . 43
7.3.4 Skalbarhet . . . 45

8 Systemarkitektur och utvärdering **47**

9 Slutsats och utvecklingsmöjligheter **50**

Referenser **51**

A Bilaga 1 **II**

B Bilaga 2 **III**

C Bilaga 3 **V**

D Bilaga 4 **VI**

1 Inledning

Autonoma system har under de senaste decennierna fått en central roll inom lagermiljöer och logistiska tillämpningar [1], [2]. Genom automatisering av processer kan arbetet effektiviseras och optimeras med snabbare arbetsflöden och förbättrad kvalitet. Inom lagerhantering och materialtransport är autonoma fordon ett alternativ för att minska tunga transporter och repetitiva manuella moment, där ett exempel illustreras i Figur 1.1. Effektiviteten är dock inte enbart beroende av enskilda komponenters prestanda, utan även av hur väl de samverkar. I takt med skiftet från industri 4.0 mot industri 5.0 förflyttas fokus från isolerad automatisering i stor skala, till utveckling av autonoma system där enheter kan samarbeta och agera självständigt [3]. I samverkande system som utökas med flera enheter där flera komponenter interagerar, uppstår utmaningar kopplade till kommunikation, planering och skalbarhet.

Figur 1.1: Autonomt fordon i logistikmiljö [4], CC BY-SA 4.0.

1.1 Bakgrund

En metod för att automatisera lagermiljöer är att använda autonoma mobila robotar (AMR) som har förmåga till att interagera med sin dynamiska miljö i realtid [2]. En AMR navigerar utifrån en tilldelad order genom att adaptivt följa en fördefinierad rutt. För att korrigera sin rörelse vid avvikelser använder den information från

lokal sensordata, som exempelvis bildbehandling från kamera eller hjulodometri. Den fördefinierade rutten kommer från en central dator som gör den överordnade ruttplaneringen. En ökning av antalet AMR:er inuti ett system kan bidra till förbättrad produktivitet men medför samtidigt ökad komplexitet och ställer högre krav på effektiv ruttplanering och samordning då risken för kollisioner och dödlägen där AMR:er fastnar ökar.

Navigering i miljöer med statiska hinder är idag väl beprövade [5], [6], där nuvarande forskning istället fokuserar på mer komplexa miljöer med dynamiska hinder. I realistiska tillämpningar, där flera enheter interagerar, ökar komplexiteten avsevärt och ställer högre krav på ruttplanering och kollisionshantering i realtid. Därför måste det dynamiska systemet inte enbart undvika kollisioner, utan även trängsel och dödlägen. Det finns därför ett behov av att utvärdera hur dessa system presterar i verkliga miljöer.

1.2 Syfte

Projektet syftar till att utveckla ett system för koordination och styrning av en skalbar flotta AMR:er i en lagermiljö. Information om ett okänt fordons position tillhandahålls av ett separat drönarbaserat avsökningssystem, som därefter används för att begränsa fordonets rörelsefömåga.

1.3 Problemformulering

För att möjliggöra styrning av flera autonoma fordon i ett skalbart system behöver det utredas hur identifikation av AMR:ernas position kan ske och kommuniceras på ett sätt som möjliggör för planering och kartläggning för både nuvarande och framtida positioner. Ett följdproblem blir därför hur information ska sändas mellan delsystemen. Vidare behöver det utvärderas om systemet kan identifiera ett okänt rörligt mål och, därefter kunna växla till ett läge för att kollektivt omringa inkräktaren och begränsa dess rörelse. En central aspekt för projektet är hur datainformation inuti systemet bör hanteras och bearbetas beroende på dess format och frekvens. Slutligen kvarstår problemet hur centraliserad ruttplanering kan sammanvägas med logistiska krav vid navigering i en fysisk och mjukvarubaserad miljö.

1.4 Avgränsningar

För att möjliggöra projektets genomförande sker följande avgränsningar:

- **Fysisk drönarplattform**
  - Projektet innefattar ej hantering av den fysiska drönarplattformen då det hanteras i ett parallellt projekt.
- **Navigering utanför täckningsområdet**

  - Studien begränsas till en känd och kontrollerad inomhusmiljö. Då det externa kamerasystemet är väggmonterat hanteras inte körning utanför täckningsområdet eftersom systemet är beroende av sensordata som erhålls från positioneringen.
- **Hantering av konkava objekt**
  - Statiska och dynamiska hinder beaktas, men avgränsas till att exkludera konkava objekt på grund av komplexitet vid detektering.
- **Komplexa sökalgoritmer**
  - Studien avgränsas från hantering av komplexa sökalgoritmer då fokus ligger på funktionalitet framför effektivitet och prestanda. Navigeringen avgränsas till stokastisk modellering av inkräktarens rörelse, vilket innebär att inkräktaren inte betraktas som en intelligent agent med eget beslutsfattande.
- **Grafisk miljö**
  - Grafen är fördefinierad och avgränsas från att lokalisera och kartlägga miljön under exekvering.
  - Kapaciteten för noder och kanter begränsas till en agent i taget.
  - Negativa kostnader för kantvikter behandlas inte eftersom dessa vikter inte är representativa för en lagermiljö. Således förutsätts deterministiska kostnader.
- **Skalbarhet**
  - Studien avgränsas till maximalt tio markstyrda AMR:er då beräkningskomplexiteten ökar markant med antalet AMR:er.

2 Ansats

En flotta av AMR:er förutsätter att flera system samarbetar. Delsystemen fattar beslut på olika nivåer utifrån instruktioner från ett centralt styrsystem, som koordinerar och distribuerar kommandon om framtida positioner. Dessa kommandon bearbetas vidare av navigeringsalgoritmer som i sin tur bestämmer AMR:ens nästa position. För att flottan ska fungera effektivt för ett skalbart antal AMR:er i logistikmiljö behöver kommunikationen och systemarkitekturen betraktas som en helhet där logik och hårdvara samverkar på ett ändamålsenligt sätt. För att strukturera arbetet delas problemet upp i flera delproblem, vilka motsvarar centrala funktioner i projektet (se Tabell 2.1). Dessa funktioner omfattar positionering för rörelsespårning samt ruttplanering för beräkning av effektiva rutter. För att hantera konflikter implementeras kollisionshantering medan inspärrning hanterar strategier för att rama in en inkräktare. Insamling av data, ruttplanering, kollisionshantering och inspärrning av inkräktare görs på en centraldator av modell NVIDIA Jetson Thor[1]. Slutligen exekverar en fysisk testplattform med AMR:er rörelsekommandon. Samtliga funktioner som har implementerats finns tillgängliga på GitLab[2].

Tabell 2.1: Insignal och utsignal för respektive delproblem för en flotta autonoma mobila robotar.

Delproblem	**Insignal**	**Utsignal**
Positionering	Emitterat infrarött ljus.	Aktuell positionsdata.
Fysisk testplattform	Aktuell position och målposition.	Fysisk förflyttning.
Ruttplanering	Aktuell position och målposition.	Rutt från start till mål.
Kollisionshantering	Aktuell position från flera AMR:er.	Nästa position för konfliktfri rutt.
Inspärrning	Aktuell position och inkräktarens position.	Målpositioner för inringning.

2.1 Positionering

Delproblemet positionering avser att behandla den aktuella positionen för varje AMR. För att möjliggöra en väl fungerande rörelsespårning krävs kontinuerlig tillgång till positionskoordinater i ett koordinatsystem. För att lösa detta problem valdes ett Motion Capture-system (MoCap) istället för traditionell RGB-kamerateknik, då det minskar behovet av resurskrävande bildanalys. Därtill motverkas risken att

[1]https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-thor/, hämtad 2026-05-13.

[2]https://git.chalmers.se/ninuss/kandidatarbete-eenx16.

en AMR behöver ta sig till sin målposition utan kännedom om sin aktuella position. Centraldatorn behöver kontinuerligt få ingående information om aktuell position vilken skickas vidare till ruttplaneringen för att göra en effektiv planering av dess bana. Delproblemet kopplas till resterande delproblem genom att dessa delsystem likaså behöver tillgång till positionsdata. Den fysiska positionen tas in i systemet, digitaliseras och görs tillgänglig för resterande delsystem. Konsekvensen om positionsdata fördröjs eller förloras innebär att ruttplaneringen blir felaktig eller att AMR:erna stannar.

2.2 Fysisk testplattform

En fysisk plattform som kan omsätta beräknad data till faktisk rörelse i fysisk miljö för att utvärdera programvara. Förflyttningen behöver ske med stabil kurs och hög noggrannhet vilket förutsätter att AMR:en kontinuerligt kan få information om sin position och korrigera sin kurs baserat på den fysiska miljön. Eventuell signalförlust eller fördröjning av positionsdata behöver hanteras på ett sätt så att AMR:en inte hamnar i ett osäkert tillstånd. Konsekvensen av bristfällig styrning är att AMR:erna blir obrukbara i drift. Om de inte regleras korrekt riskerar de att köra utanför kurs och utan att kunna identifiera hinder uppstår risken att de kolliderar eller misslyckas med att exekvera en ny rutt.

2.3 Ruttplanering

Delproblemet ruttplanering avser att planera rörelsemönstret för en AMR från startposition till målposition. I detta sammanhang behandlas en AMR som en agent då den verkar i en förenklad digital miljö. Algoritmernas uppgift är att läsa in agentens aktuella position via indata från MoCap och skicka ut information om nästa position till AMR:n. Om detta brister innebär det att den beräknade rutten inte blir tillräckligt resurseffektiv, eller att beräkningsprocessen kräver onödigt mycket minnesresurser.

2.4 Kollisionshantering

Delproblemet kollisionshantering hanterar, i likhet med ruttplaneringen, rörelsemönstret för en AMR från startposition till målposition. När systemet utökas från en agent till ett flertal agenter krävs hantering av diskreta optimeringsmål för att minimera den sammanlagda ruttkostnaden. Ett problem är att kollisioner och blockeringar i rutterna kommer att uppstå i olika situationer. Sådana problem kan vara när dödlägen (deadlocks) uppstår och agenter fastnar utan möjlighet till förflyttning eller situationer där rutter med högre prioritet blockerar rutter med lägre prioritet. Algoritmerna behöver kunna hantera dessa låsningar för att kunna generera den utdata som krävs för att styra agenterna vidare till nästa position.

2.5 Inspärrning

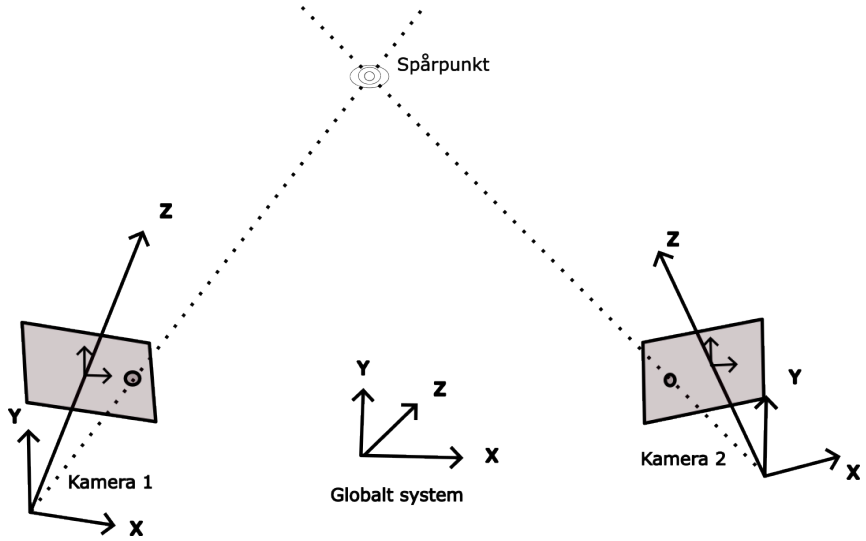
Delproblemet avser att behandla kollektiv inspärrning av en inkräktare. Baserat på ingående information om inkräktarens position samt var samtliga AMR:er befinner sig behöver systemet kunna avgöra om AMR:erna ska inta ett logistiskt rörelsemönster eller växla till ett förföljarläge för kollektiv inramning kring inkräktaren. När en inkräktare upptäckts behövs en strategi för att beräkna målpositioner för AMR:er, som därefter ska skickas till den fysiska hårdvaran. Konsekvensen om delproblemet inte löses är att systemet inte kan identifiera AMR:ernas positioner eller inkräktarens, och misslyckas med att ta fram målpositioner för att spärra in inkräktaren.

3 Positionering

Positionering är en central del i styrning av AMR:er, eftersom planering av rörelse kräver kännedom om fordonens aktuella position. Delproblemet positionering avser att undersöka hur positionsdata för AMR:erna kan erhållas i realtid samt hur denna data kan överföras till respektive fordon. Detta avsnitt beskriver de tekniker och principer som ligger till grund för inhämtning och överföring av positionsdata samt implementering av dem.

3.1 Motion Capture

Motion Capture (MoCap) är en teknik för att mäta och följa rörelser i tre dimensioner med hög noggrannhet, baserat på optisk spårning av markörer. Tekniken bygger på att flera kameror registrerar infrarött ljus som reflekteras från små markörer placerade på det objekt som ska följas. Genom att kombinera information från flera kameror kan markörernas position i rummet beräknas med triangulering (se Figur 3.1). Detta gör det möjligt att i realtid ta fram exakta positionsdata med hög precision och uppdateringsfrekvens.



Figur 3.1: Princip för triangulering mellan två kameror för bestämning av en 3D-position.

Både aktiva och passiva markörer kan användas för att spåra rörelser. Passiva markörer utgörs av små sfärer täckta med reflekterande material, vilka reflekterar infrarött ljus från kamerorna. Aktiva markörer sänder istället ut ljus enligt specifika pulssekvenser, vilket möjliggör identifiering av enskilda markörer.

3.2 Robot Operating System 2

ROS2[3] (Robot Operating System 2) är ett ramverk bestående av verktyg och bibliotek för utveckling av robotsystem. Det används för att strukturera system där flera komponenter behöver kommunicera med varandra. I ROS2 delas systemet upp i noder som utbyter data via publishers och subscribers över topics. Denna kommunikationsmodell gör det möjligt att dela upp funktionalitet i separata delar som kan köras parallellt. En fördel med detta är att systemet blir mer modulärt och enklare att utveckla, testa och vidareutveckla.

3.3 Nätverkskommunikation

Pythons standardbibliotek `socket`[4] kan användas för nätverkskommunikation, exempelvis över wifi. Data kan överföras med olika protokoll, däribland TCP (Transmission Control Protocol) och UDP (User Datagram Protocol), vilka har olika egenskaper och lämpar sig för olika typer av tillämpningar [7]. Med protokollet TCP krävs att en anslutning mellan en sändare och en mottagare etableras innan data kan skickas. Dessutom säkerställs att all data levereras i rätt ordning, vilket kan medföra högre latens. UDP är däremot anslutningslöst och data skickas utan att någon förbindelse först etableras. Varken leverans eller ordning av datan kontrolleras, vilket gör det snabbare men mindre tillförlitligt än TCP. Med UDP är det möjligt att skicka data till en specifik port utan att först etablera en anslutning, exempelvis via broadcast, vilket gör att flera enheter i nätverket kan ta emot samma information.

3.4 Implementation

Realtidsbaserad positionsdata för AMR:erna erhölls med ett Qualisys MoCap-system bestående av åtta stycken Arqus A12[5], samt programvaran QTM[6] (Qualisys Track Manager) som körs på en separat dator. Den använda kamerainstallationen visas i Figur 3.2.
Då samtidig användning av aktiva och passiva markörer inte stöds, behövde en gemensam markörtyp användas för hela systemet. Eftersom systemet dessutom skulle samverka med en parallellt utvecklad drönarplattform som kräver aktiva markörer, baserades spårningen på de aktiva markörerna.

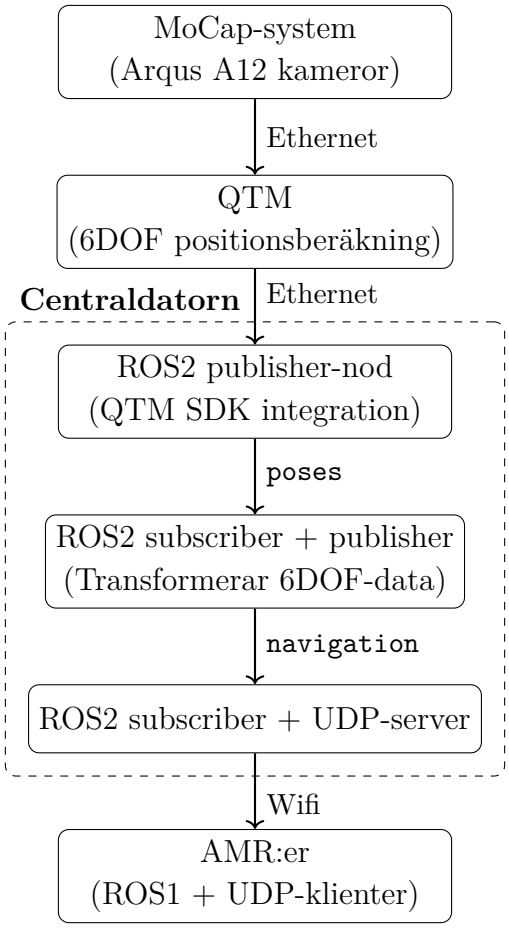
[3] https://docs.ros.org/, hämtad 2026-05-12.
[4] https://docs.python.org/3/library/socket.html, hämtad 2026-05-12.
[5] https://www.qualisys.com/cameras/arqus/, hämtad 2026-05-12.
[6] https://www.qualisys.com/software/qualisys-track-manager/, QTM version 2025.2, hämtad 2026-05-12.

