CHALMERS

Optimering av Cellkonfiguration och Robotarmssekvensering för två Robotarmar

Kandidatarbete inom Signaler och system

Ulf Hjohlman
Yuli Hua
Christian Larsen
Simon Liljeqvist
Victor Nordh
Robert Nyquist

Institutionen för Signaler och system
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2015

Abstract

In a society where more and more of the production is becoming automated, the demand for robots is increasing. Major producers of robots have begun the development of two-armed robots which have the potential to work more effectively than two one-armed robots together, and shall also promote cooperation opportunities with humans. The process to produce efficient robot program, where two robots work together in the same cell are complex and time consuming, as a result of the additional degrees of freedom. The purpose of this project is to automatize a process for producing optimally coordinated robot programs and cell configurations. The project builds on previous work in the field, which have established a solution method that this project will continue to develop and automate. The optimization will be based on three aspects; component location in the cell, were the mounting surface should be placed, as well as sequencing of robotic arm movements. The problem is seen as a *Vehicle Routing Problem* which is modeled and solved using constraint programming. An Add-In for ABB RobotStudio has been designed so the user, based on the configured robot cell and given information regarding production, automatically can calculate robot programs for both robotic arms. The results are presented in the form of robot programs that can be simulated in RobotStudio. The method was evaluated by applying it to three test cases. The test cases demonstrate that there is potential in the method but the time it takes to perform the optimization increases rapidly with the number of components that are to be assembled. Robot programs that are generated are not collision-free as a consequence of the optimization not taking into account the dynamic obstacles robotic arms pose to each other.

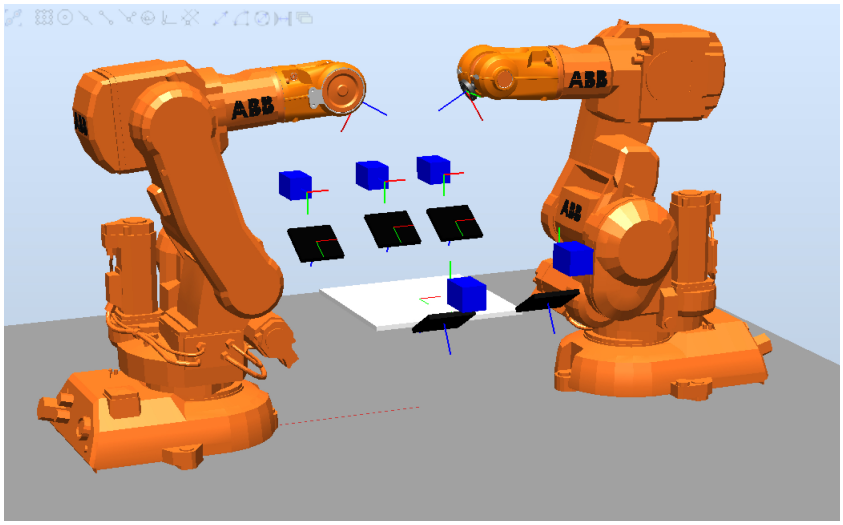
Sammanfattning

I ett samhälle där alltmer av produktionen blir automatiserad växer efterfrågan på robotar. Stora producenter av robotar har börjat utvecklingen av tvåarmade robotar som har potential att arbeta effektivare än vad två enarmade robotar skulle göra tillsammans och ska dessutom främja samarbetsmöjligheterna med människan. Processen att framställa effektiva robotprogram där två robotarmar arbetar tillsammans i samma cell blir komplex och tidskrävande pågrund av det ökade antalet frihetsgrader. Detta projekt har till syfte att automatisera en process för framtagning av optimalt koordinerade robotprogram och cellkonfigurationer. Projektet bygger vidare på tidigare arbeten inom området, där det har etablerats en lösningsmetod som i detta projekt ska utvecklas och automatiseras. Optimeringen kommer ske utifrån tre aspekter; komponenternas placering i cellen, vart monteringsytan ska placeras, samt sekvensering av robotarmarnas rörelser. Problemet ses som ett *Vehicle Routing Problem* och modelleras för att lösas med hjälp av villkorsprogrammering. Ett Add-In till ABB:s RobotStudio har konstruerats för att användaren utifrån en konfigurerad robotcell och angiven information kring produktionen automatiskt får framräknade robotprogram, för båda robotarmarna. Resultatet presenteras i form av robotprogram färdiga att simulera i RobotStudio. Metoden utvärderades genom att applicera programmet på tre testfall. Testfallen påvisar att det finns potential i metoden men tiden det tar att utföra optimeringen ökar snabbt med antalet komponenter som ska monteras. Robotprogrammen som genereras är inte kollisionsfria då optimeringen inte tar hänsyn till de dynamiska hindren robotarmarna utgör för varandra.

Innehåll

1 Introduktion

Industrins ständiga behov av att skära ner kostnader har lett till en ökad efterfrågan på användarvänliga och anpassningsbara robotar. Produktionen av tekniska produkter som till exempel mobiltelefoner och datorer blir allt större, vilket innebär en ökad mängd produktion där många små komponenter behövs sättas samman [1]. Fördelarna med att automatisera en produktionsprocess med robotar är bland annat den höga effektiviteten och den konstanta kvaliteten som levereras, oberoende av hur länge robotarna körs. För manuella produktioner kan både kvalitet och effektivitet försämras på grund av mänskliga faktorer [2]. I många fall är det dock mer lönsamt att ge instruktioner till en människa för hur produktionen ska göras, än att programmera en robot som ska utföra samma arbete. Idag ersätts därför sällan tillfälliga arbetsuppgifter av robotar på grund av robotarnas långa ställtid. Inom 10 år förutspås det att användningen av avancerade robotar kommer höja effektiviteten inom produktionen med upp till 30% i många industrier och sänka arbetskostnaderna med upp till 18% i flera stora produktionsländer, som Kina, USA och Tyskland [3].



Figur 1: Cellkonfiguration innehållande två IRB 140T-robotar, ett fixturplan (vit), fem tråg (svart) och fem kameror (blå).

En vanlig arbetsuppgift för robotarna i en robotcell är att hämta olika komponenter placerade i varsitt tråg, därefter montera ihop dem på en monteringsyta, fixturen. Ett exempel på en robotcell visas i figur 1. Varje gång roboten besöker ett tråg för att plocka en komponent med sitt plockverktyg, som i detta projektet kommer vara antingen en sugkopp eller gripdon, förflyttar den sig vidare via trågets tillhörande kamera för att kontrollera om komponenten blivit korrekt orienterad i verktyget. Komponenterna förs sedan till fixturen där de

monteras ihop med övriga komponenter.

Mer övergripande behöver robotarna en isolerad robotcell och för varje arbetsuppgift behövs ett anpassat robotprogram. Ett införande av robotinstallationer för kortvariga produktionsserier är därför inte nödvändigtvis lönsamt. Det är en av anledningarna till att många större producenter av industrirobotar valt att starta utvecklingen av tvåarmade robotar för att lättare kunna integrera robotar i produktionen.

ABB, en av världens ledande producenter av robotar, har utvecklat en tvåarmad robot vid namn YuMi, som ses i figur 2. Innan de mindre tvåarmade robotarna lanserades krävdes det stora ytor för att robotarna skulle kunna användas. YuMi som har formen av en människas överkropp, medför högre platseffektivitet på industrigolven eftersom den inte tar upp mer plats än människorna som utfört arbetet tidigare[4]. De två robotarmarna kan jobba tillsammans på ett bättre sätt än om det hade vart två fristående robotar[5]. Den största fördelen med YuMi är att den kan samarbeta med människor[3] vid till exempel ett produktionsband. Traditionella robotar måste vara inlåsta i en robotcell där samarbete med människor inte är aktuellt eftersom de inte får vistas i robotcellen under körning [6].

Figur 2: YuMi roboten.[1]

ABB valde namnet YuMi som står för "You and Me"[5], för att påvisa samarbetet mellan robot och människa. Tack vare sin höga säkerhet[4][3] med ett mjukt skal, eliminerad klämrisk mellan lederna och snabb stopptid vid eventuella kollisioner, kan YuMi installeras på arbetsstationer som i dagsläget enbart

[1]Källa: ABB, http://new.abb.com/products/robotics/yumi

bemannas av människor, utan extra anpassningar.

Robotar är stora investeringar och för att det ska bli lönsamt krävs det bra tillvägagångssätt för att få fram effektiva robotprogram som styr robotarna. Effektiva robotprogram leder till kortare produktionscykler som resulterar till en ökad produktionshastighet. Idag finns programvaror som kan simulera och generera robotprogram för industrirobotar. Varje robot behöver ett program där det specificeras hur den ska röra sig och mellan de positioner som är av intresse. Det finns två sätt att generera robotkod, online och offline. Vid onlineprogrammering programmeras robotprogrammen manuellt, direkt i produktionscellen som medför driftstopp. Vid offlineprogrammering genereras robotkoden oberoende av produktionscellen via en tredimensionell simuleringsmiljö. I båda fallen sker all programmering manuellt och det sker vanligtvis genom manuella uppskattningar. I *Virtual Engineering: Challenges and Solutions for Intuitive Offline Programming for Industrial Robot* [7] beskrivs de viktiga delarna i offlineprogrammering och utmaningarna som förekommer.

Manuell programmering är tidskrävande och resultatet beror helt på robotprogrammerarens skicklighet. Med tvåarmade robotar på horisonten blir det en ökad svårighetsgrad i att kunna koordinera robotarmarna effektivt. Framtagningen av robotprogram blir mer komplex, vilket kan leda till att det manuella tillvägagångssätten resulterar i ineffektiva robotprogram. Genom att istället grunda framtagningen av robotprogram på automatiskt utförda beräkningar, effektiviseras processen. Resultatet skulle bli bättre och företag skulle spara både tid och pengar. Samtidigt hade användningsområdena för robotar ökat eftersom de blir lätta att anpassa till varierande uppgifter.

YuMi roboten som ligger till grund för det här projektet kommer inte att användas i utförandet, då den inte finns tillgänglig i projektets utvecklingsmiljö. Två ABB IRB 140T robotar kommer istället användas. Skillnaden mellan dem är att YuMi roboten har två armar med sju leder vardera medan en IRB 140T robot har en arm med sex leder. Utformningen av de olika robotarna kommer inte påverka resultatet eftersom lösningsmetodiken fungerar för båda fallen.

Detta kandidatarbete är en fortsättning på tre tidigare studier, varav de första två är kursprojekt: *Optimization of the operation sequence and the cell layout in the YuMi cell* [8] och *Path Planning and Optimization of two armed robot* [9], samt ett examensarbete: *Implementing a Time Optimal Task Sequence For Robot Assembly Using Constraint Programming* [10].

Examensarbetet har lagt grunden till optimeringen av produktionsprocessen genom användning av villkorsprogrammering (constraint programming)[11]. En programmeringsparadigm där det sätts upp specifika villkor och låter en lösare (solver) söka efter lösningar som uppfyller dessa villkor i en avgränsad domän. De två kursprojekten har i sin tur byggt vidare på detta resultat.

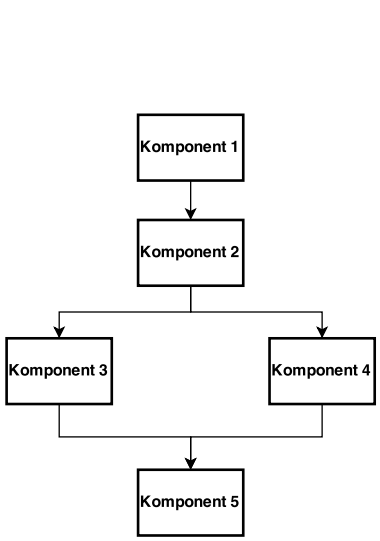
I [8] lades fokus på att effektivisera en tvåarmad robots produktionsprocess. Den största förändringen gentemot examensarbetet var att istället för att individuellt simulera alla robotrörelser antogs det att tiden för en rörelse var linjärt beroende av sträckan. Även fixturplaceringen optimerades genom en iterativ process och det undersöktes hur komponenternas placering i cellen kunde leda till ett förbättrat resultat.

Syftet i [9] var att förbättra det tidigare resultatet genom mer verklighetstrogna tidsuppskattningar för robotrörelserna. Istället för det linjära antagandet användes istället vinkeldifferenser på robotarmarnas leder för att uppskatta förflyttningstiderna.

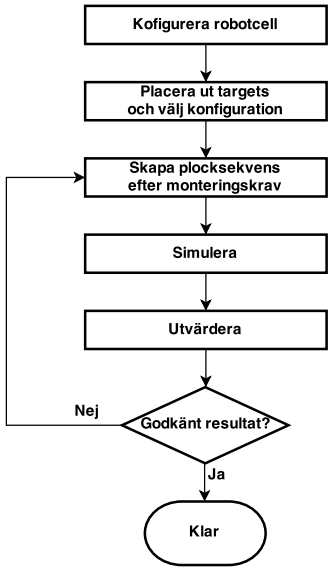
Det här projektet bygger vidare på de tidigare projekten med hänsyn till de olika metoderna som introducerats. Projektets syfte är att automatisera framtagningen av effektiva robotprogram för två robotarmar. Det som skiljer sig från tidigare projekt är att optimeringen kommer ske för generella fall av robotcellens utformning och konfigurationer. Robotarmarnas sekvenser ska optimeras automatiskt med hjälp av villkorsprogrammering och tidsberäkning av armarnas förflyttningar i robotcellen. Till skillnad från de tidigare projekten kommer optimeringsresultatet resultera i genererade robotprogram för båda robotarmarna färdiga att simulera.

2 Offlineprogrammering

Med manuell offlineprogrammering är det inte ekonomiskt möjligt att prova alla kombinationer på en robotcells utformning. I en cell med två robotarmar och flera tråg växer komplexiteten samtidigt som antalet möjliga lösningar ökar. För en robotprogrammerare skulle det ta lång tid att utvärdera alla dessa fall, dessutom är det vanligt att det förekommer monteringskrav som måste tas hänsyn till, vilket innebär att komponenterna är beroende av andra komponenter. Det leder till att det tillkommer ytterligare en faktor som även den ökar komplexiteten. Figur 3(a) visar ett exempel på ett monteringskrav.



(a) Exempel på monteringskrav visualiserat med en graf. Komponent ett måste monteras först, sen komponent två. Därefter kan komponent tre och fyra monteras i oberoende ordning innan slutligen komponent fem ska monteras.



(b) Flödesschema över hur manuell offlineprogrammering utförs idag.

Figur 3: Monteringskrav och tillvägagångsätt vid offlineprogramering.

Idag sker programmeringen intuitivt och flera cellkonfigurationer som möjligen är bättre än den som valts försvinner på grund av grova antaganden. Det är inga matematiska beräkningar bakom dessa processer och det kan ta en lång tid innan det resulterar i ett färdigt robotprogram, som anses som en effektiv lösning på problemet.

För att generera ett robotprogram till en robotcell kan detta ske på olika sätt. Ett av dem vanligare tillvägagångssätten är att använda offlineprogrammering. Vilket innebär att programmet inte skapas på robotarna som det ska appliceras

på, utan istället skapas i en simuleringsmiljö för att testas först. Detta innebär att produktionen inte behöver stoppas för att skapa ett nytt robotprogram, som det behöver om programmeringen istället görs med onlineprogrammering. Ett exempel på en vanlig process som utförs av robotprogrammeraren illustreras i figur 3(b). Först skapas en robotcell i simuleringsmiljön där alla targets (mål i cellen som robotarmarna kan besöka) placeras ut. Därefter bestäms robotarnas konfigurationer i varje target, det vill säga hur robotens leder ska positioneras i varje target. En plocksekvens för varje robotarm skapas för att utföra arbetet enligt monteringskraven. Därefter simuleras och utvärderas programmet för att upptäcka om det uppstår kollisioner eller om plocksekvenserna kan göras effektivare. Om något behöver modifieras skapas en ny sekvens och processen repeteras tills robotprogrammeraren anser att programmet är bra nog.

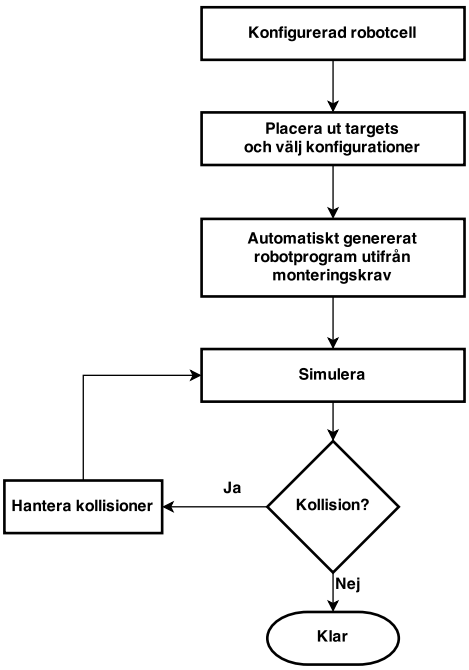
2.1 RobotStudio

RobotStudio är ABB:s egna programvara för offlineprogrammering där det är möjligt att konstruera robotceller och simulera robotprogram. ABB:s robotar styrs med RAPID-kod som genereras direkt i programmet. Fördelen med RobotStudio är att de fysiska robotarna körs med samma styrsystem som RobotStudio använder sig av i simuleringarna[12]. Detta är en stor fördel eftersom att det går att förlita sig på resultatet av en simulering.

När ett robotprogram skapas specificeras det vilken typ av rörelse roboten ska förflytta sig med, mellan alla targets, samt vilken hastighet i rummet robotens verktygsspets ska begränsas till. Det finns två olika typer av rörelser, en linjär rörelse och en ledrörelse. I ledrörelsen rör sig alla leder samtidigt och varje led håller en konstant vinkelhastighet. Dessutom anpassas varje leds hastighet till att alla leder ska nå sina slutgiltiga vinklar samtidigt, detta medför att robotens rörelse i rummet inte blir linjär. Vid en linjär rörelse anpassas ledernas vinkelhastighet till det att verktygsspetsens rörelse i rummet blir linjär, samtidigt som den håller den specificerade hastigheten. Detta medför att varje leds individuella rörelse inte blir konstant. I detta projektet kommer robotarmarna röra sig med ledrörelser.

3 Problem & Ansats

För att automatiskt generera robotprogram behövs det i likhet med den manuella offlineprogrammeringsprocessen byggas upp en robotcell, placeras ut targets samt väljas varje robots lämpliga ledkonfiguration i varje target. När den informationen finns ska det automatiskt genereras optimerade robotprogram. Detta ska göras med en optimeringsprocess som minimerar produktionstiden utifrån den angivna cellen, monteringskraven och de automatiskt beräknade förflyttningstiderna för robotarmarna mellan cellens alla noder. Resultatet presenteras som robotprogram, färdiga att simulera för att upptäcka eventuella kollisioner. Kollisioner får användaren manuellt hantera tills simuleringen blir kollisionsfri. Hela processen kan ses i figur 4.



