2013•2014
FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
master in de industriële wetenschappen: energie

Masterproef

Gerandomiseerde padplanningsalgoritmen voor opnemen van willekeurig geplaatste werkstukken met zesassige robot in simulatie

Promotor :
dr. ir. Eric DEMEESTER
dr. ir. Johan BAETEN

Gezamenlijke opleiding Universiteit Hasselt en KU Leuven

Yannik Kenzeler , Joren Deneyer
Proefschrift ingediend tot het behalen van de graad van master in de industriële wetenschappen: energie

2013•2014
Faculteit Industriële ingenieurswetenschappen
master in de industriële wetenschappen: energie

Masterproef

Gerandomiseerde padplanningsalgoritmen voor opnemen van willekeurig geplaatste werkstukken met zesassige robot in simulatie

Promotor :
dr. ir. Eric DEMEESTER
dr. ir. Johan BAETEN

Yannik Kenzeler , Joren Deneyer
Proefschrift ingediend tot het behalen van de graad van master in de industriële wetenschappen: energie

universiteit hasselt | KU LEUVEN

Woord vooraf

In april 2013 kregen we de kans om een onderwerp te kiezen voor de masterproef. Het onderwerp moest technisch zijn en er moest onderzoek aan te pas komen. Na lang afwegen hebben we ervoor gekozen om te onderzoeken wat de meest performante methode is om automatisch een bak met willekeurig geplaatste werkstukken leeg te maken. Dat dit moest gebeuren met een zesassige Epson C3 robot was een vast gegeven. Ook het gebruik van gerandomiseerde padplanningsalgoritmen, gebaseerd op het toevoegen van willekeurig geplaatste monsters in de vrije robotconfiguratieruimte, was een vaste eis. Als laatste raadde onze promotor, Prof. Dr. Ir. Eric Demeester, ons aan om te werken met OMPL. Dat is een *software*bibliotheek die verschillende gerandomiseerde padplanningsalgoritmen bevat om uit te testen op verschillende padplanningsvraagstukken.

We hebben gekozen voor dit onderwerp omdat het aansluit bij onze opleiding als Industrieel Ingenieur in de Automatisering/Energie. Vooral het aspect van de robotica interesseerde ons enorm. Om deze masterproef te mogen uitvoeren, hebben we een motivatiebrief geschreven. Tot ons genoegen kregen we de melding dat we onze eerste keuze ook effectief mochten uitvoeren.

We ervoeren het proces zeer positief en hebben veel bijgeleerd door dit onderzoek. Algemeen gezien verliep het onderzoek goed en zonder al te veel grote problemen. Er zijn echter een aantal kleine problemen voorgekomen. De meeste problemen hadden te maken met de programmeeromgeving. Ook kwamen we behoorlijk veel problemen tegen doordat de programma's nog gedeeltelijk in ontwikkeling zijn. Het onderwerp en de programma's die nodig zijn om het onderwerp te bestuderen zijn immers *state of the art*. Een groot probleem dat we het eerste semester ondervonden was het gebrek aan computercapaciteit. Door het werken met grafische applicaties in Ubuntu waren onze computers al snel onbruikbaar om het onderzoek verder te zetten. We hebben ons enkele maanden moeten focussen op de theorie en het opzoekwerk op het internet aangezien er geen computer was die de programma's aankon die we nodig hadden. Midden februari kregen we een laptop die de programma's zonder problemen kon uitvoeren. Vanaf dan hebben we *non-stop* onderzoek gedaan naar padplanningsalgoritmen, programma's en een manier om in simulatie het probleem na te bootsen of te modelleren.

We zijn zeer tevreden over het resultaat en kunnen trots terugblikken op ons geleverde werk. Ook blikken we terug op een periode waar we van veel mensen steun hebben gekregen. We danken Prof. Dr. Ir. Eric Demeester om ons met raad en daad bij te staan. Graag willen we Prof. Dr. Jeroen Lievens bedanken voor de instructies en tips om de scriptie tot een succes te brengen. We willen ook graag Prof. Dr. Ir. Johan Baeten en de projectingenieurs van ACRO bedanken voor hun tips en aanwijzingen. Ook een dankbetuiging aan familie en vrienden voor de steun en aanmoediging.

Inhoudsopgave

Lijst van tabellen

Lijst van figuren

Begrippenlijst

Child:	Het *child* van een link is de link die één plaats verder weg ligt van de oorsprong van de robot t.o.v. de beschouwde link. Het *child* van een *vertex* is de *vertex* die één plaats verder ligt van de oorsprong van de structuur t.o.v. de beschouwde *vertex.*
Configuratie:	Een robotconfiguratie komt overeen met de posite en oriëntatie van de robot.
Edge:	Een geldig (botsingsvrije) pad dat twee vertices met elkaar verbindt (zie Vertex in begrippenlijst).
Eindeffector:	Het mechanisme dat op het uiteinde van de robot geplaatst is om werkstukken mee te grijpen.
Joint:	Een scharnier tussen twee *links* dat ervoor zorgt dat de *links* ten opzichte van mekaar kunnen roteren.
Link:	Een stang of verbinding tussen twee scharnieren (*joints*).
Null-operatie:	Een operatie of een uitvoering waar de robot niet beweegt.
OMPL:	Open Motion Planning Library.
Parent:	De *parent* van een link is de link die één plaats dichter bij de oorsprong van de robot ligt t.o.v. de beschouwde link. De *parent* van een *vertex* is de*vertex* die één plaats dichter bij de oorsprong van de structuur ligt t.o.v. de beschouwde *vertex.*
Plugin:	Een aanvullend element op een computerprogramma.
Query:	Een vraag gericht aan een padplanningsalgoritme, om een pad te zoeken tussen een begin- en eindconfiguratie.
ROS:	*Robot Operating System*.
Structuur:	De verzameling van *vertices* en *edges* in de robotomgeving waarvan een zoekalgoritme gebruik kan maken om een pad te vinden.
Terminal:	Een scherm binnen de Linux omgeving van waaruit programma's opgestart of geïnstalleerd worden.
Toestand:	Een toestand geeft de huidige toestand aan van de robot wat kan overeenkomen met de positie, oriëntatie, snelheden,… van de robot. In deze scriptie komt de toestand overeen met de positie en oriëntatie van de robot.
Vertex (**vertices**):	Een configuratie toegevoegd door een padplanningsalgoritme dat geldig is (botsingsvrij) zodat de configuratie tot de structuur behoort.
Werkingsruimte:	De ruimte (set van configuraties) waar de robot kan geraken met zijn eindeffector.

Abstract

De trend naar geïndividualiseerde producten en kleine seriegroottes verhoogt de druk om productiemachines te flexibiliseren. Maar tegelijk vereisen de hoge loonkost en globale prijsdruk een verregaande automatisering. De onderzoeksgroep ACRO (KU Leuven/KHLIM) onderzoekt nieuwe technologieën zoals gerandomiseerde padplanningsalgoritmen om deze tegenstrijdige eisen te verenigen. Dit eindwerk vergelijkt en beoordeelt zulke algoritmen in simulatie op hun performantie voor een 3D-binpicking installatie. Hierbij moet een zesassige Epson C3 robot een bak met willekeurig geplaatste producten automatisch leegmaken m.b.v. een machine-visiesysteem.

Om in simulatie experimenten te kunnen uitvoeren is een model van de Epson-robot gemaakt. In een gemodelleerde testomgeving werden padplanningsalgoritmen onderzocht op hun rekentijd, betrouwbaarheid en benadering van het optimale pad voor verscheidene start- en eindconfiguraties. Ook de invloed van algemene en specifieke parameters van technieken zijn onderzocht.

Experimenten tonen aan dat de algoritmen PRM, PRM*, RRT-Connect en SBL de beste performantie halen bij deze probleemstelling. Uit onderzoek blijkt verder dat het gebruik van PRM tot boven de bak en dan RRT-Connect tot in de bak een nog betere oplossing geeft. Ook is experimenteel bewezen dat de botsingsdetectieresolutie aangepast moet zijn aan de bakwand- en productdikte om een hoge betrouwbaarheid te hebben.

Abstract in English

The trend towards individualised products and small lot sizes requires production units to be increasingly more flexible. On the other hand further automation is needed because of the high labour cost and the global price pressure. KU Leuven/KHLIM's research group, ACRO, investigates new technologies like randomised path planning algorithms to deal with these two conflicting requirements. This master's thesis compares and applies randomised planning algorithms to a 3D-bin picking setup in simulation, in which a six-axis Epson C3 robot has to clear a bin with randomly placed objects using a machine-vision system.

In order to perform experiments in simulation, a model of the Epson-robot is made. In a modelled test environment, the calculation time, reliability and resemblance to an optimal path of several planning algorithms were evaluated for different initial and goal configurations. Furthermore, the influence of general and specific parameters was investigated.

Experiments showes that the algorithms PRM, PRM*, RRT-Connect and SBL have the best performance for this problem. Research furthermore demonstrated that using PRM until above the bin and then using RRT from above the bin to the object inside the bin was the optimal solution. It was also experimentally proven that the resolution of the collision detection has to be adapted to the thickness of the product and of the bin's wall to achieve a high reliability.

1 Inleiding

1.1 Situering

Deze masterproef vindt plaats bij ACRO, een onderzoekscentrum van de Katholieke Hogeschool Limburg (KHLim) en de KULeuven. ACRO doet toegepast onderzoek naar enerzijds industriële automatisering, met opleidingen rond PLC programmering en profibus, en anderzijds robotica en visie. De masterproef situeert zich voornamelijk in het onderzoek op visie en robotica.

De trend naar geïndividualiseerde producten en kleine seriegroottes verhoogt de druk om productieprocessen te flexibiliseren. Anderzijds vereisen de hoge loonkost en globale prijsdruk een verregaande automatisering. Nieuwe technologieën zoals gerandomiseerde padplanningsalgoritmen verenigen deze tegenstrijdige eisen. Deze masterproef kadert binnen een technologietransferproject (IWT TETRA) waarin een oplossing wordt gezocht voor een probleem waar verschillende bedrijven mee kampen. Meer bepaald is de vraag gekomen om een automatische 3D-binpicking installatie te programmeren met een zesassige Epson C3 robot (figuur 1). Die moet een aangevoerde bak met willekeurig geplaatste werkstukken automatisch kunnen leegmaken gebruikmakend van een machine-visiesysteem. 3D-*bin picking* werd reeds verscheidene decennia geleden verkondig als de heilige graal van gerobotiseerde materiaalverwerking, en dat blijft het tot op heden nog steeds. 3D-*bin picking* vermijdt het gebruik van dure verenkelingsinstallaties, trilkolommen en uitlijnhulpstukken. Eén van de moeilijkheden bij 3D-*bin picking* is het snel genereren van botsingvrije paden. Het gebruik van gerandomiseerde padplanningsalgoritmen kan hiervoor een oplossing bieden, deze algoritmen zijn gebaseerd op het toevoegen van willekeurig geplaatste monsters in de robotconfiguratieruimte die dan met elkaar worden verbonden via botsingsvrije paden.

