Förstudie om reglerteknik för lagerrobot

Linus Funquist, Lisa Ståhl

2025-04-07

Version 0.2

Status

Granskad	Lisa Ståhl	2025-04-07
Godkänd	Namn	2025-xx-xx

Beställare:
Mattias Krysander, Linköpings universitet
Telefon: +46 13282198
E-post: mattias.krysander@liu.se

Handledare:
Theodor Lindberg, Linköpings universitet
E-post: theodor.lindberg@liu.se

Projektdeltagare

Namn	**Ansvar**	**E-post**
Linus Funquist		linfu930@student.liu.se
Ebba Lundberg	Dokumentansvarig	ebblu474@student.liu.se
Andreas Nordström	Projektledare	andno7733@student.liu.se
Sigge Rystedt		sigry751@student.liu.se
Ida Sonesson	Dokumentansvarig	idaso956@student.liu.se
Lisa Ståhl	Designansvarig	lisst342@student.liu.se

INNEHÅLL

DOKUMENTHISTORIK

Version	Datum	Utförda ändringar	Utförda av	Granskad
0.1	2025-01-29	Första utkast	LF, EL, AN, SR, IS, LS	LF, EL, AN, SR, IS, LS
0.2	2025-01-31	Andra utkast	LF, EL, AN, SR, IS, LS	LF, EL, AN, SR, IS, LS
1.0	2025-02-03	Första version	LF	LF, EL

1 INLEDNING

Detta dokument är en förstudie i kursen TSEA56, Elektronik kandidatprojekt och kommer att handla om att bygga och programmera en lagerrobot som autonomt ska kunna ta sig genom ett lager och hämta upp lagervaror för att sedan lämna dem på avsedd plats. För att roboten ska kunna hitta vägen genom lagret kommer den behöva följa en tejp. I denna förstudie kommer den reglerteknik som krävs för detta projekt att undersökas samt de sökalgoritmer som krävs för att beräkna vägen genom lagret.

1.1 Syfte

Syftet med denna förstudie är att undersöka vilken reglerteknik som krävs för att lagerroboten ska kunna utföra sina uppdrag och hur de ska implementeras samt hitta metoder för att beräkna kortaste vägen genom lagret.

2 PROBLEMFORMULERING

Frågeställningarna nedan är det som ämnas besvaras i förstudien:

- Vilka övergripande styrmoder är nödvändiga för att roboten ska kunna utföra sitt uppdrag?
- Hur kan man reglera roboten i de olika styrmoderna?
- Hur koordineras servona i en arm för att kunna styra gripklon till en bestämd position med en mjuk rörelse?

3 METOD

Frågeställningarna besvaras med hjälp av underlag från vetenskapliga texter.

4 SPÅRFÖLJNING

För att roboten ska kunna följa tejpen på ett stabilt sätt, utan att till exempel oscillera fram och tillbaka, behöver reglering med återkoppling användas. Det finns olika typer av reglering med återkoppling som används i olika fall [1].

4.1 Val av reglering för spårföljning

I PID-reglering (Proportional-Integral-Derivative) ges insignalen $u(t)$ baserat på avvikelsen $e(t)$ mellan önskad utsignal $r(t)$ och verklig utsignal $y(t)$, alltså $e(t) = r(t) - y(t)$ [2], även kallat återkoppling. Formeln för PID-reglering ser ut som följande:

$$u(t) = K_P e(t) + K_I \int_0^t e(\tau)d\tau + K_D \frac{d}{dt} e(t)$$

De olika termerna i ekvationen påverkar regleringen på olika sätt, vilket därmed kan styras genom att ange olika värden på K-variablerna. Den första termen $K_P e(t)$ kallas för den proportionella delen och är, precis som det låter,

proportionell mot felet $e(t)$. Därmed ger ett större fel en större justering vilket gör att den kan reagera snabbt. Det går dock inte att enbart använda sig av proportionell reglering, då det kan resultera i ett stationärt fel, det vill säga att algoritmen låser in sig på ett värde som skiljer sig från det önskade värdet. Den andra termen $K_I \int_0^t e(\tau)d\tau$ kallas för den integrerande delen. Den tar i stället hänsyn till tidigare fel och kan på så sätt eliminera det stationära felet. Den tredje termen $K_D \frac{d}{dt} e(t)$ tar, med hjälp av en deriverande faktor, hänsyn till hur snabbt avvikelsen förändras och kan på så sätt dämpa snabba variationer.