Figur 4: Ansats för en automatisk offlineprogrammering.

3.1 Optimering av produktionscykel

Vid installation av en robot och anpassning av dess cell finns det många aspekter att ha i åtanke. Vart ska robot, fixtur, tråg och kameror placeras? Hur ska komponenterna fördelas mellan trågen? I vilken ordning ska roboten besöka alla olika positioner och hur ska den röra sig? Introduceras ytterligare en robot ökar

antalet frihetsgrader, vilket gör att fler alternativa lösningar måste utvärderas. Därför måste vissa förenklingar och antaganden göras. Robotarmarnas, trågens och kamerornas placeringar antas vara förbestämda, ett troligt scenario när en robot ska ta en människas plats i en produktionslinje. Fixturen antas ha ett begränsat horisontellt plan inom vilket den kan placeras. Optimeringen kommer även utföras utifrån antagandet att inga kollisioner sker under armarnas rörelser, dock tas det hänsyn till att två armar inte får betjäna samma position samtidigt.

I projektet behandlas tre fundamentala optimeringsproblem; vilken ordning robotarmarna ska plocka komponenter, hur komponenter ska placeras i trågen samt vart fixturen ska placeras. Den grundläggande metodiken för att lösa dessa problem har etablerats av de tidigare projekten, men endast på explicita testfall. En svårighet i detta arbete blir därför att applicera metodiken på generaliserad indata och utvärdera metodikens tillförlitlighet på mer varierade scenarion. Det är också önskvärt att optimeringen inte blir lika tidskrävande som i de tidigare projekten.

3.2 Approximation av förflyttningstider

För att robotarmarnas plocksekvenser ska kunna optimeras behövs en verklighetstrogen metod för att kunna beräkna tiden det tar för robotarmarna att förflytta sig mellan alla targets i en godtycklig robotcell. I [8] har det dokumenterats att antagandet att tiden skulle vara linjärt beroende av avståndet är en dålig approximation av verkligheten. Att flytta robotarmen två sträckor som är lika långa kan variera i tid beroende på flera faktorer. Exempelvis hur robotens leder är konfigurerade i varje target, vilken typ av rörelse roboten använder under förflyttningen och vad hastigheten begränsas till i cellens rymd. I [9] har det konstaterats att vid ledrörelser kan förflyttningstiderna approximeras genom att kolla vilken av robotens leder som tar längst tid att rotera vid en förflyttning mellan två targets. Då tidsberäkningen baseras på varje leds vinkelförändring beror förflyttningstiden på vilken ledkonfiguration användaren valt.

Tiden mellan alla objekt är av stor betydelse på grund av att det är en stor del av underlaget för optimeringen. Utifrån en konfigurerad robotcell i RobotStudio behöver cellen läsas av för att samla in aktuell data. Detta leder till ett förenklat krav på mängden indata användaren behöver ange. När tiderna för robotarmarnas förflyttningar i cellen är kända ska resultatet användas tillsammans med övrig indata för att skapa optimala robotprogram.

3.3 Simulering

För att möjliggöra en simulering av ett fungerande robotprogram i RobotStudio behövs programmen automatiskt skapas för varje robot efter att optimeringsprocessen är avslutad. Ett problem är att kunna ta hänsyn till alla enskilda plockmetoder som varje komponent behöver ha. Plockmetoderna skapas manuellt och görs först efter det att sekvenserna är presenterade. Simuleringen ska visa de framräknade sekvenserna, dessutom behövs det kontrolleras om några

kollisioner uppstår innan robotprogrammen kan realiseras. Dynamiska kollisioner behöver lösas manuellt av användaren eftersom projektet inte ska ta hänsyn till det.

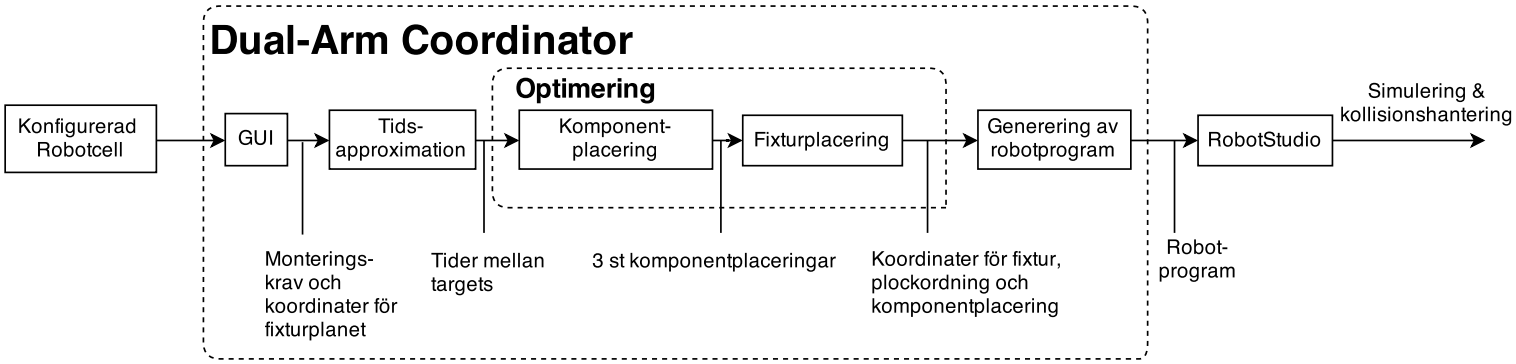
4 Optimering av cellkonfiguration och robotsekvensering

I Robotstudio finns möjligheten att skapa tilläggsprogram i form av ett Add-In, i vilket användaren kan utöka funktionaliteten [13]. Ett Add-In, vid namn Dual-Arm Coordinator (DAC), har konstruerats enligt flödesschemat i figur 5 för att lösa de olika delproblemen i kapitel 3. I nästkommande delkapitlen beskrivs det hur de olika delarna i DAC är uppbyggda och hur de fungerar med varandra.

4.1 Lösningsmodellens arkitektur

Genom RobotStudios API kan Add-in skapas i C#. För att underlätta för användaren exekverar DAC externa program automatiskt. Detta gör det även möjligt att på ett smidigt sätt skapa en användarvänlig helhetslösning i ett program.

Underlaget till optimeringen är indatan som användaren anger via ett grafiskt användargränssnitt (GUI) samt förflyttningstiderna för robotarmarna att flytta sig mellan alla targets i användarens konstruerade robotcell. Indatan från gränssnittet består av komponenterna, monteringskraven, fixturplanets gränser, information om komponenterna ska sugas eller plockas upp av robotverktygen och servicetiderna som representerar tiden det tar att plocka varje komponent i trågen samt tiden det tar att montera dem i fixturen. I figur 5 visas ej optimeringen av plockordningen. Den görs både i optimeringen av komponentplaceringen och fixturplaceringen. Utifrån optimeringens resultat genereras robotprogram för varje robotarm som är redo att simuleras.



Figur 5: Flödesschema över Dual-Arm Coordinator.

4.2 Approximation av förflyttningstider

Tidsberäkningen baseras på samma metod som presenterats i ett tidigare arbete [9], men med ett vidareutvecklat tillvägagångssätt. Förflyttningstiden mellan två punkter baseras på den led vars rörelse tar längst tid. Det som förbättras i detta projekt är att genom användning av RobotStudios API kan förflyttningstiderna beräknas för generella fall direkt i RobotStudio. Dessutom kan programmet hantera den ökade komplexiteten som tillkommer när detta sker i RobotStudio, som att en robot kan ha flera olika ledkonfigurationer för samma target.

4.2.1 Tidsberäkningen

När användaren konfigurerat cellen i RobotStudio läser DAC automatiskt av alla targets och kontrollerar samtidigt vilka targets varje robot kan nå. De targets en robot inte kan nå förbises i tidsberäkningen, för att inte försöka beräkna tiden mellan två positioner som inte är möjligt att röra sig emellan. Genom att undvika det elimineras risken att optimeringen genererar en sekvens som inte går att realisera. Framtagningen av förflyttningstiderna förenklas då robotarmens leder antas hålla konstant maxhastighet, alltså försummas både accelerations- och deaccerlationstiderna. I många rörelser är accelerations- och deaccerlationstiderna en liten del av den totala tiden det tar för robotarmen att förflytta sig, då det går fort att nå maxhastighet. För att göra en bra approximering av tiderna det tar för roboten att förflytta sig mellan de olika komponenterna och fixturen behövs det koordinater för alla tråg, fixturer och kamerorna. När koordinaterna automatiskt läses av från cellen slipper användaren att ange rena koordinater för alla targets. För att kunna läsa av robotens leder i alla targets kommer DAC att flytta robotarmarna till varje position och läsa av vinklarna på armens samtliga leder. När vinklarna på robotarnas leder i alla positioner är kända kommer tiderna att beräknas.

nT är antalet target, nL är antalet leder roboten har och $VELOCITY$ är en vektor innehållande alla ledernas maxhastighet.

$$VELOCITY = \langle v_1, ..., v_{nL} \rangle$$

För att beräkna vilken led som tar längst tid att förflytta behöver det beräknas hur långt tid det tar för varje led, l, att förflytta sig mellan starttarget, s, och sluttarget, e. Divideras resultatet med ledens hastighet ger det den förflyttningstiden som krävs för just den leden.

$$angel\_diff_{l,s,e} = abs\left(angel_{l,s} - angel_{l,e}\right)$$

$$\forall\, l \in \{1, ..., nL\}, \quad \forall\, s, e \in \{1, ..., nT\}$$

$$time\_diff_{l,s,e} = angel\_diff_{l,s,e} / VELOCITY_l$$

$$\forall\, l \in \{1, ..., nL\}, \quad \forall\, s, e \in \{1, ..., nT\}$$

För att få ut den totala tiden det tar för robotarmen att förflytta sig kontrolleras vad den längsta förflyttningstiden är för varje led, som kommer bli den slutgiltiga förflyttningstiden.

$$traveltime_{s,e} = MAX\left(time\_diff_{1,s,e}, ..., time\_diff_{nL,s,e}\right)$$

När alla tider är beräknade för alla möjliga förflyttningar kommer en matris med förflyttningstiderna mellan alla objekt att skapas och skickas vidare till optimeringen.

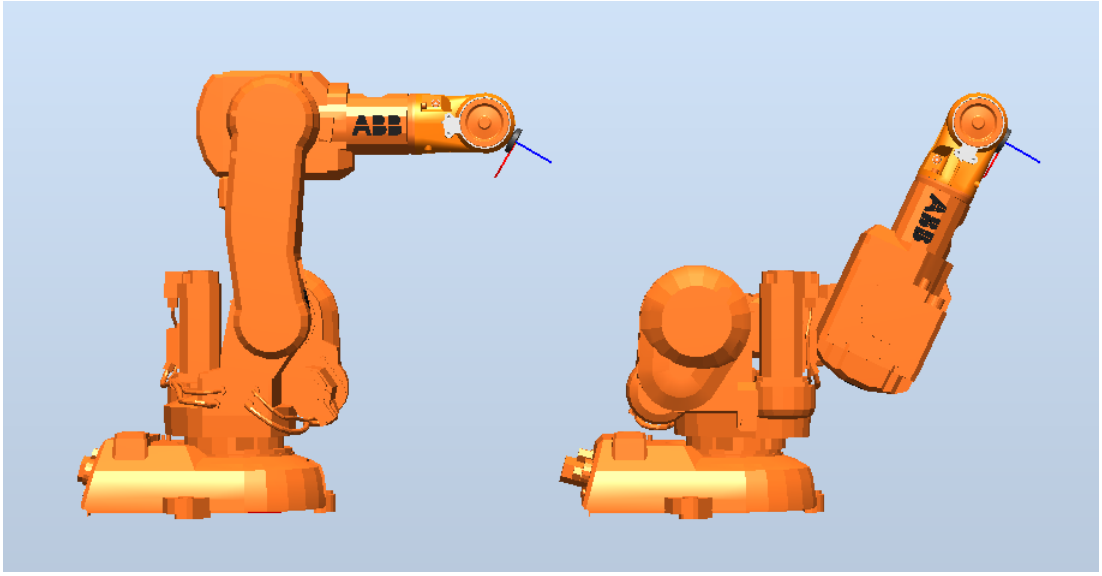
4.2.2 Ledkonfigurationer

Många av ABB:s enarmade robotar har sex olika leder som helt fristående kan roteras. YuMi roboten, som har två armar, har sju fristående leder på varje arm. YuMi, som har en extra led, ger mer flexibla rörelser som i sin tur skapar fler möjligheter att förflytta sig mellan två punkter. Dessa leder ligger som grund för hur lång tid det tar för en robot att ta sig mellan två positioner då varje led har en egen maxhastighet. Till de flesta positionerna i cellen behöver samtliga leder röra sig men det är oftast den som behöver förflytta sig längst som kommer avgöra tiden det tar för roboten att förflytta sig. När användaren specificerar en target i RobotStudio måste även en ledkonfiguration väljas för att robotarmen ska kunna förflytta sig dit. Det beror på att det alltid finns mer än en ledkonfiguration roboten kan anta för att nå en target, se figur 6. Hur användaren väljer att konfigurera robotens leder i varje target kommer därför vara av stor betydelse för tidsberäkningen då det avgör ledernas ursprungspositioner.

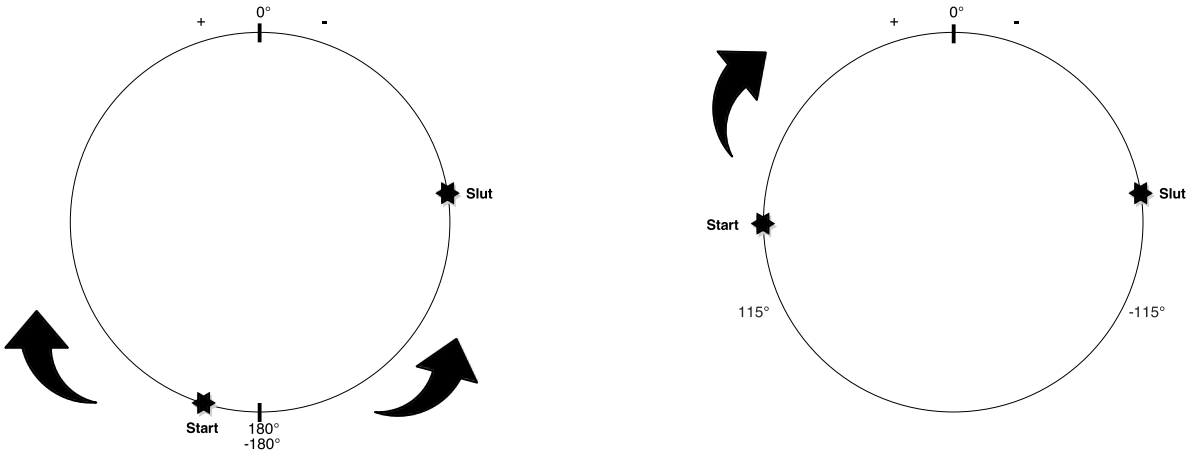
Robotarmarnas leder är begränsade till hur mycket de kan rotera. Konfigurationerna har en stor påverkan och kan som exemplet i figur 7(a) resultera i att om en led ska förflytta sig från -170° till 80°finns det två riktningar att välja mellan. Den enklaste metoden att lösa detta är att beräkna tiden det tar när leden passerar 0°, vilket alltid kommer fungera för alla leder på alla robotar. Vissa leder kan rotera mer än 360° och kan då även rotera motsols och passera 180°. I detta exemplet skulle det innebära en kortare rotation då leden enbart behöver flyttas 110° istället för 250° som den långa vägen skulle innebära. Att applicera den metoden medför dock ett problem då alla leder inte kan rotera ett helt varv, se figur 7(b) där leden är begränsad till 115°rotation åt varje håll. Genom att kontrollera om det för leden är möjligt att välja den kortare vägen och i sådana fall välja den riktningen vid de fall det blir en genväg skulle leda till effektiva rörelser. Det finns även fall där det inte är längsta vägen att passera 0°, till exempel om leden ska flytta sig från -10° till 10° så det behövs en kontroll för vilken väg som skulle vara den bästa innan det går att avgöra vilken som skulle vara den mest lämpade.

4.2.3 Problematik med tidsberäkningsmetoden

Problematiken med att hitta den bästa ledkonfigurationen är att det är många steg i DAC som påverkar och att det ändrar förutsättningarna för den optimala sekvensen. Vilken konfiguration som är mest lämpad för en target beror



Figur 6: Figuren visar samma robot som står i samma target med två olika ledkonfigurationer.



(a) Figuren visar en leds rotationsmöjligheter. Riktningen motsols som är den längre i detta fall är den som alltid fungerar, men i de leder som kan roteras hela varv runt sin axel kan det finnas möjlighet till en kortare förflyttning genom att gå motsols.

(b) Figuren visar en led som är begränsad till $\pm 115°$, i det fallet finns det bara en riktning att gå.

Figur 7: Rotationsriktningens påverkan på tidsberäkningen.

på konfigurationen i den tidigare och efterkommande targeten i sekvensen som resulterar i den minsta förändring av ledernas vinklar under rörelsen. Det är därför omöjligt att veta varje targets optimala konfiguration innan en sekvens är bestämd, dock sker tidsberäkningen innan sekvensen är framtagen. Det betyder att den framräknade sekvensen inte kan garanteras ge de optimala robotprogrammet då den utgått från targets som är konfigurerade utefter vad användaren tror är bäst. Att försöka hitta varje sekvens optimala konfiguration ökar komplexiteten i optimeringen då en sekvens kommer behöva beräknas flera gånger med alla möjliga konfigurationer. Svårigheten med att lösa problemet är att om DAC skulle ändra ledkonfigurationerna det sista när rutten är genererad skulle alla förutsättningar ändras, då det tagits hänsyn till hur konfigurationen var satt av användaren innan processen började. Detta gör att det skulle behöva ändras konfigurationer varje gång en ny plockordning testas, vilket skulle öka komplexiteten och tidsåtgången för framtagningen av robotprogrammen mycket, eftersom att det alltid finns minst två möjliga konfigurationer för varje target, för varje robot.