Figur 3.2: Översikt av det använda kamerasystemet.

Åtkomst till realtidsdata med sex frihetsgrader (6DOF) från QTM till centraldatorn möjliggjordes via ett Python-SDK[7] (Software Development Kit). SDK:t implementerades i en ROS2-publishernod för att distribuera positionsdata via ROS2-topicet `poses`, se Figur 3.3. QTM-data bestod av en tredimensionell position samt en rotationsmatris, vilken transformerades till en tvådimensionell position och en orientering kring z-axeln. Den transformerade datan publicerades på topicet `navigation`.

För dataöverföring till AMR:erna användes UDP-baserad socketkommunikation över wifi. På centraldatorn implementerades en UDP-server i en ROS2-subscribernod, vilken broadcastade positionsdata från `navigation`-topicet till AMR:erna, där motsvarande UDP-klienter kördes i publishernoder, se Figur 3.3. Valet att använda socketkommunikation framför ROS-kommunikation grundades i att centraldatorn och AMR:erna använde ROS2 respektive ROS1. Kommunikation mellan ROS1 och ROS2 skulle kräva en mer avancerad implementation utan motsvarande funktionella fördelar. UDP valdes framför TCP eftersom det möjliggör broadcast av data till flera AMR:er samt minskar risken för avbrott i anslutningen, då TCP kräver en etablerad anslutning mellan sändare och mottagare. Nackdelen med potentiell paketförlust bedömdes som acceptabel, då data skickas med hög frekvens.

[7] https://www.qualisys.com/my/qacademy/#!/tutorials/how-to-use-the-python-sdk-for-qtm, hämtad 2026-05-12.



Figur 3.3: Systemarkitektur för positionsinhämtning och distribution till AMR:er. Kommunikation in och ut från centraldatorn görs över nätverket. Inne i centraldatorn sker kommunikationen via ROS2-topics med namnen `poses` och `navigation`.

3.5 Utvärdering

Systemet möjliggjorde överföring av 6DOF-positionsdata i realtid från MoCap-systemet till AMR:erna via ROS2 och UDP. Samtliga komponenter integrerades och fungerade tillsammans enligt avsedd funktion. MoCap-systemet visade på god funktionalitet för flera AMR:er och tillhandahöll data i form av en tredimensionell position samt en rotationsmatris för varje AMR i ett globalt koordinatsystem, vilket gjorde det möjligt att distribuera nödvändig information till övriga funktioner.

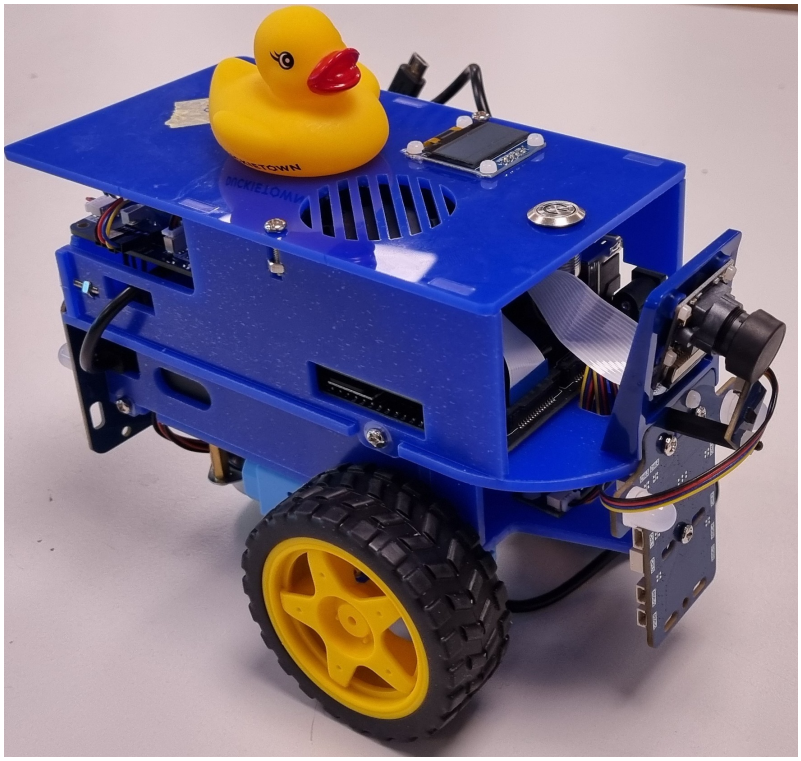
Inga kvantitativa mätningar av systemets latens genomfördes, men vid tester kunde ingen märkbar fördröjning observeras varken från MoCap-systemet eller till AMR:erna. Inte heller MoCap-systemets noggrannhet testades, men uppvisade tillräckligt goda resultat för ändamålet. Tester visade att en sändningsfrekvens från centraldatorn som översteg AMR:ernas läsningsfrekvens ledde till instabilitet i datakommunikationen, då de överbelastades med data. För att motverka detta begränsades sändningsfrekvensen till en nivå under läsningsfrekvensen.

4 Fysisk testplattform

För att kunna utvärdera funktionalitet på positioneringssytemet och tillämpning av ruttplaneringsalgoritmer behövs en fysisk plattform att testa på. AMR:er som användes i projektet är av typen Duckiebot DB21J (se Figur 4.1), en differentialstyrd robotplattform från Duckietown[8]. På vardera AMR används följande hårdvara:

- NVIDIA jetson Nano, som beräkningsenhet
- Time of Flight (ToF)- sensor
- Inertia measuring unit (IMU)
- Framåtriktad kamera RGB
- Rotary encoders på hjulen

Mjukvara på AMR:erna är Duckietowns egna Duckietown shell som inkluderar ROS1, docker och vissa förbyggda funktioner. Extern kommunikation med centraldatorn sker via UDP. Vid förlust av kontinuerlig data från centraldatorn används rotary encoders på hjulen för odometri. Nyckelfunktionalitet som inte finns färdig är reglering för banföljning och metod för kollisionsundvikning.



Figur 4.1: Duckiebot, robotplattformen som används för fysisk testning.

[8]https://docs.duckietown.com/ente/duckietown-manual/welcome-to-the-duckietown-manual.html, hämtad:2026-04-28

4.1 Styrning av AMR

För att kunna följa rutten som skickas från centraldatorn i form av noder att träffa, behövs en trajektorie tas ut och motorerna måste regleras för att kunna följa bestämd trajektorie. För att kunna följa trajektorien väl behöver regleringen fungera, för att göra en fungerande reglering måste systemet bestämmas.

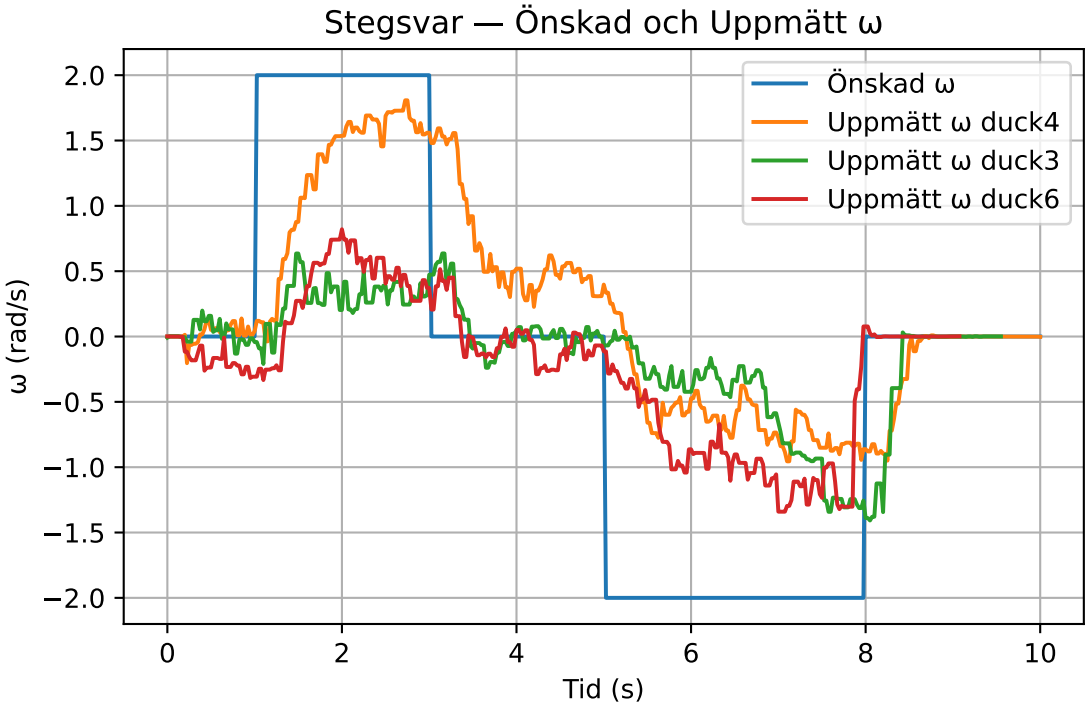
Varje AMR har en egen intern karaktäristik och behöver därför regleras var för sig. Metoden som använts är Skogestad Internal Model Control(SIMC). SIMC som beskrivet av S. Skogestad och C. Grimholt [8] lämpar sig för ett system med liten dödtid relativt tidskonstanten och för att smidigt bestämma parametrar på flera olika system. De parametrar som behöver identifieras är systemets förstärkning K, tidskonstanten τ och dödtiden θ i en första ordningens överföringsfunktion (4.1). De ekvationer som används för att ta fram PI parametrar är (4.2) för P-delen och (4.3) för I-delen. τ_c är en designparameter som rekommenderas att sättas $\tau_c = \theta$ för responsiv reglering.

$$G(s) = \frac{k}{(\tau s + 1)} e^{-\theta_s} \tag{4.1}$$

$$K_c = \frac{1}{K} \frac{\tau_I}{\tau_c + \theta} \tag{4.2}$$

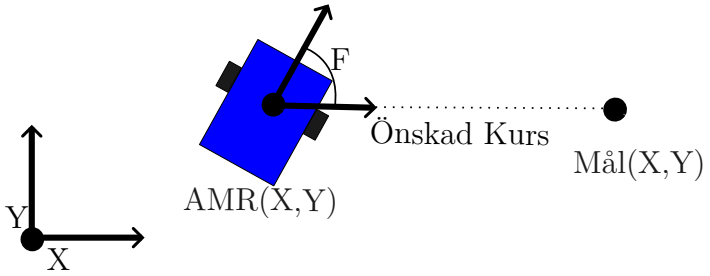
$$\tau_I = \min\{\tau, 4(\tau_c + \theta)\} \tag{4.3}$$

Systemidentifiering behövs för att approximera parametrarna K, τ och θ. För att få fram stegsvar för identifiering skickas kommandon till AMR:en att köra i en S-kruva, insignalen och rotationshastigheten som kommer från en kombination av IMU och encoders redovisas i Figur 4.2. Datan matas genom Matlabs system identification toolbox [9] som approximerar värden till ett första ordningens överföringsfunktion. Med denna funktion går det att bestämma PI reglering för varje individuell AMR.



Figur 4.2: Stegsvar för tre olika Duckiebots vid ett testtillfälle.

Informationen AMR:erna får till sig är sin nuvarande position och rotation i förhållande till ett globalt koordinatsystem samt en målnod. Med information om nuvarande position, rotation och målposition kan rotationen F som ska korrigeras beräknas för att ha en kurs mot målnoden enligt Figur 4.3. Själva regleringen av svängning sker med tidigare nämnd PI regulator.



Figur 4.3: Vinkel F är rotationen som krävs av AMR för att ha kurs mot målnod.

4.2 Kollisionsundvikning

Kollisionsundvikningen baseras på en kombination av ToF-sensor och kamerabaserad bildbehandling. ToF-sensor används som primär trigger för att avgöra när ett objekt befinner sig tillräckligt nära AMR:en. Vid detektion inom cirka 40 cm aktiveras bildbehandlingen, vilket används för att uppskatta hindrets position och identifiera en lämplig riktning för hinderundvikningen.

Den kamerabaserade analysen baseras på klassisk bildbehandling med Bird's Eye View (BEV)-transformation. BEV-transformationen innebär att kamerabilden projiceras till ett ovanifrånperspektiv, vilket underlättar analysen av hinderposition framför AMR:en. Metoden baseras på den extrinsiska kamerakalibrering som genomförs inom projektet. I Duckietown [9] beskrives den extrinsiska kalibreringen som hur kameran är monterad relativt AMR:en och hur den används för att ta fram en homografi för projektion från bildplanet till markplanet. I implementationen används den framtagna homografimatrisen tillsammans med OpenCV-funktionen `warpPerspective` för att skapa en BEV-representation av kamerabilden [10].

Den transformerade bilden bearbetas därefter med klassiska bildbehandlingsmetoder. Bilden konverteras till gråskala, filtreras med ett Gaussiskt filter och bearbetas med adaptiv tröskling samt Canny-baserad kantdetektion [11], [12], [13]. Dessa steg används för att identifiera konturer i bilden, där en kontur motsvarar ytterformen hos ett objekt eller hinder [14]. (Figur 4.4) visar ett exempel på Canny-baserad kantdetektion.

Figur 4.4: Till vänster visas originalbilden och till höger visas resultatet efter Canny-kantdetektion.

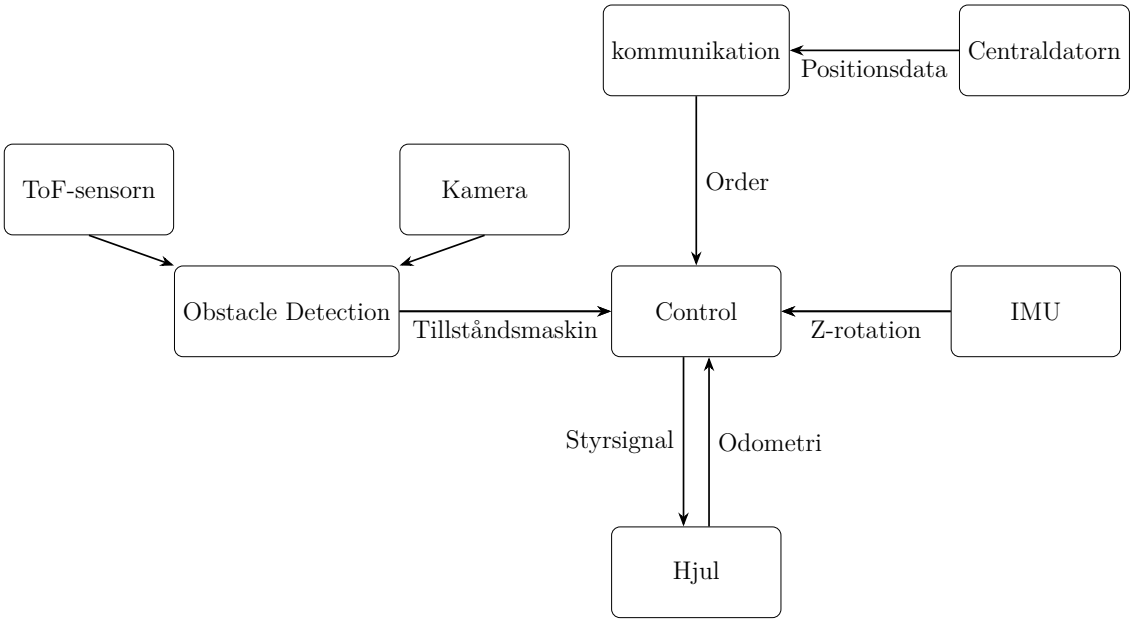
[9] Duckietown, *Camera Calibration*, `https://docs.duckietown.com/ente/duckietown-manual/20-operations/04-calibrations/duckiebot-camera-calibration.html#db-camera-calibration`, hämtad 2026-04-28.

Konturanalysen används för att identifiera den mest relevanta hinderkandidaten i området framför AMR:en och filtrera bort mindre konturer som brus. Undvikningsriktningen bestäms genom att jämföra fritt utrymme på vänster och höger sida om hindret. Om ingen sida bedöms ha tillräckligt utrymme initieras AMR:en backning.

Kollisionsundvikningen styrs av en tillståndsmaskin där ToF-detektion aktiverar scanning med BEV-analys. BEV-metoden används för att uppskatta en undvikningsriktning, varefter AMR:en ska rotera, passera hindret och återgå till sin ursprungliga riktning.