Algorithm 1 Pseudokod Regleralgoritm

Input: e(t) ▷ Felsignal
Output: u(t) ▷ Utsignal

Initiera variabler:
förra _felet = 0
K_P = proportionell konstant ▷ Ställ in konstanter (bestäms via testning)
K_D = deriverande konstant
K_I = integrerande konstant
Δt = tiden sedan förra regleringen

procedure PID_REGULATOR(e(t))
proportionell_term = $K_P * e(t)$
deriverande_term = $K_D * (\frac{e(t) - förra\_felet}{\Delta t})$ ▷ Enkel approximation av derivatan
integrerande_term = $K_I * Integralapprox.$ ▷ Ingen specifik approximation implementeras här

u(t) = proportionell_term + deriverande_term + integrerande_term
förra_felet = e(t)
return u(t) ▷ Returnerar kontrollsignalen
end procedure

5 BANA OCH RÖRELSEPLANERING

Innan roboten börjar åka kommer dimensioner på lagermiljön samt var varorna finns att anges, därefter ska roboten i autonomt läge beräkna den kortaste vägen genom lagret och sedan beräkna ny väg om hinder påträffas. I fallet där det finns fler än en vara att hämta upp kan roboten antingen åka och hämta varorna för sig eller hämta alla i samma körning och placera dem i någon form av korg som sitter på robotplattformen. Detta kommer beslutas om i ett senare skede och därför utreds här den kortaste vägen i båda fallen.

5.1 Algoritm för att hämta varorna en och en

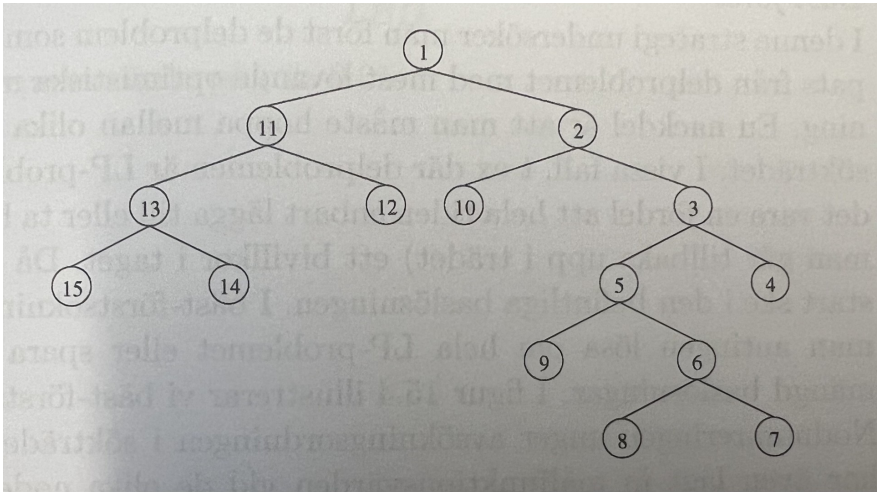
I fallet att varorna ska hämtas var för sig behöver snabbaste vägen till nästa vara beräknas från robotens startposition, vid varuavlämningen i ett av lagrens hörn, alternativt vid påträffat hinder.

Ett sätt att gå tillväga är att ange alla korsningar som noder och vägen mellan dem som kanter där alla vägar mel-

lan två närliggande noder kostar lika mycket. Avståndet mellan en nod i en fyrvägskorsning och en nod där vara finns är kortare men lika lång från båda hållen vilket innebär att man kan bortse från längdskillnaden. Dessa noder kan sedan ritas upp som ett träd och därmed kan trädsökning användas. Det finns många olika sätt att genomföra en trädsökning som är olika effektiva beroende på hur trädet ser ut och vad syftet med sökningen är [3].

5.1.1 *Djupet-Först-Sökning*

Djupet-Först-Sökning börjar med att först söka i en gren så långt som möjligt (exempelvis alltid till höger) tills en lövnod nås. Om målet inte hittats stegar den tillbaka och fortsätter utforska nästa möjliga gren på samma sätt [3]. Detta säkerställer att alla möjliga vägar genomsökts tills målet hittats, se figur 1.



Figur 1: Träddiagram Djupet-Först-Sökning [3].

Denna metod fungerar bra då en väg behöver hittas snabbt men som då inte nödvändigtvis är den kortaste vägen. I figur 2 visas pseudokoden för Djupet-Först-Sökning.