Figuur 1: De zesassige Epson C3 robot die op ACRO geïntegreerd is in een 3D-bin picking setup [1]

1.2 Doelstellingen

Het doel van het eindwerk is om gerandomiseerde padplanningsalgoritmen in simulatie te onderzoeken, toe te passen op een 3D-*bin picking* situatie en te beoordelen op hun performantie. Performantie wordt beoordeeld op vlak van rekentijd, betrouwbaarheid en benadering van het optimale pad voor verschillende begin- en eindconfiguraties. Met deze algoritmen moet de robot in staat zijn een bak met willekeurig geplaatste werkstukken leeg te maken, gebruikmakend van een beschikbaar visiesysteem. Deze robotarm moet in 95% van de gevallen een werkstuk correct verplaatsen of als resultaat geven dat er geen pad gevonden is. De robot moet bovendien de opgenomen werkstukken op een gedefinieerde positie kunnen plaatsen zonder te botsen met zichzelf of de omgeving, maar met dynamische obstakels wordt geen rekening gehouden. Dat zijn obstakels die er eerst niet waren en plots de omgeving binnenkomen zoals de arm van een mens. Voor dit eindwerk zijn de eisen dat de robot:

- nooit met zichzelf in botsing mag treden;
- niet mag botsen met elementen uit zijn statische omgeving dus geen rekening houdend met dynamische obstakels;
- volledig automatisch moet werken door gebruik te maken van gerandomiseerde padplanningalgoritmen.

Een padplanningsalgoritme heeft een voldoende hoge performantie voor deze probleemstelling indien het algoritme voldoet aan enkele strenge eisen. Het padplanningsalgoritme moet:

- minstens 95% van de keren een geldig botsingsvrij pad als resultaat geven als er daadwerkelijk een pad bestaat. In de andere 5% van de gevallen moet het algoritme besluiten dat er geen pad werd gevonden, zodat gepaste acties ondernomen kunnen worden (vb. schudden met bak);
- Nooit een foutief pad als resultaat geven;
- een gemiddelde rekentijd hebben die minder is dan 5 seconden;
- het optimale pad redelijk goed benaderen.

1.3 3D-*Bin Picking* opstelling

De robot staat opgesteld bij ACRO. De eindeffector is eventueel nog aan te passen. Ook het visiesysteem is reeds aanwezig. Het bestaat uit een laser en een camera aangevuld met een eventuele belichting. Er zijn enkele belichtingen aanwezig bij ACRO, maar het gebruik ervan is niet noodzakelijk en afhankelijk van de omgeving en de situatie. Bovendien is er een kalibratieplaat beschikbaar waardoor kalibratie van het visiesysteem en de robot mogelijk is. Er is ook een laptop beschikbaar om simulatie mee te kunnen uitvoeren. Als laatste zijn de werkstukken met het overeenkomstige CAD-model aanwezig waardoor het uittesten van het programma mogelijk is. Het eindwerk focust zich niet op het visiegedeelte, wat betekent dat het er van uitgaat dat de input van de positie van een werkstuk via een visiesysteem gegeven is.

1.4 Gevolgde methode en overzicht

Om tot het uiteindelijke resultaat te komen zijn er verschillende stappen nodig die ook gestructureerd in deze scriptie vermeld zijn. Van sterk belang is de theoretische achtergrond van padplanningsalgoritmen en de verschillende elementen die hierbij behoren. Hoofdstuk 2 bevat daarom een deel van de literatuurstudie wat een eerste stap is in het proces. Aangezien de literatuurstudie voor dit eindwerk zeer breed was en verschillende delen van de literatuurstudie niet essentieel zijn voor de scriptie, zijn enkel de echt relevante onderwerpen beschreven in hoofdstuk 2. Om padplanningsalgoritmen en hun parameters te kunnen beoordelen is een grondige theoretische kennis nodig over deze algoritmen. Hoofdstuk 3 bespreekt de theorie van enkele gerandomiseerde padplanningsalgoritmen. Bij de evaluatie van de algoritmen, in hoofdstuk 6, kan naar deze theorie terugverwezen worden. De tweede stap van het proces is een kennismaking met de *software*. Aangezien dit eindwerk bestaat uit onderzoek in simulatie zijn er verscheidene programma's nodig. Om experimenten te kunnen uitvoeren is er een programma nodig dat het gebruik van deelprogramma's noodzakelijk maakt. Ook de modellering van de robot en de testomgeving gebeurt via software. Daarom behandelt hoofdstuk 4 uitvoerig de werking en betekenis van de verschillende softwareprogramma's die nodig zijn om het resultaat uit te voeren of die nodig zijn om tot dat resultaat te komen. Om het zoeken naar een pad in simulatie mogelijk te maken moeten verschillende stappen doorlopen worden. Hoofdstuk 5 bespreekt de derde stap in het eindwerk namelijk het operationeel maken van de robot in simulatie zodat er padplanningsalgoritmen op kunnen uitgevoerd worden. De modellering van zowel de robot als de omgeving is in dit hoofdstuk besproken net als de rest van het effectieve programmeerwerk. Naast het modelleren van een robot in simulatie is de belangrijkste stap van het proces het testen, meten en evalueren van padplanningsalgoritmen. Hoofdstuk 6 beschouwt daarom de metingen, analyseert de resultaten en verklaart deze resultaten met een eventuele verwijzing naar de theoretische achtergrond van hoofdstuk 5. Aangezien het uitvoeren van padplanningsalgoritmen in simulatie geen eindstation is, is er toekomstig werk mogelijk. De stappen die mogelijk zijn na de resultaten van dit eindwerk staan samen met de conclusie beschreven in Hoofdstuk 7.

2 Literatuurstudie

2.1 Inleiding

2.1.1 *Vertices* en *edges*

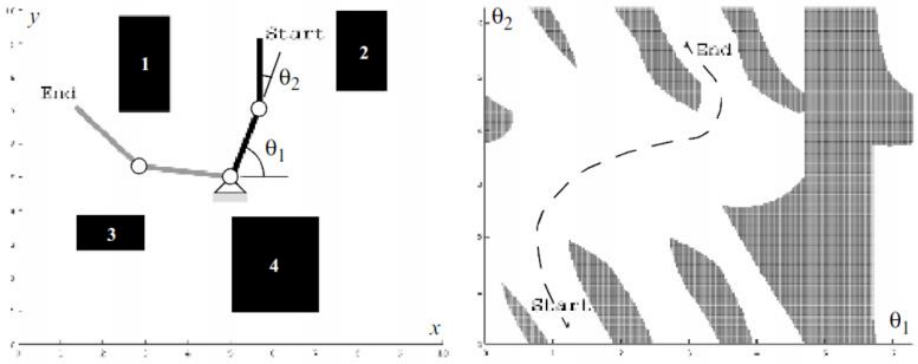
Zoals vermeld in de inleiding is het de bedoeling om een bak met willekeurig geplaatste werkstukken leeg te maken. Om dit te voltooien moet de eindeffector van de robot tot in de bak geraken. De robot moet dus eerst vanaf een ingestelde positie naar een positie in de bak gaan waar een werkstuk ligt. Daarna moet de robot het werkstuk grijpen en dit verplaatsen naar een vooropgestelde positie waar de eindeffector het werkstuk terug loslaat. De bedoeling is om een pad te vinden die een begintoestand met een eindtoestand verbindt zodat de robot een beweging kan maken van één plaats naar een andere plaats. De reden waarom een pad robotconfiguraties verbindt en niet posities is om het feit dat er voor iedere 3D-positie van de eindeffector meerdere 3D-oriëntaties mogelijk zijn. Een robotconfiguratie ligt vast als alle hoeken van de scharnieren van een robot gekend zijn. In het geval van een zesassige robot komt een robotconfiguratie overeen met een positie (drie cartesische coördinaten) en een rotatie (drie hoeken) van de eindeffector.

Om een pad te berekenen zijn er mogelijk tussentoestanden nodig die ervoor zorgen dat het pad soepeler is en obstakels ontwijkt. Deze tussentoestanden, toegevoegd door een algoritme, heten *vertices*. Een gerandomiseerd padplanningsalgoritme voegt min of meer willekeurig monsters toe aan de werkingsruimte van de robot. Ieder monster is een robottoestand en indien het monster geldig is (‘2.2.1 Geldigheid van *vertices* en *edges*’), voegt het algoritme dat monster toe aan de omgeving en vanaf dan heet zo een monster een *vertex*. Vanaf dat moment kan het algoritme deze vertex gebruiken om een pad te zoeken. Natuurlijk zijn er niet alleen *vertices* nodig om een pad te vormen, maar ook verbindingen tussen die *vertices*. Net als bij *vertices* berekent het algoritme of de verbinding geldig is (‘2.2.1 Geldigheid van *vertices* en *edges*’). Wanneer de verbinding geldig is voegt het algoritme deze verbinding toe aan de omgeving en vanaf dat moment heet deze verbinding een *edge*. [2]

2.1.2 Configuratieruimte

Aangezien een pad uit toestanden bestaat en niet uit posities is het eenvoudiger om in een ruimte te zoeken die bestaat uit robottoestanden in plaats van posities. Zo een ruimte heet een configuratieruimte. De configuratieruimte is een verzameling van alle mogelijke toestanden die een robot kan aannemen. De dimensie ervan is afhankelijk van het aantal vrijheidsgraden. Een zesassige robot heeft zes vrijheidsgraden waardoor de configuratieruimte van deze robot uit zes dimensies bestaat. [3]

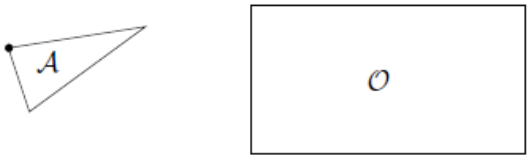
Een voorbeeld van een configuratieruimte staat in figuur 2 afgebeeld. In dat voorbeeld is in het linkse deel van de figuur een twee dimensionale robot omgezet naar de configuratieruimte van die robot (rechtse deel van de figuur). De twee vrijheidsgraden zijn de twee hoeken. In de configuratieruimte wordt de robot herleid tot een punt, wat het plannen van paden in de configuratieruimte sterk vereenvoudigt. Ieder punt in de rechtse afbeelding komt overeen met een configuratie van de robot wat in dit geval twee hoeken zijn. De begin- en eindconfiguratie van een padplanningsprobleem kunnen in de configuratieruimte toegevoegd worden zoals hier geïllustreerd is. Het algoritme tracht in deze configuratieruimte een pad te zoeken die de twee configuraties met elkaar verbindt. Zoals in de figuur is weergegeven, is het veel eenvoudiger om een pad te vinden in de configuratieruimte (rechtse deel van de figuur) dan in de gewone cartesische ruimte (linkse deel van de figuur). Obstakels in de cartesische ruimte (de zwarte rechthoekige obstakels in figuur 2, links) moeten ook naar de configuratieruimte getransformeerd worden (grijze zones in figuur 2, rechts). Deze operatie is erg rekenintensief en de obstakels in de configuratieruimte kunnen vaak slechts benaderend, niet algebraïsch beschreven worden.