4.3 Utvärdering av körförmåga

Integrationen mellan hinderdetektionen och körningslogiken genomfördes med hjälp av ROS1. Där informationsutbyte mellan systemets noder möjliggjorde samordning mellan sensorbaserad hinderidentifiering och styrning av AMR:erna. Det övergripande informationsflödet illustreras i Figur 4.5.



Figur 4.5: Figuren visar systemets informationsflöde, där data från IMU, ToF-sensorn, kameran och order används av Control för att styra hjulen, samtidigt som odometri återkopplas till systemet.

Testet av hinderdetektionen genomfördes i en vanlig inomhusmiljö utan körfältsmarkeringar (se Figur 4.4). Miljön innehöll varierande bakgrunder, exempelvis fönster, väggar och andra objekt, vilket kunde påverka kamerabilden. Syftet med testet var att undersöka om roboten kunde detektera ett hinder framför sig, genomföra en undvikande rörelse och därefter passera hindret.

BEV-metoden visade delvis fungerande beteende i de praktiska testerna. ToF-sensorn detekterade hinder framför AMR:en, vilket gjorde att AMR:en stannade innan kollision. I ett fall lyckades AMR:en genomföra hela hinderundvikninggen genom att först rotera och därefter passera hindret. Detta visar att grundlogiken i systemet var möjlig att genomföra.

Resultatet var däremot inte tillräckligt stabilt för att metoden skulle kunna användas i den slutliga implementationen. I resterande fall roterade AMR:en på plats utan att därefter fortsätta framåt, vilket innebär att hinderundvikningen inte slutfördes. Eftersom samma AMR och samma kod användes vid samtliga tester kunde orsaken till det inkonsekventa beteendet inte fastställas. Sammantaget fungerade ToF-sensorn för att detektera hinder och stoppa AMR:en innan kollision, men BEV-metoden kunde inte verifieras som en fungerande lösning för kollisionsundvikning.

AMR:ernas styrning visade sig vara oregelbunden, vilket kan tillskrivas flera faktorer. Stegsvaret (se Figur 4.2) från systemidentifikationen uppvisar hög brusnivå och skiljer sig markant mellan olika enheter, vilket tyder på opålitliga sensorer eller mekaniska komponenter. Under tester mottog AMR:erna kommandon och beräknade riktning korrekt på mjukvarunivå. Det faktiska körbeteendet varierade avsevärt mellan körningar med identisk mjukvara, från korrekt målpunktsstyrning till ingen rörelse alls och okontrollerad cirkeldrift. En trolig bidragande orsak är att motorerna är underdimensionerade. Vid låg styrsignal saknar de tillräckligt vridmoment för att övervinna friktionen mot underlaget, medan en högre styrsignal resulterar i alltför snabb svängning. Med mer tillförlitlig hårdvara och sensordata skulle en mer systematisk utvärdering av körbeteendet vara möjlig, vilket inte var genomförbart med befintlig utrustning.

5 Ruttplanering

Det finns många olika sökalgoritmer [15], dessa löser planering på skilda sätt och förutsätter att viss information finns tillgänglig. Informationen som förutsätts i detta projekt är att lagermiljön kan modelleras som en 2D graf och att målets position i grafen är känd. En graf är en naturlig, flexibel modell för att applicera komplexa relationer och interaktioner [16].

5.1 Sökalgoritmer

För att en agent ska kunna ta sig från start till mål måste den ha en planerad rutt att följa, där Dijkstras och A* är etablerade metoder som undersöks närmare i detta projekt.

5.1.1 Dijkstras algoritm

Dijkstras algoritm hittar den kortaste vägen mellan noder i en viktad graf med icke-negativa kantvikter, från en given startpunkt till alla andra noder. Algoritmen bygger på en girig strategi där den successivt väljer den nod som för tillfället har det lägsta kända avståndet från startpunkten.

Algoritmen inleds med att avståndet till startnoden sätts till noll, medan resterande noder förblir outforskade. Därefter väljs den obesökta nod med lägst avstånd, när en nod är markerad som besökt är kostnaden till noden garanterat minimal från startnoden. Processen fortsätter tills alla noder har behandlats och kortaste vägen till alla noder är funnen.

5.1.2 A*

A* är en informerad vägsökningsalgoritm som hittar den kortaste vägen från start till mål givet en viktad graf om det finns en väg till målet [15]. Den hittade vägen är kostnadsminimerad och algoritmen undersöker en minimal mängd noder givet att den har perfekt heuristik. A* bestämmer vägen genom att expandera en trädstruktur en kant i taget från start-noden tills den är framme vid målet. När A* initieras expanderar den noder som minimerar den uppskattade kostnaden f(n) i ekvation (5.1).

$$f(n) = g(n) + h(n) \tag{5.1}$$

Där n är nästa nod på vägen, g(n) är kostnaden från start till nod n, och h(n) är en heuristik funktion som uppskattar vägen kvar mellan noden n och målet. Enligt [17] är A* en fundamental algoritm för att lösa Multi Agent Path Finding (MAPF) problemet.

5.2 Heuristik för ruttplanering

En heuristik är en funktion som uppskattar den återstående kostnaden från en given nod till målnoden. Inom informerad sökning, såsom A*, används heuristiken för att prioritera vilka noder som bör utforskas först. För att A* ska vara optimal krävs det att heuristiken är underskattande och konsekvent. I detta avsnitt presenteras tre vanliga heuristiska kostnadsfunktioner för ruttplanering: Manhattan-, diagonal- och euklidisk heuristisk.

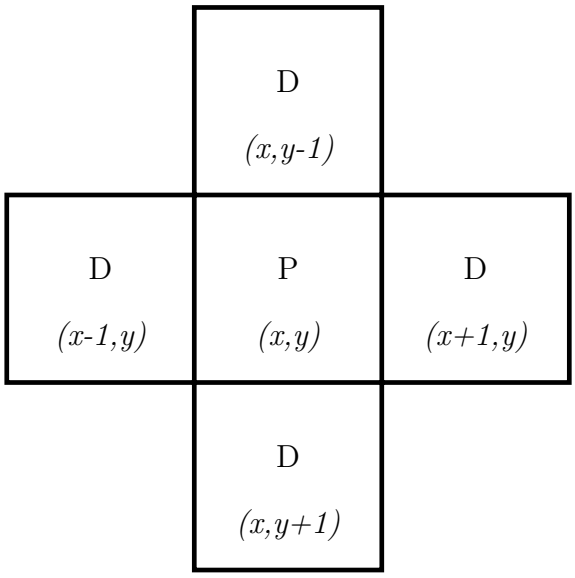
5.2.1 Manhattan

Manhattan-kostnadsfunktionen är väl lämpat för ett Von Neumann-grannskap eftersom avståndet beräknas utifrån summan av horisontella och vertikala steg. Manhattan kostnadsfunktionen beräknas enligt ekvation (5.2).

$$h_M(n) = |n.x - goal.x| + |n.y - goal.y| \tag{5.2}$$

Von Neumann-grannskapet av radie 1 kring punkten (i, j) består av punkten själv och dess fyra närmaste grannar i horisontell och vertikal riktning. Om någon av dessa punkter ligger utanför domänen D (koordinatdomänen av V), tas de bort från mängden, se ekvation (5.3).

$$N_1(i, j) = \{(k, l) \in D \mid |k - i| + |l - j| \leq 1\}. \tag{5.3}$$



Figur 5.1: En illustration av ett Von Neumann grannskap. En nod på position P kan endast ha kanter till de första noderna D i positiv och negativ vertikal och horisontell riktning [18], CC0.

5.2.2 Diagonal

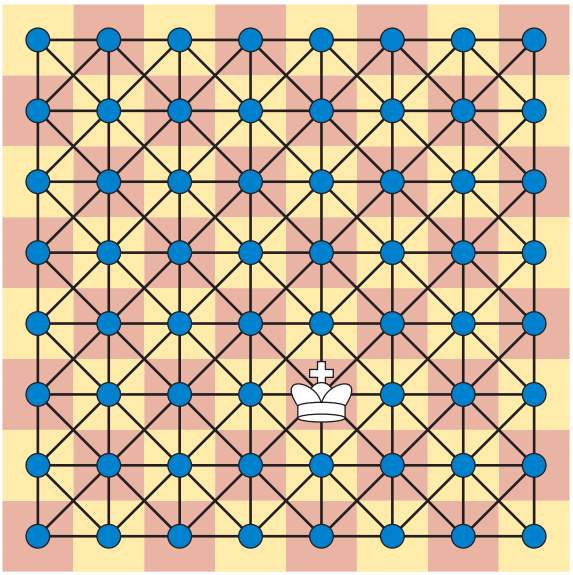
Diagonal-kostnadsfunktionen ger en optimal sökning på en kungsgraf och beskrivs i Figur 5.2. Det är en optimal sökning eftersom avståndet beräknas på horisontella, vertikala och diagonala steg. Diagonala kostnadsfunktionen beräknas enligt ekvation (5.4), där variablerna är definierade i ekvation (5.5).

$$h_D(n) = c_d \cdot d_{min} + c_n \cdot (d_{max} - d_{min}) \tag{5.4}$$

$$\begin{aligned} d_{max} &= max(|n.x - goal.x|, |n.y - goal.y|) \\ d_{min} &= min(|n.x - goal.x|, |n.y - goal.y|) \\ c_n &= \text{cost of non-diagonal movement} \\ c_d &= c_n \cdot \sqrt{2} \end{aligned} \tag{5.5}$$

En kungsgraf av radie 1 kring punkten (i, j) består av punkten själv och dess åtta närmaste grannar i horisontell, vertikal och diagonal riktning. Om någon av dessa punkter ligger utanför domänen D (koordinatdomänen av V), tas de bort från mängden, se ekvation (5.6).

$$N_1(i, j) = \{(k, l) \in D \mid \max(|k - i|, |l - j|) \leq 1\}. \tag{5.6}$$



Figur 5.2: En illustration av hur en kungsgraf ser ut. En agent kan röra sig precis som en kung i schack [19], CC0 1.0.

5.2.3 Euklidisk

En euklidisk-kostnadsfunktion mäter fågelvägen mellan en nod och målnoden, detta är garanterat inte en överskattning då fågelvägen är det kortaste avståndet mellan

två geografiska punkter men blir ofta en underskattning av resvägen. Euklidiska kostnadsfunktionen beräknas enligt ekvation (5.7).

$$h_E(n) = \sqrt{(n.x - goal.x)^2 + (n.y - goal.y)^2} \tag{5.7}$$

5.3 Definition av miljö och modell

Varför en graf används
För att representera lagermiljön används en graf, där noder representerar traverserbara positioner och kanter representerar möjliga förflyttningar mellan positioner. Grafrepresentationer är ett etablerat sätt att genomföra ruttplanering i diskreta miljöer [20], [21], eftersom navigeringsproblemet kan formuleras som en sökning mellan noder med väldefinierade kanter. Detta är särskilt lämpligt i lagermiljöer där agenter rör sig längs fördefinierade gångar och inte fritt i kontinuerligt rum.

Formell definition av grafen
Grafen defineras formellt som ekvation (5.8).

$$G = (V, E, w) \tag{5.8}$$

Där V är mängden noder som representerar diskreta positioner i miljön, E är mängden riktade kanter som representerar tillåtna förflyttningar mellan noder och w är viktfunktionen som tilldelar varje kant en kostnad.

Koppling till ruttplanering
Grafmodellen är direkt kopplad till ruttplanering eftersom planeringsalgoritmerna använder grafen för att planera en rutt. Riktningen på kanterna begränsar vilka rutter som är möjliga och därmed hur sökrymden ser ut. En riktad graf kan exempelvis modellera enkelriktade gångar eller flödesriktningar i ett lager, vilket i sin tur påverkar vilka rutter som är tillåtna.

5.3.1 Modellering av lagermiljön

Uppbyggnad av graf
Grafen består av totalt 45 noder i ett tvådimensionellt koordinatsystem enligt ekvation 5.9.

$$(x, y), \quad x \in \{-2, -1.5, -1, ..., 2\}, y \in \{-1, -0.5, 0, 0.5, 1\} \tag{5.9}$$

Upplösningen 0.5 meter används på grund av fysiska begränsningar i den verkliga testmiljön som grafen appliceras på. Noderna representerar diskreta positioner en agent kan befinna sig på, exempelvis i korsningar, punkter framför hyllor eller positioner i grafen för att möjliggöra fler samtidiga agenter i systemet. Dessutom

representeras hinder i miljön som noder utan kanter, vilket gör de otillgängliga för alla agenter.

Kantstruktur

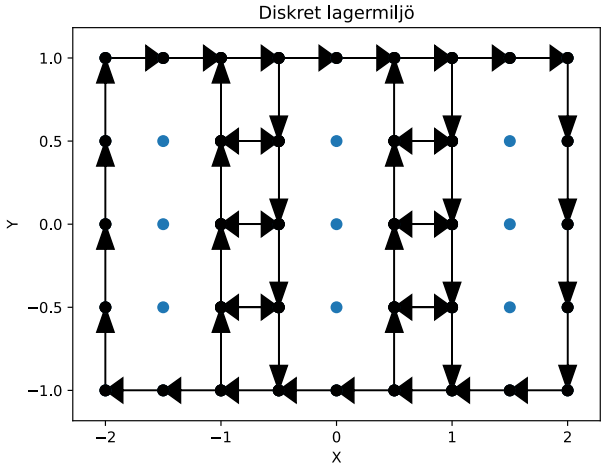
Kantstrukturen i grafen består huvudsakligen av enkelriktade kanter vilket minskar mängden konflikter som kan uppstå mellan agenter. I gångarna mellan hyllraderna används däremot dubbelriktade kanter för att möjliggöra riktningsbyte i en gång mellan hyllor. Kombinationen av enkelriktade- och dubbelriktade kanter begränsar antalet möjliga konflikter samtidigt som agenter kan färdas flexibelt genom grafen.

Begränsningar och antaganden

För att möjliggöra planering av flera agenter införs även kapacitetsbegränsningar i grafen. Det innebär att varje nod, enkelriktade och dubbelriktade kant endast tillåter en agent åt gången, vilket förhindrar att två agenter befinner sig på samma kant samtidigt.

Grafens struktur

Figur 5.3 visar den graf som används i implementationen. Den illustrerar nodernas placering, kanternas riktning och de hinder som ingår i miljön. Grafens noder är placerade i ett regelbundet rutnät eftersom det är en enkel modell att generera en graf på. Figuren fungerar som en visuell referens för att verifiera att grafen som genereras stämmer överens med den modell som beskrivits i detta avsnitt.



Figur 5.3: En illustration av grafen som används i projektet. Grafen används för att göra en ruttplanering på en diskret miljö som representerar verkligheten. Grafen innehåller enkelriktade och dubbelriktade kanter för att bättre representera en verklig lagermiljö.

5.4 Implementation

Detta avsnitt hanterar implementation av ruttplanering i projektets grafmiljö baserat på teorier kring sökalgoritmer, heuristik och grafstrukturer som beskrivits i tidigare avsnitt. Implementationen ska bidra till en robust ruttplanering som kan användas i flera typer av lagerlayouter.

Avsnitt 5.4.1 beskriver hur Dijkstras och A* har implementerats, med fokus på datastrukturer, algoritmflöde och hantering av ogiltiga eller blockerade noder. Avsnitt 5.4.2 beskriver valet av heuristik och den bakomliggande motiveringen, baserat på de tekniska egenskaperna hos heuristiker som analyserades i avsnitt 5.2. Tillsammans redogör avsnitten för implementationen av ruttplaneringssystemet, där både algoritmiskt och heuristiskt designval motiveras utifrån projektets krav och teoretiska förutsättningar.

5.4.1 Implementation av sökalgoritm

I detta avsnitt beskrivs implementation av Dijkstras algoritm och A* i projektets graf-modell definierad i avsnitt 5.3. Ruttplaneringsalgoritmen som valdes i projektet är baserad på de teoretiska egenskaperna som presenterades i avsnitt 5.1. Syftet med implementationen är att ruttplaneringsalgoritmen ska kunna appliceras på många olika typer av grafer samt effektivt bestämma en väg.

Datastrukturer
Grafen representeras av en adjacency-lista (`adj`) där varje nod är kopplad till sina grannar tillsammans med respektive kantkostnad. Lagring av ackumulerad kostnad från start till noden lagras i en dictionary (`g_score`), där varje nod initieras med oändlig kostnad utom startnoden som sätts till noll. Den uppskattade totala kostnaden sparas i en separat dictionary (`f_score`), där värdet beräknas enligt ekvation (5.1). För att rekonstruera den slutgiltiga vägen används en parent-dictionary (`came_from`), som lagrar ett uppslag från uppslagen nod till noden som har den minsta kostnaden `f_score`. Noder som ska hanteras sparas i ett set (`open_set`), där noden med lägst f-värde väljs varje iteration. Även om en mängd inte är lika effektivt som en prioritetskö är den tillräckligt effektiv för projektets syfte och ger en tydlig och enkel implementation. Hinder undviks med hjälp av en lista med blockerade noder (`blocked`), som gör det möjligt att blockera noder som inte får beträdas.