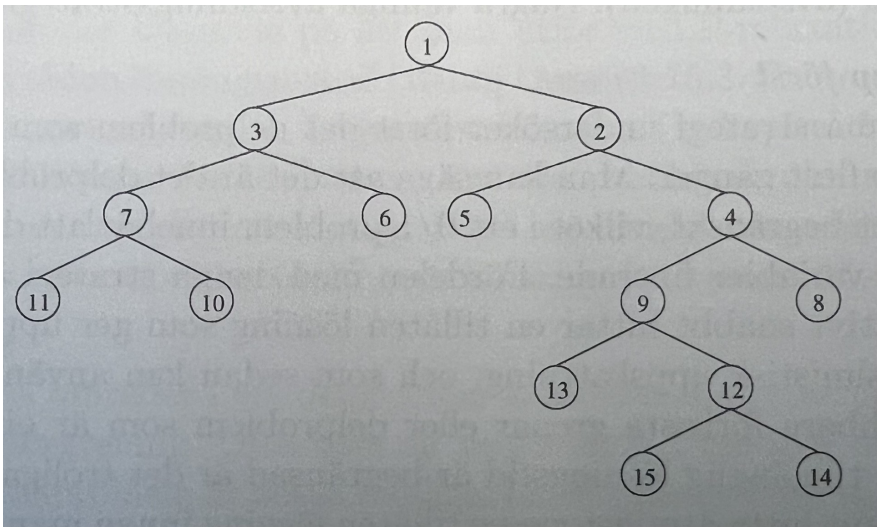
Algoritm 2: En DFS algoritm.
Input: En GrannListad graf $U = \{n_1, n_2, \ldots, n_n\}$ av noder, en sökt nod *soughtNode*
Output: Den kortaste vägen A till mål noden. (Om det inte finns någon väg till mål noden så, $A = \emptyset$).

```
DFS(U,soughtNode)
(1)     Stack pathTo
(2)     node = U.root
(3)     while endNotReached
(4)         if node == soughtNode
(5)             return pathTo
(6)         else
(7)             foreach node.neighbours
(8)                 if !neighbour.visited
(9)                     neighbour.visited = true
(10)                    path.add(neighbour)
(11)                    neighbourFound = true
(12)                    break
(13)            if neighbourFound != true
(14)                node = path.pop()
(15)        if node == root
(16)            endNotReached = false
(17)    return ∅
```

Figur 2: Pseudokod Djupet-Först-Sökning [4].

5.1.2 *Bredden-Först-Sökning*

Bredden-Först-Sökning är en algoritm som används för att hitta kortaste vägen från en nod till en annan genom att först undersöka alla delproblem på samma nivå i trädet, för att sedan ta ett steg ner enligt figur 3 [3].



Figur 3: Träddiagram bredden först [3].

I figur 2 visas pseudokoden för Bredden-Först-Sökning.

Algoritm 1: En BFS algoritm.
Input: En Grannlistad graf $U = \{n_1, n_2, \ldots, n_n\}$ av noder, en sökt nod $soughtNode$
Output: Den kortaste vägen A till målnoden. (Om det inte finns någon väg till målnoden så, $A = \emptyset$).

```
BFS(U,soughtNode)
(1)     Queue queue
(2)     queue.add(U.root)
(3)     while endNotReached
(4)         node = queue.pop() if node == soughtNode
(5)             return (pathTo(node))
(6)         else
(7)             foreach node.neighbours
(8)                 if !neighbour.visited
(9)                     neighbour.visited = true
(10)                    queue.add(neighbour)
(11)        if queue.size() == 0
(12)            endNotReached = false
(13)    return ∅
```

Figur 4: Pseudokod Bredden-Först-Sökning [4].

5.2 Algoritm för att hämta alla varor i en färd

Ska alla varor hämtas i en tur går problemet istället att formulera som ett handelsresandeproblem. Då lagret beskrivs som ett rutnät där alla avstånd är lika långa blir det en väldigt förenklad variant av den klassiska modellen.

Om målet är att alltid hitta den optimala vägen mellan alla noder så är det säkraste sättet att studera alla möjliga rutter och hitta den med kortast längd [5]. Detta är väldigt tidskrävande och för större system blir det snabbt väldigt beräkningstungt. Därför kan det vara relevant att istället studera en girig algoritm, eller annan heuristik.