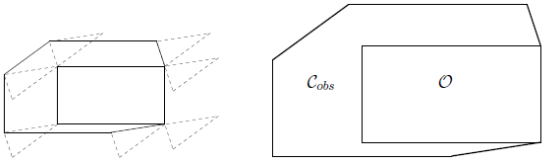
Figuur 2: Voorbeeld van een 2-dimensionale configuratieruimte (rechts) overeenkomend met de cartesische ruimte (links) [4]

Een pad is geldig wanneer dit pad volledig in de vrije ruimte ligt. De manier waarop het algoritme dit berekent staat uitgelegd in '2.2.2 Werking van botsingsdetectie'. Of een pad geldig is, is te zien in figuur 2 waar het pad volledig in de vrije (witte) configuratieruimte moet liggen. De grijs gearceerde ruimte komt overeen met de verzameling van niet-toegankelijke toestanden. Een toestand kan niet-toegankelijk of ongeldig zijn indien deze toestand buiten het bereik van de robot ligt. Wanneer een deel van de robot een obstakel of ander deel van de robot raakt behoort die toestand ook tot de niet-toegankelijke ruimte, wat in dat geval ook obstakelruimte genoemd wordt.

De obstakelruimte bestaat niet alleen uit toestanden waar de eindeffector een obstakel raakt, maar ook uit toestanden waar een ander deel van de robot een obstakel raakt. De configuratieruimte houdt hier meteen rekening mee aangezien zo een toestand ongeldig is waardoor het een grijze toestand wordt. Het algoritme moet weten wanneer een bepaalde toestand voor een botsing zal zorgen. Dit gebeurt door in de cartesische ruimte het bestaande obstakel te vergroten zodat de robot in geen geval het effectieve obstakel zal raken. Maar hoe wordt dit in de praktijk geïmplementeerd? Stel dat de werkingsruimte tweedimensionaal is, dat de robot een driehoek is en dat het obstakel een rechthoek is zoals in figuur 3 afgebeeld staat. De robot 'A' bevat een punt, wat de oorsprong van de robot voorstelt. De oorsprong moet steeds op een bepaalde afstand van het obstakel blijven zodat het onmogelijk is om het obstakel te raken. De robot kan enkel transleren. In figuur 4 (linkse deel) is te zien hoe de robot op een bepaalde afstand van het obstakel blijft om een botsing te voorkomen. Het rechtse deel van figuur 4 illustreert het nieuwe, vergrootte obstakel. Dat komt overeen met het obstakel in de configuratieruimte waar de robot herleid wordt tot een punt [2]



Figuur 4: 2D-robot met rechthoekig obstakel [2]



Figuur 3: Omvorming van obstakel [2]

Voor 2D translerende robots en polygonale obstakels is het dus haalbaar om obstakels in de configuratieruimte expliciet in algebraïsche vorm te berekenen. In het algemene 6D geval met willekeurige obstakels is het complex (en niet altijd mogelijk) om de obstakels in de configuratieruimte te berekenen. In dat geval is het wel nog steeds mogelijk om monsters uit de configuratieruimte te nemen en te berekenen of die configuraties vrij of in botsing zijn.

2.1.3 Padplanning

Het plannen van een pad gebeurt door zogenaamde padplanningsalgoritmen. Zulke algoritmen moeten de begin- en eindtoestand(en) gegeven hebben zodat het algoritme naar een oplossing kan zoeken. Hoe het pad eruit ziet, hangt af van het probleem, het algoritme, de parameters van het algoritme en de robot. De invloed van het algoritme en zijn parameters op het pad is besproken in 'Hoofdstuk 3: Padplanningsalgoritmen in theorie'. Ook de roboteigenschappen kunnen het verloop van het pad beïnvloeden omdat robots een verschillend werkingsgebied of een verschillend aantal vrijheidsgraden kunnen hebben. [5]

Een berekend pad bestaat uit tussen*vertices*, begin- en eind*vertex* en de *edges* hiertussen. Een algoritme stuurt de toestanden door naar een simulatieprogramma of naar de robot zelf. De robot of het simulatieprogramma gaat dan doorheen deze punten zodat het pad gevolgd wordt. Deze scriptie focust zich op het plannen van paden door middel van simulatie. In simulatie is het handiger om de performantie van algoritmen te onderzoeken. Ieder algoritme heeft een andere aanpak waardoor de algoritmen vooral bruikbaar zijn bij specifieke problemen. Het beoordelen van een algoritme slaat terug op de rekentijd die nodig is om het pad te vinden, het aantal *vertices* dat het resulterende pad bevat en de nauwkeurigheid van een pad (hoe benadert het berekende pad het optimale pad?). [2]

2.2 Botsingsdetectie

2.2.1 Geldigheid van *vertices* en *edges*

Botsingsdetectie is een kernelement en *bottleneck* van alle padplanningsalgoritmen. Het berekent of een *edge* of *vertex* botsingsvrij is. Het is een van de belangrijkste elementen van een padplanningsalgoritme en is het onderdeel dat bij de meeste algoritmen het meeste tijd in beslag neemt. Een uitzondering hierop vormen algoritmen die gebruikmaken van *lazy collision checking* of luie botsingsdetectie ('Hoofdstuk 3: Padplanningsalgoritmen in theorie') die hier minder tijd aan besteden.

Geldigheid van een *vertex:* De controle of een monster (robotconfiguratie) geldig is, gebeurt op het moment dat het padplanningsalgoritme dat monster probeert toe te voegen aan de omgeving. De botsingsdetectie controleert simpelweg of dat monster binnen een obstakel ligt of niet. Indien het monster niet botsingsvrij is, is het ongeldig en verwerpt het algoritme dat monster ook meteen. Indien het monster in de vrije ruimte ligt, is het geldig en voegt het algoritme dat monster toe als vertex in de omgeving zodat het algoritme hier gebruik van kan

maken om een pad te vinden. Er bestaan verschillende manieren om een toestand op zijn geldigheid te controleren (‘2.2.2 Werking botsingsdetectie’).

Geldigheid van *edges:* De geldigheid van een verbinding controleren is echter complexer en er bestaan twee methoden hiervoor. De eerste methode is de exacte methode. Deze methode geeft een zeer goed resultaat, maar hiertegenover staat een extra tijdsduur die nodig is om berekeningen uit te voeren. De exacte methode maakt gebruik van de volledig opgebouwde configuratieruimte (‘2.1.2 Configuratieruimte’). Om de configuratieruimte op te bouwen moet iedere mogelijke toestand gecontroleerd worden, wat enorm veel tijd in beslag neemt aangezien een robot immens veel toestanden kan aannemen. Nadat de volledige configuratieruimte is opgebouwd, kan de botsingsdetectie eenvoudig controleren of een verbinding volledig binnen de vrije ruimte ligt. Indien dat het geval is, is deze verbinding geldig en voegt het algoritme deze verbinding als *edge* toe aan de omgeving. Indien de verbinding niet geldig is, verwerpt het algoritme deze verbinding. Ook zal het algoritme opnieuw een volledige configuratieruimte moeten maken indien er een obstakel bijkomt of een obstakel van plaats verandert. Doordat de exacte methode zo veel tijd in beslag neemt, maakt men in de praktijk bijna altijd gebruik van de tweede methode. [2] [5] [6]

De tweede methode, de benaderende methode, deelt een verbinding op in deeltoestanden. Het aantal deeltoestanden dat op een verbinding toegevoegd wordt, hangt af van de situatie en is een praktische afweging tussen nauwkeurigheid en tijdsduur. Als de berekeningen te lang duren zijn er te veel deeltoestanden gegenereerd op de verbinding. Als het algoritme daarentegen een te lage nauwkeurigheid haalt, dan is het aantal deeltoestanden op de verbinding te laag. In de praktijk deelt de botsingsdetectie de verbinding niet op in een bepaald aantal deeltoestanden, maar hangt dat aantal af van de lengte van de verbinding. De botsingsdetectie zal met andere woorden deeltoestanden toevoegen waar de afstand tussen twee toestanden een vaste parameter is. Er bestaat geen optimale afstand aangezien deze optimale waarde afhangt van het algoritme en het probleem. Algemeen gezien is de afstand tussen twee toestanden een praktische afweging tussen nauwkeurigheid en tijdsduur. De botsingsdetectie controleert de deeltoestanden op dezelfde manier als het de monsters controleert. De mogelijke manieren waarop de botsingsdetectie dit doet staan vermeld in ‘2.2.2 Werking van botsingsdetectie’.

Er bestaan twee verschillende volgordes waaraan de botsingsdetectie zich kan houden om de verschillende toestanden op een verbinding te controleren. De eerste manier is de eenvoudigste manier en begint met de controle op een toestand die aan een uiteinde van de verbinding ligt. Daarna volgt de botsingsdetectie de verbinding richting het andere uiteinde en controleert stap voor stap iedere toestand die het tegenkomt. Stel bijvoorbeeld dat er een lijn van 0 tot 1 loopt ([0,1]) en dat deze uit 9 tussentoestanden bestaat of 11 toestanden indien de uiteinden meegerekend zijn. De botsingsdetectie controleert eerst de toestand die op $1/10^{de}$ ligt van de verbinding (0 en 1 zijn *vertices* en zijn al gecontroleerd). Daarna controleert de botsingsdetectie de geldigheid van de toestand die op $2/10^{de}$ van de verbinding ligt. Vervolgens controleert het de toestand die op $3/10^{de}$ ligt, $4/10^{de}$ ligt,… tot $9/10^{de}$. De botsingsdetectie blijft toestanden op een verbinding controleren tot er een toestand ongeldig is of tot alle toestanden gecontroleerd zijn.