Dijkstra
Dijkstras algoritm används som baslinjeimplementation och bygger på att inkrementellt uppdatera utforskade noder baserat på kortast kända ackumulerade kostnad. Varje iteration väljs den nod i `open_set` som har den lägsta ackumulerade kostnaden, nodens föregångare sparas i `came_from`. När målnoden är hittad rekonstrueras

vägen med hjälp av att följa `came_from` bakåt. Pseudokod för algoritmen finns i Bilaga A.

A*
A* är implementerat på samma grundstruktur som Dijkstra, men använder istället ett f-värde enligt ekvation (5.1), där heuristiken uppskattar återstående kostnad till målet. Varje iteration väljs den nod med lägst f-värde i `open_set`, vilket resulterar i att A* nästan alltid utforskar färre noder än Dijkstra. När en granne utvärderas beräknas ett tentativt f-värde. Om värdet är lägre än alla andra noder i `open_set` tas det bort och uppdaterar `came_from`, och grannarna läggs till i `open_set`, när målnoden nås rekonstrueras vägen på samma sätt som för Dijkstras. Pseudokod för algoritmen finns i Bilaga B.

Jämförelse
Skillnaden mellan A* och Dijkstras algoritm är att A* är en informerad sökalgoritm vilket innebär att den har en uppfattning om vart målet är. A* utforskar däremot färre noder än vad Dijkstras gör och därmed är A* mer effektivt vilket gör den mest lämplig att använda i projektet.

Felhantering och robusthet
Om algoritmen får en ogiltig start- eller målnod kommer algoritmen att skicka en tom lista och ett nytt uppdrag genereras. Om ingen väg hittas efter 60 försök tilldelas agenten ingen rutt och förblir stillastående. Detta säkerställer att systemet inte fastnar i oändliga sökningar vid blockerade eller otillgängliga start och mål.

5.4.2 Heuristikdesign och teknisk motivering

Heuristiken som valdes i projektet är baserat på de teoretiska egenskaperna som presenterades i avsnitt 5.2 . Grafen som används i projektet är ett Von Neumann grannskap och har därför endast vertikala och horisontella kanter i grafen. Det visades i avsnitt 5.2 att Manhattan-heuristiken är den mest informativa och effektiva heuristiken i projektets typ av graf.

Däremot är projektets mål att utveckla en metod som kan generaliseras till flera typer av modeller som även tillåter icke ortogonala rörelser. I verkliga lagerlayouter förekommer ofta diagonala gångar, oregelbundna avstånd och geometrier som inte följer ett strikt rutnät. För sådana miljöerna är Manhattan-heuristiken ofta varken underskattande eller konsekvent, medan euklidisk-heuristik både förblir underskattande och konsekvent på alla grafer där kostnaden för ett steg som minst motsvarar avståndet mellan noderna.

Relationen mellan de heuristiska funktionerna i avsnitt 5.2 formuleras enligt ekvation

(5.10).

$$h_E(n) \leq h_D(n) \leq h_M(n) \tag{5.10}$$

Ekvationen innebär att den euklidiska heuristiken ofta kommer uppskatta avståndet till målet som mycket mindre än både diagonal- och Manhattan-heuristiken. Konsekvensen blir att A* expanderar fler noder i ett rent Von Neumann grannskap när uppskattningen blir sämre. Däremot euklidiskt avstånd kommer uppnå kraven för heuristik om diagonal- eller Manhattan-heuristik gör det.

Eftersom projektet prioriterar generalitet över maximal prestanda säkerställer en euklidisk-heuristik att metoden även fungerar i miljöer där diagonala eller oregelbundna mönster kan förekomma. Det innebär att sökkostnaden ökar, men det betraktas som acceptabelt relativt den erbjudna robustheten heuristiken erbjuder.

Projektet valde därmed euklidisk-heuristik som estimerande kostnadsfunktion då den uppfyller kraven för underskattning och konsekvens, samtidigt som den möjliggör generalisering till flest antal typer av grafstrukturer, där projektet applicerade ett riktat Von Neumann grannskap. Valet är därmed motiverat både utifrån teoretiska egenskaper och projektetes övergripande mål.

5.5 Resultat

Projektet använder en mängd (`open_set`) för att välja nästa nod att expandera, vilket innebär att den värsta komplexiteten för både Dijkstra och A* är $O(V^2 + E)$. Detta beror på att lägsta f- och g-kostnaden hittas med en linjär sökning i mängden för varje iteration.

Däremot expanderar A* betydligt färre noder än Dijkstra i praktiken då heuristiken styr sökningen mot målet. Med en underskattande och konsekvent heuristik hittar A* vägen med lägst kostnad. I bästa fall med perfekt heuristik reduceras tidskomplexiteten till $O(d)$, där d är kantantalet på den optimala vägen.

Enligt tabell 5.1 expanderade A* med euklidisk heuristik i snitt 14 noder medans A* utan heuristik utforskade 19 noder, vilket visar att heuristiken styr A* mot målet. Eftersom båda använder samma datastrukturer blir skillnaden i exekveringstid och expanderade noder liten i små grafer, men växer snabbt i större grafer där A* med heuristik skalar bättre. Eftersom A* med heuristik gör mindre arbete per sökning är den bättre lämpad för realtidsplanering i en autonom flotta, där många samtidiga sökningar måste utföras. Resultaten följer förväntningen att A* med heuristik utforskar färre noder än utan heuristik som även efterliknar beteendet av Dijkstras algoritm.

Resultaten visar att A* är något snabbare med heuristik än utan, den är även

snabbare än Dijkstra eftersom Dijkstra behöver utföra minst lika mycket beräkning som A* utan heuristik, vilket även kan beskrivas som ekvation (5.11). Detta gör att A* är mer lämpad för realtidsplanering i en autonom flotta, där snabb omplanering och låg beräkningskostnad är avgörande för att hantera dynamiska hinder.

$$t_{A*}(\text{euklidisk}) \leq t_{A*}(\text{utan heuristik}) \leq t_{Dijkstra} \tag{5.11}$$

Tabell 5.1: Resultat efter en miljon sökningar på grafen i Figur 5.3.

	Medeltid per sökning (ms)	Medel expanderade noder (st)
A* (utan heuristik)	0.104	19.0
A* (euklidisk)	0.084	14.0

6 Kollisionshantering

Kollisionshantering är en central utmaning som uppstår när flera agenter agerar i en gemensam miljö. Det är viktigt att hantera kollisioner för att undvika skador på agenter, omgivningen och potentiella människor som rör sig i den. Utmaningar med kollisionshantering är att en optimal planering av kollisionsfria banor med Multi Agent Path Finding (MAPF) är ett NP-hard problem. [22], [23]

För att möjliggöra robust kollisionshantering behövs en teoretisk grund i vilka MAPF-algoritm som finns, förutbestämda kriterier och omständigheter kollisionshantering sker inom och en uppfattning av vilka kollisioner som kan uppstå vid hantering av flera agenter.

6.1 Planeringsparadigm: val av modell

En livslång, dynamisk, tidsdiskret och suboptimal planeringsmodell används för att hantera kontinuerliga miljöer. Modellen motiveras av att agenterna ska kunna fortsätta planera över tid och anpassa sig till andra planerade scheman samt oförutsägbara situationer. Att modellen väljs suboptimal bidrar till att minska beräkningstiden, medan den diskreta representationen gör att modellen behöver beräkna mindre men är fortfarande applicerbar i en kontinuerlig verklighet.

6.1.1 Multi Agent Path Finding – definition och antaganden

Multi Agent Path Finding (MAPF) är ett problem inom artificiell intelligens och robotik som handlar om att planera kollisionsfria rutter för flera agenter i en gemensam miljö [24]. Grundläggande egenskaper hos MAPF-algoritmer är att den garanterar kollisionsfri planering [24], [25], optimerar rutter med avseende på väglängd, tid eller andra kostnader och att flera agenter delar en gemensam miljö [26]. Utmaningar med MAPF är att när antalet agenter ökar blir problemet snabbt mer komplext och svårare att lösa [27], och att agenter måste samarbeta för att undvika trängsel [28], och dödlägen [29]. Lösningar på MAPF problemet kan bli kategoriserade inom 5 olika typer: optimala-, suboptimala-, anytime-, lärnings-baserade- och specialiserade-lösningar som beskrivs tydligare i Tabell 6.1.

Tabell 6.1: Kategorisering av lösningar på MAPF-problemet.

Optimal lösning	Garanterar den bästa lösningen för en specifik kostnadsfunktion.
Suboptimal lösning	Beräkningsmässigt enklare men kan inte garantera att den bästa lösningen hittas.
Anytime lösning	Hittar snabbt en giltig lösning och förbättrar den iterativt inom en given tidsram, vilket skapar en balans mellan kvalitet och körtid.
Lärningsbaserad lösning	Använder ML eller RL för att träna agenterna och möjliggöra decentraliserade eller adaptiva strategier.
Specialiserad lösning	MAPF anpassas till ett specifikt område, exempelvis dynamisk eller livslång MAPF.

6.1.2 Lifelong och One-Shot

One-Shot MAPF innebär att planera en kollisionsfri rutt för alla agenter från en start till ett mål vanligtvis i en statisk miljö där alla agenters mål är kända i förväg, ofta kallad det klassiska MAPF problemet [27]. Planeringen är vanligtvis centraliserad och optimerar vanligtvis antingen total kostnader (Sum of Cost, SoC) eller tid att slutföra alla agenters uppdrag (makespan). Algoritmen strävar efter att beräkna optimala eller nära optimala vägar och eftersom beräkningen görs en gång kan beräkningstiden vara längre än för ett system som kräver kontinuerlig uppdatering.

Lifelong MAPF är utformad för miljöer där agenter kontinuerligt får nya uppdrag när de är klara med sitt förra uppdrag [27]. Planeringen är vanligtvis decentraliserad eller en hybrid-planering för att öka genomströmning och responsförmåga. Optimeringen fokuserar ofta på att maximera systemets genomströmning och minimera latenstid snarare än att optimera en enskild global kostnad. Algoritmen kräver inkrementell planering för att dynamiskt hantera nya uppdrag och se till att lösningen är rimlig i realtid.

6.1.3 Dynamisk Multi Agent Path Finding

Dynamisk MAPF (DMAPF) är bättre i verkligheten än traditionell MAPF eftersom den klarar att hantera dynamiska förändringar av agenter som läggs till eller tas bort, rörliga hinder och förändrade omgivningar genom att kontinuerligt övervaka, omplanera och reparera vägar [30]. Detta gör att lösningarna blir mer anpassningsbara, robusta och skalbara i verkliga scenarier, där verklighetens osäkerheter och operativa krav, som kinematiska begränsningar och tidskritiska uppdateringar, explicit tas i beaktande.

Förändring i omgivningen innebär statiska eller dynamiska förändringar av hinder, där forskning lägger större tyngd på dynamiska hinder då dessa uppstår under operation av agenter och har en större påverkan på vägplanering än dynamiska förändringar i statiska hinder [31], [32]. Ett sätt att lösa DMAPF-problemet effektivare än med traditionella MAPF-algoritmer är med hjälp av inkrementell planering. En sådan algoritm är Incremental Conflict Based Search (ICBS) som presenteras i [17], som löser dynamisk multiagentnavigering mer effektivt än Conflict Based Search (CBS).

6.1.4 Tidsrepresentationer och implikationer

Diskret MAPF förenklar världen kraftigt eftersom världen är kontinuerlig. Fördelen med diskreta MAPF algoritmer är att de är beräkningsmässigt effektiva och skalbara och kan ofta hantera upp till hundratals och tusentals agenter i ett system [33], [34]. Begränsningen med diskret MAPF är att diskretisering av vägen kan leda till extremt begränsad trajektoriekvalitet [33], vilket minskar optimaliteten av algoritmen i verkligheten.

Kontinuerlig MAPF (Multi-Agent Path Finding with Real-valued times, MAPFR) bygger på MAPF och arbetar i kontinuerlig tid. Kontinuerlig tidsrepresentation tillåter att kanter har icke uniforma längder och tillåter agenter att köra på dem över vilken positiv tid som helst [35], vilket är väldigt användbart när man vill testa olika hastighetsbegränsningar i agenterna [36]. En central utmaning med MAPFR är att prestandan av algoritmer generellt försämras kraftigt när man skalar problemet till flera agenter [33]. Men MAPFR skapar istället bättre vägar och tar hänsyn till rörelser i verkligheten, vilket gör det väldigt användbart i system som behöver planering av precisa rörelser. Detta stöds av [36], som visar att användningen av rörelsebanor i MAPFR miljöer skapar mer realistiska och körbara trajektorier under kinodynamiska förhållanden.

6.2 Konflikthanterande algoritmer: översikt

I detta avsnitt kommer den etablerade MAPF-algoritmen Priority Based Search (PBS) att presenteras. Avsnittet inkluderar även en expansion av PBS, Lifelong PBS (LPBS), samt en egenutvecklad MAPF-algortim inspirerad av de två tidigare algoritmerna, Locally Collision Resolving A* (LCRA*). Avsnittet behandlar även MAPF-algoritmernas egenskaper och hur de fungerar på en abstrakt nivå.

6.2.1 Priority Based Search

Definition
PBS är en suboptimal prioritetsbaserad MAPF-algoritm som bygger en prioritetsgraf mellan agenter för att avgöra vilka agenter som prioriteras vid konflikter och

planerar individuella tidsparametriserade rutter sekventiellt [23].

Mekaniken
Algoritmen arbetar i diskret tid och identifierar konflikter mellan agenters rutter vid varje tidssteg. Vid upptäckta konflikter uppdateras den partiella ordningen och berörda agenter omplaneras.

Körningsmodell
PBS används vanligtvis som en One-Shot algoritm där hela planeringen utförs innan exekvering påbörjas. För att följa prioritetsordningen används en topologisk sortering av prioritetsgrafen inför planeringen.

Begränsningar
Eftersom planeringen endast utförs en gång är standardversionen av PBS mindre lämplig i dynamiska miljöer där hinder och agenter förändras över tid.

6.2.2 Lifelong Priority Based Search – förbättringar av PBS

I dynamiska och lifelong MAPF-miljöer krävs kontinuerlig hantering av nya uppgifter och förändrade förutsättningar, vilket inte stöds av standard-PBS. Därför har detta projekt skapat LPBS som utökar PBS genom att introducera löpande uppgiftstilldelning och inkrementell planering, där nya uppgifter tilldelas agenter när en uppgift är avklarad och befintliga planer uppdateras vid konflikter eller miljöförändringar. Istället för att planera hela problemet igen uppdateras endast berörda agenter medan övriga planer behålls. Detta gör att tidigare planeringsresultat kan återanvändas, men kräver kontinuerlig uppdatering av konfliktinformation.

6.2.3 Locally Collision Resolving A* – vår metod

Detta projekt introducerar även en modell av LPBS kallad lokalt konfliktlösande A* (LCRA*), avsedd för kontinuerliga och dynamiska MAPF-scenarier. Metoden är inspirerad av prioritetsbaserade planeringsstrategier men reducerar koordinationen mellan agenter till lokal konfliktlösning snarare än global prioritetshantering.

LCRA* planerar individuella rutter för varje agent med hjälp av A*, där planeringen sker i statisk rumsrepresentation (x, y) utan explicit modellering av tidsdomänen. Till skillnad från klassiska MAPF-formuleringar i den tidsutvidgade sökrymden (x, y, t) separeras tidsaspekten från själva ruttplaneringen och hanteras under exekvering.

Eftersom ingen global tidskoordinering utförs, upptäcks och hanteras kollisioner lokalt under körning. Vid konflikt mellan agenter justeras berörda rutter genom lokal omplanering, medan övriga agenter fortsätter enligt sina befintliga planer. Detta

reducerar beräkningskostnaden jämfört med globalt koordinerade metoder, men innebär samtidigt att optimalitet och global konfliktfrihet inte kan garanteras.

6.3 Konflikter i MAPF: kategorier och betydelse

Här kommer vanliga konflikttyper och deadlocks att presenteras och förklaras. Detta görs för att förstå vad agenter behöver undvika för att utan problem ta sig från start till mål och det som särskiljer single agent path finding (SAPF) mot MAPF. För att representera agenter i formella definitioner använder projektet beteckningen π som representerar en agent.

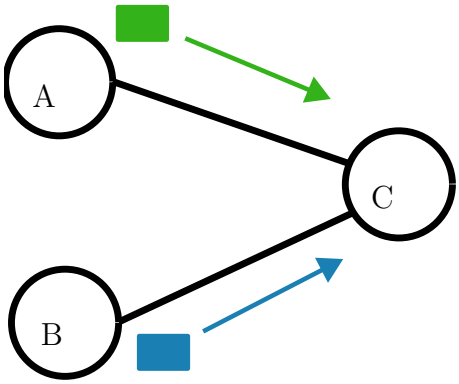
6.3.1 Kollisionstyper med exempel

För att öka effektiviteten i ett logistiksystem är en lösning att öka mängden agenter i systemet. Att öka mängden agenter skapar dock potentiella konflikter mellan agenter. Tre olika kollisionstyper har varit i huvudfokus under detta projekt: nodkonflikter, byteskonflikter och cykelkonflikter.