Den lättaste giriga heuristiken att tillämpa kallas nearest neighbor [5]. Den går ut på att roboten alltid väljer att ta vägen till närmaste noden som inte redan besökts. När alla noder har besökts återvänder roboten till startpunkten. Denna algoritm är väldigt billig, men oftast fås inte den optimala rutten. Däremot är rutten oftast mycket bättre än godtycklig slumpad rutt.

En annan relevant algoritm är Held-Karp-algoritmen [6]. Som när alla möjliga rutter jämförs fås här den optimala lösningen, men den är betydligt mer effektiv för medelstora system. Först beräknas avstånden mellan alla punkter som vill besökas. Sedan gås systematiskt alla delmängder av noder igenom. Först beräknas totala längden mellan alla

kombinationer av tre noder, sedan läggs en fjärde nod till, och så fortsätter det tills alla noder är tillagda. Då är bästa rutten hittad och problemet är löst. Som märks av processen blir det snabbt väldigt många beräkningar om det finns många noder att besöka, men i fallen med medelstora system (som detta projekt troligen är) bör metoden vara snabb nog.

6 STYRNING AV ROBOTARMEN

I projektet kommer en robotarm av typen PhantomX reactor användas. Med rotation av basen och "handleden" inräknat har den 5 frihetsgrader, själva armen har då tre vridningspunkter. Servona som sitter i armen är av typen Dynamixel AX-12A. För att beräkna vilka vinklar servona ska ställas in på används invers kinematik. Den typen av problem kan i vissa fall lösas analytiskt, men ibland måste numeriska metoder användas [7].

För att lösa invers kinematik måste armens modelleras och det brukar göras med Denavit-Hartenberg (D-H) parametrar [8]. Parametrarna beskriver varje länks egenskaper, vilket tillsammans ger en bra beskrivning av hela armen. De parametrar som ingår beskrivs nedan:

- a_i : Länklängd, längden av armen efter vridningspunkten
- α_i : Länkvridning, skillnaden i två på varandra följande länkars vridningsaxel.
- d_i : Länkförskjutning
- θ_i : Länkvinkel, specifika vinkeln servon är inställd på, ändras beroende på armens specifika konfiguration.

I de flesta fall är en analytisk lösning bäst, om det går att hitta en. Beroende på hur komplicerad robotarmen är och framförallt hur många frihetsgrader den har, kan en analytisk lösning vara svår att härleda. Om armen skulle vara redundant, vilket betyder att den har för mångafrihetsgrader, kan en analytisk lösning också vara extra svår att tillämpa. Detta då det kan finnas många olika sätt, i vissa fall oändligt många, att orientera armen så att gripklon når en specifik position.

En annan lösning som bör vara mycket lättare att implementera, men som inte är lika allmän, hade varit att hårdkoda in servonas positioner. Om avståndet från mittlinjen till objektet som ska plockas upp är känt och konstant, så borde armen kunna göra exakt samma rörelse för att plocka upp föremålet varje gång. Den enda riktiga nackdelen med detta är om det finns osäkerheter i robotplattformens position när den ställer sig för upplockning, men om den styrningen är bra nog är det här ett alternativ.

En annan utmaning är att få armen att styras med en mjuk rörelse. Servona som används i projektet, Dynamixel AX-12A, har mycket funktionalitet både för att styra och läsa av relevant mätdata [9]. Just för att få en mjuk och enhetlig kontroll av armen går det att ställa in servons hastighet. För en mjuk och koordinerad rörelse går det att bestämma servonas hastigheter utifrån hur långt den ska förflyttas. På så sätt går det att få så att alla servon börjar röra sig samtidigt, och når sin önskade position samtidigt. Till exempel går det, om en maximal vinkelhastighet bestäms, att sätta hastigheten för den servon som ska förflyttas längst till den hastigheten, och sedan sätta hastigheterna på resterande servon till den hastigheten multiplicerat med en kvot av servonas respektive kommande förflyttning.

7 DISKUSSION OCH SLUTSATSER

7.1 Nödvändiga styrmoder

De styrmoder som behövs för att roboten ska kunna utföra sina uppdrag är spårföljning, så att roboten följer tejpen och vägnavigering så att roboten hittar i lagret.