Na controle van de helft van de toestanden is nog maar de helft van de verbinding gecontroleerd, de andere helft van de sequentie is helemaal niet bekeken. Dus na de controle van de 5de sample, die op 5/10de van de verbinding ligt, heeft de botsingsdetectie alle toestanden gecontroleerd die tussen 1/10de en 5/10de liggen terwijl alle toestanden die tussen 6/10de en 9/10de liggen nog niet gecontroleerd zijn. Voor deze actie te voltooien gebruikt de botsingsdetectie de helft van de totale tijd. Als er dan een groot obstakel ligt, dat zich uitstrekt van 6/10de tot 9/10de van de verbinding, is deze na de helft van de berekeningstijd nog niet ontdekt terwijl de helft van de maximale rekentijd al gebruikt is. Om de rekentijd efficiënter te benutten is er een tweede methode ontwikkeld die complexer is, maar die sneller verschillende delen van de verbinding controleert. [2]

De tweede manier maakt gebruik van de zogenaamde 'van der Corput' sequentie. Het grote nadeel hiervan is dat deze sequentie enkel toepasbaar is op eenheidssequenties ([0,1]). De methode maakt gebruik van omgekeerde binaire codes om de sequentie te verkennen. Stel dat een eenheidsinterval uit 16 toestanden bestaat. De methode schrijft de omgekeerde binaire codes neer van 0 tot 15/16de. Dat zorgt ervoor dat de volgorde verandert van 0 tot 15/16de naar 0, 1/2de, 1/4de, 3/4de,… . De volgorde van de omgekeerde binaire code en het volledige voorbeeld zijn gedemonstreerd in figuur 5 op de volgende pagina.

i	Naive Sequence	Binary	Reverse Binary	Van der Corput	Points in $[0,1]/\sim$
1	0	.0000	.0000	0	
2	1/16	.0001	.1000	1/2	
3	1/8	.0010	.0100	1/4	
4	3/16	.0011	.1100	3/4	
5	1/4	.0100	.0010	1/8	
6	5/16	.0101	.1010	5/8	
7	3/8	.0110	.0110	3/8	
8	7/16	.0111	.1110	7/8	
9	1/2	.1000	.0001	1/16	
10	9/16	.1001	.1001	9/16	
11	5/8	.1010	.0101	5/16	
12	11/16	.1011	.1101	13/16	
13	3/4	.1100	.0011	3/16	
14	13/16	.1101	.1011	11/16	
15	7/8	.1110	.0111	7/16	
16	15/16	.1111	.1111	15/16	

Figuur 5: De 'Van der Corput' methode om een verbinding efficiënter te controleren op zijn geldigheid [2]

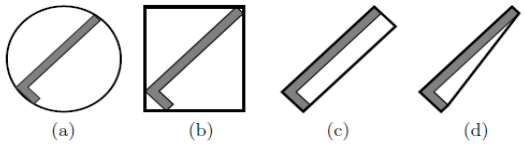
Indien het aantal toestanden een macht van twee is, is de verbinding gelijkmatig verdeeld. Dat het lijnstuk soms ongelijkmatig verdeeld is, vormt geen problemen aangezien dit enkel tot gevolg heeft dat de nauwkeurigheid niet overal hetzelfde is bij de verbinding. Het grote voordeel van de methode is de vermindering aan berekeningstijd indien er een obstakel ligt op de verbinding. Dat komt doordat de methode sneller verschillende delen van de verbinding gaat controleren zoals in figuur 5 afgebeeld staat waardoor obstakels sneller opgemerkt worden. Indien er geen obstakel ligt dan is de rekentijd voor beide methodes gelijk aangezien de botsingsdetectie dan alle toestanden moet controleren. Als iedere deeltoestand geldig is dan is

de verbinding ook geldig waardoor het algoritme deze verbinding als *edge* toevoegt aan de omgeving. [2] [7]

De Halton/Hammersley-methode is een uitbreiding op de van der Corput sequentie en werkt het grote nadeel van de eenheidssequentie weg. De methode is een n-dimensionele generalisering van de van der Corput sequentie en maakt gebruik van verschillende basissen waardoor algoritmen deze methode kunnen gebruiken voor grotere sequenties. Een ander alternatief is om een verbinding met willekeurige lengte naar een eenheidssequentie te schalen. Op deze eenheidssequentie past het algoritme dan de van der Corput methode toe. Iedere toestand op de eenheidssequentie moet dan terug naar de oorspronkelijke afstand geschaald worden waardoor de botsingsdetectie kan controleren of deze toestand geldig is of niet. De methode is voor de rest analoog met als voordeel dat deze methode een generalisering is van de van der Corput sequentie en als nadeel dat de methode iets meer rekentijd in beslag neemt. [2] [8]

2.2.2 Werking van botsingsdetectie

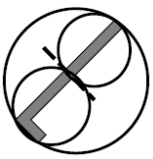
Het vorige deel ('2.2.1 Geldigheid van *vertices* en *edges*') besprak de elementen waarop botsingsdetectie uitgevoerd wordt, maar hoe voert de botsingsdetectie deze controle concreet uit? Voor gerandomiseerde padplanningsalgoritmen, gebaseerd op het toevoegen van monsters, is de werking van de botsingsdetectie gebaseerd op een hiërarchische methode. Deze methode beperkt de berekeningstijd van de botsingsdetectie door gebruik te maken van geometrische figuren. Het volledige lichaam of obstakel, dat de botsingsdetectie moet controleren, ligt binnen deze geometrische figuur. De geometrische figuur kan zowel een bol als een kubus als eender welk ander figuur zijn. Het opbouwen van de figuur gebeurt door de uiterste hoekpunten van het lichaam te nemen als deel van de geometrische figuur. Een voorbeeld van mogelijke geometrische figuren die een obstakel omringen staat in figuur 6 afgebeeld. De afgebeelde geometrische figuren zijn een bol (a), een kubus (b), een balk (c) en een combinatie van geometrische figuren (d). Van links naar rechts is de figuur steeds nauwkeuriger en neemt de figuur steeds meer de vorm aan van het lichaam. Het nadeel is echter dat de botsingsdetectie ook steeds meer tijd in beslag neemt naarmate de figuur nauwkeuriger wordt. [2]



Figuur 6: Mogelijke geometrische figuren rond een lichaamals benadering voor botsingsdetectie [2]

a) bol
b) kubus
c) balk
d) combinatie geometrische figuren

De botsingsdetectie doorloopt enkele stappen om te controleren of een monster binnen het obstakel ligt of niet. Eerst controleert de botsingsdetectie of het monster binnen de geometrische figuur ligt. Indien dat niet zo is, treedt er zeker geen botsing op waarop het algoritme dat monster toevoegt aan de omgeving als *vertex*. Indien het monster wel in de figuur ligt, splitst de geometrische figuur zich op in twee zogenaamde 'kindergeometrische figuren' wat dezelfde vorm heeft als de oorspronkelijke figuur, maar die de figuur opdeelt in twee identieke figuren die half de grootte hebben van de oorspronkelijke figuur zoals in figuur 7 te zien is. In de figuur splitst de geometrische figuur, de cirkel, zich op in twee kindercirkels. [2]



Figuur 7: Opsplitsing figuur naar kinderfiguren voor de botsingsdetectie [2]

Vervolgens controleert de botsingsdetectie bij beide geometrische figuren of het monster in de figuur ligt. Indien het monster in geen van beide figuren ligt dan is het monster botsingsvrij met dit lichaam. Indien het monster daarentegen in één van de figuren ligt dan splitst die figuur opnieuw in twee kinderfiguren. De andere figuur zal echter niet verder splitsen aangezien het monster hier niet in ligt. Deze stappen blijven zich herhalen tot het monster niet meer in een geometrische figuur ligt en dus botsingsvrij/geldig is of tot de geometrische figuur de grootte en vorm van het lichaam aanneemt. Als het monster dan nog binnen de geometrische figuur ligt dan ligt het ook in het obstakel, waardoor een botsing optreedt. Het monster is daardoor ongeldig en het algoritme verwerpt dat monster dan ook meteen. Deze manier van werken beperkt de berekeningstijd doordat de botsingsdetectie niet meteen op het niveau van het lichaam moet controleren wat veel meer berekeningstijd in beslag zou nemen dan het controleren van een geometrische figuur. Het komt immers vaak voor dat monsters niet binnen een obstakel liggen waardoor ze via deze methode veel sneller beschouwd kunnen worden als geldige monsters in plaats van lang te berekenen of ze in het lichaam liggen of niet.

Er zijn twee soorten botsingsdetecties, namelijk een botsingsdetectie die men uitvoert om te kijken of een monster binnen een lichaam ligt, zoals hiervoor is uitgelegd, en een botsingsdetectie die men uitvoert om een botsing te registreren tussen twee lichamen. De tweede soort botsingsdetectie is bijna volledig analoog aan de eerste soort die al verklaard is. De botsingsdetectie berekent bij de tweede soort echter of een geometrische figuur (figuur 6) van het ene lichaam snijdt met de geometrische figuur van een ander lichaam. Indien dat het geval is, delen de geometrische figuren zich op in kinderfiguren waarop de controle opnieuw gebeurt tot de twee figuren elkaar niet meer snijden of tot de geometrische figuren de vorm van het lichaam aannemen. Indien bij dat laatste nog steeds een snijding aanwezig is, treden de lichamen in botsing. [2]

2.3 Gerandomiseerde padplanningsalgoritmen gebaseerd op het toevoegen van configuratieruimtemonsters

2.3.1 Combinatorische- en bemonsteringsalgoritmen

Om een pad te berekenen is er een padplanningsalgoritme nodig. Er bestaan twee soorten algoritmen die elk een eigen toepassingsgebied hebben. Het eerste soort algoritme maakt gebruik van een combinatorische techniek. Het tweede soort algoritme is gebaseerd op het toevoegen van monsters of toestanden. Aangezien het eerste soort algoritme niet bruikbaar is voor de toepassing van dit eindwerk (zie verderop in dit deel), namelijk het leegmaken van een bak met willekeurig geplaatste werkstukken m.b.v. een zesassige robot, wordt dat soort algoritme enkel in het kort besproken. [2]

Combinatorische methoden maken gebruik van de configuratieruimte die volledig op voorhand is opgebouwd. Het opbouwen van de configuratieruimte duurt zeer lang aangezien het algoritme iedere toestand van de robot moet controleren op zijn geldigheid (‘2.2.1 Geldigheid van *vertices* en *edges*’). Ook is er enorm veel geheugen nodig om de configuratieruimte in op te slaan. Het voordeel van deze methode is dat het een exacte methode is waardoor het algoritme optimale paden als resultaat teruggeeft. Voor iedere toestand weet het algoritme namelijk met zekerheid of het geldig is of niet aangezien iedere toestand gecontroleerd is op zijn geldigheid. Dat voordeel weegt niet op tegen de nadelen zoals de hoge rekentijd en de nood aan een groot geheugen.