Nodkonflikt
En nodkonflikt uppstår när två agenter planerar att befinna sig på samma nod vid samma tidpunkt, vilket bryter mot säkerhetsregeln att endast en agent får uppehålla sig i en nod åt gången. Den formella definitionen av en nodkonflikt är beskrivet i ekvation (6.1).

$$\pi_i[x] = \pi_j[x] \tag{6.1}$$

Nodkonflikter måste hanteras för att undvika kollisioner i modellen. Nodkonflikter kan upptäckas genom att jämföra agenternas planerade positioner över tid, och de löses vanligtvis genom att införa begränsningar, omplanering eller prioriteringsregler som förhindrar att båda agenterna väljer samma nod samtidigt. En bild på innebörden av en nodkonflikt beskrivs i Figur 6.1 där både en agent på nod A och nod B vill röra sig till nod C samtidigt.



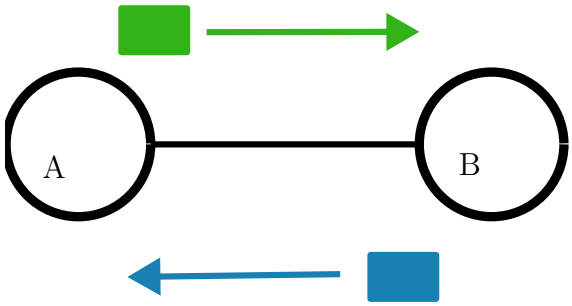
Figur 6.1: Illustration av en nodkonflikt mellan två agenter. Den gröna och blåa rutan illustrerar två olika agenter. Pilarna illustrerar att agenterna vill till nod C samtidigt.

Byteskonflikt

En byteskonflikt uppstår när två agenter vill använda samma kant i motsatt riktning samtidigt. Om byteskonflikter inte hanteras korrekt kan antingen en kollision eller dödläge uppstå. En byteskonflikt kan endast ske i en graf som har tvåvägskanter, men konflikterna är relativt vanliga när mängden tvåvägskanter blir större. Den formella definitionen för en byteskonflikt beskrivs i ekvation (6.2).

$$\pi_i[x+1] = \pi_j[x], \quad \pi_j[x+1] = \pi_i[x] \tag{6.2}$$

vilket innebär att agenterna försöker byta plats med varandra. Se Figur 6.2 för hur en byteskonflikt ser ut.

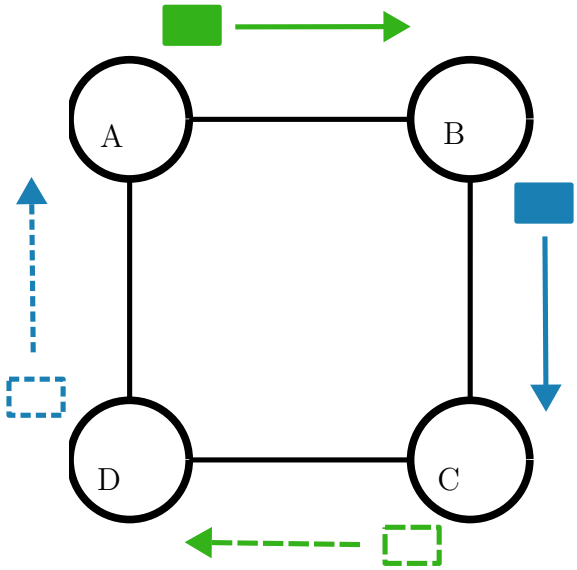


Figur 6.2: Illustration av en byteskonflikt mellan två agenter. Den gröna och blåa rutan illustrerar två olika agenter. Pilarna illustrerar att agenterna vill använda en kant samtidigt.

Cykelkonflikt

En cykelkonflikt uppstår när en mängd agenter $\pi_1, \pi_2, \pi_3, ..., \pi_n$ bildar en sluten kedja av beroenden, ett exempel visas i Figur 6.3. Det innebär att varje agent π_i väntar på att nästa agent π_{i+1} ska frigöra noden och för den sista agenten π_n gäller att den väntar på π_1. Detta skapar en cyklisk vänterelation där ingen agent kan fortsätta sin rörelse. När en cykelkonflikt uppstår utan lösning hamnar agenterna i dödläge

och eftersom fler agenter kan reservera noderna som ingår i dödläget riskerar hela systemet att stanna.



Figur 6.3: Illustration av en cykelkonflikt mellan 4 st agenter. De gröna och blåa rutorna illustrerar 4 olika agenter. Pilarna illustrerar att agenterna vill röra sig i en cykel men att noderna för tillfället är upptagna.

6.3.2 Dödlägen och relaterade problem i MAPF

Dödlägen är ofta ett utmanande problem när flera agenter samarbetar i en graf, och problemet är särskilt svårt när grafen är stor och en stor andel av alla noder ockuperas av agenter [37]. De uppstår ofta när flera agenter planerar vägar med överlappande noder, vilket kan resultera i ett statiskt låst läge där ingen agent kan fortsätta [38]. För att hantera dessa situationer finns flera metoder varav en som dynamiskt omordnar agenternas exekveringsscheman för att bibehålla synkronisering och undvika dödlägen även vid förseningar till nästa nod [39].

Dödlägen är inte det enda problemet som kan uppstå i MAPF-system, även livelock och starvation kan förekomma. Livelock innebär att agenter fortsätter att röra sig i grafen utan att komma närmare sina mål, exempelvis att två agenter omplanerar sina rutter, speglar varandras rörelser och därmed fastnar i en oändlig loop. Starvation innebär att en agent aldrig får möjlighet att röra sig eftersom andra agenter alltid har högre prioritet.

6.4 Implementation

Implementation av LCRA* bygger på en realtidsorienterad beslutsmodell för multiagentnavigering, där målet är att undvika kollisioner och dödlägen utan att förlita sig på global omplanering. Lösningen kan ses som en hybrid mellan prioriteringsbasera-

de MAPF-metoder och lokala regler. Den tar inspiration från etablerade algoritmer som PBS och LPBS, men anpassas för en miljö där agenter kontinuerligt rör sig och där beslut måste fattas snabbt och inkrementellt. I Bilaga D beskrivs pseudokoden för LCRA* i algoritm 7-12.

Lokala beslut
Till skillnad från klassiska MAPF-algoritmer, som ofta genererar fullständiga tidsutrymmen och globala planer, använder implementationen ett lokalt och temporärt beslutsramverk. Varje simuleringssteg betraktas som en isolerad beslutsenhet där systemet identifierar vilka noder och kanter som för närvarande är upptagna eller reserverade. Dessa markeras som otillgängliga under resten av steget.

Blockeringsrelationer och partiella ordningar
En central komponent i implementationen är identifieringen av blockeringar mellan agenter. Genom att analysera varje agents nästa planerade rörelse skapas beroenderelationer som motsvarar partiella ordningar. Om en agent är på väg mot en nod som för närvarande är upptagen av en annan agent etableras en prioritet där den blockerande agenten måste röra sig först.

Detta är konceptuellt besläktat med de prioriteringsstrukturer som används i PBS och LPBS, där konflikter löses genom att etablera ordningsrelationer mellan agenter. I denna implementation används dock en förenklad och mer dynamisk variant, där prioriteringskedjor genereras och utvärderas i realtid rekursivt snarare än som en del av en global sökprocess. Den rekursiva hanteringen av beroendekedjor gör att systemet kan lösa lokala flaskhalsar utan att behöva omplanera alla agenter.

Regelbaserad konflikthantering
För att avgöra om en agent får röra sig till sin nästa nod tillämpas ett antal regler. Reglerna säkerställer att nästa nod inte är upptagen, att den inte redan reserverats av en annan agent, att den inte använts tidigare under samma steg och att ingen annan agent är på väg att använda samma kant i motsatt riktning.

Denna regelbaserade modell kan ses som en form av lokal begränsning för agenter, där systemet implicit upprätthåller en konfliktfri delmängd av tillståndsutrymmet. Det möjliggör effektiv hantering av många agenter utan den omfattande globala sökning som ofta krävs i klassiska MAPF-lösningar.

Hantering av möteskollisioner och omplanering
Byteskonflikter uppstår när två agenter försöker byta plats längs samma kant. Då tillämpas en deterministisk prioriteringsregel där den agent med lägst prioritet tvingas omplanera sin rutt. Omplaneringen sker genom en ny sökning med hjälp av ruttplaneringen där den blockerande noden temporärt undantas. Detta motsvarar en

lokal reparation av planen. Denna mekanism tar bort risken för dödlägen på tvåriktade kanter och säkerställer att systemet kan fortsätta framåt även när två agenter möts frontalt.

Cykeldetektion och dödläges-hantering
Cykliska beroenden kan även uppstå, särskilt i täta miljöer där flera agenter blockerar varandra. Implementationens cykeldetektion bygger på att analysera de agenter som står stilla och kartlägga deras beroenden. Cykler identifieras genom att en Depth First Search (DFS) utförs varje iteration på de agenter som står stilla och deras beroenderelationer. Om en cykel upptäcks tillåts samtliga agenter i cykeln att röra sig samtidigt.

6.5 Resultat och analys

I grafen presenterad i avsnitt 5.3.1 fungerar LCRA* för att undvika kollisioner mellan agenter samtidigt som de kan röra sig till ett mål. Detta bestäms som ett adekvat resultat då projektet menade att producera rutter till agenter. Algoritmen går att skala till en stor mängd agenter då prioritetsscheman används. Algoritmens tidskomplexitet utvärderas inte då projektets mål är att utvärdera om det går att använda och inte hur bra det görs.

7 Inspärrning av inkräktare

Inspärrning av inkräktare utgör det delproblem som i detta projekt behandlas i syfte att integrera två samverkande system för att utföra ett gemensamt uppdrag. Drönarplattformen ansvarar för att lokalisera ett dynamiskt objekt genom områdesavsökning. Information om det dynamiska objektets position används för att förflytta AMR:erna så att det dynamiska objektet innesluts.

7.1 Introduktion

Pursuit–Evasion Problem (PE-problemet) studerar situationer där en eller flera förföljare ska stänga in en eller flera undvikande agenter i en gemensam miljö [40]. Det finns flera strategier för att angripa PE-problemet. Tidigare arbeten har bland annat använt maskininlärningsbaserade metoder, där exempelvis q-inlärning [41] och neurala nätverk [42] har tillämpats för att välja handlingar inom systemet eller för att optimera delproblem, såsom evaluering av tillstånd.

En annan strategi är sannolikhetsbaserade metoder, exempelvis för att approximera förföljarens position när den är okänd [43] eller för att vikta möjliga förflyttningar och därefter välja den optimala handlingen [44]. Två ytterligare metoder är Hamilton-Jacobi-Issacs (HJI)[45] och Hamilton-Jacobi-Bellman (HJB) [41] som approximerar optimala strategier genom att skapa en partiell differentialekvation. Dessa ekvationer kan approximeras numeriskt eller med maskininlärning. Sist finns hybridlogiker och egendefinerade metoder vilket kombinerar metoder eller använder andra strategier som inte tydligt tillhör nämnda strategier. Ett exempel på en hybrid och egen strategi nämns av A. Kolling i [46], vilket bemöter PE-problem med en graf-baserad strategi som genomsöker området genom att trädstrukturera grafen och blockerar kanter som leder till cykler i grafen.

7.2 Praktisk grund och designval

För att hantera PE-problemet i detta projekt har förenklingar av systemet genomförts. Vidare har två testmetoder används i syfte att effektivisera utvecklingen av programkod. Den övergripande strategin för att bemöta PE-problemet har även utvärderas och presenteras på en konceptuell nivå i detta avsnitt.

7.2.1 Situation och antaganden

PE-problemet tillämpas i detta projekt på ett fåtal AMR:er vars syfte är att omringa en inkräktare i en grafmiljö. Grafen består av noder och kanter som representerar positioner respektive vägar. En inkräktare definieras som omringad när samtliga noder som är direkt nåbara från inkräktarens nuvarande position är blockerade av AMR:er i förföljarläge. Delproblemet inspärrning av inkräktare initieras när en in-

kräktare identifieras inom lokalområdet. Därefter fortsätter programmet att köras i förföljarläge fram till omstart. När inkräktaren upptäcks skickar drönarplattformen inkräktarens position till systemet. Inkräktaren tilldelas därefter den närmaste noden vid initiering av PE-problemet. Scenariot anses fullbordat då inkräktaren är infångad, varpå samtliga AMR:er stannar.

7.2.2 Testning och utvärderingsgrund

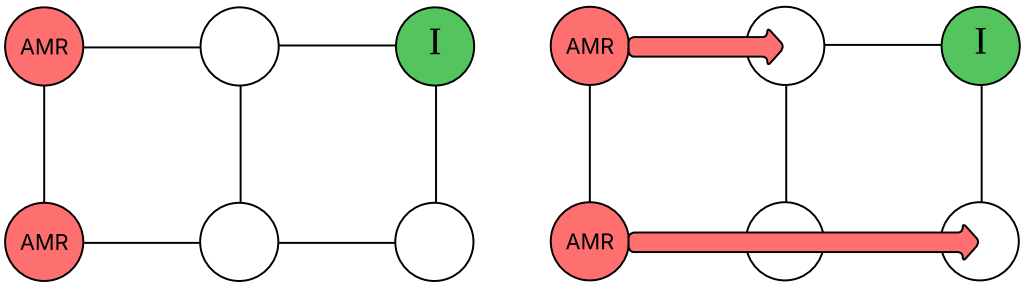
Testning av systemet genomfördes med två metoder. Under utveckling av programkoden utvärderades systemet med en animation av förloppet. Förenklingar som gjordes för denna metod var att inkräktaren rör sig slumpmässigt i grafen genom att förflytta sig till en slumpmässig grannod från nuvarande positions nod. Förflyttning sker iterativt, där både inkräktaren och AMR:erna förflyttas längs grafens kanter i lika stora diskreta steg. Den andra testmetoden utgjordes av verklig hårdvarutestning, där både inkräktaren och AMR:erna representeras som noder i systemets graf baserat på deras uppmätta positioner. När inkräktaren eller AMR:erna är inom ett bestämt avstånd till en grannod från nuvarande positions nod, uppdateras nuvarande position till grannoden.

Programkoden och strategier utvärderades utifrån två bedömningsgrunder. Den första bedömningsgrunden var AMR:ernas beteendemönster och den andra hur snabbt AMR:erna uppnår målet att omringa inkräktaren. Tiden för att uppnå målet har dock endast analyserats på en övergripande observationsnivå och kvantitativa mätningar har endast genomfördes vid analysering av skalbarhet. Majoriteten av testningen utfördes med animation på grund av dess snabba återkoppling, justerbara grafmiljö och ett enkelt skalbart antal AMR:er. Testning i verklighet med hårdvara gjordes men i lägre utsträckning på grund av metodens höga tidskostnad och bristande repeterbarhet.

7.2.3 Implementering och metodval

Maskininlärning implementerades inte, främst eftersom deterministiska metoder är lättare att utvärdera, men även för att strategin ansågs vara för komplex inom projektets omfattning. Sannolikhetsbaserade metoder bedömdes ge begränsad nytta eftersom inkräktaren antas förflytta sig slumpmässigt i grafen. Att utforma sannolikhetslogiken med en människostyrd inkräktare i åtanke skulle avsevärt öka komplexiteten och ansågs ligga utanför projektets ramar. Matematikbaserade metoder exkluderades på grund av implementationskomplexitet. Därför definierades egen logik och hybridlogik, då det möjliggör en mer hanterbar implementation. Två angreppssätt utformades: en blockeringsmetod och avgränsningsmetod, baserade på var sin logik.

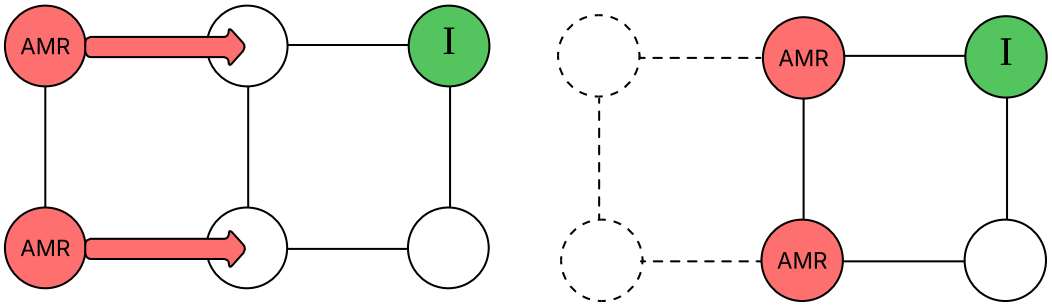
Blockeringsmetoden syftar till att begränsa inkräktarens rörelseförmåga genom positionering av AMR:er nära inkräktaren. Metoden baseras på att AMR:er tilldelas noder i inkräktarens rörelseriktning i syfte att blockera inkräktarens förflyttningsmöjligheter (se Figur 7.1).