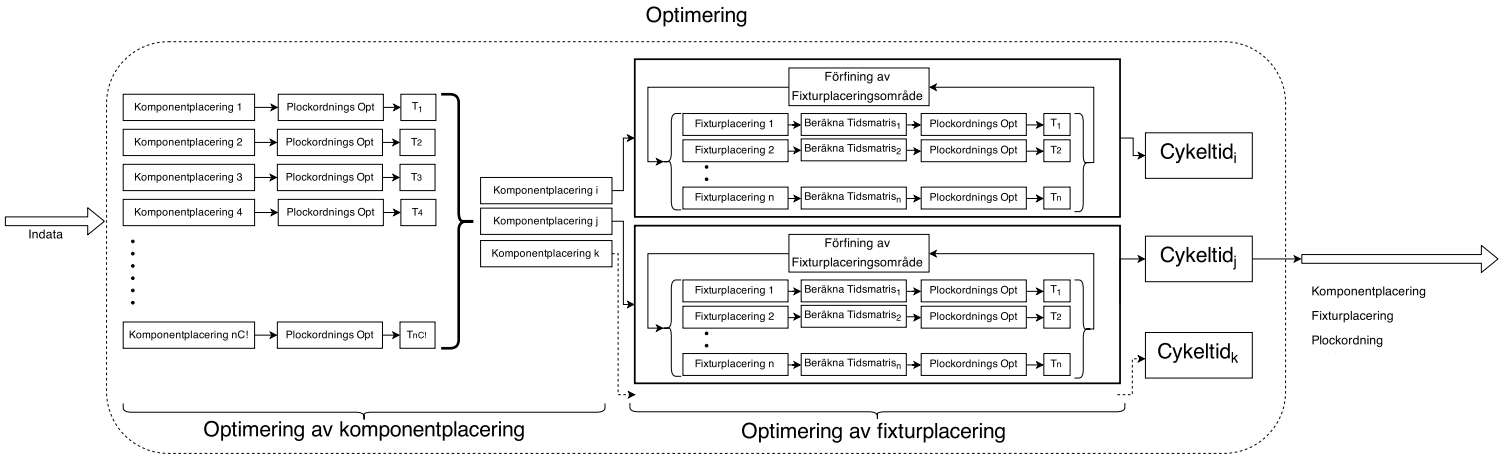
Den beräknade tiden blir missvisande eftersom metoden tar hänsyn till varje leds maxhastighet vid varje leds individuella tidsåtgång att förflyttas mellan två positioner. Det är ett problem att varje led har en maxhastighet som inte är kopplad till den maxhastighet som användaren kan sätta på robotarmens verktygsspets på en rörelse i robotprogrammen. Det gör att det alltid utförs en uppskattning av tiden med avseende på alla leders maximala vinkelhastighet, vilket inte är ett rimligt antagande eftersom att alla leder inte roteras med full hastighet vid en förflyttning. Resultatet blir att approximationen av de beräknade förflyttningstiderna blir mindre än vad den simulerade tiden blir, trots att tiden roboten inte rör sig i simuleringen förbises. Den funktionalitet som finns tillgänglig i RobotStudios API tillåter ingen lämplig lösning på problemet. För att kunna förlita sig på de framtagna tiderna skulle det bästa vara att implementera funktionalitet i RobotStudios API för att automatiskt kunna simulera och mäta simuleringstider eftersom det hade blivit exakt samma tider som det kommer vara i simuleringen av robotprogrammen. Det skulle leda till att optimeringsresultatet skulle bli mer verklighetstroget.

4.3 Optimering

För att minska komplexiteten på optimeringsproblemet till något som är överkomligt med dagens beräkningskraft delas problemet upp i tre mindre beståndsdelar: plockordning, komponentplacering samt fixturplacering. Under optimeringen av var och en av dessa delproblem kan vissa variabler hållas fixa för att minska antalet frihetsgrader och underlätta en lösning. Se figur 8 för hela optimeringsflödet.

4.3.1 Förarbete av Indata

Mellan det att en komponent plockas och att den placeras på fixturen måste komponenten visas upp för en kamera. En möjlig händelse är att armen även



Figur 8: En överblick av hela optimeringsprocessen.

utför annat arbete mellan dess att komponenten plockats och visats för kameran. Detta har dock inte uppstått under något explicit testfall eftersom trågen och deras respektive kameror alltid är placerade nära varandra. Ett mer sannolikt fall är att komponenten direkt förs till sin kamera efter upplockning, vilket är ett antagande som därför görs i optimeringen.

Ur datan från tidsberäkningen sparas därför förflyttningstiderna mellan tråg och deras kameror undan som extra servicetider för trågnoderna. Därutöver justeras tidsmatrisen till att tider från trågen är tider från trågens kameror. Förenklingen att kameran alltid besöks direkt minskar lösningsrummet och bidrog till en tiofaldig förminskning av tiden som krävdes för att utföra optimeringsprocessen.

4.3.2 Plockordning - Ett *Vehicle Routing Problem*

Plockordnings-problemet är att utifrån förbestämda positioner av alla objekt i robotcellen hitta den optimala ordningen för roboten att besöka alla nödvändiga positioner, för att fullfölja en produktionscykel. Fixturens position och komponenternas placering hålls alltså här fixerade. Den uträknade besöksordningen måste uppfylla monteringskraven, ta hänsyn till robotens begränsningar (t.ex. att varje arm endast kan bära en plockbar och en sugbar komponent i taget) och inte låta två armar betjäna en geografisk position samtidigt. Plockordningsoptimeringen kommer att anropas av komponent- och fixturplaceringsoptimeringen ett flertal gånger med olika indata. Det gör plockordningsoptimeringen till den mest centrala beståndsdelen i hela optimeringsprocessen och det är av största vikt att den kan lösas effektivt.

Problemet kan ses som ett Vehicle Routing Problem (VRP) i vilket ett antal

fordon ska besöka en mängd utspridda noder med olika distanser emellan sig. Rutter planeras med villkoret att alla noder besöks exakt en gång, av endast ett fordon. Målet är att minimera den längsta av dessa fordonsrutter.

I det aktuella fallet består fordonen av robotarmarna, medan besöksnoderna är trågen, kamerorna, hempositionerna och fixturen. Distanserna mäts i tid, dvs. avståndet mellan två noder består således av tiden det tar en robotarm att rör sig däremellan, givet av tidsberäkningen. Denna tid varierar beroende på vilken arm som ska utföra rörelsen, eftersom att olika konfigurationer krävs för att armarna ska nå samma punkt i robotcellen.

Ett problem som uppstår är att fixturen i praktiken kommer behövas besökas multipla gånger, men en nod får endast besökas en gång i ett VRP. Detta löses genom att betrakta fixturen som flera olika noder placerade i samma geografiska punkt, utan någon rörelsekostnad sinsemellan. Därför är det alltid en ny nod som besöks av roboten när den går till fixturen, vilket gör rörelsen tillåten inom VRP:ets ramar. Samma logik kan även appliceras om ett tråg behöver besökas multipla gånger för hämtning av flera exemplar av en likadan komponent.

Mer precist kan denna optimering ses som en specifik variant av ett VRP vid namn Capacitated Vehicle Routing Problem with Time Windows (CVRPTW). Varje fordon har då en maxkapacitet på hur mycket den kan bära och kan endast besöka noder under vissa tidsfönster. Sådana krav förekommer i detta projekt. De två robotarmarna får inte befinna sig i två noder med samma geografiska position samtidigt, vilket kan ses som en begränsning i tidsfönstret och varje robotarm har en maxkapacitet på en plock- och en sugbar komponent i taget. För mer ingående diskussion om VRP se [14].

VRP och CVRPTW tillhör mängden av NP-svåra problem vilka i största allmänhet är svårlösta. Ett lämpligt tillvägagångssätt har visat sig vara villkorsprogrammering, se [15], vilket även är det tillvägagångssätt som använts i detta projekt.

4.3.3 Villkorsprogrammering

Den grundläggande idén med villkorsprogrammering är att deklarera ett antal förbestämda parametrar, okända *decision variables*, och *constraints* som beskriver hur dessa ska förhålla sig till varandra. En solver söker sedan efter möjliga värden på variablerna som uppfyller alla *constraints*, samt satisfierar en given *objective function*. De specifika stegen en solver ska utföra deklareras dock aldrig explicit. Istället är detta upp till solvern själv att avgöra, även om vissa riktlinjer kan ges av användaren.

Ett exempel på ett trivialt problem formulerat som ett villkorsprogrammeringsproblem:

$$parametrar : z = 8$$

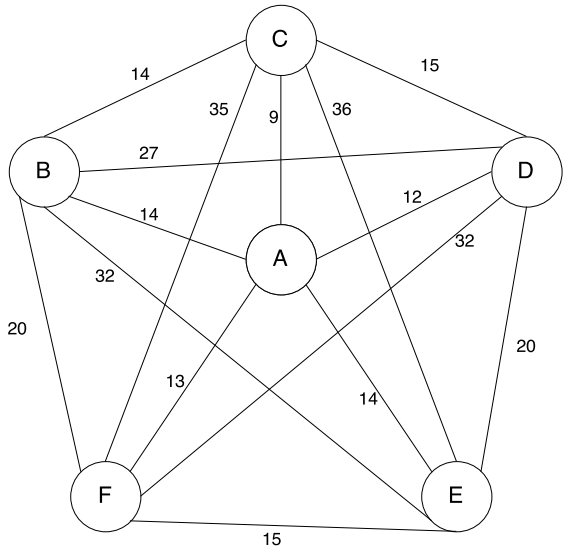
$$decision\ variables : x \in \{1, ..., 10\}, \quad y \in \{3, ..., 10\}$$

$$constraints : x > y, \quad x + y > z$$

$$objective\ function : minimize(x)$$

En solver hittar enkelt lösningen $x = 5, y = 4$.

Villkorsprogrammering ska nu appliceras på ett enkelt VRP för att ge en inblick i tillvägagångssättet. I problemet ska sex noder besökas av ett fordon. Problemet illustreras i figur 9.



Figur 9: Vehicle Routing Problemet är att bilda en Hamilton-cykel (en sluten loop som går genom alla noder exakt en gång) med minsta möjliga totalkostnad (distans). Eftersom endast ett fordon används kan detta också ses som ett Traveling Salesman Problem[16].

Parametrarna i det här problemet består av $NODES$ och $distance$. $NODES$ är mängden noder där varje nod tillskrivits ett numeriskt värde. A representeras av 1, B representeras av 2, och så vidare. $distance$ är en matris vars element, $distance_{i,j}$ är distansen mellan två noder i och j angivna i figur 9.

$$NODES = \{1, ..., 6\}$$

$$distance = \begin{pmatrix} 0 & 14 & 9 & 12 & 14 & 13 \\ 14 & 0 & 14 & 27 & 32 & 20 \\ 9 & 14 & 0 & 15 & 36 & 35 \\ 12 & 27 & 15 & 0 & 20 & 32 \\ 14 & 32 & 36 & 20 & 0 & 15 \\ 13 & 20 & 35 & 32 & 15 & 0 \end{pmatrix}$$

$successor$ är en array vars element, $successor_i$, anger vilken nod som ska besökas efter i. För att förtydliga: värdet på $successor_2$ är alltså noden som ska besökas

efter nod B. Eftersom dessa värden är okända och ska beräknas, anges *successor* som en *decision variable* och tar sina värden i $NODES$.

$$successor_i \in NODES \quad \forall\, i \in NODES$$

För att formulera detta problem kan matematiskt fördefinierade *constraints* användas, kallade *global constraints*. Detta är ofta fördelaktigt eftersom *global constraints* är väl studerade och solvers har ofta speciella metoder för att hantera dem. *alldifferent(x)* anger att alla värden i en array x ska anta olika värden. *circuit(x)*, där x är en array med *successor* struktur, anger att elementen i x ska bilda en sluten loop. Båda är villkor som *successor* måste uppfylla.

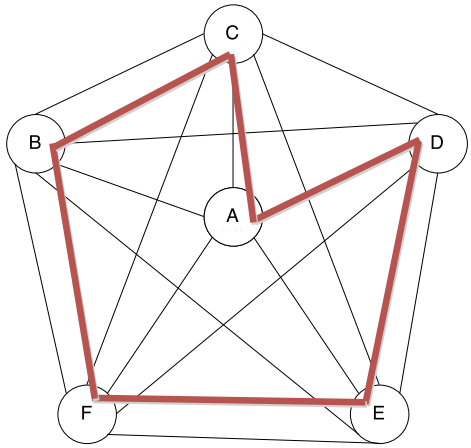
$$alldifferent(successor)$$

$$circuit(successor)$$

Slutligen anges en *objective_function* som anger optimeringsmålet: att minimera den totala ruttens sträcka.

$$minimize\,(distance_{1,successor_1} + ... + distance_{6,successor_6})$$

Genom att explicit skriva in modellen ovan i ett villkorsprogrammeringsspråk kan den exekveras, vilket medför att en solver löser problemet. Lösningen visas i figur 10.



Figur 10: Lösning till VRP:et definerad i figur 9. Den totala distansen för denna rutt är 90.

4.3.4 MiniZinc Modellen

För att formulera CVRPTW:et som ett villkorsprogrammeringsproblem användes MiniZinc 2.0.1. Ett högnivå modelleringsspråk som har ett gränssnitt till FlatZinc, vilket är ett lågnivåspråk som kan sammankopplas med olika solvers. Solvern använd under detta projekt har varit *G12 Finit Domain Solver* utvecklad av The University of Melbourne. En stor fördel med MiniZinc är möjligheten att dela upp optimeringsproblemet i en modellfil och en datafil. I datafilen anges all indata medan i modellfilen beskrivs och löses problemet utifrån en generell datafil. På det sättet kan samma modellfil återanvändas för att hantera problemet för olika indata.

Modellfilen består huvudsakligen av parametrar, *decision variables*, *constraints*, och en *objective function*, medan en datafil endast består av parametrar. En utförlig förklaring till uppställningen av parametrar, *decision variables*, och *constraints* redovisas i de nästkommande avsnitten. Modellen som använts är en vidare utveckling av den som skapats i [8][9], och för att vara konsekvent har objekt namngivits på samma vis.

4.3.4.1 Parametrar

Objekt som inte explicit deklareras som en annan typ blir automatiskt av typen *par* (parameter) i MiniZinc. Det betyder att dess värde är förbestämt och behöver inte avgöras av solvern. Det är oftast numeriska konstanter, men kan även innefatta andra saker som numeriska mängder. Alla parametrar måste vara angivna antingen i modellfilen eller i datafilen innan modellen exekveras. Några av de viktigare parametrarna i modellen följer nedan:

nA - Antalet robotarmar.

nC - Antalet komponenter. Ekvivalent till antalet tråg.

nGL - Antalet geografiska positioner i form av tråg och fixturen i cellen. Armarnas hempositioner och kameror är ej medräknade.

nN - Antal noder som ska besökas. Start och slut noderna i hempositionen är ej medräknade. Är därför lika med $nN = TrågNoder + FixturNoder = 2 * nC$.

LOCATIONS - Ett set av integers $= \{1, 2, ..., nGL + nA\}$ där varje värde motsvarar en position. De första nC är tråg. Sedan kommer fixturen följd av hempositioner. När en variabel tar ett värde i $LOCATIONS$ är det ett matematiskt sätt att tilldela den en av dessa positioner.

$$LOCATIONS_i = \begin{cases} tråg_i & , \ i \in \{1, ..., nC\} \\ fixtur & , \ i = nC + 1 = nGL \\ hemposition_{i-nGL} & , \ i \in \{nGL + 1, ..., nGL + nA\} \end{cases}$$

service_time - En array med servicetiderna för varje position i $LOCATIONS$. Servicetid är tiden det tar för en robotarm att utföra dess uppgift i en nod. Till exempel att släppa av eller plocka upp en komponent.

camera_time - För varje tråg och robotarm finns en associerad kameratid. Detta är tiden det tar för armen att föra en komponent från tråget till dess kamera och analysera om den greppats rätt efter upplockningen.

NODES - Ett set av integers = $\{1, 2, ..., nN+2*nA\}$. Här motsvarar varje värde en av VRPets noder. Har liknande struktur som $LOCATIONS$, men notera att en geografisk position kan motsvara flera noder. I enlighet med diskussionen i avsnitt 4.3.2 finns det en unik fixturnod för varje komponent. Även armarnas hempositioner ses som två separata noder. En startnod och en slutnod per arm.

$$NODES_i = \begin{cases} tråg_i & , \ i \in \{1, ..., nC\} \\ fixtur_{i-nC} & , \ i \in \{nC+1, ..., 2*nC\} \\ start_{i-2*nC} & , \ i \in \{2*nC+1, ..., 2*nC+nA\} \\ slut_{i-2*nC-nA} & , \ i \in \{2*nC+nA+1, ..., 2*nC+2*nA\} \end{cases}$$

geo_loc - En array med information om vilka värden i $NODES$ som befinner sig på samma plats. Två noder med samma geografiska position kan inte betjänas samtidigt.

gripper_component - En nC lång boolean array. Om $gripper\_component_i$ är $true$ ska komponent i plockas med armens plockverktyg, om den är $false$ ska den plockas med sugverktyget.

precedence - En $2 \times$ *antal monteringskrav* matris som anger vilka komponenter som har företräde. Komponent $precedence_{i,1}$ har prioritet över $precedence_{i,2}$ $\forall i$ och måste därför anlända till fixturen före den andra.

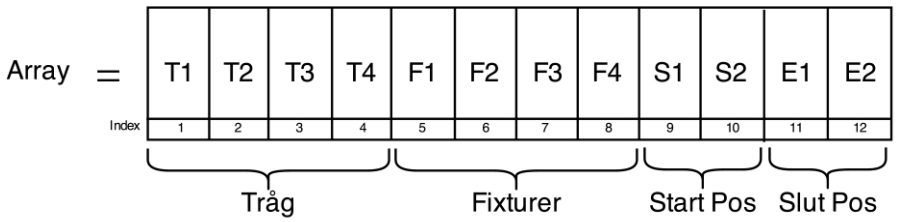
reachable - En $nC \times nA$ boolean-matris som anger huruvida en robotarm når ett tråg eller inte. Om $reachable_{(i-1),j} = true$ kan arm i nå tråg j. Alla tråg måste nås av minst en arm.

distance - En matris med tiderna det tar en arm att röra sig mellan två $LOCATIONS$. Eftersom olika armar kräver olika lång tid för att fullfölja dessa rörelser är denna matris $LOCATIONS \times (LOCATIONS * nA)$ stor. För arm ett tar det $distance_{i,k}$ tid att röra sig mellan $LOCATIONS_i$ och $LOCATIONS_k$, medans arm två utför samma rörelse på tiden $distance_{i+nA+nGL,k}$. Tider från tråg beräknas från trågens respektive kameror eftersom det är där robotarmen befinner sig då arbetet på noden är slutfört. Se sektion 4.2.1 för mer information angående hur dessa tider beräknas.