7.2 Reglering av styrmoder

I regleringen för tejpföljning kommer enbart PD (Proportionell respektive Deriverande reglering) att behöva användas. Felet för tejpföljning fås från tejpsensorn på bilen, och är ett mått på hur långt från mitten av bilen tejpen är. Ett fel e(t) = 0 innebär att tejpen ligger mitt under bilen, och rimligtvis önskas då att bilen fortsätter köra rakt. u(t) kommer att representera hur mycket åt något håll bilen måste svänga, och om insignalen (felet) e(t) = 0 kommer självklart utsignalen u(t) = 0. I detta fall finns då inget kvarstående reglerfel, och en I del behövs därför inte [1].

Däremot behövs en D del i algoritmen, då den används för att minska oscillationer i regleringen. Oscillationer är ofta ett stort problem om enbart en P del används (framförallt vid höga värden på K_P, dvs snabb reglering) så en deriverande del är här väldigt relevant, hur den hjälper i detta fallet är väldigt tydligt. Derivatan av felet är hur fort felet ändras, och det den termen gör här är att göra svängningen mindre aggressiv om den redan är på väg åt rätt håll, och börja svänga ännu mer om bilen är på väg åt fel håll. Ekvationen som kommer användas blir då den nedan, konstanterna tas lättast fram genom testning.

$$u(t) = K_P e(t) + K_D \frac{d}{dt} e(t)$$

Vägnavigeringen regleras med hjälp av input i form av karta och position på varor samt kostnader för olika vägar.

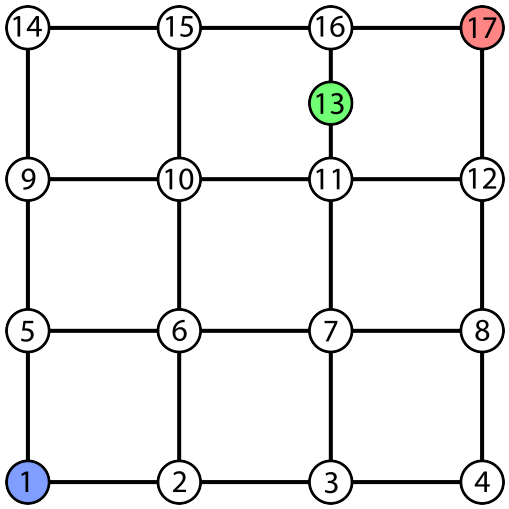
7.3 Styrning av robotarm

För styrning av robotarmen är det lättaste, och helt klart mest realistiska, att hårdkoda in de servovärden som krävs för att plocka upp föremålet. För det som robotarmen skall användas till i projektet behövs inte mycket mer än så, iallafall inte för det autonoma fallet. Om resten av robotens reglering görs tillräckligt bra kommer avståndet mellan armen och föremålet den ska plocka upp att vara samma varje gång, så att hårdkoda in armens rörelse bör vara en bra lösning. Detta görs lättast genom att ställa upp roboten i positionen där objektet plockas, ställa armen i en position där föremålet nås, och sedan läsa av all relevant information om servonas inställning. Detta går enkelt att göra då servona som används i armen har många inbyggda sensorer som kan läsas av [9].

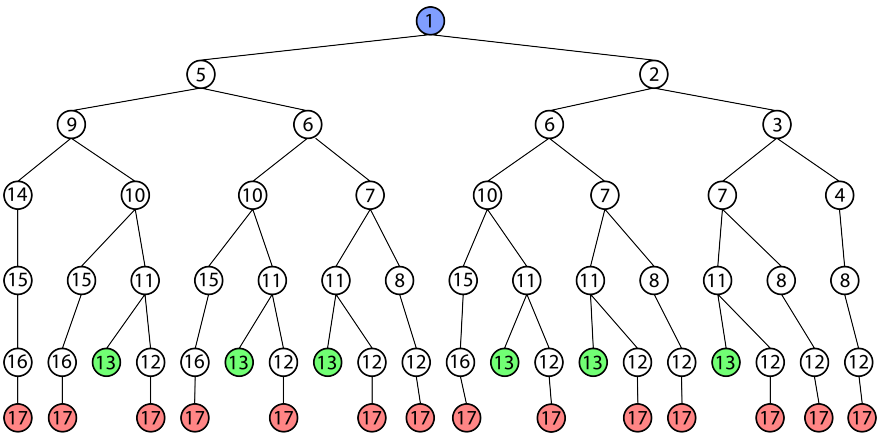
Beroende på hur det implementeras kan manuell styrning av robotarmen bli lite knepigare. En implementering, som nog är lättast för användaren, är om robotarmen tar emot kommandon såsom framåt, bakåt, vänster och liknande. Detta lär dock vara väldigt svårt att implementera, då beräkningarna för hur servona ska röras lär behöva göras med invers kinematik. Detta kan, vilket konstaterades tidigare, vara väldigt komplicerat, framför allt för en robotarm med många leder. Den mer realistiska, men kanske inte riktigt lika användarvänliga lösningen är att i manuellt läge styra en servo i taget. Detta ger även en större frihet för användaren, men det tar längre tid att ställa in armen i önskad position och är allmänt inte lika intuitivt.