Figur 7.1: Illustration av en grafmiljö. Röda noder (AMR) rör sig för att positionera sig nära den gröna inkräktarnoden (I).

Blockeringsmetoden omfattar strategier för koordineringen mellan AMR:er samt identifiering av lämplig blockeringsnoder. Vilken nod som är mest lämplig prioiteras i detta projekt utifrån nodens avstånd till inkräktarens nuvarande position, där närliggande noder ges högst prioritet. En breadth first search (BFS) tillämpas för att prioritera noder utifrån deras avstånd från inkräktarens position. BFS returnerar därefter en lista av noder i prioriterad ordning. Koordineringen kan sedan använde listan för att tilldela AMR:er blockeringsnoder enligt denna prioritering.

Avgränsningsmetoden syftar till att begränsa inkräktarens rörelseområde genom positionering av AMR:er på utvalda noder (se Figur 7.2). Metoden baseras på att successivt minska det område som inkräktaren kan nå från sin nuvarande position. Rörelseområdet definieras här som mängden noder som inkräktaren kan nå från den aktuella positionen. Metoden implementeras med målet att möjliggöra omringning av inkräktaren.



Figur 7.2: Illustration av en grafmiljö. Röda noder (AMR) rör sig för att minska den gröna inkräktarnodens (I) rörelseområde. Streckade noder beskriver blockerade områden.

Avgränsningsmetodens strategi är att varje AMR gör ett lokalt optimalt val oberoende av de övriga AMR:erna. Ett optimalt val definieras som en förflyttning från nuvarande position till en grannod som resulterar i en minskning av inkräktarens rörelseområde. För att identifiera inkräktarens rörelseområde används en BFS, med justering noder som blockeras av AMR:er inte inkulderas i beräkningen.

Hybridstrategin utvecklades i syfte att kompensera för de enskilda metodernas respektive svagheter. Strategin utformades så att de två metoderna anropas iterativt, vilket innebär att båda påverkar det slutliga utfallet.

7.3 Resultat

I detta projekt har en enklare lösning till PE-problemet implementerats där en egen hybridstrategi vuxit fram som grundas i grafteori. Denna strategi bygger på att kombinera två separata metoder genom att anropa en av dem iterativt, där metoden som körs bestäms av specifika villkor. Den första metoden löser PE-problemet genom att skicka förföljare till noder nära inkräktarens nuvarande position. Den andra metoden löser PE-problemet genom att successivt minska ytan inkräktaren kan röra sig på genom att blockera noder.

7.3.1 Blockeringsmetoden, Blockering av noder nära inkräktare

Implementeringen av blockeringsmetoden baserades på BFS-prioritering utifrån nodernas avstånd till inkräktaren. Strategin möjliggjorde tilldelning av AMR:er till olika och viktiga målnoder. Detta resulterade i ett system som anpassar blockering till antalet tillgängliga AMR:er. Pseudokod för blockeringsmetoden är presenterad i algoritm 1 nedan.

Algorithm 1 Blockeringsmetoden

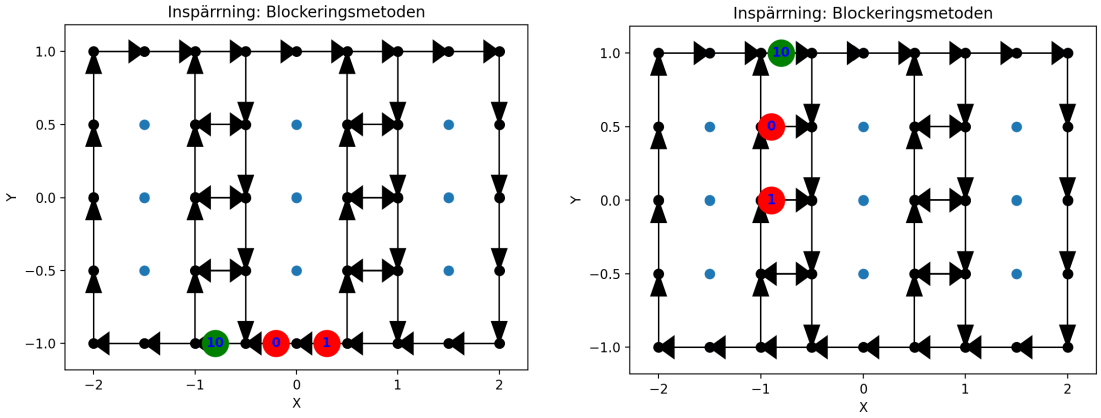
Input: $AgentObjs$: all AMR in the system $IntruderObj$: the intruder in the system
Output: update path for AMR

```
1: function LocalBlocker(AgentObjs,IntruderObj)
2:     PossibleNextNodes ← BFS(IntruderNode)
3:     for AgentObj in AgentObjs do
4:         set AgentObj to not have an assignment
5:     for target in PossibleNextNodes do
6:         ClosestAgent ← GetClosestAgent(target)
7:         set ClosestAgent as busy
8:         NewPath ← Astar(ClosestAgentNode,target)
```

Programmet för blockeringsmetoden initieras med en BFS som prioriterar noder baserat på avstånd till inkräktaren. Vid lika avstånd prioriteras noden med flest grannoder. BFS avbryts när listans längd motsvarar antalet AMR:er i systemet. Därefter sätts samtliga AMR:er i tillståndet ledig. Målnoder genereras genom att iterera över BFS-listan och tilldelas AMR:er baserat på minsta avstånd samt nodernas prioritetsordning. När en AMR tilldelas en målnod markeras den som upptagen för att förhindra dubbel tilldelning. Slutligen används A* för att generera en rutt från respektive AMR:s aktuella position till den tilldelade målnoden, och AMR:ens rutt uppdateras därefter.

Resultatet visar att strategin i samtliga genomförda tester lyckas omringa inkräktaren i den använda grafmiljön när en eller fler AMR:er används och ingen tidsbegränsning tillämpas. Det ska dock påpekas att testning endast utfördes på en enkel graf med fasta egenskaper och primärt genom animationstestning. Testmetoden begränsar därmed resultatets generaliserbarhet.

Teoretiskt kan strategin inte garantera inspärrning. Under testning med få AMR:er riskerar samtliga att följa efter inkräktaren snarare än att omringa den. Om inkräktaren skulle agera optimalt med syfte att undvika att omringas skulle detta kunna leda till situationer där AMR:erna endast följer efter inkräktaren utan att någonsin lyckas omringa den (se Figur 7.3).



Figur 7.3: Illustration av ledmönstret via en stillbild från animation. Röda markeringar representerar AMR:er och den gröna markeringen representerar inkräktaren.

Beteendemönstret beror på hur A* arbetar med strategin. När AMR:erna åker mot inkräktarens nuvarande position och inkräktaren sedan förflyttar sig kan AMR:erna positioneras bakom inkräktaren. När A* genererar en ny rutt försöker AMR:erna ta sig framför inkräktaren genom att köra igenom inkräktarens nuvarande position. Kollisionshanteraren tillåter inte körningen, vilket kan resultera i att AMR:erna följer efter inkräktaren. Genom att tilldela AMR:erna olika målnoder och att målnoderna kräver minst två grannoder motverkas beteendet.

Fördelen med strategin är dock att AMR:erna snabbt kan komma nära inkräktarens nuvarande position. Om AMR:en är på en bra startposition i förhållande till inkräktaren så kan denna strategi snabbt spärra in inkräktaren. För att effektivisera strategin ytterligare krävs en positionspredikering och anpassning för inkräktares position i ruttplaneringen.

7.3.2 Avgränsningsmetoden, minska inkräktarens rörelseområde

Implementeringen av avgränsningsmetoden resulterade i en strategi där varje AMR fattar egna beslut oberoende av övriga AMR:er. Besluten fattas baserat på vilken förflyttning till grannod som minimerar inkräktarens rörelseområde vid den aktuella tidpunkten. Rörelseområdet beräknas med en BFS variant som returnerar en lista av noder inkräktaren kommer åt med hänsyn till blockerade noder. Förflyttningar som resulterar i en ökning av inkräktarens rörelseområde tillåts inte. Om inget alternativ finns behåller AMR:en sin nuvarande position.

Pseudokod för avgränsningsmetoden är presenterad i algoritm 2 nedan.

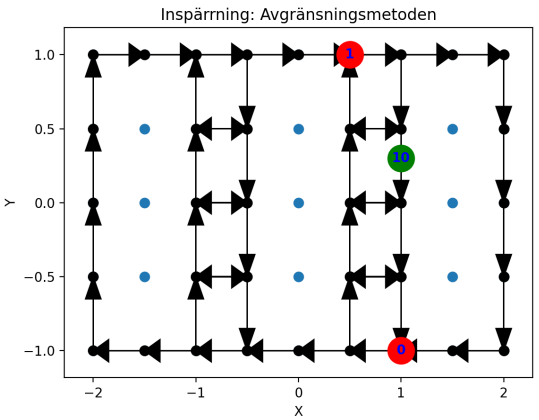
Algorithm 2 Avgränsningsmetoden

Input: *AgentObjs*: all agents in the system *IntruderObj*: the intruder in the system

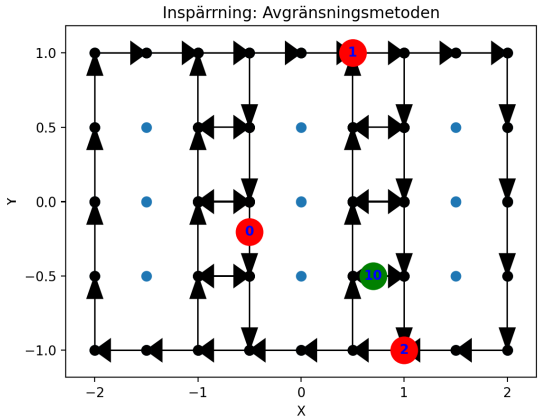
Output: total amount of movement this iteration

```
1: function ReduceArea(AgentObjs,IntruderObj)
2:     QueueCorrelation, MovmentThisIteration ← empty list
3:     AgentsOccupyNodes ← list of nodes agents are currently occupying
4:     IntruderReachableNodes ← length of BFS(IntruderNode,AgentsOccupyNodes)
5:     for AgentObj in AgentObjs do
6:         BestAgentMovment ← IntruderReachableNodes
7:         for AgentNeighbor in AgentNeighbours do
8:             if AgentNeighborNode in AgentsOccupyNodes then
9:                 if AgentNeighborNeighbor not in AgentsOccupyNodes then
10:                    QueueCorrelation.append(AgentNeighbor)
11:            else
12:                replace AgentNode to AgentNeighborNode in
    AgentsOccupyNodes
13:                IntruderReachableNodes2 ← BFS(IntruderNode,AgentsOccupyNodes)
14:                if IntruderReachableNodes < BestAgentMovment then
15:                    BestAgentMovment ← AgentNeighbor
16:        MovmentThisIteration.append([AgentObj,BestAgentMovment])
17:    update AgentObj path specified in MovmentThisIteration
18:    assign agentObj in QueueCorrelation to make random movement to neigh-
    bor
       return len(BestAgentMovment+QueueCorrelation)
```

Resultatet visar att det finns två situationer då avgränsningsmetoden själv inte kan stänga in inkräktaren. Den första situationen uppstår när alla AMR:er i systemet är positionerade på viktiga noder och alla förflyttningar resulterar i en ökning av inkräktarens rörelseområde. Samtliga AMR:er förblir då stillastående (se Figur 7.4). Den andra situationen inträffar när en AMR oavsett förflyttning inte påverkar inkräktarens rörelseområde. Detta kan behandlas genom att slumpmässigt förflytta AMR:en till en grannod, vilket resulterar i en mycket ineffektiv körning (se Figur 7.5). Den slumpmässiga förflyttningen har valts bort till förmån för kombinationen mellan blockeringsmetoden och avgränsningsmetoden.



Figur 7.4: Låssituation där alla förflyttningar resulterar i immobilisering. Den röda pricken representerar AMR:er och den gröna pricken representerar en inkräktare.



Figur 7.5: Låssituation med två immobiliserade positioner och en i slumpmässig rörelse. Den röda pricken representerar AMR:er och den gröna pricken representerar en inkräktare.

Avstängningsmetoden behöver flera AMR:er i systemet för att bli effektiv, men kan effektiviseras ytterligare med tillägg. Om AMR är på en position som inte påverkar inkräktarens rörelseområde, oavsett förflyttning, ska AMR:en köra en direktväg mot inkräktaren. Ytterligare behandling av situationen när alla AMR:er är låsta och alla förflyttningar ökar inkräktarens rörelseområde kan implementeras. Effektivisering kan realiseras genom att skicka AMR:en som är längst bort från inkräktaren till en annan nod, vilket bryter låsningen. Alternativt kan en prognos göras som förflyttar AMR:erna mot nästa låsningstillstånd.

Fördelen med strategin för detta projekt är att fångst uppnås i den definierade grafen så länge det finns tre eller fler AMR:er i systemet och det finns möjliga förflyttningar att göra. Lösningen är inte tidseffektiv, men kan för flera situationer garantera en fångst.

7.3.3 Hybridlösning, kombination

Kombinationen av blockeringsmetoden och avgränsningsmetoden bygger på att de två metoderna anropas iterativt. Kombinationen har implementerats med detta i åtanke och är strukturerad så att den ena metoden eller den andra anropas under en iteration beroende på två villkor.

Pseudokod för kombinationen är presenterad i algoritm 3 nedan.

Algorithm 3 Hybridlogik

Input: $AgentObjs$: all AMR in the system $IntruderObj$: the intruder in the system
Output: update path for AMR

```
1: function IntruderCatch(AgentObjs,IntruderObj)
2:     if CurrentAlgorithm = algorithm one then
3:         AmountOfMovement ← ReduceArea(AgentObjs,IntruderObj)
4:         if AmountOfMovement = 0 then
5:             CurrentAlgorithm ← algorithm two
6:     if CurrentAlgorithm = algorithm two then
7:         LocalBlocker(AgentObjs,IntruderObj)
8:         Counter += 1
9:     if Counter = 300 then                    ▷ 300 is a selected value
10:        CurrentAlgorithm ← algorithm one
11:        Counter ← 0
12:    if intruder have no available neighbours then
13:        set agents to stand still
```

Programmet initieras med att `current_algorithm` sätts till blockeringsmetoden vid första körningen av jaktalgoritmen. Metoden körs därefter i 300 iterationer, varefter `current_algorithm` växlar till avgränsningsmetoden. Denna körs tills inga ytterligare optimala förflyttningar kan göras. Därefter återgår `current_algorithm` till blockeringsmetoden, som återigen körs i 300 iterationer. Detta cykliska beteende upprepas tills samtliga grannoder till inkräktarens aktuella nod är ockuperade av AMR:er och inkräktaren därmed är inlåst. Vid inlåsning uppdateras systemets variabler för att indikera att fångst är uppnådd.

Strategierna arbetar med varandra genom att blockeringsmetoden skickar AMR till noder nära inkräktaren och avgränsningsmetoden kan sedan garantera en instängning oberoende av inkräktarens förflyttning. Detta gör att blockeringsmetoden kompenserar för avgränsningsmetoden algoritmen genom att göra förflyttningar när avgränsningsmetoden inte kan det. Dessutom gör detta att AMR snabbt kan komma nära inkräktaren.

Kombinationen arbetar mot varandra genom att när blockeringsmetoden kallas efter avgränsningsmetoden så bryts inlåsningen, vilket gör att kombinationen inte kan garantera en fångst. Dessutom om ett lågt antal AMR används och en situation där AMR följer efter inkräktaren inträffar så har avgränsningsmetoden svårt att komma ur detta tillstånd då alla AMR är samlade på en och samma position.

Kombination skulle kunna effektiviseras ytterligare om sättet integrationen effektiviseras. Exempelvis när avgränsningsmetoden inte kan göra en rörelse så körs bloc-

keringsmetoden endast för ett visst antal utvalda AMR som inte bryter inlåsningstillståndet får avgränsningsmetoden. Eller så används bara en utav dessa strategier beroende på hur många AMR som finns i grafen och hur stor grafen är.

För grafer inom ramen för projektet har problemet alltid observerats till att kunna stänga in inkräktaren så länge man ger det tillräckligt lång tid.
Majoriteten av testningarna genomfördes med animationer med en förenklad dynamik. Fysisk testning utfördes i ett begränsat antal, där inget komplett test av den slutliga programkoden kunnat utföras. Funktionaliteten av programkoden som löser logikproblemet för inspärrning av inkräktaren i fysisk miljö kan därmed inte garanteras. Vidare testning krävs för att säkertälla måluppfyllnad av delproblemet inspärrning av inkräktare. Det bör dock påpekas att verklig testning har ett krav på synlighet för inkräktare och AMR när en nod nås. För att i verklig miljö låta programkod kunna uppdatera dess positioner.

7.3.4 Skalbarhet

Projektets skalbarhet undersöktes genom att utvärdera blockeringsmetoden och avgränsningsmetoden separat. Eftersom systemet vid varje iteration använder metoderna alternativt, utgör respektive metods skalbarhet en direkt indikation på systemets totala skalbarhet.