4.3.4.2 Decision Variables

Typen *var* anger ett objekt som en *decision variable*. Det är dessa variabler solvern ska bestämma optimala värden på. När en *var* deklareras är det även fördelaktigt att ange ett begränsat domän där variabeln tillåts ta sina värden. Detta underlättar ofta för solverns framtida sökning. I denna MiniZinc modell är flera av de viktigaste *decision variables* utformade som arrays och har en uppbyggnad som liknar $NODES$ där varje array element beskriver de tillhörande attributen i en nod. Se illustrationen i figur 11. I utdatan från MiniZinc är det i synnerhet *successor* och *arrival_time* som är av intresse då dessa anger armarnas rörelser respektive tiderna rörelserna beräknas ta.

arm - Anger vilken arm som ska besöka noden. Tar sina värden i $\{1, 2\}$ i fallet med två robotarmar.



Figur 11: En exempel array med nod strukturen som återfinns i flera *decision variables*. I detta exempel har vi fyra komponenttråg och två armar. (Arrayindex börjar på 1 i MiniZinc).

successor & predecessor - Anger vart armen som besöker noden ska gå till därefter respektive vart den kom ifrån. Tar värden i $NODES$.

arrival_time & departure_time - Anger tiden då en arm anländer respektive lämnar en nod, räknat från robotens första rörelse. Skillnaden mellan dessa tider utgörs av nodens *service_time*, *camera_time*, och *wait_time*.

start_time - Tiden då en arm börjar utföra arbete på en nod. Antar samma värde som *arrival_time*, såvida det inte uppstår en *wait_time*.

wait_time - Om en arm anländer till en nod som redan arbetas på av en annan arm måste den invänta sin tur att serva noden. I ett optimalt fall undviks helst väntetider vilket även återspeglas i VRP:s lösningar där *wait_time* ofta är noll eller väldigt små för alla noder.

Utöver dessa finns vissa *decision variables* som döljs ifrån utdatan men fortfarande är viktiga internt i modellen för att formulera villkoren.

gripper_occupied & suction_occupied - Anger huruvida nodens arm har sitt grip- respektive sug-verktyg ockuperat efter att arbetet i noden utförts.

holding_components - En $nC \times NODES$ matris som anger vilka komponenter nodens arm bär på efter att arbetet i noden är utfört.

4.3.4.3 Constraints

Det karakteristiska draget för villkorsprogrammering är typen *constraint*. Den anger hur alla modellens variabler och parametrar ska förhålla sig till varandra för att en lösning ska accepteras. *Constraints* utformas som booleanska uttryck och en vanligt förekommande konstruktion i detta projekt är att de loopar över samlingar av noder. Totalt består modellen av 37 *constraints* varav de intressantare redovisas nedan.

Constraints som uttrycker krav som ställs på ordningen armarna får besöka noderna.

***alldifferent*(*successor*)**, ***alldifferent*(*predecessor*)** - Ett alldifferent *constraint*

säger att varje värde i en array måste vara unikt, vilket här medför att varje nod endast kan besökas en gång. Alldifferent är ett globalt *constraint* vilket innebär att många solvers har speciella metoder för att hantera det effektivt under beräkningen.

$\boldsymbol{successor_{predecessor_n} = n} \quad \forall\, \boldsymbol{n \in NODES}$ - Nästkommande nods föregående nod är den aktuella noden. Samma *constraint* återfinns även med omvända platser på *successor* och *predecessor* för att garantera att de följer en konsekvent struktur.

$\boldsymbol{arm_{successor_n} = arm_n} \quad \forall\, \boldsymbol{n \in NODES}$ - Samma arm ska användas hela vägen längs en rutt. Lösningen kommer på så vis innefatta en stängd rutt per arm.

$\boldsymbol{reachable_{1,component_n} \wedge \neg reachable_{2,component_n} \Rightarrow arm[n] = 2,}$
$\boldsymbol{reachable_{2,component_n} \wedge \neg reachable_{1,component_n} \Rightarrow arm[n] = 1} \quad \forall\, \boldsymbol{n \in NODES}$ - Om en nod endast nås av en robotarm kommer den betjänas av den armen. Om en nod nås av mer än en arm specificeras inte betjäningsarmen. Det lämnas åt solvern att avgöra vilken arm som vore optimal.

$\boldsymbol{holding\_component_{component_n,predecessor_n}} \quad \forall \boldsymbol{n \in FIXTURES}$ - För att besöka en fixturnod måste rätt komponent bäras på i föregående nod. Med $FIXTURES$ menas mängden noder som är fixturnoder.

$\boldsymbol{component_i = precedence_{k,1} \wedge component_j = precedence_{k,2} \wedge i \neq j \Rightarrow departure\_times_i \leq start\_time_j \; \forall i, j \in FIXTURES, \; \forall k}$ - Ser till att byggkraven följs. Ska komponent i placeras före j enligt byggkraven ser detta *constraint* till att fixturnod i är besökt innan arbete börjar i fixturnod j.

Constraints som behandlar tidsaspekter. Typiska för ett VRP-TW där anländnings- och avgångs-tider bokförs för att koordineras med tillåtna betjäningsfönster.

$\boldsymbol{wait\_time_n = start\_time_n - arrival\_time_n} \quad \forall\, \boldsymbol{n \in NODES}$ - Definierar hur väntetiden förhåller sig till de andra tiderna.

$\boldsymbol{start\_time_n + service\_time_n + distance_{geo\_loc_n,geo\_loc_{successor_n}} \leq}$
$\boldsymbol{\leq arrival\_time_{successor_n}} \quad \forall\; \boldsymbol{n \in \{NODES \backslash TRAYS : arm_n = 1\}}$ - Från det att en arm börjar betjäna en nod, som inte är ett tråg, tar det minst servicetiden och tiden för rörelsen till nästa nod innan armen är framme i nästa nod. För noder servade av arm 2 gäller samma krav med skillnaden att rörelsetiden är $distance_{geo\_loc_n+nGL+nA,geo\_loc_{successor_n}}$ istället. Med $TRAYS$ menas mängden noder i $NODES$ som är tråg. För dessa gäller istället nästa *constraint*.

$\boldsymbol{start\_time_n + service\_time_n + distance_{geo\_loc_n,geo\_loc_{successor_n}} +}$
$\boldsymbol{+camera\_time_{1,n} \leq arrival\_time_{successor_n}} \; \forall\, \boldsymbol{n \in \{TRAYS : arm_n = 1\}}$ - Liknande *constraint* som det ovan, men här gäller det för trågnoder. Skillnaden är att armen måste gå via kameran innan den når nästa nod. Den tidigare uträknade tiden för tråg till kamera rörelsen läggs då till. För arm 2 gäller även här andra index i *distance* samt $camera\_time_{2,n}$.

$\boldsymbol{geo\_loc_i = geo\_loc_j \ \wedge \ i \neq j \Rightarrow departure\_time_j \leq start\_time_i \vee departure\_time_i \leq start\_time_j \quad \forall\, i, j \in NODES}$ - Förhindrar flera armar från att betjäna en geografisk position samtidigt. Notera att matematiskt sägs inget om att två robotarmar inte får befinna sig på samma plats samtidigt. Armarna kan dock inte utföra arbete parallellt.

Det behövs även ett antal *constraints* för att hålla koll på armarnas status medan de rör sig mellan noderna.

$\boldsymbol{gripper\_component_{component_n} \Rightarrow gripper\_occupied_n \wedge suction\_occupied_n = suction\_occupied_{predecessor_n} \quad \forall\, n \in TRAYS}$ - Om ett tråg som innehåller en greppbar komponent besöks kommer greppverktyget bli ockuperat medan sugverktygets status förblir oförändrad. Ett omvänt *constraint* gäller för sugbara komponenter.

$\boldsymbol{gripper\_component_{component_n} \Rightarrow \neg gripper\_occupied_n \wedge suction\_occupied_n = suction\_ocuppied_{predecessor_n} \quad \forall\, n \in FIXTURES}$ - När en fixtur för en greppbar komponent besöks kommer greppverktyget bli ledigt medan sugverktygets status förblir oförändrad. Ett omvänt *constraint* gäller för sugbara komponenter.

$\boldsymbol{holding\_components_{component_n,n} \quad \forall n \in TRAYS}$ - En komponent plockas upp efter att dess trågnod besökts.

$\boldsymbol{\neg holding\_components_{component_n,n} \quad \forall n \in FIXTURES}$ - En komponent hålls inte efter att dess fixturnod besökts.

Constraints som upprätthåller start- och slutvärden. Startnoderna är $START\_NODES = \{nN+1, ..., nN+nA\}$ och slutnoderna $END\_NODES = \{nN+nA+1, ..., nN+2nA\}$.

$\boldsymbol{start\_time_n = arrival\_time_n = 0 \quad \forall\, n \in START\_NODES}$ - Tiden är noll då produktionscykeln startar. Eftersom inget arbete utförs i startnoderna blir även *departure_time* här lika med noll och robotarmarna börjar sin rutt.

$\boldsymbol{arm_n = n - nN \quad \forall\, n \in START\_NODES}$ - Tilldelar startnoderna varsin arm.

$\boldsymbol{arm_n = n - nA - nN \quad \forall\, n \in END\_NODES}$ - Tilldelar slutnoderna varsin arm.

$\boldsymbol{\neg suction\_occupied_n \wedge \neg gripper\_occupied_n \quad \forall\, n \in START\_NODES \cup END\_NODES}$ - Vid start och slut av produktionscyckeln ska robotens verktyg inte vara ockuperade.

$\boldsymbol{\neg holding\_components_{k,n} \quad \forall\, k \in \{1, ..., nC\},\ \forall\, n \in START\_NODES \cup END\_NODES}$ - Inga komponenter bärs i början eller slutet av produktionscyckeln.

4.3.4.4 Objective Function

Det är *objective* funktionen som avgör vad som uppfattas av lösaren som en bra lösning. I denna modell formuleras den som:

$$minimize\,(max(arrival\_time_i)) \quad \forall\, i \in END\_NODES$$

där *EndNodes* är indexen för slutpositionerna $\{2nC + 1, 2nC + 2\}$. Med andra ord är problemet som överlämnas till solvern att minimera den största av armarnas sluttider genom att bestämma värden på *decision* variablerna. Detta utan att bryta något *constraint*.

Det anges även ett antal söknings riktlinjer i samband med *objective* funktionen. Deras uppgift är att vägleda solvern under sitt val av *desicion variables*. Till exempel är det fördelaktigt att testa olika noders *arrival_time* innan *departure_time*. I modellen formulerades dessa riktlinjer på följande sätt.

```
solve :: seq_search
([ int_search([arm[n] | n in NODES], first_fail,
                                    indomain_min, complete),
int_search([successor[n] | n in NODES], first_fail,
                                    indomain_min, complete),
int_search([predecessor[n] | n in NODES], first_fail,
                                    indomain_min, complete),
int_search([arrival_time[n] | n in NODES], first_fail,
                                    indomain_min, complete),
int_search([departure_time[n] | n in NODES], first_fail,
                                    indomain_min, complete),
int_search([start_time[n] | n in NODES], first_fail,
                                    indomain_min, complete),
int_search([wait_time[n] | n in NODES], first_fail,
                                    indomain_min, complete)])
```

När solvern ska bestämma sig för en variabel att testa ett nytt värde på kommer den i första hand ändra på *arms* följt av *successor*, *predecessor*, *arrival_times*, *departure_times*, *start_time* och *wait_time*. Inom varje av dessa arrays väljs enligt *first_fail* det element som har den minsta domän storleken och *indomain_min* betyder att den ska sättas till domänets minsta värde. Om det till exempel ska väljas en ny test variabel och det står mellan $successor_3 \in \{2, 3, 5, 6\}$, $successor_4 \in \{3, 4, 5\}$, och $predecessor_2 \in \{1, 2, 3\}$ så väljs $successor_4$ och sätts till 3.

4.3.5 Förbättring av MiniZinc Modellen

Inom villkorsprogrammering är det svårt att förutsäga hur problem ska modelleras på bästa sätt för att en solver ska kunna hantera problemet snabbast möjligt. Det finns en stor frihet i val av variabler, formulering av villkor, uppställning av *objective function*, med mera. Allmänt kan det sägas att det är fördelaktigt med få *decision variables*, små domän, konsekventa villkorsformuleringar och

användning av globala villkor där det är möjligt [17]. Denna praxis har följts under utvecklingen av modellen utefter bästa förmåga, men det är troligt att den erfarne villkorsprogrammeraren kan tillföra förbättringar.

En aspekt som experimenterades med var val av sökriktlinjer. Till exempel testades olika kombinationer av följande:

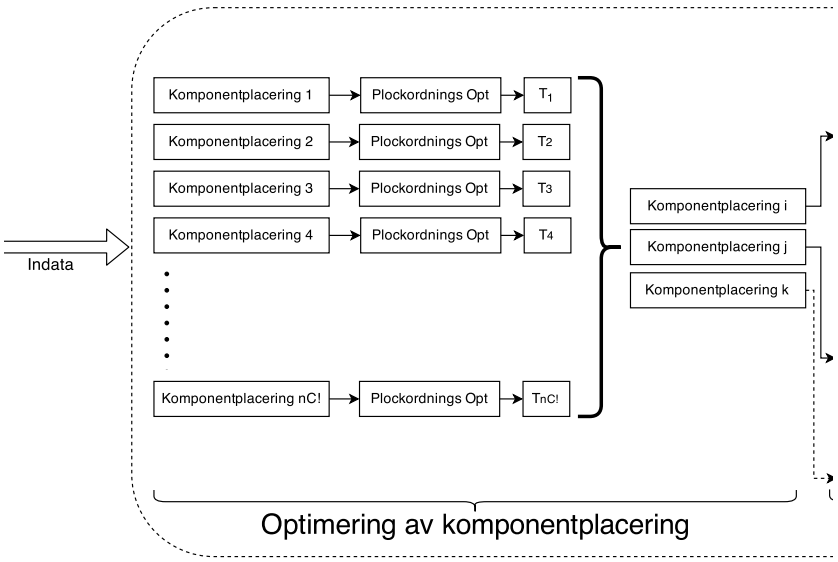
```
solve :: seq_search
([ int_search([arm[n] | n in NODES], first_fail,
                              indomain_random, complete),
int_search([successor[n] | n in NODES], first_fail,
                              indomain_random, complete),
int_search([predecessor[n] | n in NODES], first_fail,
                              indomain_random, complete),
int_search([arrival_time[n] | n in NODES], smallest,
                              indomain_min, complete),
int_search([departure_time[n] | n in NODES], smallest,
                              indomain_min, complete),
int_search([start_time[n] | n in NODES], smallest,
                              indomain_min, complete),
int_search([wait_time[n] | n in NODES], smallest,
                              indomain_min, complete)
```

En skillnad mellan detta och den som slutgiltligen användes (se avsnitt 4.3.4.4) är att här ombedes solvern välja ett slumpmässigt värde på element i *arms*, *successor*, och *predecessor* när en av dessa ska preciseras. Detta kan tyckas vara logiskt eftersom vilken av armarna som besöker en nod i det optimala fallet borde vara slumpmässigt av symmetriskäl och det samma gäller vilken nod som ska besökas efter och före en slumpmässig nod. Här anges även att elementen i tidsvariablerna prioriteras i ordningen av vilken som har det minsta värdet i sitt domän (*smallest*), istället för den som har minsta domänstorleken (*first_fail*) vilket tidigare var fallet.

Liksom tidigare test kring *objective* funktionen i [8] gav denna experimentation tyvärr ytterst otydliga resultat. För ett testfall kunde flerfaldig förbättring i exekveringstid erhållas med koden ovan, samtidigt som den orsakade en lika stor försämring för ett annat testfall. För att avgöra vad som kan anses vara ett optimalt val av sökriktlinjerna måste ett mycket större antal testscenarion undersökas med många upprepade testkörningar för att minimera de statistiska fluktuationerna i exekveringstid som uppstår. En sådan undersökning är dock utanför ramarna av detta projekt.

4.3.6 Komponentplacering

MiniZinc modellen som beskrevs ovan beräknar plockordningen för en fix komponent- och fixturplacering. Antalet möjliga komponentplaceringar i cellen ökar med fakultet gentemot antalet komponenter. Att genomföra hela optimeringsprocessen för alla fall vore mycket tidskrävande och skulle samtidigt innebära att många dåliga alternativ skulle ta upp värdefull beräkningstid.



Figur 12: Inzoomning på komponentplaceringsdelen i figur 8.

För att effektivisera exekveringstiden görs först plockordningsoptimeringen på alla komponentplaceringarna med fixturen preliminärt placerad i mitten av fixturplanet. De tre mest lovande permutationerna väljs ut för att utvärderas vidare i fixturoptimeringen. Flödesschemat för komponentplaceringen visas i figur 12.

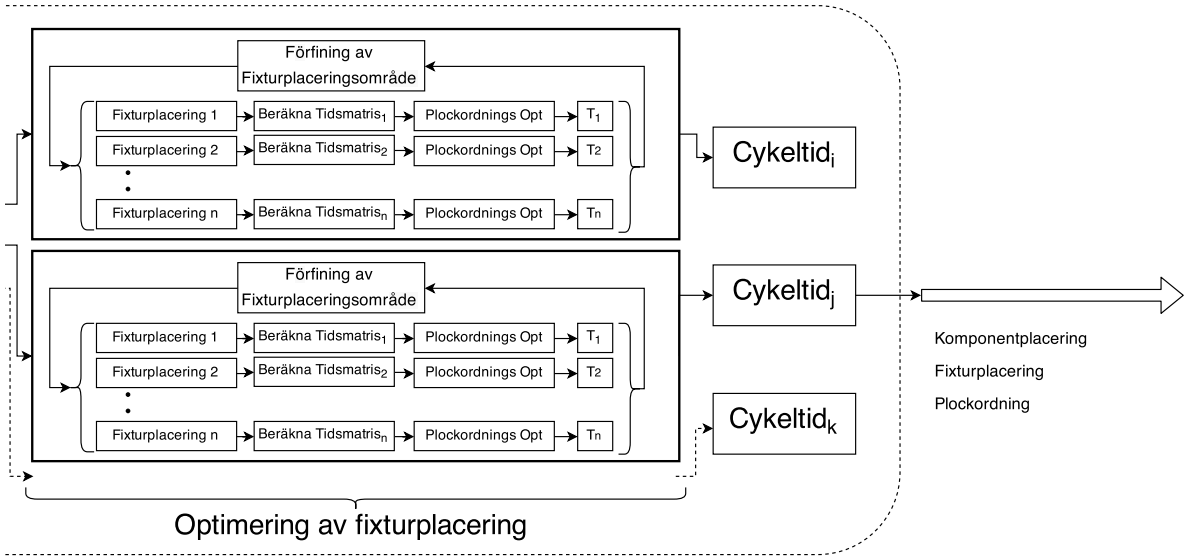
I problemet är N den permuterade vektorn med alla nC komponenter, där varje har ett värde mellan ett och nC.

$$N = \langle n_1, ..., n_{nC} \rangle$$

Villkoren på komponenterna i en permutation N har endast kravet att alla komponenter ska vara olika

$$\forall\, n_i, n_j \in N : i \neq j \Rightarrow n_i \neq n_j$$

Varje gång plockordningsoptimeringen utförs för en ny komponentplacering, behöver indatan till MiniZinc uppdateras. Detta för att VRP:ets noders attributer följer villkoren på komponenterna som är i dem, till exempel servicetiderna och monteringsvillkoren. Permuteras komponentplaceringen måste även denna indata permuteras.