7.4 Bana och rörelseplanering

När position för varan matas in av användaren placeras en nod där avsticket till varan finns. Då roboten väl kommer fram till rätt kant känner den själv av vilken sida om tejpen varan är på. I fallet där varorna hämtas en och en är användningen av Bredden-Först-Sökning mest effektiv för att hitta denna nod då den hittar kortaste vägen och inte bara första bästa som Djupet-Först-Sökning gör. Eftersom lagret är utformat som ett rutnät behöver inte alla vägar ut ur en nod tillåtas för att hitta den snabbaste vägen. Om startpunkten antas vara i nedre vänstra hörnet enligt figur 5 behöver noderna exempelvis endast avsökas uppåt och till höger då ingen snabbare väg kommer kräva steg till vänster eller nedåt. Då fås träd enligt figur 6.

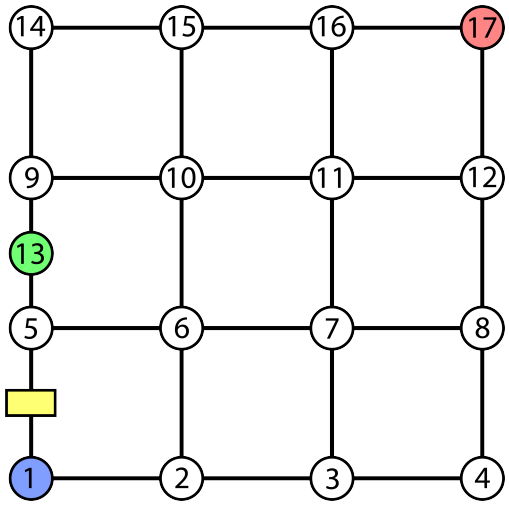


Figur 5: Exempel på lagermiljö med numrerade noder.



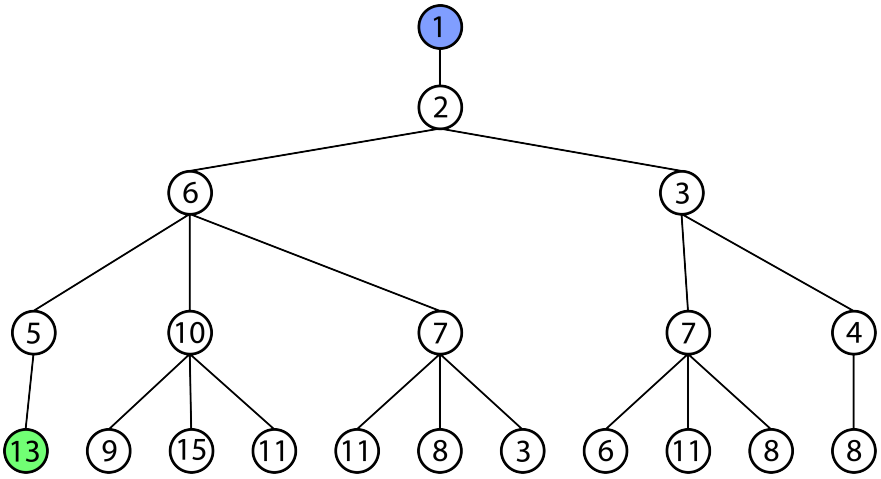
Figur 6: Träddiagram över noder i lagermiljö.

Detta tillsammans med Bredden-Först-Sökning fungerar utmärkt i detta fall, men i verkligheten kommer det även finnas hinder, vilket visas som gul rektangel i figur 7.



Figur 7: Exempel på lagermiljö med numrerade noder och hinder.

Vid påkommet hinder tar sig roboten tillbaka till närmaste nod, beräknar ny väg samt raderar vägen som hindret låg på. Detta innebär att snabbaste vägen kan kräva steg till vänster eller nedåt för att komma till varan. Därför måste efter påkommet hinder, enligt figur 8, alla vägar avsökas.