En tidskomplexitetsanalys har undersökts för båda delmetoderna. För blockeringsmetoden beräknades komplexiteten till $O(A(V^2+E))$ och för avgränsningsmetoden till $O(A(V+E))$, där A är antal AMR i systemet, V är antal noder och E antal kanter. Detta visar på att blockeringsmetoden påverkas i högre grad av grafens storlek än vad avgränsningsmetoden gör.

Empirisk data har samlats in genom animationstestning där medelvärdet för en iteration beräknades för de respektive metoderna. Resultatet av testningen presenteras i förenklad form i Tabell 7.1. Resultatet visar att tidsåtgången för blockeringsmetoden ökar kraftigare vid grafskalning än för avgränsningsmetoden. Det bör dock noteras att blockeringsmetoden i majoriteten av fallen kräver färre iterationer för att omringa målet.
Samtliga mätvärden presenteras i Bilaga C.1.

Tabell 7.1: Grafstorleken ges av antal noder i x och y led. Tid, A* och BFS är medelvärdet per iteration. Iter. beräknas som är antal iterationer till fångst uppnåtts

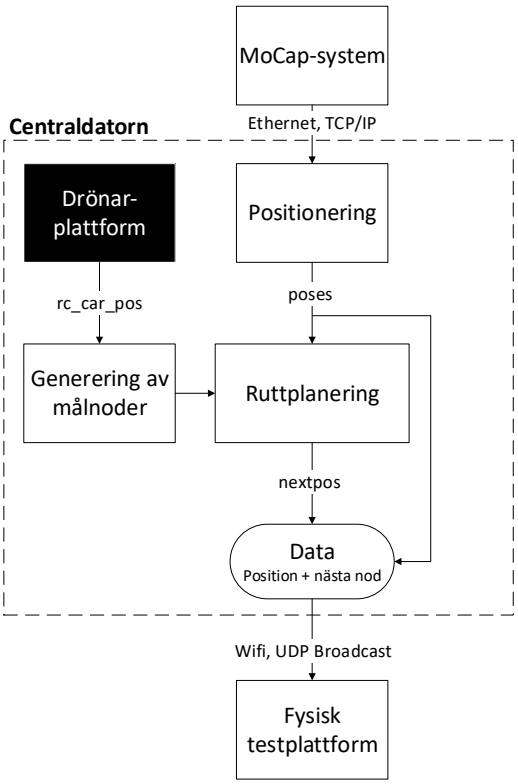
Antal AMR & Graf-storlek	**Metod**	**Tid (ms)**	**A* (st.)**	**BFS (st.)**	**Iter. (st.)**
AMR:3, Graf: 9x5	Blockering	0.254	3	1	11
	Avgränsning	0.211	0	6.2	16
AMR:15, Graf: 18x10	Blockering	10.8	15	1	2
	Avgränsning	1.39	0	23.0	4
AMR: 30, Graf: 90x10	Blockering	399	30	1	13
	Avgränsning	17.0	0	50.0	42

Sammanfattningsvis visar testerna att systemet hanterar grafer med upp till 900 noder (90×10) och 30 AMR:er inom rimliga tidsramar. Utifrån detta dras slutsatsen att en hybridlogik mellan metoderna kan skalas upp till testningens maximala omfattning, vilket överstiger de krav som ställts inom ramen för projektet.

8 Systemarkitektur och utvärdering

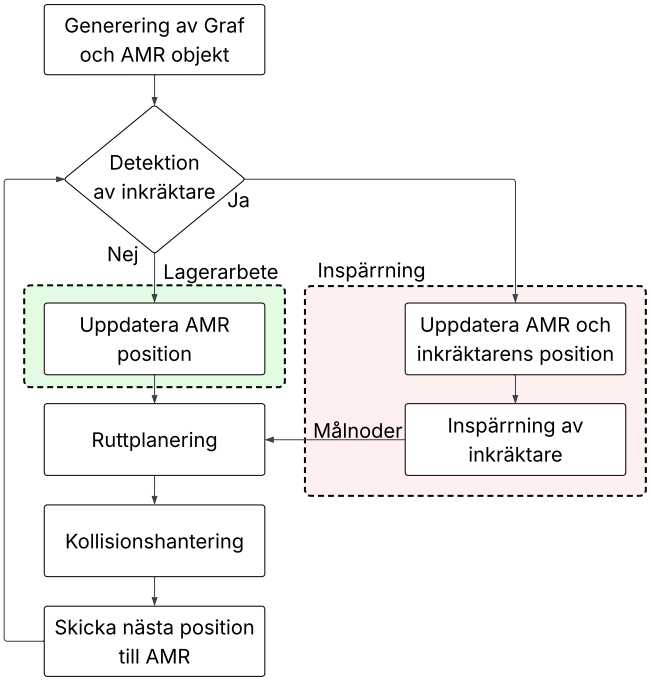
Avsnittet innehåller en beskrivning av hur lösningarna för respektive delproblem byggs upp till det slutgiltiga systemet samt en utvärdering av systemet.

Systemets arkitektur sammanfattas i Figur 8.1. Figuren visar hur 6DOF-positionsdata från MoCap-systemet skickas till centraldatorn via Ethernet. I *positionering* transformeras positionsdatan till en 2D-koordinat och en orientering kring z-axeln, varefter den distribueras till *ruttplanering* via `poses`. När inkräktaren inte har upptäckts genereras målnoder slumpmässigt. När inkräktaren upptäcks genereras målnoderna istället av *inspärrning av inkräktare* som tar emot inkräktarens position från den externa drönarplattformen via `rc_car_pos`. Utifrån målnoderna beräknar ruttplaneringen rutter för respektive AMR och publicerar därefter nästa nod i rutten via `nextpos`. Slutligen kombineras nästa nod för varje AMR med dess aktuella position och distribueras till samtliga AMR:er via UDP-broadcast över wifi.



Figur 8.1: Översikt av systemarkitektur. Kommunikation in till och ut från centraldatorn görs via Ethernet respektive wifi. Inne i centraldatorn sker kommunikationen med ROS2 via topics med namnen `rc_car_pos`, `poses` och `nextpos`.

En mer detaljerad systembeskrivning av delmoment *Generering av målnoder* och *ruttplanering* från Figur 8.1 visas i Figur 8.2. När programmet initieras inleds exekveringen med *Generering av graf och AMR-objekt*. Därefter kontrolleras om en inkräktare har detekterats i systemet. Beroende på detta körs antingen *Lagerarbete* eller *Inspärrning*, vilka i figuren representeras av gröna respektive röda områden. Därefter utförs ruttplanering och kollisionshantering, varefter information om nästa position skickas till *Data* i Figur 8.1. Slutligen återgår programmet till *detektion av inkräktare*, där en ny iteration påbörjas.



Figur 8.2: Mer detaljerad systemarkitektur för delproblem ruttplanering, kollisionshantering och inspärrning av inkräktare. Programmet initieras med generering av graf och AMR objekt. Uppdatering av positioner fås från MoCap-system och drönarplattform.

Tester visade att med positionsdata från MoCap, tar ruttplaneringen fram rutter för alla AMR:er och ger ut nästa nod i rutten. Nästa nod tillsammans med positionsdata skickades utan märkbar latens eller dataförlust till alla AMR:er som kan beräkna en körriktning för att nå den önskade noden. En begränsning i systemet är att endast envägskommunikation mellan centraldatorn och den fysiska testplattformen implementerades. Det medför att logiken blir helt beroende av att MoCap-systemet ser alla AMR:er vid alla tidpunkter. Även om varje AMR vet sin position ungefärligt med odometri finns inget sätt att meddela ruttplaneraren att nästa nod är nådd om den nås medan AMR:en inte är synlig för MoCap-systemet.

Den globala kollisionshanteringen fungerade i simulerad miljö, där agenterna undvek kollisioner och nådde sina målpunkter. Metoden genererade kollisionsfria rutter utifrån deterministiska agentmodeller och tog inte hänsyn till återkoppling från AMR:ernas faktiska positioner under körning. Eftersom AMR:erna i fysisk miljö uppvisade oförutsägbara rörelser kunde global kollisionsundvikning därför inte garanteras. På AMR:erna implementerades ingen lokal kollisionshantering, vilket innebar att kollisionsfri drift i fysisk miljö inte kunde säkerställas.

Inspärrning av inkräktare har observerats fungera i datorsimulerad miljö. Programkoden kan ta in information, generera målnoder och skicka nästa position till AMR:erna, men inspärrningen har inte kunnat verifieras med den fysiska plattformen.

Vid design av ruttplanering och kollisionshantering har skalbarheten av AMR:er varit en viktig del. Digitalt har systemet testats med grafer upp till ett hundratal noder och ett tiotal AMR:er. Programkoden har fungerat som specificerat under testerna. Vid test i verkligheten har det endast testats med maximalt tre AMR:er och ett ental drönare. Skalbarheten i den fysiska plattformen har endast kunnat utvärderas i begränsad omfattning på grund av att endast tre AMR:er varit tillgängliga under projektet. Systemet anses därmed skalbart inom projektets ramar då programkoden är utformad för att klara en betydligt större mängd AMR:er.

9 Slutsats och utvecklingsmöjligheter

Projektet har på simuleringsnivå demonstrerat att en skalbar flotta av AMR:er kan automatisera förflyttningar i en lagermiljö. När lagermiljönavigeringen applicerades på en fysisk plattform var de tillgängliga AMR:erna inte tillräckligt stabila för att bevisa att metoden fungerar.

Projektet demonstrerade även att systemet kan spärra in dynamiska mål, vilket innebär att målet inte behöver vara statiskt för att agenterna ska kunna nå och omringa det. Detta kan vara fördelaktigt i lagermiljöer där objekt förflyttas under drift, eftersom målpositionen då kan uppdateras dynamiskt medan AMR:erna kontinuerligt anpassar sina positioner. Funktionen kan även användas som en säkerhetsmekanism, där övriga AMR:er kan spärra in en felaktigt fungerande eller övertagen AMR för att minska risken för skador på människor och omgivning.

Projektet har gett en grund för att installera MoCap-system i riktiga lagermiljöer. Under tester på den fysiska plattformen har MoCap med god precision detekterat AMR:ers positioner. Vid tillämpning i verkligheten är MoCaps infraröda kameror även en potentiell fördel ur integritetssynpunkt då bilder inte sparas.

För nästa testplattform som används rekommenderas ROS2-kompatibilitet, vilket underlättar tvåvägskommunikation och möjliggör datainsamling från testplattformen, såsom rapportering av hinder eller löpande diagnostik. Ruttplaneringsmetoden kan expanderas för att täcka konflikttyper som ej hanterats inom ramen för projektet, exempelvis prioriteringskonflikter och navigering i smala passager. En ytterligare förbättringsmöjlighet är att utvärdera följdkonflikter, det vill säga att ta hänsyn till avståndet mellan agenter under färd. Om en agent är betydligt långsammare än en bakomliggande agent riskerar en kollision att uppstå i verkligheten, trots att den inte detekterades i simulering. Detta scenario är mindre sannolikt att fångas digitalt men förekommer oftare i fysiska driftsmiljöer där förutsättningarna varierar. En annan utvecklingspotential är att beräkna mjuka kurvor så att det blir enklare för AMR:er att följa rutten i ej diskreta steg.

Sammantaget visar projektet att det är möjligt att integrera positionering, ruttplanering och kollektiv koordination i ett autonomt system, ett konkret steg mot den automatiserade och adaptiva lagermiljö som industri 5.0 eftersträvar. Det visar också på de fortsatta utmaningar som finns med att integrera fysisk hårdvara även med bra simuleringar.

Referenser

[1] P. Y. Leong och N. S. Ahmad, ”Exploring Autonomous Load-Carrying Mobile Robots in Indoor Settings: A Comprehensive Review,” *IEEE Access*, årg. 12, s. 131 395–131 417, Jul. 2024, DOI: 10.1109/ACCESS.2024.3435689.

[2] Q. G. Zheng Zhang Juan Chen, ”Application of Automated Guided Vehicles in Smart Automated Warehouse Systems: A Survey,” *Computers & Industrial Engineering*, årg. 134, nr 3, s. 1–35, Sep. 2022. DOI: 10.32604/cmes.2022.021451.

[3] A. Zia och M. Haleem, ”Bridging Research Gaps in Industry 5.0: Synergizing Federated Learning, Collaborative Robotics, and Autonomous Systems for Enhanced Operational Efficiency and Sustainability,” *IEEE Access*, årg. PP, nr 99, s. 1–1, Jan. 2025. DOI: 10.1109/ACCESS.2025.3541822.

[4] MakoGomez90, *AGV con carro*, Wikimedia Commons [Fotografi], Hämtad: 2026-05-08., 2026. URL: https://commons.wikimedia.org/wiki/File:AGV_con_carro.jpg.

[5] H. Jiang, T. Mao, S. Wu, M. Xu och Z. Wang, ”A Local Evaluation Approach for Multi-Agent Navigation in Dynamic Scenarios,” i *Proceedings of the 13th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and its Applications in Industry (VRCAI '14)*, Shenzhen, Kina, 2014, ss. 89–93. [Online]. Tillgänglig: 10.1145/2670473.2670493, Hämtad: 2026-05-05.

[6] G. Li, J. Wang och H. Wang, ”Cooperative Collision Avoidance Strategy for Multi-Agent Based on Dynamic Game,” i *Proceedings of the 2023 3rd International Conference on Electronic Information Engineering and Computer Science (EIECS)*, Changchun, Kina, 2024, ss. 907–912. [Online]. Tillgänglig: 10.1109/EIECS59936.2023.10435374, Hämtad: 2026-05-05.

[7] GeeksforGeeks. ”TCP vs. UDP.” 2026. [Online]. Tillgänglig: https://www.geeksforgeeks.org/computer-networks/differences-between-tcp-and-udp/ (hämtad: 2026-04-29).

[8] S. Skogestad och C. Grimholt, ”The SIMC Method for Smooth PID Controller Tuning,” i *PID Control in the Third Millennium*, ser. Advances in Industrial Control, R. Vilanova och A. Visioli, utg., London: Springer, 2012, kap. 5, 147–175, [Online]. DOI: 10.1007/978-1-4471-2425-2_5. hämtad 28 april 2026.

[9] *System Identification Toolbox, Version: 25.2 (R2025b)*, [Mjukvara], Natick, Massachusetts, USA: The MathWorks Inc., 2025.

[10] OpenCV. ”Geometric Image Transformations.” 2026. [Online]. Tillgänglig: https://docs.opencv.org/4.x/da/d54/group__imgproc__transform.html (hämtad: 2026-04-28).

[11] OpenCV. ”Smoothing Images.” 2026. [Online]. Tillgänglig: https://docs.opencv.org/4.x/d4/d13/tutorial_py_filtering.html (hämtad: 2026-05-07).

[12] OpenCV. ”Image Thresholding.” 2026. [Online]. Tillgänglig: https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html (hämtad: 2026-05-07).

[13] OpenCV. ”Canny Edge Detection.” 2026. [Online]. Tillgänglig: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html (hämtad: 2026-05-07).

[14] OpenCV. ”Contours: Getting Started.” 2026. [Online]. Tillgänglig: https://docs.opencv.org/4.x/d4/d73/tutorial_py_contours_begin.html (hämtad: 2026-05-07).

[15] L. Rocha och K. Vivaldini, ”Analysis and Contributions of Classical Techniques for Path Planning,” i *2021 Latin American Robotics Symposium (LARS)*, Natal, Brasilien, 2021, ss. 54–59. [Online]. Tillgänglig: 10.1109/LARS/SBR/WRE54079.2021.9605425, Hämtad: 2026-05-05.

[16] S. Sakr, *Large-Scale Graph Processing Systems*, Schweiz, ser. SpringerBriefs in Computer Science, 2016, kap. 4, ss. 35–51. [Online] Tillgänglig: https://doi.org/10.1109/CCDC.2009.5192696, Hämtad: 2026-05-05.

[17] X. Xing, B. Wang och J. Cheng, ”Incremental CBS: Adaptive Multi-Agent Pathfinding in Dynamic Environments,” i *2024 7th International Conference on Robotics, Control and Automation Engineering (RCAE)*, Wuhu, Kina, 2024, ss. 164–169. [Online]. Tillgänglig: 10.1109/RCAE62637.2024.10834259, Hämtad: 2026-05-05.

[18] R. Duck, *Von Neumann neighborhood*, Wikimedia Commons [Fotografi], Hämtad: 2026-05-13, 2016. URL: https://commons.wikimedia.org/wiki/File:Von_Neumann_neighborhood.svg.

[19] D. Eppstein, *King's graph with white king*, Wikimedia Commons [Fotografi], Hämtad: 2026-05-13, 2019. URL: https://commons.wikimedia.org/wiki/File:King%27s_graph_with_white_king.svg.

[20] S. Gong, R. Chen, Z. Xie och X. Li, ”Major Approaches to Robot Environmental Modeling,” i *2021 6th International Symposium on Computer and Information Processing Technology (ISCIPT)*, Changsha, Kina, 11–13 juni. ss. 218–222. [Online]. Tillgänglig: 10.1109/ISCIPT53667.2021.00051, Hämtad: 2026-05-05.