Figur 13: Inzoomning på fixturplaceringsdelen i figur 8.

4.3.7 Fixturplacering

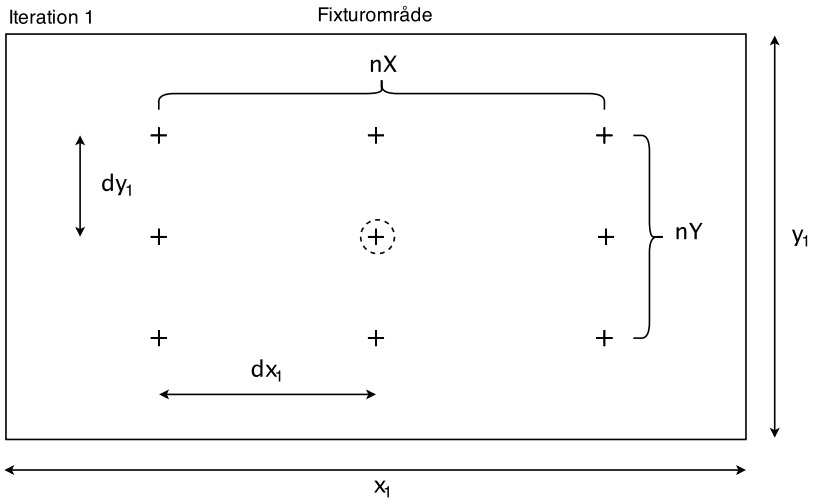
Optimeringen av fixturplaceringen kommer utföras på de tre bästa komponentplaceringarna. Varje komponentplacering går igenom en iterativ process, se figur 13, där det tillåtna fixturområdet och monteringstiden successivt förminskas. Den komponentplaceringen med lägsta monteringstid efter fixturoptimeringen kan därefter väljas ut.

För att hitta en tidsminimerad fixturplacering för en specifik komponentplacering utförs plockordningsoptimeringen flera gånger. Utifrån fixturplanets gränser som användaren specificerat i det grafiska gränssnittet genereras ett jämnt fördelat och lågupplöst nät med $nX * nY$ stycken möjliga fixturplaceringar, se figur 14. Avståndet mellan varje fixturplacering beräknas med ekvationerna 1 och 2.

$$dx_n = \frac{x_n}{nX + 1} \tag{1}$$

$$dy_n = \frac{y_n}{nY + 1} \tag{2}$$

Varje fixturplacering i nätet medför att avståndet och därmed förflyttningstiden från fixturen till andra cellpositioner ändras. Innan en plockordningsoptimering kan göras måste därför en ny tidsmatris beräknas. Tidsmatrisen uppdateras och plockordningsoptimering utförs på alla fixturplaceringar i nätet.



Figur 14: Första iterationen i fixturoptimeringen. Varje kryss representerar en fixturplacering. Den streckade ringen är den fixturplacering som presenterar bäst resultat i iterationen.

Kring den fixturplacering som ger bäst tidsförbättring förfinas rutnätet. Detta görs genom att uppdatera det förfinade rutnätets höjd och bredd, enligt ekvation 3 och 4. När höjden och bredden är uppdaterad kan ekvation 1 och 2 beräknas på nytt och $nX * nY$ nya fixturplaceringar genereras kring den placering som presenterade bäst resultat, se figur 15.

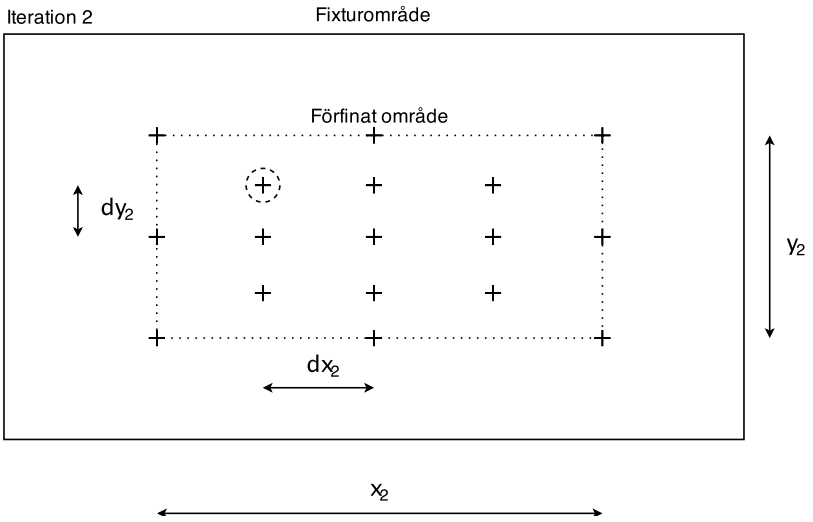
$$x_n = dx_{n-1} \cdot 2 \tag{3}$$

$$y_n = dy_{n-1} \cdot 2 \tag{4}$$

Optimeringen fortgår sedan på samma sätt tills varje iteration bara medför att fixturen förflyttas 15 mm, beräknad med ekvation 5.

$$\sqrt{dx_n^2 + dy_n^2} < 15mm \tag{5}$$

Flera tester, se kapitel 6, utfördes för att bestämma lämpliga värden på nX och nY. Eftersom det existerar ett flertal frihetsgrader i optimeringsproblemet är det svårt att förutsäga hur tiden kommer variera på fixturplanet. Tiden bestäms bland annat av plockordningen, ledkonfigurationer och förflyttningssträckan. Alla dessa faktorer gör att tiden inte varierar linjärt på fixturområdet. För att tillvägagångssättet ska lyckas krävs det därför att det första nätet är fint nog för att inte exkludera något område samtidigt som det är tillräckligt lågupplöst för att inte behöva köra onödigt många plockordningsoptimeringar per iteration. Att konstanterna nX och nY är udda säkerställer att den ursprungliga fixturplaceringen i mitten av fixturplanet finns kvar i det första rutnätet och även att



Figur 15: Andra iterationen i fixturoptimeringen. Ett förfinat nät har genererats kring den fixturplacering som presenterade bäst resultat i föregående iteration.

den fixturplaceringen som väljs för att göra ytterligare en iteration kring har kvar fixturplaceringen som valdes. Det medför att tiden för en produktionscykel bara kan bli antingen bättre eller oförändrad. Det bedömdes därför att ett lämpligt värde på båda konstanterna nX och nY är 3. Testfallen visade dessutom att ingen märkbar tidsförbättring tillkom efter det att fixturen flyttades mindre än 15 mm. Fixturoptimeringen är därför utförd när villkor 5 uppfylls.

4.4 Simulering

När optimeringen är utförd ska ett robotprogram skapas för varje robot utifrån de sekvenser som beräknats. Plockmetoder det vill säga hur komponenterna plockas genereras inte i DAC. Istället läggs den angivna servicetiden in i robotprogrammet som pauser för att representera tiden det tar att utföra en plockmetod.

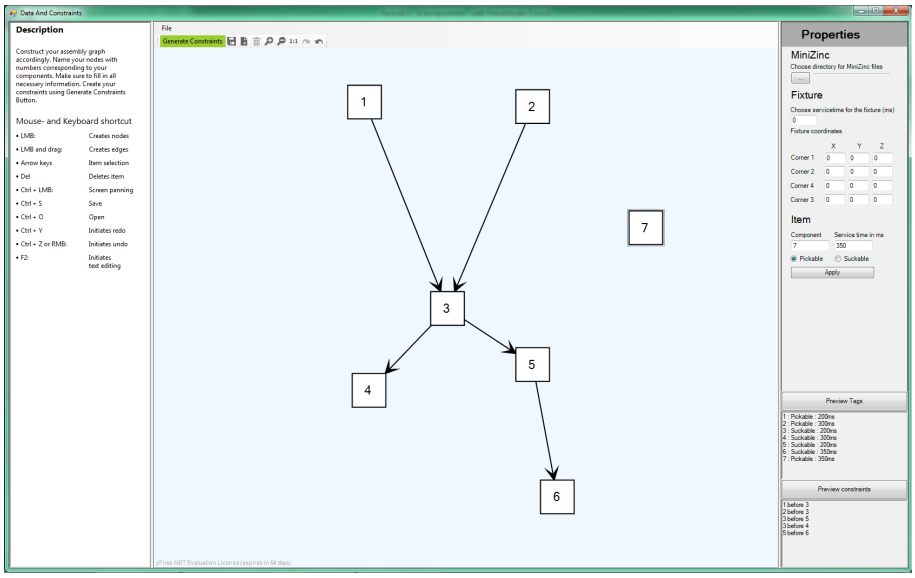
I detta stadie finns en genererad sekvens för varje robot som är redo att simuleras i RobotStudio. Det behöver manuellt undersökas om eventuella kollisioner uppstår då det inte har tagits hänsyn till dynamiska kollisioner vid framtagningen av sekvenserna. Uppstår det kollisioner kan användaren lösa det genom att lägga in pauser för en av armarna för att den andra armen ska hinna förflyttas och inte utgöra ett hinder. Alternativt kan viapunkter införas för att modifiera robotarmens bana för att undvika kollisioner.

När programmet går att köra utan kollisioner kommer användaren manuellt implementera plockmetoder som ska bytas ut mot de redan implementerade fördröjningstiderna.

5 Dual Arm Coordinator : ett RobotStudio Add-In

För att DAC ska kunna möjliggöra framtagningen av robotprogram direkt i RobotStudio behövs det manuellt förarbete. När detta är genomfört löser DAC beräkningarna automatiskt. Det manuella arbetet handlar om att användaren behöver konfigurera en robotcell i RobotStudio och därefter modellera produktionskraven i ett gränssnitt. Till exempel behövs en monteringsgraf skapas för att genera monteringskraven, se figur 16.

När alla krav är uppfyllda och robotarna är rätt konfigurerade startas DAC genom Execute i den implementerade fliken Dual-Arm Coordinator i RobotStudio. Under tiden som DAC körs uppdateras användaren kontinuerligt i RobotStudios terminal. Där får användaren reda på vilken del av processen som för närvarande beräknas. När processen är färdig presenteras de framräknade och simulerbara robotprogrammen för varje robot direkt i RobotStudio. Har kollisioner uppstått kan de lösas här och plockmetoderna implementeras för att sedan överföras till de fysiska robotarna.

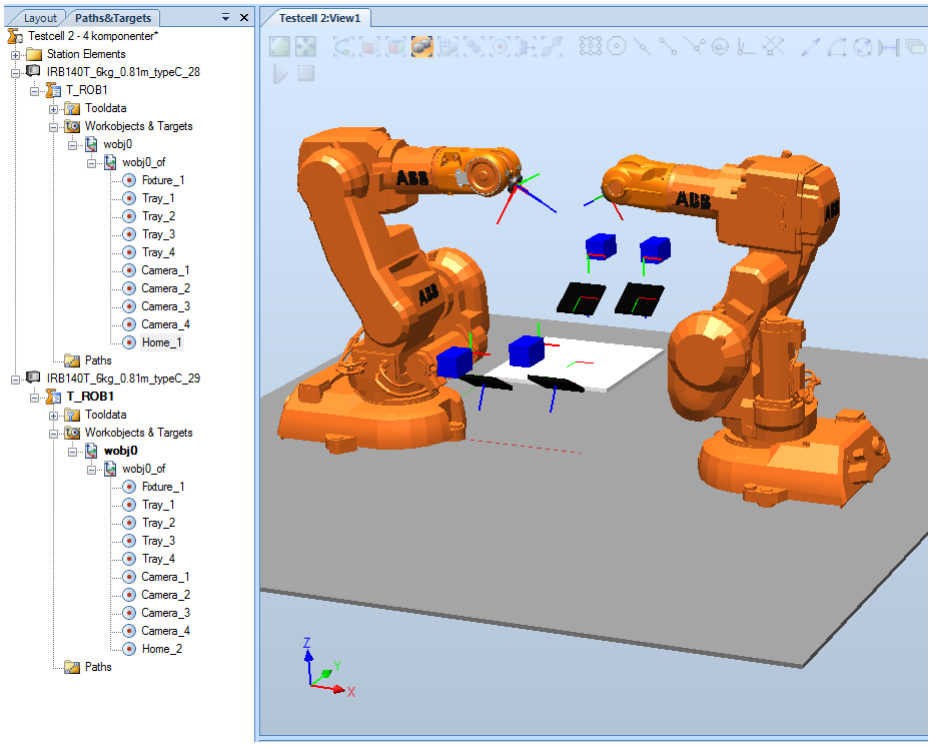


Figur 16: Figuren visar det grafiska användargränssnittet och dess funktionalitet, samt en uppritad monteringsgraf. Pilarna pekar i den ordning komponenterna ska monteras. Komponent ett och två måste monteras innan komponent tre och så vidare. Komponent sju är en fristående komponent som är monteringsoberoende och kan monteras när som helst.

5.1 Robotkonfigurationer och inställningar för robotarmarna i robotstudio

När två enarmade robotar tillämpas behövs ett gemensamt koordinatsystem specificeras. Detta för att användaren ska kunna ange fixturplanet utifrån en gemensam referenspunkt. Användaren gör det möjligt genom att sätta robotarnas *Task Frame* i RobotStudio i origo för det globala koordinatsystemet.

DAC behöver även känna till antalet tråg, kameror och fixturer. Därför krävs det att användaren döper samtliga targets enligt: Fixture_x, Tray_x, Home_x och Camera_x, där x börjar på siffran ett och ökar stegvis för varje objekt, som visas till vänster i figur 17.



Figur 17: Figuren visar två IRB 140T robotar, fyra tråg, fyra kameror och en fixtur. Figuren visar även hur varje robot har definierade targets som namngetts för att DAC ska kunna särskilja dem.

5.2 DAC:s gränssnitt Data And Constrains

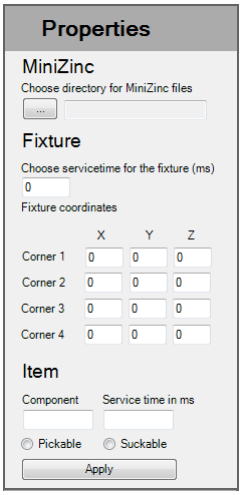
Krav som inte kan definieras genom konfigurationer i RobotStudio samt de krav som inte automatiskt kan avläsas av DAC, anges i DAC:s gränssnitt. Som visas i figur 18.

För att DAC automatiskt ska kunna anropa MiniZinc-modellen i optimeringsmetoderna krävs det att det är installerat och att användaren anger den lokala sökvägen till MiniZinc.

Fixturens servicetime behöver också definieras. Till skillnad från komponenternas individuella servicetid, som definieras för varje komponent, kommer användaren behöva definiera en allmän och gemensam servicetid för monteringen i fixturen.

Fixturplanets område är en annan parameter som DAC behöver ha kännedom om. Användaren anger x,y,z-koordinaterna i tur och ordning för områdets fyra hörn.

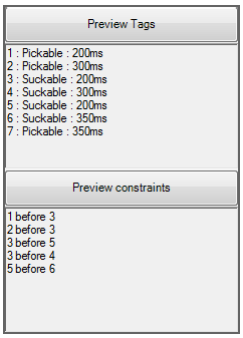
Varje komponent måste också specificeras med tre uppgifter. Dels ska komponenterna numreras för att monteringsgrafen ska kunna genereras. Dels ska användaren ange komponenternas individuella servicetime och plockmetod, pickable eller suckable. Som figur 5.3 visar kan användaren få en överblick på alla krav som definierats. När användaren definierat all data för varje komponent, skapas avslutningsvis alla krav genom Generate Constraints i menyn.



Figur 18: Figuren visar Properties-fältet och är en inzoomning av DAC:s gränssnitt.

5.3 Monteringsgrafer

Största fördelen med det grafiska gränssnittet är den överskådliga inmatningen av monteringskraven, se figur 16. Användaren skapar sin monteringsgraf, som innebär att alla komponenter visualiseras i en graf där det specificeras i vilken ordning komponenterna måste monteras i fixturen. Skulle det vara komponenter som inte är beroende av andra komponenter placeras den ut i grafen utan några pilar, då tar optimeringen hänsyn till det och placerar in den där det är mest passande för att få en effektiv användning av robotarmarna. Det är en stor fördel för användaren att kunna visualisera sina monteringskrav eftersom att det är betydligt lättare att kontrollera sin inmatning och går snabbare att rita upp en graf, än att exempelvis ange monteringskraven i en textfil. Graferna går även att sparas ner för att undvika att skapa en helt ny vid nästa körning.



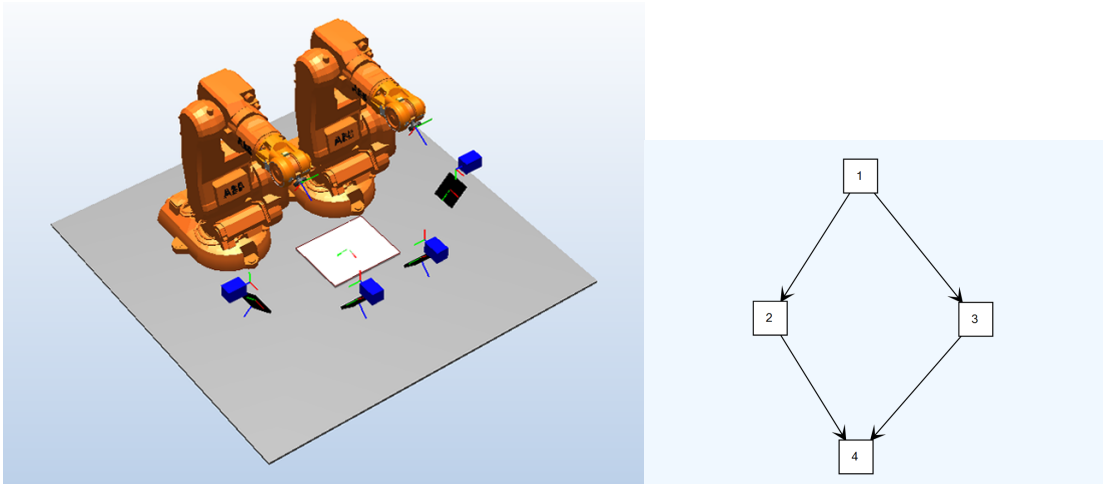
Figur 19: Figuren visar knapparna PreviewTags och PreviewConstraint. De tillåter en förhandsgranskning över de angivna kraven.