Combinatorische methoden zijn bijgevolg enkel nuttig om te gebruiken bij problemen die laag dimensionaal, eenvoudig en statisch zijn. Een dynamische omgeving zal er namelijk voor zorgen dat het algoritme, iedere keer dat er iets verandert in de omgeving, een nieuwe configuratieruimte moet opbouwen wat enorm tijds- en rekenintensief is. Bij een hoge complexiteit zoals bij deze masterproef duurt het heel lang om een configuratieruimte op te bouwen aangezien het in dit geval 6-dimensionaal is en dus uit enorm veel mogelijke toestanden bestaat. [2]

Het tweede soort algoritme, dat gebaseerd is op het toevoegen van monsters, is wel bruikbaar in de probleemstelling van deze masterproef. Dit soort algoritme maakt gebruik van de benaderende methode ('2.2.1 Geldigheid van *vertices* en *edges*': tweede methode voor het controleren van een verbinding: de benaderende methode). De hele werkwijze van dit soort padplanningsalgoritme is gebaseerd op het toevoegen van willekeurige monsters in het werkingsgebied van de robot. Dat wil zeggen dat de algoritmen een willekeurige toestand genereren en deze controleren op haar geldigheid. Na verschillende stappen te doorlopen, die afhankelijk zijn van het algoritme ('Hoofdstuk 3: Padplanningsalgoritmen in theorie'), genereert het algoritme een volgend monster. Dat blijft zich herhalen tot de eindvoorwaarden bereikt zijn. Deze eindvoorwaarden zijn ook afhankelijk van het algoritme.

Aangezien dit soort algoritme gebruikmaakt van 'willekeurig' gegenereerde monsters is er minder geheugen nodig en is de methode compacter dan de combinatorische methode. De gegenereerde monsters zijn theoretisch gezien niet volledig willekeurig aangezien de meeste algoritmen als voorwaarde hebben dat de monsters binnen een bepaald gebied moeten liggen. Een voorbeeld van zo een restrictie die geldt bij ieder algoritme is dat het algoritme enkel monsters mag genereren die binnen het werkingsgebied van de robot liggen. Door de willekeurigheid van de gegenereerde monsters is het berekende pad vaak niet het optimale pad. De kans is namelijk groot dat er een andere toestand in de buurt van het gegenereerde monster ligt, dat voor een optimaler pad zorgt dan het berekende pad. Dat nadeel weegt niet zwaar door in de beoordeling van het type algoritme aangezien de berekende paden vaak het optimale pad goed benaderen. De kleine verbetering die nog mogelijk is tot het optimale pad staat in schril contrast met de nadelen, namelijk extra rekentijd en meer nood aan geheugen. Daardoor hebben de algoritmen meer tijd beschikbaar om complexere problemen op te lossen. Het terugkrijgen van een botsingsvrij pad is immers al een zeer goed resultaat, ook al is dat pad niet volledig optimaal.

Algoritmen die gebaseerd zijn op het genereren van willekeurige monsters bevatten nog een tweede nadeel ten opzichte van combinatorische algoritmen: de kans dat het algoritme een pad vindt is lager. Algoritmen gebaseerd op monsters hebben meestal een parameter dat het maximaal aantal geïntroduceerde monsters bepaalt. Als dat maximum bereikt is, is er een eindvoorwaarde bereikt en stopt het algoritme met nieuwe monsters toe te voegen. De kans om een pad te vinden is voor beide types gelijk indien het aantal geïntroduceerde monsters wiskundig naar oneindig gaat. In de praktijk staat er echter altijd een limiet op het aantal toegevoegde monsters waardoor er minder kans is dat dit type algoritme een pad vindt. Ook dit nadeel heft bij complexe problemen het voordeel van de verminderde rekentijd niet op.

Nochtans vinden algoritmen die gebruikmaken van monsters in sommige situaties wel een oplossing terwijl de combinatorische technieken falen. Dat komt doordat algoritmen gebaseerd op monsters een vaak voorkomend nadeel overbruggen: lokale minima. Lokale minima zijn plaatsen waar een padplanningsalgoritme in vast kan blijven steken. Stel bijvoorbeeld dat de structuur die tot nu toe bekomen is zeer dicht bij de eindtoestand ligt en dat de structuur nog een kleine stap verder moet zetten om het doel te bereiken. Het algoritme weet echter niet dat

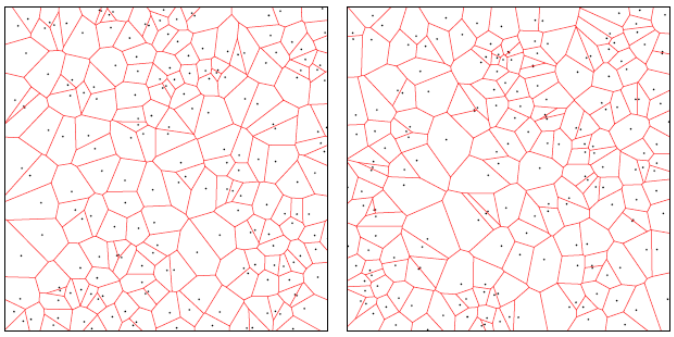
tussen het einde van de structuur en de eindtoestand een heel breed obstakel ligt zoals op figuur 8 te zien is. Combinatorische methoden blijven in de richting van de eindtoestand gaan aangezien het algoritme denkt dat het er kortbij zit. Dat algoritme zit vast in een zogenaamd lokaal minimum en blijft aan de rand van het obstakel steken zoals in figuur 8 te zien is. Algoritmen gebaseerd op monsters voegen daarentegen willekeurige monsters toe waardoor het algoritme ook richting de andere kant ontdekt. Uiteindelijk zal het algoritme rond het obstakel tot de eindtoestand geraken terwijl dat bij combinatorische algoritmen niet het geval is. [2] [5] [6] [9]

Figuur 8: Algoritme dat vastzit in een lokaal minimum [2]

2.3.2 Het toevoegen van monsters

Zoals in het vorige deel vermeld, is de probleemstelling van dit eindwerk oplosbaar met behulp van padplanningsalgoritmen die gebaseerd zijn op het toevoegen van monsters, maar hoe voegt het algoritme deze monsters toe? Het is belangrijk om te weten dat er verschillende eisen gesteld worden aan de monsters. Ten eerste mogen ze niet allemaal op dezelfde plaats liggen, ze moeten dus verspreid over het werkingsgebied liggen. Ten tweede mag het algoritme de monsters ook niet uniform verdelen over het werkingsgebied aangezien dit de willekeurigheid tegenspreekt. Om aan deze twee eisen te kunnen voldoen, maakt het algoritme gebruik van zogenaamde *Voronoi*-gebieden. Die gebieden illustreren hoe ver monsters van elkaar liggen en zorgen voor een controle zodat er geen grote niet-verkende gebieden meer overblijven in de werkingsruimte. Grote *Voronoi*-gebieden komen overeen met grote afstanden tussen monsters, wat overeenkomt met een gebied dat weinig verkend is.

Het construeren van de *Voronoi*-gebieden gebeurt op een bepaalde manier. Om dit te illustreren staan er twee voorbeelden geïllustreerd in figuur 9. Ieder punt op de afbeelding komt overeen met een monster dat gegenereerd is door een algoritme. In de eerste stap wordt tussen een bepaald monster en zijn buren een lijn getrokken die even ver ligt van beide monsters. Dat doet het systeem voor alle monsters. In stap twee houdt het systeem enkel de delen van een lijn over waar die lijn de dichtstbijzijnde lijn is voor de monsters waarbij die lijn hoort. Op die manier ontstaat er een veelhoek rond ieder monster. Het kenmerk van *Voronoi*-gebieden is dat ieder punt binnen het gebied van de veelhoek als dichtstbijzijnde monster het monster van dat gebied heeft. [2] [10]

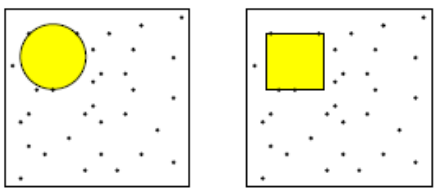


Figuur 9: Voorbeelden Voronoi-gebieden [2]

Algoritmen voegen dus monsters toe aan de werkingsruimte tot een maximaal aantal van toegevoegde monsters bereikt is of tot er een pad gevonden is. De tweede eindvoorwaarde geldt enkel bij bepaalde algoritmen ('2.3.4 Algoritmen gebaseerd op een boomstructuur'). Een eerste manier om ervoor te zorgen dat het volledige werkingsgebied verspreid bemonsterd wordt, is gebruikmaken van de *Voronoi*-gebieden. Algoritmen geven namelijk de voorkeur om te bemonsteren in grote *Voronoi*-gebieden aangezien daar nog niet veel verkend is. Door dit herhaald toe te passen zijn de monsters verspreid over het werkingsgebied terwijl het algoritme de (beperkte) willekeurigheid behoudt. [2] [3] [5] [9]

Een tweede manier maakt gebruik van totale willekeurigheid om een verspreide bemonstering in een werkingsgebied te verkrijgen. Aangezien het systeem hier geen voorkeur heeft om te bemonsteren in gebieden waar nog niet veel verkend is, is de kans reëel dat er op het begin lege gebieden ontstaan. Om dit te voorkomen voegt het systeem "oneindig" veel monsters toe. Wiskundig gezien is de kans namelijk 100% dat een werkingsgebied verspreid bemonsterd is indien het systeem oneindig veel monsters toevoegt. Aangezien het onmogelijk is om in de praktijk oneindig lang te bemonsteren blijft er een kans dat er lege gebieden in de werkingsomgeving zijn. Vandaar dat algoritmen deze manier in de praktijk niet toepast.

Er is nog een derde manier om een werkingsgebied verspreid te bemonsteren. Deze manier maakt gebruik van geometrische figuren en heet *low-disperion sampling* of 'lage-spreiding bemonstering'. Net als bij de eerste manier maakt deze manier gebruik van een beperkte willekeurigheid aangezien de bemonstering beperkt wordt tot een bepaald gebied. In het hele werkingsgebied genereert de methode een geometrische figuur zoals in figuur 10 (links een cirkel en rechts een vierkant).