[21] J. Sun och X. Li, ”Indoor evacuation routes planning with a grid graph-based model,” i *2011 19th International Conference on Geoinformatics*, Shanghai, Kina, 2011, ss. 1–4. [Online]. Tillgänglig: 10.1109/GeoInformatics.2011.5980680, Hämtad: 2026-05-05.

[22] P. Guturu och R. Dantu, ”An Impatient Evolutionary Algorithm With Probabilistic Tabu Search for Unified Solution of Some NP-Hard Problems in Graph and Set Theory via Clique Finding,” *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, årg. 38, nr 1, s. 253–257, Jun. 2008. DOI: 10.1109/TSMCB.2008.915645.

[23] S.-H. Chan, R. Stern, A. Felner och S. Koenig, ”Greedy Priority-Based Search for Suboptimal Multi-Agent Path Finding,” i *Proceedings of the 16th International Symposium on Combinatorial Search (SoCS)*, Prag, Tjeckien, 2023, ss. 11–19. [Online]. Tillgänglig: 10.1609/socs.v16i1.27278, Hämtad: 2026-05-05.

[24] A. Bogatarkan, ”Flexible and Explainable Solutions for Multi-Agent Path Finding Problems,” i *Proceedings of the 37th International Conference on Logic Programming (ICLP)*, Istanbul, Turkiet, 2026, ss. 41034 – 41035. [Online]. Tillgänglig: 10.4204/EPTCS.345.40, Hämtad: 2026-05-05.

[25] A. Bogatarkan, ”Flexible and Explainable Solutions for Multi-Agent Path Finding Problems,” i *Proceedings of the 37th International Conference on Logic Programming (ICLP)*, Porto, Portugal, 2021, ss. 240–247. [Online]. Tillgänglig: 10.4204/EPTCS.345.40, Hämtad: 2026-05-05.

[26] W. Ye, H. Wang och W. Chen, ”Path Coordination for Robust, Fast, and Scalable Multi-Agent Path Finding Under Unforeseen Delays,” *IEEE Transactions on Automation Science and Engineering*, årg. 22, 2025, ss. 22396–22409, Jan. 2025, doi: 10.1109/TASE.2025.3615989.

[27] Z. Chen, D. Harabor, J. Li och P. J. Stuckey, ”Traffic Flow Optimisation for Lifelong Multi-Agent Path Finding (Extended Abstract),” i *Proceedings of the 17th International Symposium on Combinatorial Search (SoCS)*, Alberta, Kanada, 2024. ss. 265–266. [Online]. Tillgänglig: 10.1609/socs.v17i1.31573, Hämtad: 2026-05-05.

[28] T. Geft, ”Fine-Grained Complexity Analysis of Multi-Agent Path Finding on 2D Grids,” i *Proceedings of the 16th International Symposium on Combinatorial Search (SoCS)*, Prag, Tjeckien, 2023, ss. 20–28. [Online]. Tillgänglig: 10.1609/socs.v16i1.27279, Hämtad: 2026-05-05.

[29] Z. Ma, Y. Luo och H. Ma, ”Distributed Heuristic Multi-Agent Path Finding with Communication,” i *Proceedings of the 2021 IEEE International Conference on Robotics and Automation (ICRA)*, Xi’an, Kina, 2021, ss. 8699–8705. [Online]. Tillgänglig: 10.1109/ICRA48506.2021.9560748, Hämtad: 2026-05-05.

[30] J. X. m.fl., ”Improved Communication and Collision-Avoidance in Dynamic Multi-Agent Path Finding,” i *Proceedings of the 2024 International Joint Conference on Neural Networks (IJCNN)*, Yokohama, Japan, 2024, ss. 1–9. [Online]. Tillgänglig: 10.1109/IJCNN60899.2024.10651091, Hämtad: 2026-05-05.

[31] S. Kim, K. Gil och J. Shin, ”Optimization-Based Path Planning With Artificial Potential Function,” *IEEE Access*, årg. 13, ss. 141717–141731, Jan. 2025, doi: 10.1109/ACCESS.2025.3597311.

[32] B. Sun och Z. Lv, ”Multi-AUV Dynamic Cooperative Path Planning with Hybrid Particle Swarm and Dynamic Window Algorithm in Three-Dimensional Terrain and Ocean Current Environment,” *Biomimetics*, årg. 10, nr. 8, art. nr. 536, Aug. 2025, doi: 10.3390/biomimetics10080536.

[33] J. Liang, S. Koenig och F. Fioretto, ”Discrete-Guided Diffusion for Scalable and Safe Multi-Robot Motion Planning,” i *Proceedings of the 40th AAAI Con-*

ference on Artificial Intelligence, Singapore, 2026, ss. 23417–23424. [Online]. Tillgänglig: 10.1609/aaai.v40i28.39512, Hämtad: 2026-05-05.

[34] S. Zhou, S. Zhao och Z. Ren, "Loosely Synchronized Rule-Based Planning for Multi-Agent Path Finding with Asynchronous Actions," i *Proceedings of the 39th AAAI Conference on Artificial Intelligence*, Philadelphia, USA, 2025, ss. 14763–14770. [Online]. Tillgänglig: 10.1609/aaai.v39i14.33618, Hämtad: 2026-05-05.

[35] A. Combrink, S. F. Roselli och M. Fabian, "Prioritized Planning for Continuous-time Lifelong Multi-agent Pathfinding," i *Proceedings of the 11th International Conference on Control, Decision and Information Technologies (CoDIT)*, Split, Kroatien, 2025, ss. 1454–1459. [Online]. Tillgänglig: 10.1109/CoDIT66093.2025.11321711, Hämtad: 2026-05-05.

[36] N. Zavarzin och K. Yakovlev, "Empirical Evaluation of Motion Primitives in Multi-Agent Path Finding with Kinodynamic Constraints," i *Proceedings of the 23rd International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, Sirius, Ryssland, 2025, ss. 467–478. [Online]. Tillgänglig: 10.1007/978-3-032-13612-$1_4$1, Hämtad: 2026-05-05.

[37] F. K. S. C. m.fl., "Multi-Agent Pathfinding for Deadlock Avoidance on Rotational Movements," i *Proceedings of the 17th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, Singapore, Singapore, 2022, ss. 765–770. [Online]. Tillgänglig: 10.1109/ICARCV57592.2022.10004303, Hämtad: 2026-05-05.

[38] Y. Ding, J. Li och K. Tei, "Multi-Agent Path Finding with Priority Transfer Accompanied with Moving Obstacles," i *Proceedings of the 2022 IEEE 20th Jubilee International Symposium on Intelligent Systems and Informatics (SISY)*, Subotica, Serbien, 2022, ss. 103–108. [Online]. Tillgänglig: 10.1109/SISY56759.2022.10036293, Hämtad: 2026-05-05.

[39] A. Berndt, N. van Duijkeren, L. Palmieri, A. Kleiner och T. Keviczky, "Receding Horizon Re-Ordering of Multi-Agent Execution Schedules," *IEEE Transactions on Robotics*, årg. 40, s. 1356–1372, Jan. 2024, DOI: 10.1109/TRO.2023.3344051.

[40] J. Fan, J. Ruan, Y. Liang och L. Tang, "A PSO solution for pursuit-evasion problem of randomly mobile agents," i *Proceedings of the 2009 Chinese Control and Decision Conference (CCDC)*, Guilin, Kina, 2009, ss. 4032–4035. [Online]. Tillgänglig: 10.1109/CCDC.2009.5192696, Hämtad: 2026-05-04.

[41] X. Dong, H. Zhang och Z. Ming, "Adaptive Optimal Control via Q-Learning for Multi-Agent Pursuit-Evasion Games," *IEEE Transactions on Circuits and Systems II: Express Briefs*, årg. 71, nr 6, s. 3056–1372, Jan. 2024. DOI: 10.1109/TCSII.2024.3354120.

[42] J. Zhang, H. Zhang och W. Zhao, "Distributed Optimal Pursuit-Evasion Strategy of Multiple-Pursuer Single-Evader Game via Reinforcement Learning," i *Proceedings of the IECON 2023- 49th Annual Conference of the IEEE In-*

dustrial Electronics Society, Singapore, Singapore, 2023, ss. 1–6. [Online]. Tillgänglig: 10.1109/IECON51785.2023.10312136, Hämtad: 2026-05-04.

[43] J. Zheng, H. Yu, W. Liang och P. Zeng, "Probabilistic strategies to coordinate multiple robotic pursuers in pursuit-evasion games," i *Proceedings of the 2007 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, Sanya, Kina, 2007, ss. 559–564. [Online]. Tillgänglig: 10.1109/ROBIO.2007.4522223, Hämtad: 2026-05-04.

[44] J. P. Hespanha, H. J. Kim och S. Sastry, "Multiple-agent probabilistic pursuit-evasion games," i *Proceedings of the 38th IEEE Conference on Decision and Control (CDC)*, Phoenix, USA, 1999, ss. 2432–2437. [Online]. Tillgänglig: 10.1109/CDC.1999.831290, Hämtad: 2026-05-04.

[45] V. G. Lopez, F. L. Lewis, Y. Wan, E. N. Sanchez och L. Fan, "Solutions for Multiagent Pursuit-Evasion Games on Communication Graphs: Finite-Time Capture and Asymptotic Behaviors," *IEEE Transactions on Automatic Control*, årg. 65, nr 5, ss. 1911–1923, Jul. 2020, doi: 10.1109/TAC.2019.2926554.

[46] A. Kolling och S. Carpin, "Pursuit-Evasion on Trees by Robot Teams," *IEEE Transactions on Robotics*, årg. 26, nr 1, 2010, ss. 32–47, Dec. 2010, doi: 10.1109/TRO.2009.2035737.

A Bilaga 1

Algorithm 4 Dijkstras Algoritm

Input: *start* the startnode.

Output: A map from each goal node to nodes ending in the startnode.

```
1:  function DIJKSTRAS ALGORITM(start)
2:      OpenSet ← {start}
3:      CameFrom ← an empty map
4:      GScore ← map all nodes with default value of ∞
5:      GScore[start] ← 0
6:      while OpenSet is not ∅ do
7:          current ← the node in OpenSet having the lowest GScore value
8:          for each neighbour of current do
9:              TentativeGScore ← GScore[neighbour] + EdgeCost
10:             if TentativeGScore < GScore[neighbour] then
11:                 CameFrom[neighbour] ← current
12:                 GScore[neighbour] ← TentativeGScore
13:                 if neighbour not in OpenSet then
14:                     OpenSet.add(neighbour)
15:     return CameFrom
```

B Bilaga 2

Algorithm 5 A stjärna

Input: *start*: the startnode, *goal*: the goalnode, *blocked*: a list of blocked nodes
Output: the shortest path from the startnode to the goalnode

```
1: function A*(start, goal, blocked)
2:     if blocked is None then blocked ← ∅
3:     OpenSet ← {start}
4:     CameFrom ← an empty map
5:     GScore ← a map with default value of ∞
6:     GScore[start] ← 0
7:     FScore ← map with default value of Infinity
8:     FScore[start] ← HEURISTIC(start, goal)
9:     while OpenSet is not ∅ do
10:        current ← the node in OpenSet having the lowest FScore value
11:        if current = goal then
12:            return ReconstructPath(CameFrom, current)
13:        for each neighbour of current do
14:            if neighbor is not blocked or current != start then
15:                TentativeGScore ← GScore[neighbour] + EdgeCost
16:                if TentativeGScore < GScore[neighbour] then
17:                    CameFrom[neighbour] ← current
18:                    GScore[neighbour] ← TentativeGScore
19:                    FScore[neighbour] ← TentativGScore+
   HEURISTIC(neighbour, goal)
20:                    if neighbour not in OpenSet then
21:                        add neighbour to OpenSet
22:    return an empty list
```

Algorithm 6 ReconstructPath

Input: $CameFrom$: a map between nodes directed towards the startnode, $Current$: the node to reconstruct a path to

Output: A list from the startnode to the endnode

```
1: function ReconstructPath(CameFrom, Current)
2:     TotalPath ← [Current]
3:     while Current in CameFrom do
4:         Current ← CameFrom[Current]
5:         Preprend Current to TotalPath
6:     return TotalPath
```

C Bilaga 3

Tabell C.1: Omringning av inkräktare, jämförelse mellan två metoder. Tid, A* och BFS är medelvärdet per iteration.

Scenario	Metod	Tid (s)	A* (st.)	BFS (st.)	Iterationer (st.)
AMR 3, Graf 9x5	Blockering	0.0002208091	3	1	11
	Blockering	0.0002535143	3	1	7
	Blockering	0.0003068429	3	1	14
	Blockering	0.0000856143	0	3.9	14
	Avgränsning	0.0002108500	0	6.2	16
	Avgränsning	0.0002153579	0	4.7	19
AMR 15, Graf 18x10	Blockering	0.0108366000	15	1	2
	Blockering	0.0247605500	15	1	4
	Blockering	0.0087766526	15	1	19
	Avgränsning	0.0010305000	0	24.1	18
	Avgränsning	0.0036626133	0	25.1	30
	Avgränsning	0.0013888500	0	23.0	4
AMR 30, Graf 90x10	Blockering	0.6972494050	30	1	20
	Blockering	0.3994482923	30	1	13
	Blockering	0.2201218000	30	1	18
	Avgränsning	0.0132959341	0	51.1	88
	Avgränsning	0.0170124333	0	50.0	42
	Avgränsning	0.0215564500	0	51.2	126
AMR 10, Graf 90x10	Blockering	0.1209678000	10	1	48
	Blockering	0.0733786500	10	1	16
	Blockering	0.1126940782	10	1	55
	Blockering	0.0077666061	0	16.3	99
	Avgränsning	0.0066504638	0	16.2	47
	Avgränsning	0.0076439270	0	17.3	137
AMR 100, Graf 90x10	Blockering	1.2825954000	100	1	2
	Blockering	1.0838606500	100	1	2
	Blockering	1.5116666437	100	1	16
	Avgränsning	0.0348397617	0	154.6	47
	Avgränsning	0.0344196143	0	148.4	42
	Avgränsning	0.0375024000	0	160.0	1

D Bilaga 4

Algorithm 7 LCRA*

Input: *agents*: all agents in the system, *intruder*: the intruder in the system
Output: Agents allowed to move

```
1: procedure LCRA*(agents, intruder)
2:     ResetFrameState()
3:     MarkOccupiedNodes(agents, intruder)
4:     PO ← ComputeBlockingRelations(agents)
5:     allowed ← all agents initially marked as not allowed to move
6:     for each agent in increasing priority order do
7:         curr ← agent's current node
8:         next ← agent's next planned node
9:         if MeetingCollision(curr, next) then
10:            Replan the lower–priority agent
11:            continue
12:        if ViolatesSafetyRules(next) then continue
13:        Mark agent as allowed to move
14:        Reserve transition curr → next
15:        allowed ← ExpandPriorityChain(agent, PO, allowed)
16:    allowed ← ResolveCycles(allowed, agents)
17:    return allowed
```

Algorithm 8 MarkOccupiedNodes

Input: agents: all agents in the system, intruder: the intruder in the system
Output: make the node unavailable for this iteration

```
1: procedure MarkOccupiedNodes(agents, intruder)
2:     if intruder is active then
3:         Mark intruder's node as occupied
4:     for each agent do
5:         Mark the agent's current node as occupied
```

Algorithm 9 ExpandPriorityChain

Input: agent: all agents in the system, PO: tha partial order, allowed: a map between each agent and a bool if they can go or not
Output: Change in state of allowed

```
1: procedure ExpandPriorityChain(agent, PO, allowed)
2:     if agent already visited then
3:         return allowed
4:     Mark agent as allowed
5:     if agent blocks another agent in PO then
6:         child ← dependent agent
7:         return ExpandPriorityChain(child, PO, allowed)
8:     return allowed
```

Algorithm 10 MeetingCollision

Input: curr: node id of an agents current position, next: node id of an agents next position in the path
Output: Bool determining of there is an switch conflict or not

```
1: function MeetingCollision(curr, next)
2:     for each other agent do
3:         if curr = their next and next = their curr then
4:             return true
5:     return false
```

Algorithm 11 ComputeBlockingRelations

Input: agents: all the agents in the system
Output: the partial ordering between the agents

```
1: function ComputeBlockingRelations(agents)
2:     PO ← ∅
3:     for each agent do
4:         next ← agent's next node
5:         if another agent occupies next then
6:             Add dependency (blocker → blocked) to PO
7:     return PO
```

Algorithm 12 ResolveCycles

Input: allowed: a map between agents and a bool if they are allowed to move, agents: all agents in the system.
Output: an updated map of allowed agents

```
1: function ResolveCycles(allowed, agents)
2:     Identify all agents that are stopped but intend to move
3:     Construct a dependency graph based on their next nodes
4:     for each such agent do
5:         Perform DFS to detect a cycle
6:         if cycle is found then
7:             Mark all agents in the cycle as allowed
8:     return allowed
```