6 Utvärdering

För att utvärdera DAC:s resultat och se hur olika produktionsscenarion hanteras har tre olika cellkonfigurationer skapats. DAC är konstruerat för att vara generellt och klara av att hantera celler med olika utformning. De tre testfallen är skapade för att utvärdera hur bra resultatet blir med avseende på variation i cellkonfiguration och monteringskrav. Servicetider har satts inom samma storleksordning som rörelsetiderna för att skapa intressanta testscenarion.

6.1 Testfall 1

Testfall 1 ställdes upp för att emulera en möjlig installation av en tvåarmad robot. Två IRB 140T är placerade bredvid varandra där de ska efterlikna en tvåarmad robot, se figur 20(a). Cellen består av fyra tråg med tillhörande kameror. Det tillåtna området för fixturens placering har angetts till en rektangulär yta framför robotarna. Varje robot kan nå tre av de fyra trågen. Av de fyra komponenterna har två angetts till plockbara, medan de andra två kräver sugverktyget. Monteringskraven är angivna enligt monteringsgrafen i figur 20(b). För mer detaljerad information angående testfallet se bilaga A.



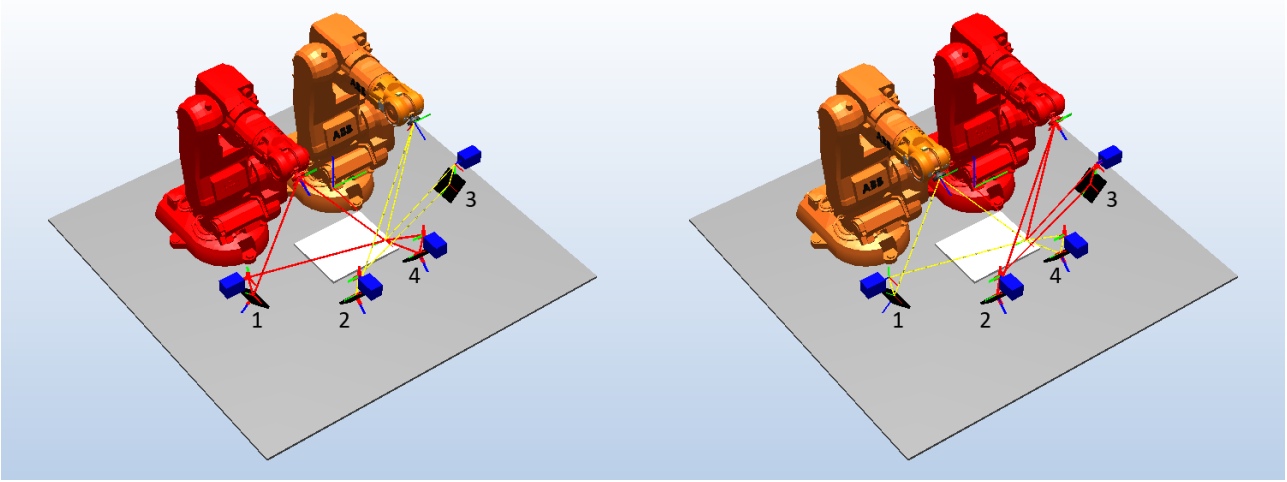
(a) Cellkonfiguration för testfall 1. Den vänstra roboten benämns med robot A och den högra med robot B.

(b) Monteringsgraf för testfall 1.

Figur 20: Övergripande information för testfall 1.

I figurerna 21(a) och 21(b) syns de genererade robotprogrammen. Robot A börjar med att hämta komponent ett och monterar den i fixturen. Samtidigt hämtar robot B komponent två och tre som ska monteras efter komponent ett.

När robot A har monterat komponent ett plockar den komponent fyra medan robot B monterar komponent två och tre i fixturen. Precis efter att robot B lämnar fixturen monterar robot A komponent fyra.



(a) Resulterande robotprogram för robot A.

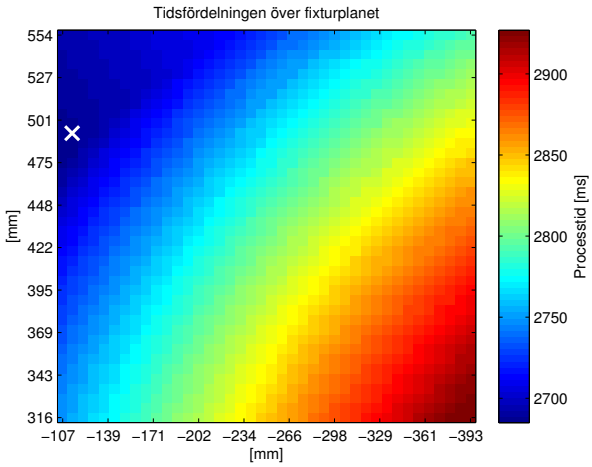
(b) Resulterande robotprogram för robot B.

Figur 21: Framräknade robotprogram för testfall 1. Siffrorna brevid trågen visar hur komponenterna har placerats.

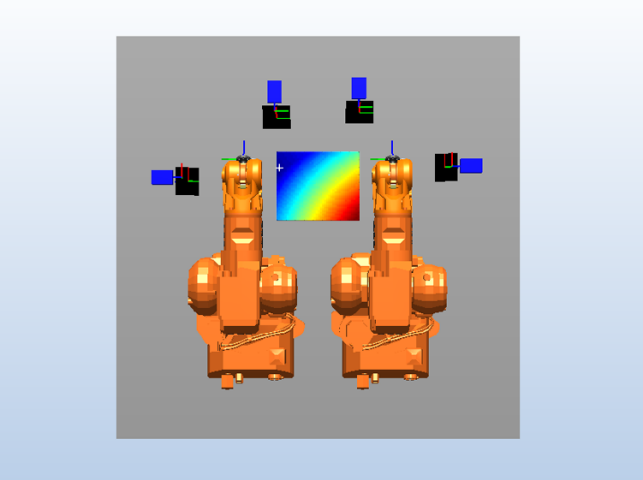
Figur 22(a) och 22(b) visar hur tiderna varierar beroende på var fixturen placeras på fixturplanet. Det vita krysset som syns på figur 22(a) är där DAC beräknat att fixturens mittpunkt ska placeras efter fixturoptimeringen, som tidigare var positionerad i mitten av fixturen. Alltså är det där monteringen kommer utgå ifrån. Av de tre komponentplaceringarna som utvecklades ytterligare i fixturoptimeringen, var det inte den preliminärt bästa komponentplaceringen som resulterade i det bästa resultatet efter fixturoptimeringen.

Gantt-schemat för testfall 1 i figur 23 visar hur de beräknade förflyttningstiderna förhåller sig till de simulerade tiderna i RobotStudio. Som figuren visar är den beräknade tiden betydligt kortare än den simulerade. Det beror på problematiken med tidsberäkningen beskriven i 4.2.3 samt att det uppstår fördröjningar i RobotStudio när simuleringen körs.

Trots att tidsberäkningen bara är en approximation befinner sig robotarna aldrig matematiskt i samma nod under simuleringen. En kollision uppstår dock när robot B lämnar fixturen samtidigt som robot A anländer några millisekunder senare. I verkligheten tar det en viss tid för roboten att flytta sig ifrån noden på grund av sin volym. Kollisionerna går i det här fallet inte att undvika genom att sätta in pauser i rutten. Komponent fyra som monteras sist ligger i tråget närmast fixturen. Robot B måste ha monterat komponent två och tre innan robot A kan röra sig från kameran till fixturen med komponent fyra. Eftersom tråget ligger nära fixturen kommer alltid robotarna kollidera oberoende



(a) Figuren representerar hur fixturtiden varierar på det angivna fixturplanet för testfall 1. Figuren genererades genom att köra 1600 plockordningsoptimeringar uniformt fördelade på fixturplanet. Det vita korset representerar den slutgiltiga fixturplaceringen som uppnåddes efter 27 plockordningsoptimeringar.



(b) Hur tiden varierar på fixturplanet tillsammans med testcell 1.

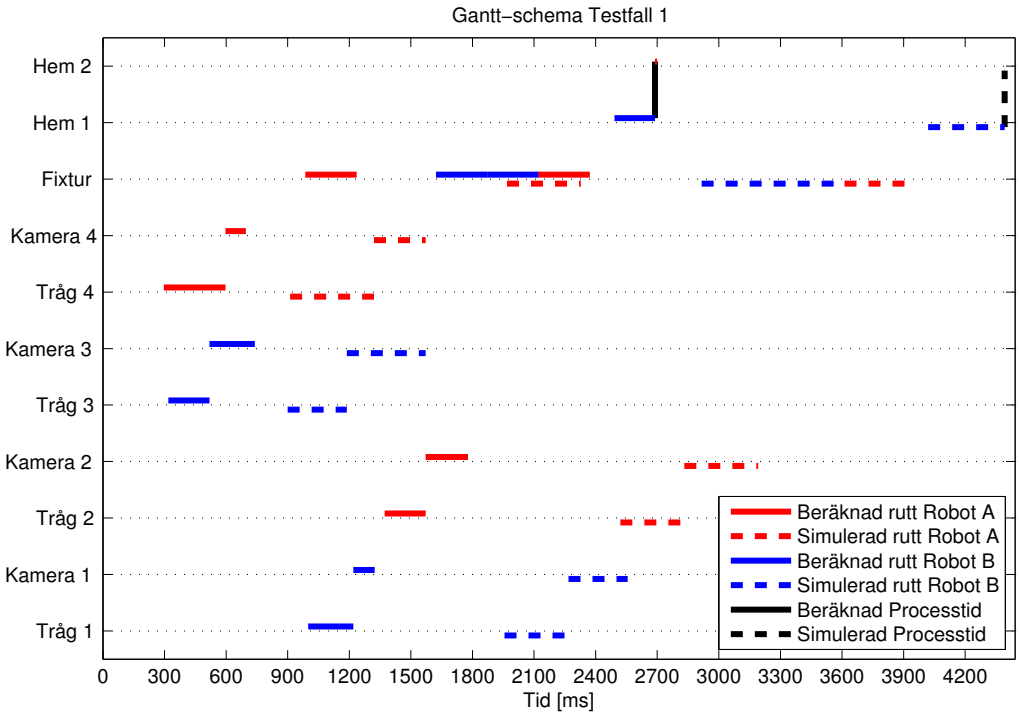
Figur 22: Fixturoptimeringens resultat.

av hur länge robot A måste vänta i kameran. För att lösa kollisionen behövs viapunkter manuellt läggas in från kameran till fixturen som flyttar på robot A och tillåter robot B att arbeta i fixturen.

Den optimeringsdel som tog längst tid i det här fallet var fixturoptimeringen, se tabell 1. Det beror på att beräkning av tidsmatrisen tar längre tid än plockordningsoptimeringen. Eftersom fixturen är fixerad under komponentplaceringen behövs tidsmatrisen bara beräknas en gång. Varje gång fixturen flyttas vid fixturoptimeringen behövs en ny tidsmatris beräknas. Det gör att det tar noterbart längre tid att utföra fixturoptimeringen än komponentplaceringen.

Tabell 1: Tider för testfall 1.

Optimeringstider	
Tid per plockordningsoptimering	0.33 s
Tid per tidsmatrisberäkning	3 s
Komponentplacering för 24 permutationer	16 s
Fixturoptimering för de tre bästa komponentplaceringarna	6 min 20 s
Totaltid för optimeringen	6 min 36 s

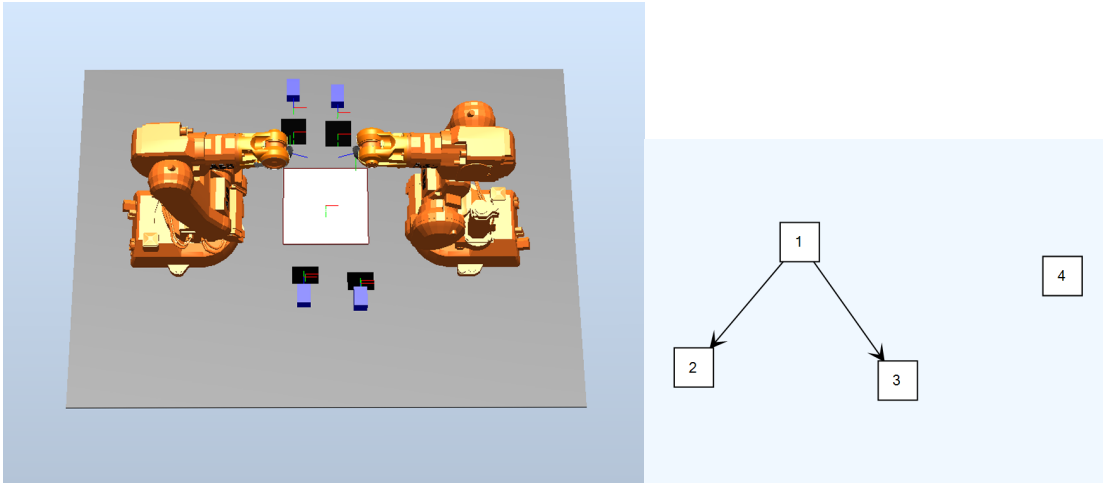


Figur 23: Gantt-schema för testfall 1. Schemat visar hur den beräknade rutten förhåller sig till den simulerade rutten i RobotStudio. Varje streck visar under vilken tid en robot befinner sig i respektive position. Tiden emellan positionerna visar tiden det tar att flytta sig mellan positionerna.

6.2 Testfall 2

I testfall 2 står två IRB 140T robotar på var sin sida av fixturplanet. I cellen finns fyra tråg, fyra kameror och en fixtur, se figur 24(a). Båda robotarna kan nå alla tråg. För mer detaljerad information angående testfallet se bilaga B. Monteringskraven är specificerade i figur 24(b). Genom att komponent fyra är fristående i grafen tillåts den monteras när som helst, den är därför inte bunden till de övriga komponenterna. Det tillåtna området för fixturens placering har angetts till en rektangulär yta mellan robotarna. Av de fyra komponenterna har två angetts till plockbara, medan de andra två kräver sugverktyget.

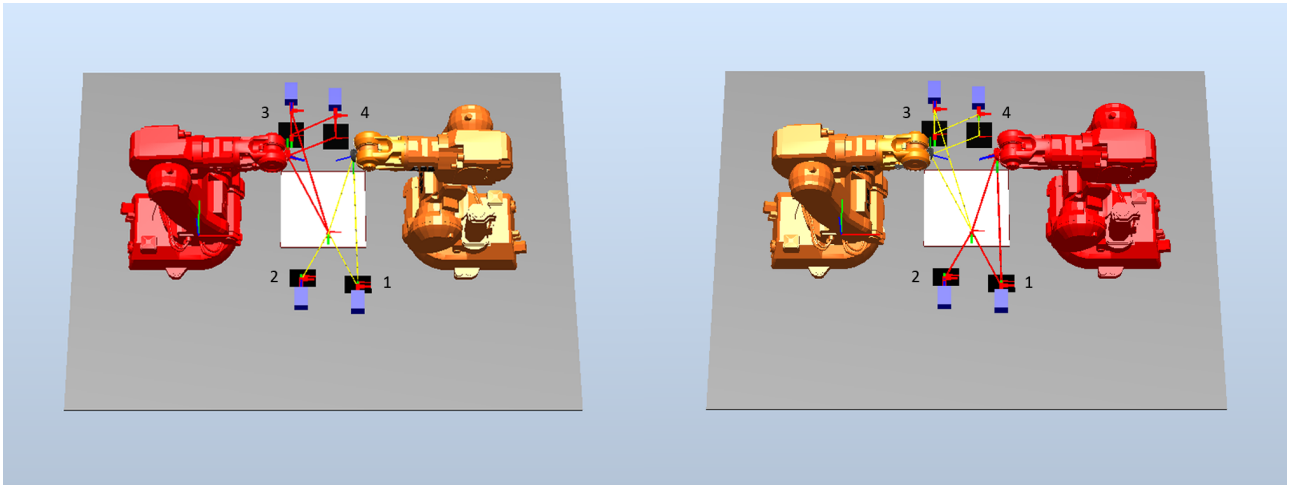
Optimeringen resulterade i banorna visualiserade i figur 25(a) och 25(b). Robot B börjar med att hämta komponent ett för att direkt placera den i fixturen. Samtidigt plockar robot A komponent tre och fyra. Eftersom robot A måste röra sig en längre sträcka hinner robot B flytta sig ifrån fixturen. Medan robot A monterar komponent tre och fyra i fixturen plockar robot B komponent två. Robotprogrammet är färdigt när robot B har monterat komponent



(a) Cellkonfiguration för testfall 2. Den vänstra roboten benämns med robot A och den högra med robot B.

(b) Monteringsgraf för testfall 2.

Figur 24: Övergripande information för testfall 2.



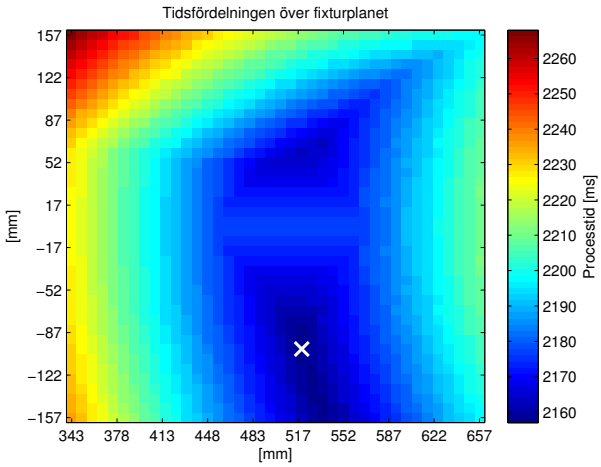
(a) Resulterande robotprogram för robot A.

(b) Resulterande robotprogram för robot B.