Figuur 10: low-dispersion sampling geometrische figuren [2]

Dit gebeurt door zo groot mogelijke geometrische figuren te maken in de werkingsomgeving zonder dat er een monster binnen de geometrische figuur valt. Door dit overal in het werkingsgebied toe te passen, genereert de methode verschillende geometrische figuren. Daarna bepaalt de methode de grootste geometrische figuur aan de hand van de afmetingen of de oppervlakte, omdat deze toch proportioneel zijn ten opzichte van elkaar. De volgende bemonstering zal vervolgens plaatsvinden binnen deze geometrische figuur zodat dit grote niet-verkende gebied verder verkend wordt waardoor de monsters uiteindelijk verspreid zijn over de werkingsruimte. De methode maakt meestal gebruik van een cirkel als geometrische figuur aangezien deze het makkelijkste te implementeren is. [2]

Een vierde manier maakt gebruik van een gestructureerd rooster om verspreid te bemonsteren in een werkingsgebied waardoor het een onwillekeurige manier is. Uniform bemonsteren en de methode van Sukharev zijn hier enkele voorbeelden van. Aangezien dit niet willekeurig is en omdat algoritmen pas kunnen beginnen zoeken naar een pad indien de volledige werkingsruimte bemonsterd is, maken algoritmen hier geen gebruik van in de praktijk waardoor deze niet verder besproken worden in de scriptie. Dat wil zeggen dat algoritmen altijd gebruikmaken van de tweede of derde manier van bemonsteren, namelijk het gebruik van *Voronoi*-gebieden of geometrische figuren. In ieder geval maken algoritmen gebruik van beperkte willekeurigheid om te bemonsteren. Dat omdat volledige willekeurigheid te veel tijd in beslag neemt (oneindig lang bemonsteren) en omdat bij onwillekeurig bemonsteren gewacht moet worden tot de volledige werkingsruimte bemonsterd is. [2] [11] [12]

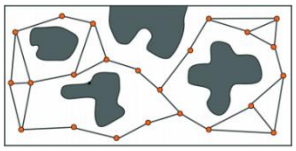
Er zijn twee categorieën van gerandomiseerde padplanningsalgoritmen. Een eerste categorie baseert zich op het maken van een wegenkaart. De algemene stappen van deze categorie staan vermeld in ‘2.3.3 Algoritmen gebaseerd op een wegenkaart’. De werking en eigenschappen van de specifieke algoritmen die behoren tot deze categorie staan besproken in ‘3.1 Wegenkaartalgoritmen’. De tweede categorie maakt gebruik van een boomstructuur om paden te vinden. De algemene stappen van deze categorie staan vermeld in ‘2.3.4 Algoritmen gebaseerd op een boomstructuur’. De werking en eigenschappen van de specifieke algoritmen die gebruikmaken van een boomstructuur zijn besproken in ‘3.2 Boomstructuuralgoritmen’. Het verschil in toepassingsgebied staat vermeld in ‘2.3.5 Verschil wegenkaart- en boomstructuuralgoritmen’. [2] [3] [5] [9]

2.3.3 Algoritmen gebaseerd op een wegenkaart

De eerste categorie van algoritmen maakt gebruik van een *roadmap* of wegenkaart. In de eerste stap genereren de algoritmen willekeurige monsters zoals in '2.3.2 Het toevoegen van monsters' besproken is. Het bemonsteren blijft doorgaan tot het maximaal aantal gegenereerde monsters bereikt is, wat een instelbare parameter is. Een voorbeeld van een bemonsterde omgeving staat afgebeeld in figuur 11 waar de grijze gebieden overeenkomen met obstakels en de rode punten met monsters of robottoestanden.

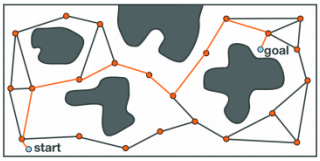
Figuur 11: Een bemonsterde werkingsruimte [5]

Deze categorie van algoritmen verbindt in de tweede stap de *vertices* met elkaar zodat er *edges* ontstaan in de configuratieruimte. Deze verbindingen mogen niet doorheen obstakels gaan om botsingen te voorkomen. Figuur 12 illustreert deze stap, toegepast op het voorbeeld. Na deze stap is ook meteen duidelijk waarom de naam van deze categorie verwijst naar een wegenkaart. De naam vindt namelijk zijn oorsprong in de visuele gelijkenis tussen de structuur, wat de combinatie van de *vertices* en *edges* is, en een wegenkaart waar de *vertices* overeen komen met steden en de *edges* met verbindingswegen.



Figuur 12: Het creëren van edges [5]

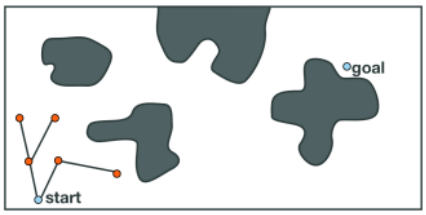
In de laatste stap voegen de algoritmen de begin- en eindtoestand toe aan de configuratieruimte als *vertices*. De algoritmen proberen dan deze *vertices* te verbinden met de wegenkaart op dezelfde manier als in de tweede stap. Indien dit met succes is gebeurt, hebben de algoritmen een pad gevonden en geven ze dit pad als resultaat terug. In figuur 13 is deze stap te zien, toegepast op het voorbeeld. [2] [3] [5] [9]



Figuur 13: Het toevoegen en verbinden van begin- en eindtoestand [5]

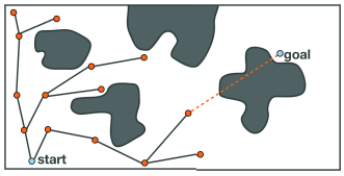
2.3.4 Algoritmen gebaseerd op een boomstructuur

De tweede categorie van algoritmen maakt gebruik van een boomstructuur. De algoritmen van deze categorie voegen als eerste stap de begintoestand toe aan de structuur. Vervolgens voegen de algoritmen een willekeurig monster toe aan de structuur. Het toevoegen van willekeurige monsters gebeurt analoog als in '2.3.2 Het toevoegen van monsters'. Daarna verbindt het algoritme de nieuwe vertex meteen met de bestaande boomstructuur zoals in figuur 14 afgebeeld staat.

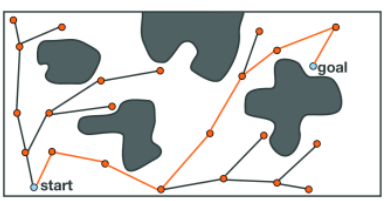


Figuur 14: Start van de boomstructuur [5]

Het toevoegen en verbinden van monsters blijft zich herhalen tot er een pad gevonden is, maar hoe kan er een pad gevonden worden? Iedere keer dat een *edge* tot stand is gebracht en er een nieuw monster toegevoegd moet worden is er een bepaalde kans dat de eindtoestand de nieuwe monster is. De kans dat dit voorvalt is een instelbare parameter. De algoritmen trachten vervolgens de eindtoestand te verbinden met de boomstructuur. Lukt dit niet, dan verwijdert het algoritme de eindtoestand terug en herhaalt de procedure zich zoals in figuur 15 geïllustreerd staat. Als de verbinding daarentegen wel geldig is, dan is er een pad gevonden zoals in figuur 16 te zien is. De algoritmen geven dan dat pad als resultaat terug. De benaming van boomstructuur vindt zijn oorsprong in de visuele gelijkenis tussen een boom en de structuur van de algoritmen. De algoritmen maken in het begin namelijk grote verbindingen (dikke takken) en naarmate de tijd vordert voegen de algoritmen steeds kleinere verbindingen toe (dunnere takjes op de dikke takken). [2] [3] [5] [9]



Figuur 15: Foutieve verbinding van begintoestand met boomstructuur [5]



Figuur 16: Correcte verbinding van begintoestand met boomstructuur [5]

2.3.5 Verschillen wegenkaart- en boomstructuuralgoritmen

Ondanks de gelijkenissen van het bemonsteren bij beide categorieën zijn er ook enkele verschillen. Boomstructuuralgoritmen proberen sneller naar een oplossing te grijpen dan de wegenkaartalgoritmen aangezien deze eerst de omgeving volledig bemonsteren. Boomstructuuralgoritmen zijn bijgevolg vooral bruikbaar in dynamische, grote omgevingen. Wegenkaartalgoritmen zijn daarentegen vooral nuttig in statische omgevingen aangezien de volledige omgeving bemonsterd is en die bemonstering dan behouden kan worden omdat de omgeving toch niet verandert. Bij dynamische toepassingen komt het regelmatig voor dat de wegenkaart niet meer correct is zodat het algoritme opnieuw moet beginnen met de bemonstering. Dat houdt ook in dat de omgeving niet te groot mag zijn aangezien het dan veel langer duurt om een volledige omgeving te bemonsteren.

Toch zijn er verscheidene problemen met een zeer grote omgeving die gebruikmaken van een wegenkaart. Hierbij moet de omgeving wel zeer statisch zijn. Een voorbeeld hiervan is een huis waar een robot in kan rondwandelen. De muren en meubels van het huis veranderen niet (of nauwelijks) waardoor een wegenkaart in dat geval een betere keuze is dan iedere keer een boomstructuur op te bouwen. Een ander voorbeeld van een groot statisch probleem is een landkaart met wegen op. Deze veranderen ook nauwelijks en op deze kaart kan men iedere keer opnieuw een route uitstippelen zonder de kaart opnieuw op te bouwen. [2] [3] [5] [9]

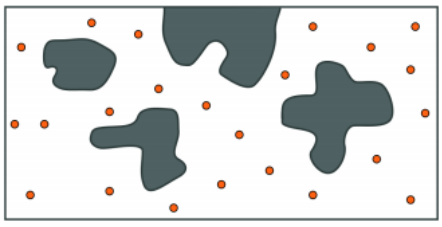
3 Padplanningsalgoritmen in theorie

In de literatuurstudie staat vermeld dat er twee categorieën van algoritmen zijn. Dat zijn algoritmen die zich baseren op een wegenkaart en algoritmen die zich baseren op een boomstructuur. De algemene stappen van de categorieën zijn reeds in de literatuurstudie besproken. Dit deel bespreekt specifieke varianten van de twee basisalgoritmen, die behoren tot de twee categorieën, en die in de *Motion Planning* bibliotheek OMPL zijn geïmplementeerd. De werking, de opbouw en de voor- en nadelen van deze algoritmen zijn in dit hoofdstuk besproken. In 'Hoofdstuk 6: Resultaten' worden de resultaten besproken, in de evaluatie wordt er regelmatig naar dit theoretisch hoofdstuk terugverwezen.