Figur 8: Träddiagram över lagermiljö med noder och hinder.

I fallet där alla varor hämtas i en körning finns det några alternativ som skulle vara möjliga att implementera bra. Av de idéer som diskuterades tidigare är den minst användbara nearest neighbor. Den är snabb att beräkna, men eftersom att optimala rutten inte är garanterad så är den tiden som sparas i beräkningen inte mer än tiden som tappas av att roboten åker en suboptimal väg. Däremot hade det nog inte, i de situationer som kommer dyka upp i projektet, varit

en helt orimlig strategi att beräkna alla möjliga kombinationer av upplockningsordningar. Om noderna blir många blir det snabbt väldigt många beräkningar, men även om det är 6 objekt som ska plockas upp så blir det endast 720 kombinationer att beräkna. Detta låter mycket, men till projektet används en Raspberry PI med 1.4 GHz klockfrekvens [10] (vilket betyder att den kan utföra 1.4 miljarder beräkningar per sekund), så även om det skulle bli många beräkningar att göra skulle den extra tiden det tar inte vara jämförbar med den extra tiden en längre väg skulle ta.

Den strategin som känns bäst för att lösa detta problem blir då Help-Karp algoritmen. Den beräknar alla möjliga kombinationer och ger därför alltid den optimala lösningen, men den gör det mer effektivt, framförallt eftersom att den slipper göra samma beräkning flera gånger. Även med denna metod blir det då, som metoden ovan, många beräkningar när antalet noder ökar. Help-Karp metoden är som sagt mer effektiv än den ovan, och blir därför en bra lösning.

8 REFERENSER

REFERENSER

[1] L. University. “Föreläsning 6 - AVR.” Tillgänglig: 24 februari 2025. (2025), URL: https://liuonline.sharepoint.com/sites/Lisam_TSEA56_2025VT_M8/CourseDocuments/Forms/AllItems.aspx?id=%2Fsites%2FLisam%5FTSEA56%5F2025VT%5FM8%2FCourseDocuments%2FProjektmodul%2FF%C3%B6rel%C3%A4sningar%2FF%C3%B66%2DAVR%2Epdf&parent=%2Fsites%2FLisam%5FTSEA56%5F2025VT%5FM8%2FCourseDocuments%2FProjektmodul%2FF%C3%B6rel%C3%A4sningar.

[2] T. Glad och L. Ljung, *Reglerteknik: Grundläggande teori*, 4. utg. Lund, Sweden: Studentlitteratur, 2003, ISBN: 978-91-44-02308-1.

[3] J. Lundgren, M. Rönnqvist och P. Värbrand, *Optimeringslära*, 2. utg. Lund, Sweden: Studentlitteratur, 2010, ISBN: 978-91-44-05512-9.

[4] J. Nordström. “Combinatorial Search.” Accessed: 2025-04-07. (2009), URL: https://www.csc.kth.se/utbildning/kth/kurser/DD2458/popup15/material/oldnotes/combsearch.F4.09.pdf.

[5] W3Schools, *Traveling Salesman Problem*, https://www.w3schools.com/dsa/dsa_ref_traveling_salesman.php, Accessed: 2025-02-24, n.d.

[6] CompGeek, *Held-Karp Algorithm for TSP*, https://compgeek.co.in/held-karp-algorithm-for-tsp/, Accessed: 2025-02-24, n.d.

[7] U. of Illinois - Motion Group, *Inverse Kinematics*, Accessed: 2025-02-22, 2024. URL: https://motion.cs.illinois.edu/RoboticSystems/InverseKinematics.html.

[8] T. Abaas, A. Khleif och M. Abbood, “Kinematics Analysis of 5 DOF Robotic Arm,” *Engineering and Technology Journal*, årg. 38, nr 3A, s. 412–422, 2020. DOI: 10.30684/etj.v38i3A.475. URL: https://etj.uotechnology.edu.iq/article_168857.html.

[9] ROBOTIS, *AX-12A e-Manual*, Accessed: 2025-02-23, n.d. URL: https://emanual.robotis.com/docs/en/dxl/ax/ax-12a/.

[10] R. P. Foundation, *Raspberry Pi 3 B+ Product Brief*, Accessed: 2025-04-07, 2018. URL: https://datasheets.raspberrypi.com/rpi3/raspberry-pi-3-b-plus-product-brief.pdf.

9 APPENDIX