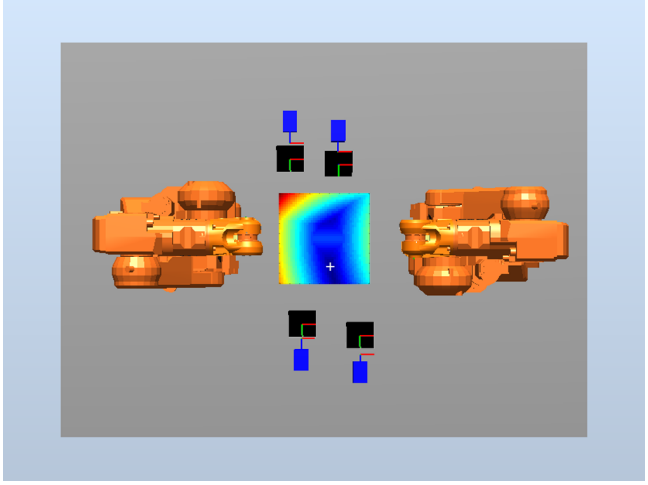
Figur 25: Framräknade robotprogram för testfall 2. Siffrorna bredvid trågen visar hur komponenterna har placerats.

två i fixturen. Plockordning medför att robotarna kan arbeta parallellt vilket resulterar i ett effektivt robotprogram.

Figur 26(b) och 26(a) visar hur tiderna varierar beroende på vart fixturen



(a) Figuren representerar hur fixturtiden varierar på det angivna fixturplanet för testfall 2. Figuren genererades genom att köra 1600 plockordningsoptimeringar uniformt fördelade på fixturplanet. Det vita korset representerar den slutgiltiga fixturplaceringen som uppnåddes efter 27 plockordningsoptimeringar.



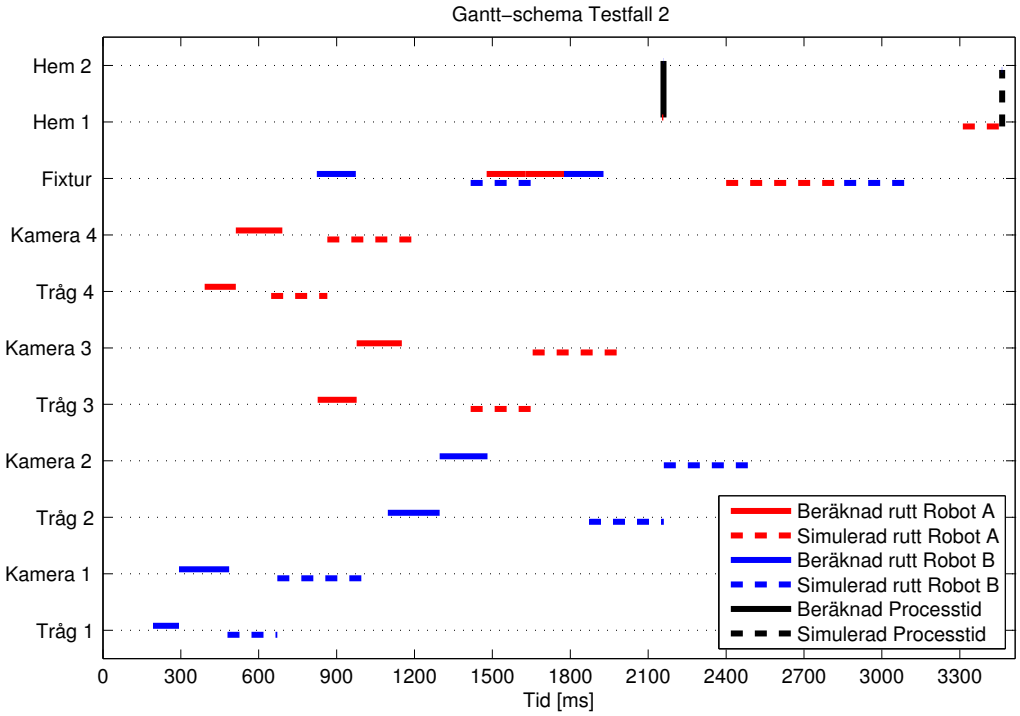
(b) Hur tiden varierar på fixturplanet tillsammans med testcell 2.

Figur 26: Fixturoptimeringens resultat.

placeras på fixturplanet. Fixturoptimeringsmetoden träffade den bästa fixturplaceringen efter tre iterationer. Processtiden bestäms av den robot som är klar sist. I det här fallet är det robot B. Robot B plockar komponent två och placerar den sist i fixturen. Den slutgiltiga fixturplaceringen är en övervägning mellan tiden det tar att placera komponent två i fixturen och att gå tillbaka till hempositionen från fixturen. Fixturen placeras därför mellan trågets tillhörande komponent två och robot B:s hemposition.

Rutten som respektive robot tar beskrivs i figur 27. Precis som i testfall 1 uppstår det en fördröjning i RobotStudio för de simulerade robotprogrammen. Även i detta testfall befinner sig aldrig robotarna matematiskt på samma ställe. En kollision uppstår även här när robot A inte hinner flytta sig ifrån fixturen. I det här testfallet kan kollisionen undvikas genom att låta robot B vänta 500 ms innan den anländer till fixturen och monterar komponent två. Detta medför dock att produktionens cykeltid förlängs med 500 ms.

Den optimeringsdel som tog längst tid var precis som i testfall 1 fixturoptimeringen. Även fast båda fallen innefattade fyra komponenter återspeglas den ökade komplexiteten i totaltiden för optimeringen. I skillnad från testfall 1 kan båda robotarna nå alla tråg. Det gör att komplexiteten i problemet ökar eftersom att det finns fler möjligheter att plocka komponenterna.



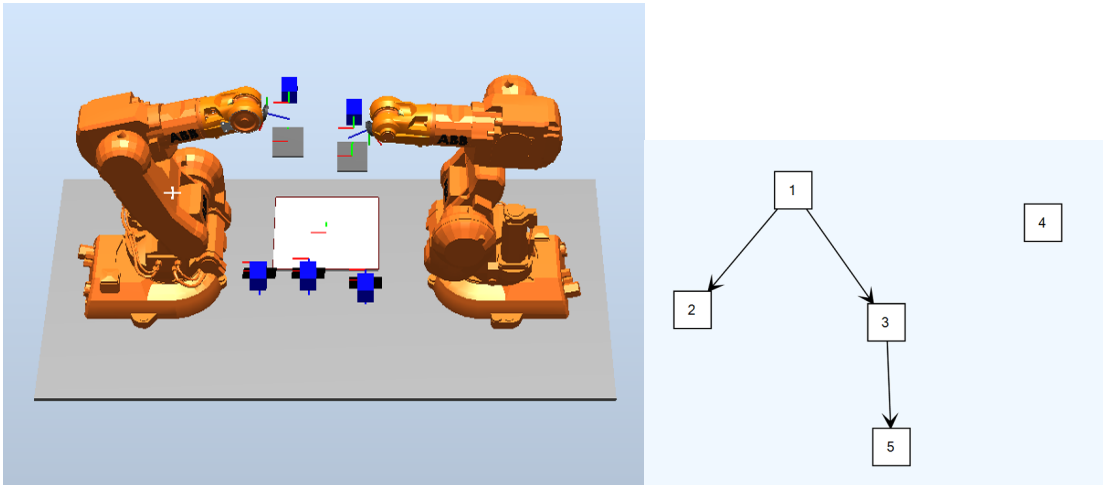
Figur 27: Gantt-schema för testfall 2. Schemat visar hur den beräknade rutten förhåller sig till den simulerade rutten i RobotStudio.

Tabell 2: Tider för testfall 2.

Optimeringstider	
Tid per plockordningsoptimering	0.8 s
Tid per tidsmatrisberäkning	5.8 s
Komponentplacering för 24 permutationer	27 s
Fixturoptimering för de tre bästa komponentplaceringarna	9 min 15 s
Totaltid för hela optimeringen	9 min 42 s

6.3 Testfall 3

För att undersöka hur totaltiden för optimeringen ökar när komplexiteten ökar introducerades en extra komponent till testfall 2, se figur 28(a). Båda robotarna kan precis som i testfall 2 nå alla tråg. Byggkraven ställdes upp enligt figur 28(b). Tre komponenter använder plockverktyget medan två använder sugverktyget. För mer detaljerad information om testfallet se bilaga C.



(a) Cellkonfiguration för testfall 3. Den vänstra roboten benämns med robot A och den högra med robot B.

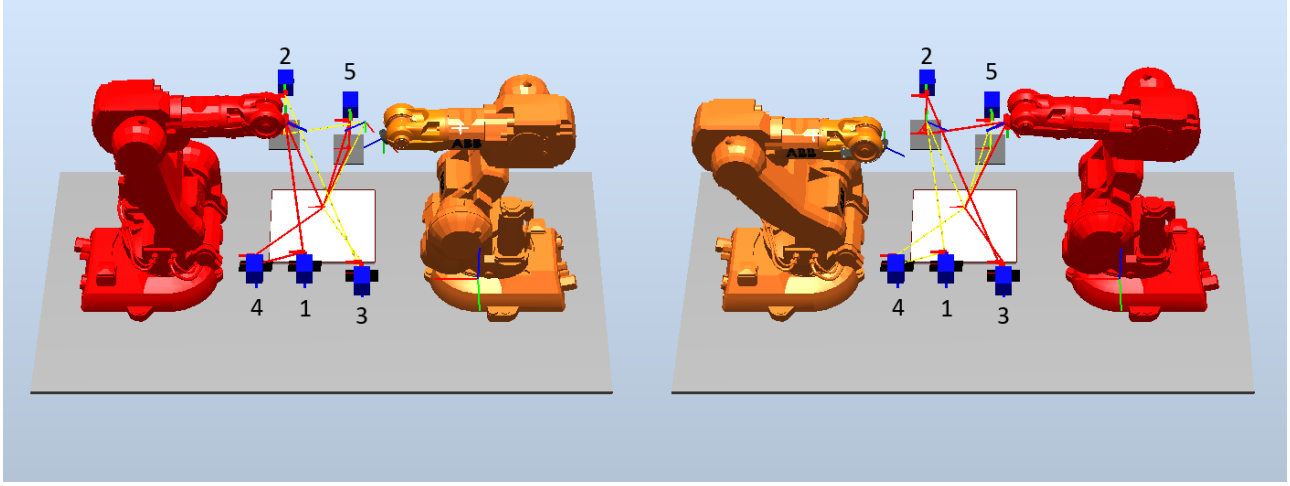
(b) Monteringskrav för testfall 3.

Figur 28: Övergripande information för testfall 3.

Sekvensen som respektive robot tar visas i figur 29(a) och 29(b). Robot A plockar komponent ett och fyra för att direkt placera dem i fixturen. Samtidigt plockar robot B komponent två och tre. Komponent två och tre är placerade på motsatt sida av fixturen. Det gör att robot B måste göra en stor förflyttning vilket medför att robot A hinner flyttas från fixturen lagom tills robot B ska montera i fixturen. Medan robot B monterar i fixturen plockar robot A den sista komponenten och monterar den i fixturen precis efter robot B börjar flytta sig ifrån fixturen. Banorna medför att robotarna arbetar parallellt och effektivt utan väntetider.

Figur 30(a) och 30(b) visar hur tiderna varierar beroende på var fixturen placeras på fixturplanet. Fixturoptimeringsmetoden träffade den bästa fixturplaceringen efter tre iterationer precis som i de andra testfallen. Processtiden bestäms av den robot som är klar sist. Precis som i testfall 2 är det robot A som plockar den sist komponenten. Den slutgiltiga fixturplaceringen är en övervägning mellan tiden det tar att placera komponent fem i fixturen och att gå tillbaka till hempositionen från fixturen.

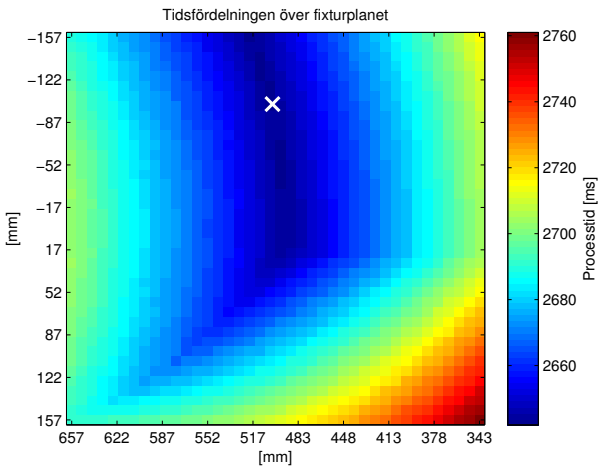
Figur 31 visar den resulterade sekvensen. Simuleringen i RobotStudio påvisar även här en fördröjning och felet i tidsberäkningen återspeglas tyd-



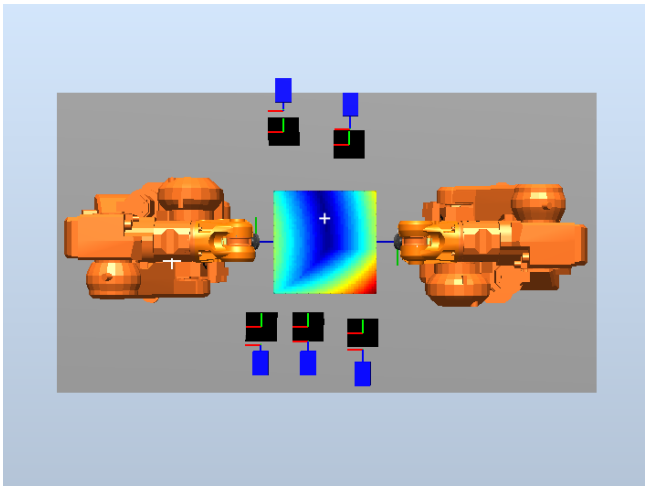
(a) Resulterande robotprogram för robot A.

(b) Resulterande robotprogram för robot B.

Figur 29: Framräknade robotprogram för testfall 3. Siffrorna bredvid trågen visar hur komponenterna har placerats.

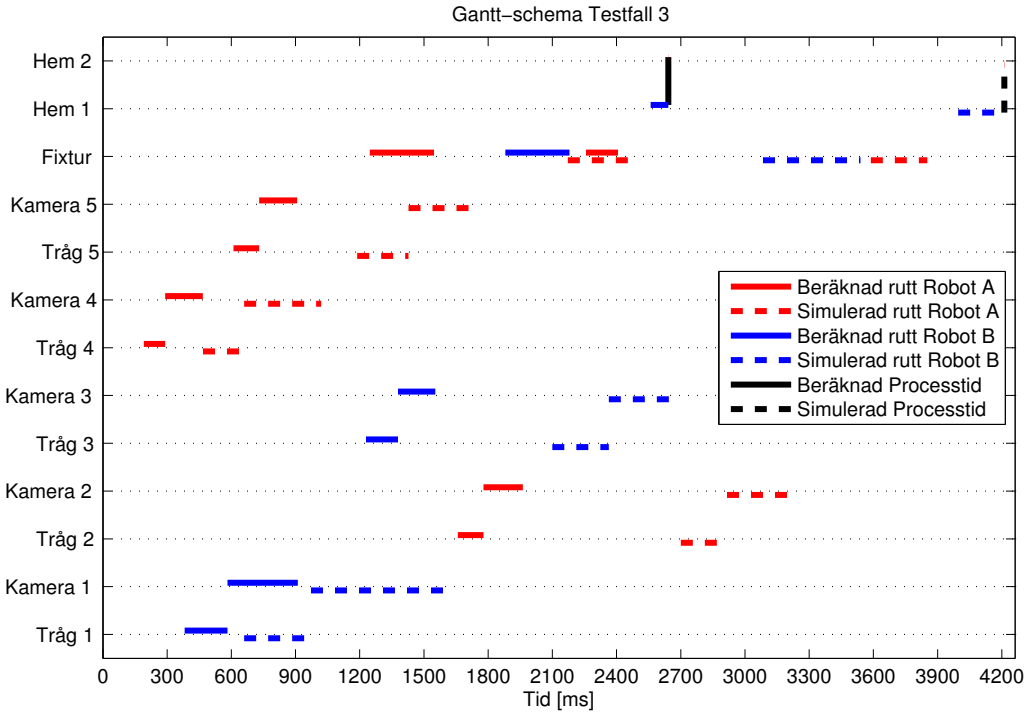


(a) Figuren representerar hur fixturtiden varierar på det angivna fixturplanet för testfall 3. Figuren genererades genom att köra 1600 plockordningsoptimeringar uniformt fördelade på fixturplanet. Det vita korset representerar den slutgiltiga fixturplaceringen som uppnåddes efter 27 plockordningsoptimeringar.



(b) Hur tiden varierar på fixturplanet tillsammans med testcell 3.

Figur 30: Fixturoptimeringens resultat.



Figur 31: Gantt-schema för testfall 3. Schemat visar hur den beräknade rutten förhåller sig till den simulerade rutten i RobotStudio.

ligt. I kamera ett jobbar roboten ovanligt länge som en följd av den förbestämda ledkonfigurationen har valts till ett olämpligt värde.

En kollision uppstår när robot B plockar komponent två för att sedan plocka komponent tre på andra sidan av fixturen samtidigt som robot A monterar i fixturen. Kollisionen kan undvikas med en viapunkt genom att styra undan robot B när den rör sig över fixturplanet. Viapunkten medför dock att förflyttningstiden blir längre som i sin tur medför att de två robotarna krockar i fixturen. Kollisionen undviks genom att låta robot A vänta med att montera i 900 ms. Utförs den manuella kollisionshanterigen på detta sätt ökar då produktionscykeltiden med 900 ms.

Ökningen av antalet komponenter bidrar till en ökning av tiden som krävs för att utföra optimeringsprocessen. Att problemets komplexitet ökar med flera komponenter återspeglar sig i plockordningsoptimeringens tiodubblade tidsåtgång. Dock är det tiden för komponentplaceringen som växer snabbast, vilket är väntat på grund av problemets fakultetskaraktär. Noterbart är hur tiden för en tidsmatrisberäkning inte påverkas i samma utsträckning.

Tabell 3: Tider för testfall 3.