3.1 Wegenkaartalgoritmen

3.1.1 PRM

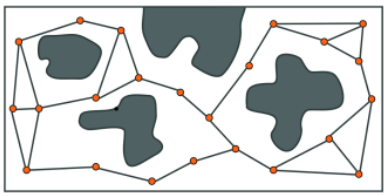
Het eerste en tevens meest gebruikte algoritme dat gebruikmaakt van een wegenkaart is PRM of *Probabilistic Road Map*. Het algoritme bestaat uit twee fasen, namelijk de opbouw- en de afvraagfase. In de opbouwfase vertrekt het algoritme van een volledig lege structuur. In deze lege structuur voegt het algoritme willekeurige monsters toe ('2.3.2 Het toevoegen van monsters'). De botsingsdetectie controleert ieder monster op zijn geldigheid ('2.2 Botsingsdetectie'). De monsters die geldig zijn worden toegevoegd aan de structuur of kaart als *vertices* terwijl de andere monsters verwijdert worden. PRM blijft bemonsteren tot de maximale waarde van toegevoegde monsters bereikt is. Deze maximale waarde is een instelbare parameter voor het algoritme. De *map* of kaart is nu opgebouwd uit monsters zoals in figuur 17 te zien is. In de figuur komen de grijze gebieden overeen met obstakels en de rode punten met monsters of robottoestanden.



Figuur 17: Een bemonsterde werkingsruimte [5]

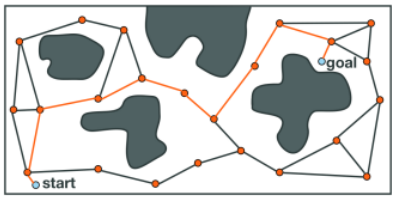
In het tweede deel van de opbouwfase gaat het PRM-algoritme voor iedere *vertex* controleren of het algoritme deze *vertex* kan verbinden met andere *vertices* in de structuur. De botsingsdetectie controleert de verbinding, die twee *vertices* met elkaar verbindt, op zijn geldigheid ('2.2 Botsingsdetectie'). Indien de verbinding geldig is, voegt het algoritme deze verbinding als *edge* toe aan de structuur. Indien de verbinding ongeldig is, verwijdert het algoritme de verbinding. PRM probeert dus iedere *vertex* te verbinden met andere *vertices* in de structuur, maar niet met alle andere *vertices* aangezien dit veel berekeningstijd in beslag neemt. PRM gebruikt daarom een parameter om aan te duiden hoeveel connecties een *vertex* maximaal mag proberen te maken.

Er zijn twee manieren om deze parameter te implementeren. Een eerste manier zorgt ervoor dat een *vertex* enkel verbinding mag proberen te maken met de 'k' dichtstbijzijnde punten (met k als instelbare parameter). Een tweede manier is dat het algoritme met alle *vertices*, die binnen een bepaalde radius 'r' liggen, een verbinding probeert te maken (met r als instelbare parameter). Deze tweede mogelijkheid kan nog altijd veel rekentijd in beslag nemen als het aantal gegenereerde monsters zeer hoog is. Daarom gebruikt deze tweede mogelijkheid vaak een tweede beperking die een maximum stelt aan het aantal mogelijke verbindingen. Als deze verbindingen voor iedere *vertex* gemaakt zijn, is er een volledige wegenkaart of 'Road Map' opgebouwd zoals in figuur 18 te zien is.



Figuur 18: Een opgebouwde wegenkaart [5]

In de tweede fase, de afvraagfase, voegt het algoritme de begin- en eindtoestand toe aan de structuur. Het PRM-algoritme probeert vervolgens deze twee *vertices* te verbinden met de wegenkaart. Deze toestanden maken maar één verbinding met de wegenkaart. Indien deze verbinding geldig is, voegt het algoritme de verbinding toe aan de structuur als *edge.* Indien dat niet het geval is, verwijdert het algoritme de verbinding en probeert het algoritme een nieuwe verbinding te zoeken met de wegenkaart. Als beide toestanden verbinding hebben gemaakt met de structuur heeft het algoritme een pad gevonden waarop het algoritme dat pad als resultaat teruggeeft. Een mogelijk pad voor het voorbeeld is weergegeven in figuur 19. [3] [5] [9] [13]



Figuur 19: Een gevonden pad als oplossing [5]

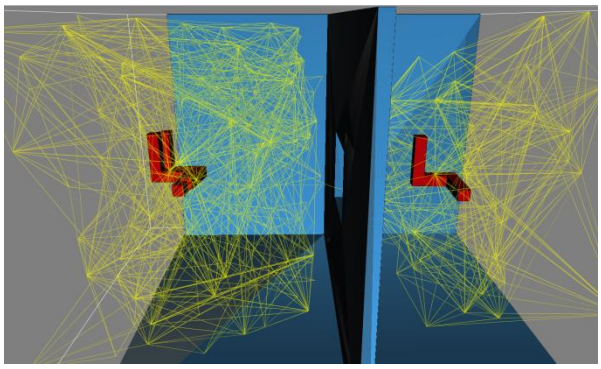
3.1.2 PRM*

PRM* of *Probabilistic Road Map Star* of *Optimal Probabilistic Road Map* is een variant van PRM met één groot verschil. PRM* zoekt namelijk niet naar eender welk geldig pad, maar zoekt naar een optimaal pad. De verwijzing ster (*) geeft namelijk aan dat het een optimale versie is van de standaardmethode. PRM* bevat dus dezelfde stappen als PRM, op de kleine aanpassing na om het pad optimaler te maken. Het resultaat is dus beter, maar er is meer berekeningstijd voor nodig.

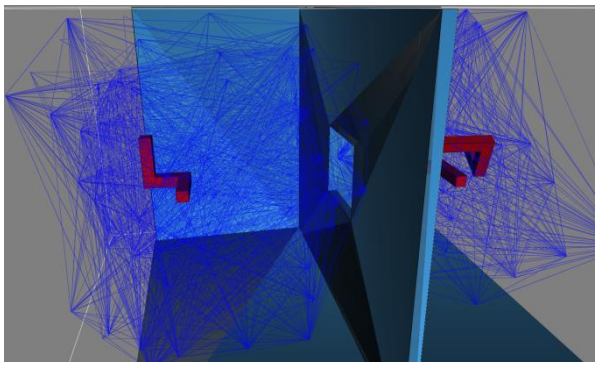
Het eerste deel van de opbouwfase, het generen van monsters en dus *vertices*, gebeurt volledig analoog als bij PRM. Het tweede deel van de opbouwfase, het creëren van verbindingen en dus *edges*, bevat echter een kleine aanpassing. PRM probeert iedere *vertex* met een aantal (k) van zijn dichtstbijzijnde buren te verbinden of probeert iedere *vertex* met alle *vertices* binnen een bepaalde radius (r) te verbinden, met een eventuele bovengrens voor het aantal verbindingen. PRM* verbetert dat door de radius in functie van het aantal toegevoegde monsters (n) te schrijven. De grootheden zijn omgekeerd evenredig ten opzichte van elkaar. Dat wil zeggen dat indien er veel monsters zijn toegevoegd op het einde van het eerste deel van de opbouwfase, iedere *vertex* enkel verbinding zal proberen te maken met andere *vertices* die binnen een kleine radius liggen. Als er daarentegen maar weinig toegevoegde monsters zijn, zal iedere *vertex* verbinding proberen te maken met alle *vertices* die binnen een grote radius liggen. Op die manier bekomt het algoritme optimalere resultaten dan bij PRM.

PRM* heeft dezelfde toepassingsgebieden als PRM namelijk statische, kleine omgevingen. De algoritmen worden soms ook wel *multiple-query* algoritmen genoemd aangezien het algoritme de opbouwfase niet opnieuw moet doen indien de omgeving niet verandert is. Het algoritme kan dan simpelweg het nieuwe probleem, met zijn nieuwe begin- en eindtoestand, beginnen bij de afvraagfase. Het grote voordeel van PRM* ten opzichte van PRM is het gebruik van de parameter die de radius in functie van het aantal toegevoegde monsters beschrijft. Door deze verbetering heeft het algoritme altijd een optimale verhouding tussen de nauwkeurigheid (veel verbindingen) en rekentijd (niet te veel verbindingen). Bij PRM is het mogelijk dat de nauwkeurigheid te laag ligt of de rekentijd te hoog is als de parameters slecht ingesteld zijn. PRM heeft dus nog een fijnregeling nodig van zijn parameters terwijl PRM* dit automatisch doet. De volgende figuren illustreren dat de oplossing van PRM afhangt van de instellingen van de parameter terwijl PRM* hier een balans tussen zoekt en iedere keer ongeveer hetzelfde aantal *vertices* en *edges* verkrijgt. [3] [5] [9] [13] [14]

Figuur 20 maakt gebruik van PRM en heeft als parameter k = 15 (aantal buren waarmee een *vertex* verbinding probeert te maken). Figuur 21 laat daarentegen een voorbeeld van PRM zien met k = 60. Er is een groot verschil te zien tussen de twee instellingen. Figuur 21 heeft namelijk enorm veel *edges* en *vertices* terwijl figuur 20 er beduidend minder heeft.

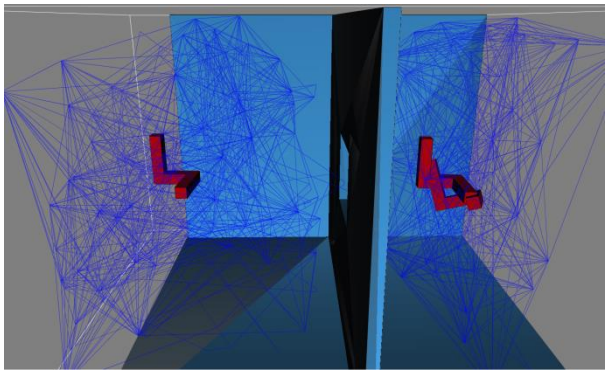


Figuur 20: Voorbeeld PRM met k = 15

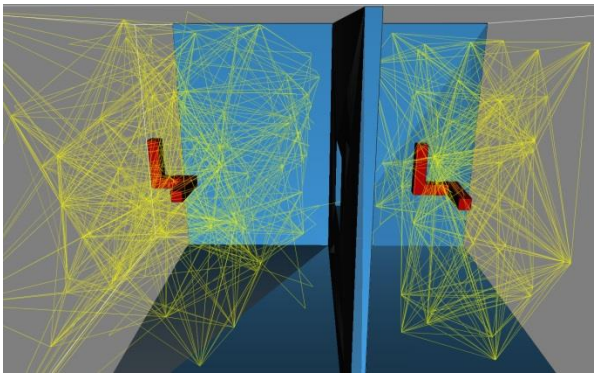


Figuur 21: Voorbeeld PRM met k = 60

Figuren 22 en 23 laten hetzelfde voorbeeld zien, maar opgelost met PRM*. Zoals op de figuren te zien is, ligt het totaal aantal *vertices* en *edges* tussen de twee extreme waarden van PRM in. Ook blijft deze hoeveelheid altijd ongeveer hetzelfde bij PRM*, wat duidelijk maakt dat het algoritme voor hetzelfde probleem bijna altijd hetzelfde resultaat geeft. Ook blijkt uit de figuren dat er een goede balans is tussen de nauwkeurigheid en rekentijd bij PRM*.