Optimeringstider	
Tid per plockordningsoptimering	8 s
Tid per tidsmatrisberäkning	7 s
Komponentplacering för 120 permutationer	15 min 15 s
Fixturoptimering för de tre bästa komponentplaceringarna	20 min 22 s
Totaltid för hela optimeringen	35 min 48 s

6.4 Intryck från Testfallen

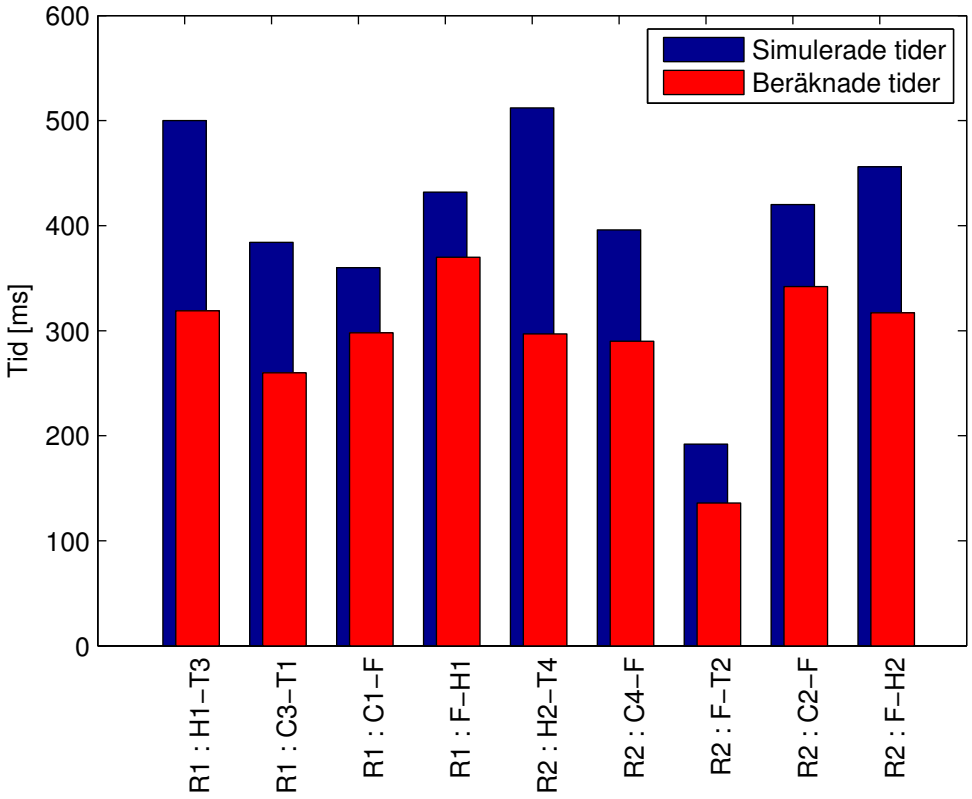
Testfallen ger en inblick för metodens för- och nackdelar. DAC:s styrka är att användaren får en matematisk grundad placering av komponenterna och fixturen samt i vilken ordning komponenterna ska plockas direkt i RobotStudio. Banorna som genereras är inte felfria, men eftersom de genereras direkt i RobotStudio minimeras det manuella arbetet som krävs för att få dem realiserbara.

I alla testfall uppstod kollisioner under simuleringen av den resulterande sekvensen. Genom att lägga till pauser eller viapunkter kunde kollisionerna undvikas, men följden är att det matematiska underlaget för robotprogrammet försvinner. Överlag påvisar testfallen den starka tendensen till kollisioner när två robotarmar ska samspela i en produktionscell.

Testfallen kördes bara med fyra och fem komponenter. Skillnaden i optimeringstiden i testfall två och tre påvisar att metoden inte är lämpad för scenarion med många komponenter. Tiden som krävs för att utföra komponentplaceringsoptimeringen ökar i en avsevärd takt när antalet komponenter stiger.

I testfallen observerades att en fördröjning uppstod när de framräknade banorna simulerades i RobotStudio, se figur 23, 27 och 31. Konsekvent för alla

testfall var att robotarna stod stilla i 400 ms innan de började röra sig. Detta påverkar dock inte banorna utan fördröjer bara processtiden. Kompenseras den simulerade processtiden med 400 ms kan det observeras att den beräknade processtiden fortfarande inte stämmer överens med den simulerade. Ett stort bidrag till detta är tidsberäkningsmetoden som används. Som figur 32 visar uppstår tydliga missvisningar i tidsapproximationerna. Alla rörelser tog längre tid i simuleringen än vad som räknats med under optimeringen, vilket bidrog till en längre produktionscykeltid under simuleringen.



Figur 32: Jämförelse mellan beräknade och simulerade tider för olika rörelser i testfall 1. RX står för robot X, TX för tråg X, F för fixtur, CX för kamera X och HX för hemposition X.

7 Diskussion

Arbetet resulterade i ett Add-In till RobotStudio. Genom ett grafiskt gränssnitt tillåts användaren modellera ett allmänt produktionsscenario för två robotarmar. Villkorsprogrammering appliceras för att optimera produktionscykeln med avseende på plockordning, komponenternas placering i trågen och fixturens placering. För att genomföra optimeringen krävs kunskap om hur lång tid det tar för robotarmarna att utföra olika rörelser inom cellen. Dessa tider approximeras genom att räkna på armledernas vinkeldifferenser i förhållande till deras maxhastighet. Utifrån resultatet från optimeringen genereras preliminära robotprogram som användaren får korrigera för att hantera eventuella kollisioner.

Styrkan med DAC är den matematiska grunden till ett effektivt robotprogram, samt att processen tar relativt kort tid i jämförelse med dagens manuella arbetssätt. Generellt är tiden en ingenjör arbetar mer dyrbar än tiden en dator kan utföra motsvarande beräkningar. Även fast DAC inte tar fram helt fullständiga robotprogram med avseende på kollisioner och plockmetoder skapas en gedigen grund för vidare arbete och på så vis sparas dyrbar ingenjörstid.

En stor svaghet med resultatet är att hela optimeringsdelen förlitar sig på förflyttningstider som inte är helt korrekta. Som diskussionen i kapitel 4.2.3 tog upp så ger beräkningarna utförda på ledernas vinkeldifferenser inte bra nog tidsapproximationer. Detta fel fortplantar sig i beräkningarna och det framräknade resultatet kanske inte är optimalt i verkligheten. Dessutom kan avvikelser i förflyttningstiderna medföra att ytterligare kollisioner uppstår eftersom optimeringens garanti att armarna inte betjänar samma nod samtidigt blir ogiltig.

I projektet har hela optimeringen utförts under antagandet att inga kollisioner uppstår under armarnas rörelser. Testfallen har dock påvisat att kollisioner tenderar att uppstå när armarna ska samspela i trånga utrymmen. För att minimera det manuella arbetet som krävs för att göra robotprogrammen realiserbara vore en naturlig förbättring att integrera en metod för kollisionshantering på det uträknade resultatet. En sådan metod beskrivs i [18] där svepvolymer utnyttjades för att begränsa robotarna från att befinna sig i ett område samtidigt. En nackdel med att applicera en sådan lösning i efterhand är att ju mer banan modifieras desto mer frångås det matematiskt optimala resultatet. Det finns därför ingen garanti på att det modifierade resultatet är den bästa genomförbara lösningen.

I testfallen observerades det hur optimeringstiden ökade kraftigt med antalet komponenter. På grund av komponentplaceringsmetodens fakultetskaraktär utgör den en begränsning för metoden som helhet. Fixturplacerings- och plockordningsoptimeringen skulle dock kunna appliceras på komplexare fall. En möjlig lösning vore att kräva förbestämda komponentplaceringar för scenarion med många komponenter och endast utföra en optimering av plockordning och fixturplacering. Att utföra hela optimeringsprocessen på samma sätt som gjorts i detta arbete vore dock inte hållbart när en cell med många komponenter ska konfigureras. Även i ett kortvarigt produktionsscenario där små serier produ-

ceras kan fokus ligga på att framtagningen av robotprogram ska gå snabbt. Möjligen till den grad att endast en plockordningsoptimering utförs.

7.1 Utvidgning av DAC:s Funktionalitet

En naturlig fortsättning på arbetet vore att implementera funktionalitet som låter DAC appliceras på ytterligare scenarier som kan uppstå i industrin. Ett vanligt förekommande scenario är att samma komponent ska plockas multipla gånger. I nuläget kan detta hanteras genom att lägga till en helt separat instans av denna komponent med ett eget tråg. Optimeringen kan då genomföras på vanligt vis och producera ett acceptabelt resultat. Dock uteblir informationen att två komponenter är ekvivalenta, vilket skulle kunna underlätta i plockordningsoptimeringen. Dessutom skulle det vara fördelaktigt att ha friheten att placera flera lika komponenter i samma tråg, vilket skulle ge ett mer önskvärt resultat och snabba upp komponentplaceringsoptimeringen. Under projektets gång experimenterades det med en MiniZinc modell för detta syfte. Modellen fungerade väl, men NP-svår kategoriseringen av problemet gjorde sig fort påmind när mer komplexa fall skulle lösas. Multipla upplockningar ökar antalet noder i VRP:t och tiden för exekvering ökar drastiskt. Fokus lades istället på kärnfunktionaliteten och multipick modellen implementerades inte i DAC.

Ytterligare en funktionalitet som skulle vara fördelaktig att kunna applicera är användningen av två fixturer. Det skulle möjliggöra monteringen på två områden och kunna dela upp monteringen i två mindre produkter innan de slutligen monteras ihop. Det gör det möjligt att när en robot är upptagen i ena fixturen kan nästa robot jobba i den andra fixturen. Många elektroniska produkter monteras i mindre delar innan de tillslut sätts ihop till en större produkt, två fixturer skulle möjliggöra det i samma utrymme på ett effektivt sätt vilket skulle leda till en lönsam process då två robotarmar kan jobba på flera processer samtidigt.

För att snabba upp DAC:s beräkningstid skulle flera processer kunna parallelliseras och på så sätt utföras snabbare. Lösningsmodellen är fortfarande beroende av ordningen vilket innebär att varje optimeringsmoment skulle kunna parallelliseras, men kräver att varje optimeringsmoment blir fullständigt innan nästa kan påbörjas.

I projektet har det endast tagits hänsyn till en fixtur och flera tråg med kameror. I industrin förekommer det många fler objekt i robotcellerna som har avgränsats från detta arbete. DAC:s funktionalitet skulle kunna utvidgas genom att kunna ta hänsyn till komponenter som behöver till exempel en luftpistol innan en montering.

Arbetet som grundade sig i utvecklingen av tvåarmade robotar har enbart tagit hänsyn till fall där två robotarmar är involverade. Metoden är dock applicerbar på scenarion med fler än två armar. För att implementera denna funktionalitet behöver mindre förändringar göras till modellen och diverse datahanteringsfunktioner.

Sekvenseringen av robotrörelser är ett aktuellt ämne och det kommer allt fler arbeten med olika förhållningssätt till problemet. I [19] ses det som ett

"Job Shop Problem", och löses under andra förutsättningar. I detta projekt har ett "Vehicle Routing Problem" visat sig vara ett lämpligt tillvägagångssätt. En viss problematik kvarstår dock innan metodiken är redo att appliceras i verkligheten.

Referenser

[1] ABB, "The future of robotics and automation depends on humans and robots working together." `http://www.abb.se/robotics`, 2015. [Online; accessed 13-May-2015].

[2] W. Ingdahl, "Underskatta inte behovet av förändring i industrin." http://www.nyteknik.se/asikter/debatt/article3899842.ece, Mars 2015. Ny-Teknik, [Online; posted 13-April-2015].

[3] ABB, "YuMi®, världens första samarbetande tvåarmade robot." `http://www.abb.se/cawp/seitp202/2b3cc28be5a5ec5dc1257e260021a0c0.aspx`, 2015. [Online; accessed 13-March-2015].

[4] ABB, *"YuMi® creating an automated future together You and Me* , 2015.

[5] ABB, "IRB 14000, YuMi®." `http://www.abb.com/product/seitp327/CF7BDCF65F724F66C1257E2D002E7025.aspx`, 2015. [Online; accessed May-2015].

[6] B. Matthias, S. Kock, H. Jerregard, M. Kallman, I. Lundberg, and R. Mellander, "Safety of collaborative industrial robots: Certification possibilities for a collaborative assembly robot concept," *IEEE Std. 1516-2000*, 2011.

[7] L. Qi, X. Yin, H. Wang, and L. Tao, "Virtual engineering: challenges and solutions for intuitive offline programming for industrial robot," in *Robotics, Automation and Mechatronics, 2008 IEEE Conference on*, pp. 12–17, Sept 2008.

[8] R. Andersson, M. Gronback, P. Lokur, and O. Noresson, "Optimization of the operation sequence and the cell layout in the yumi cell." Technical report in a design project at Systems, Control and Mechatronics, Chalmers University of Technology, 2014.

[9] E. Kiniklis, K. M.-M. Rathai, K. Mitritsakis, S. Xu, and T. Wang, "Path planning and optimization of two armed robot." Technical report in a design project at Systems, Control and Mechatronics, Chalmers University of Technology, 2014.

[10] J. Ejenstam, "Implementing a time optimal task sequence for robot assembly using constraint programming," Master's thesis, Uppsala Universitet, Ångströmslaboratoriet, Lägerhyddsvägen 1, 75121 Uppsala, 6 2014.

[11] K. R. Apt, *Principles of Constraint Programming*. CWI, Amsterdam, The Netherlands: Cambridge University Press, 2003.

[12] ABB, "RobotStudio, it's just like having the real robot on your PC!." `http://new.abb.com/products/robotics/robotstudio`, 2014. [Online; accessed 10-May-2015].

[13] ABB, "RobotStudio API." `http://developercenter.robotstudio.com/Index.aspx?DevCenter=RobotStudio&OpenDocument&Title=References`, 2015. [Online; accessed 18-May-2015].

[14] P. Toth and D. Vigo, *The vehicle routing problem*. SIAM monographs on discrete mathematics and applications, Philadelphia (Pa.): SIAM, 2002. SIAM : Society for Industrial and Applied Mathematics.

[15] B. D. Backer, V. Furnon, P. Shaw, P. Kilby, and P. Prosser, "Solving vehicle routing problems using constraint programming and metaheuristics," *Journal of Heuristics*, vol. 6, pp. 501–523, September 2000.

[16] G. Pesant, M. Gendreau, J.-Y. Potvin, and J.-M. Rousseau, "An exact constraint logic programming algorithm for the traveling salesman problem with time windows," *Transportation Science*, vol. 32, no. 1, pp. 12–29, 1998.

[17] K. Marriott and P. J. Stuckey, "A minizinc tutorial."

[18] H. Flordal, M. Fabian, K. Åkesson, and D. Spensieri, "Automatic model generation and plc-code implementation for interlocking policies in industrial robot cells," *Control Engineering Practice*, vol. 15, no. 11, pp. 1416–1426, 2007.

[19] T. Kvant, "Task scheduling for dual-arm industrial robots through constraint programming," *LU-CS-EX 2015-10*, 2015.

A Testfall 1

Fixturplanet			
Hörn	**X [mm]**	**Y [mm]**	**Z [mm]**
h1	560	-100	160
h2	310	-100	160
h3	310	-400	160
h4	560	-400	160

Byggkrav		
Komponent	**Plockmetod**	**Servicetid**
1	Suck	300 ms
2	Pick	200 ms
3	Suck	220 ms
4	Pick	200 ms
Fixture	———	250 ms

Optimeringsresultat					
Komponent-placering	**Permutation**	**Tid innan fixturoptimering**	**Tid efter fixturoptimering**	**Nätförfiningar**	**Fixturplacering**
0	[2, 1, 3, 4]	2777	2766	3	[0,40375, -0,2125, 0,16]
1	[3, 1, 2, 4]	2777	2766	3	[0,40375, -0,2125, 0,16]
2	[3, 4, 2, 1]	2799	2689	3	[0,4975, -0,11875, 0,16]

Övrig info	
Processtid robot A	4.3 s
Processtid robot B	3.9 s
cfx i fixtur robot A	0
cfx i fixtur robot B	0
nX	3
nY	3
Exit radius	15 mm
Byggkravsfil	4comp.graphml
Robotstudio cell	Testcell 1.rsstn
CPU	Intel COre i7-4700K CPU @ 3.50 GHz
RAM	16 GB

B Testfall 2

Fixturplanet			
Hörn	**X [mm]**	**Y [mm]**	**Z [mm]**
h1	335	-165	240
h2	335	165	240
h3	665	165	240
h4	665	-165	240

Byggkrav		
Komponent	**Plockmetod**	**Servicetid**
1	Pick	100 ms
2	Suck	200 ms
3	Pick	150 ms
4	Suck	120 ms
Fixture	———	150 ms

Optimeringsresultat					
Komponent-placering	**Permutation**	**Tid innan fixturoptimering**	**Tid efter fixturoptimering**	**Nätförfiningar**	**Fixturplacering**
0	[1, 2, 3, 4]	2178 ms	2158 ms	3	[0,520625, -0,103125, 0,24]
1	[1, 2, 4, 3]	2178 ms	2158 ms	3	[0,520625, -0,103125, 0,24]
2	[4, 3, 1, 2]	2208 ms	2177 ms	3	[0,582500, -0,144375, 0,24]

Övrig info	
Processtid robot A	3.4 s
Processtid robot B	3.6 s
cfx i fixtur robot A	0
cfx i fixtur robot B	0
nX	3
nY	3
Exit radius	15 mm
Byggkravsfil	Testfall 2.graphml
Robotstudio cell	Testcell 2 - 4 komponenter.stn
CPU	Intel COre i7-4700K CPU @ 3.50 GHz
RAM	16 GB

C Testfall 3

Fixturplanet			
Hörn	**X [mm]**	**Y [mm]**	**Z [mm]**
h1	335	-165	240
h2	335	165	240
h3	665	165	240
h4	665	-165	240

Byggkrav		
Komponent	**Plockmetod**	**Servicetid**
1	Pick	100 ms
2	Suck	200 ms
3	Pick	150 ms
4	Suck	120 ms
5	Pick	120 ms
Fixture	———	150 ms

Resultat					
Komponent-placering	**Permutation**	**Tid innan fixturoptimering**	**Tid efter fixturoptimering**	**Nätförfiningar**	**Fixturplacering**
0	[2, 5, 3, 1, 4]	2645	2642 ms	3	[0,5, -0,0825, 0,24]
1	[2, 5, 3, 4, 1]	2645	2642 ms	3	[0,5, -0,0825, 0,24]
2	[3, 5, 2, 1, 4]	2645	2642 ms	3	[0,5, -0,0825, 0,24]

Övrig info	
Processtid robot B	4.1 s
Processtid robot A	4,3 s
cfx i fixtur robot A	0
cfx i fixtur robot B	0
nX	3
nY	3
Exit radius	15 mm
Byggkravsfil	Testfall 3.graphml
Robotstudio cell	Testcell 3 - 5 komponenter.rsstn
CPU	Intel Core i7-4700K CPU @ 3.50 GHz
RAM	16 GB