*Figuur 23: Voorbeeld 1 van PRM**



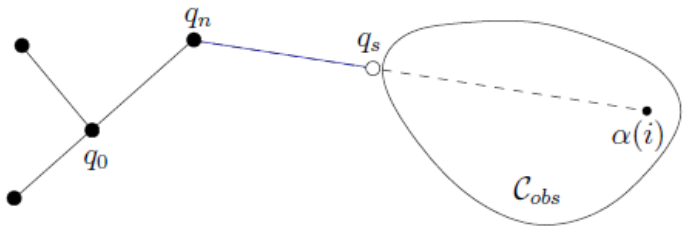
*Figuur 22: Voorbeeld 2 van PRM**

3.2 Boomstructuuralgoritmen

3.2.1 RRT

Het eerste en tevens meest gebruikte algoritme is RRT (*Rapidly exploring Random Tree*). Het is een algoritme dat efficiënt paden vindt in hoog-dimensionale en dynamische ruimtes. RRT bouwt een boomstructuur incrementeel op. Dat wil zeggen dat het de structuur stap voor stap opbouwt. Initieel bevat de werkingsruimte net als bij de andere algoritmen nog geen *vertices* en *edges*.

Als eerste stap voegt het algoritme de begin- en eindtoestand toe aan de werkingsruimte als *vertices*. Daarna voegt het algoritme als tweede stap een willekeurig gegenereerd monster toe ('2.3.2 Het toevoegen van monsters') aan de structuur. Vervolgens probeert het algoritme een verbinding te maken tussen dat monster en de boomstructuur. Op dit moment bestaat de boomstructuur enkel uit de begintoestand dus probeert het toegevoegde monster verbinding te maken met de begintoestand. Is deze verbinding geldig, dan voegt het algoritme de verbinding toe aan de structuur als *edge* en het monster als *vertex*. Als deze verbinding niet geldig is, dan deelt het algoritme deze verbinding op in deeltoestanden en voegt het algoritme de deeltoestand toe aan de structuur die de langst geldige verbinding maakt met de structuur. Ook wanneer een monster ongeldig is, voegt het algoritme een *vertex* toe aan de structuur die op de maximale geldige lengte ligt van de structuur zoals op onderstaande afbeelding te zien is (figuur 24).



Figuur 24: RRT met ongeldig toegevoegd monster [5] met:

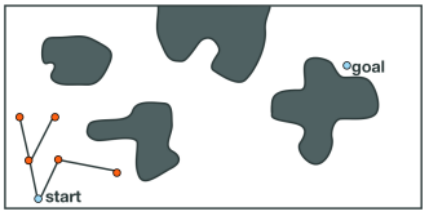
- α(i): het gegenereerde, ongeldige monster

- C_{obs}: het obstakel

- q_n: vertex van de boomstructuur dat het dichtst bij het gegenereerde monster ligt

- q_s: de nieuwe vertex die de verst mogelijke, geldige verbinding maakt tussen de boomstructuur en het ongeldige monster

De tweede stap, het genereren van nieuwe monsters, herhaalt zich. RRT voegt altijd een nieuw monster toe in de grootste *Voronoi*-gebieden van de werkingsruimte ('2.3.2 Het toevoegen van monsters': *Voronoi*-gebieden). Ieder *Voronoi*-gebied krijgt met andere woorden een waarschijnlijkheid dat het volgende willekeurige monster binnen dat *Voronoi*-gebied ligt die proportioneel is met de grootte van het *Voronoi*-gebied. Dat zorgt ervoor dat het algoritme snel grote delen verkent en daarna steeds meer in detail gaat zoeken (kleinere takken van een boom). Het algoritme probeert ieder nieuw monster te verbinden met de boomstructuur zodat de boomstructuur steeds aan grootte toeneemt (figuur 25).

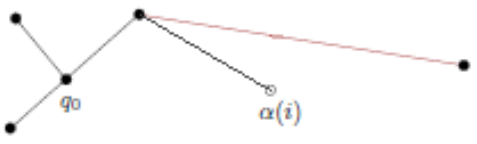


Figuur 25: Het opbouwen van boomstructuur door toevoegen en verbinden van monsters [5]

Deze tweede stap blijft zich herhalen tot een eindvoorwaarde bereikt is. Een algoritme stopt met doorzoeken indien:

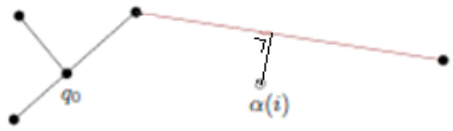
- de maximale tijd verstreken is;
- het maximaal aantal toegevoegde monsters bereikt is;
- de structuur voldoende dicht is (alle *Voronoi*-gebieden zitten onder een bepaalde grootte);
- een pad gevonden is.

Het verbinden van een gegenereerde *vertex* (α(i)) met de boomstructuur gebeurt op een andere manier dan bij wegenkaartalgoritmen. De *vertex* zal namelijk maar één verbinding maken met de boomstructuur. Er zijn verschillende manieren om een nieuwe vertex te verbinden met de boomstructuur. In een eerste manier tracht de *vertex* een verbinding te maken met de dichtstbijzijnde vertex van de boomstructuur zoals in figuur 26. Deze manier is het minst nauwkeurig van de drie manieren, maar neemt daarentegen ook het minste rekentijd in beslag.



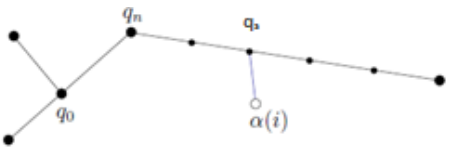
Figuur 26: Verbinding vertex met dichtstbijzijnde vertex van boomstructuur [2]

In een tweede manier probeert het algoritme de toegevoegde *vertex* te verbinden met het dichtstbijzijnde punt van de boomstructuur zoals in figuur 27 afgebeeld staat. Dat punt kan zowel een *vertex* als een *edge* zijn waardoor de nieuwe *edge* loodrecht op de boomstructuur staat, tenzij de *edge* verbinding maakt met een *vertex*. Deze manier is het meest nauwkeurige van de drie manieren, maar neemt ook het meeste tijd in beslag.



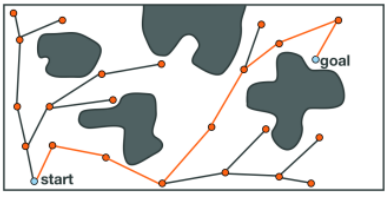
Figuur 27: Verbinding vertex met dichtstbijzijnde punt van boomstructuur [2]

In de derde manier deelt het algoritme de dichtstbijzijnde *edge* op in deel*vertices* met een vaste afstand tussen de deel*vertices*. Het algoritme verbindt het nieuwe monster met de dichtstbijzijnde *vertex* (of deel*vertex*) van de boomstructuur zoals in figuur 28 gedemonstreerd is. Deze manier ligt tussen de twee vorige manieren in aangezien het algoritme hier een middenweg zoekt tussen de nauwkeurigheid en de rekentijd. Om die reden passen algoritmen deze derde manier toe in de praktijk.

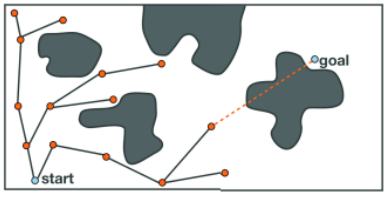


Figuur 28: Verbinding vertex met dichtstbijzijnde (deel)vertex van boomstructuur [2]

Zoals eerder vermeld kan het algoritme ook vroegtijdig stoppen indien er al een pad gevonden is, maar hoe kan de eindtoestand een verbinding maken met de boomstructuur? Indien het algoritme een *edge* heeft geconstrueerd, moet het algoritme een nieuw monster toevoegen. Iedere keer dat dit gebeurt, bestaat de kans erin dat de eindtoestand dat nieuwe monster is. Deze kans is een parameter van RRT en noemt *GoalBias*. Indien de eindtoestand het nieuwe monster is, probeert het algoritme een verbinding te maken met de boomstructuur. Als dat met succes gebeurt, is er een pad gevonden en geeft het algoritme dat pad als resultaat terug zoals in figuur 29. Als de verbinding daarentegen niet geldig is, verwerpt het algoritme de verbinding zoals in figuur 30 en vervolgt RRT zijn procedure met stap twee. [2] [5] [9] [15] [16] [17]

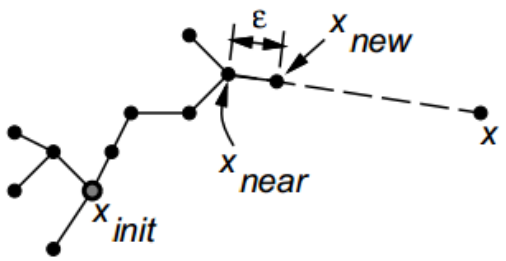


Figuur 29: Correcte verbinding en gevonden pad [5]



Figuur 30: Ongeldige verbinding en verwerping verbinding [5]

Een kleine variant op RRT beperkt de maximale lengte van een verbinding door middel van een instelbare parameter. Stel bijvoorbeeld dat een nieuw monster is toegevoegd en dat deze een geldige verbinding kan maken met de boomstructuur, maar dat deze verbinding een lengte heeft van 15 centimeter. Als de parameter echter bepaalt dat een verbinding een maximale afstand van 10 centimeter mag overbruggen, dan zal op 10 centimeter van de boomstructuur in de richting van het toegevoegde monster een *vertex* gecreëerd worden. Het algoritme verwijdert vervolgens het oorspronkelijke monster. Een voorbeeld hiervan is in figuur 31 afgebeeld. De maximale lengte van een *edge* is aangeduid met ε. In de figuur is x het willekeurige monster, x_{near} de kortste *vertex* van de boomstructuur tot x. X_{new} is de nieuwe *vertex* op de maximale afstand ε van de boomstructuur in de richting van x. [2] [16] [18]



Figuur 31: De maximale lengte van een edge [16]

3.2.2 RRT-Connect

RRT-Connect is een uitbreiding op het algemene algoritme RRT. Het algoritme draagt soms ook de naam van bi-directionele-RRT omdat het gebruikmaakt van twee boomstructuren die naar elkaar toegroeien wat tevens het grote verschil is met RRT. Eén boomstructuur vertrekt vanaf de begintoestand en de andere boomstructuur vertrekt vanaf de eindtoestand. Het algoritme bouwt beide boomstructuren op zoals dat bij RRT het geval is. De enige wijziging in opbouw van de structuur is dat er om beurt één boomstructuur verder groeit. Zo zal bijvoorbeeld eerst de boomstructuur vertrekkende van de begintoestand een monster toevoegen en proberen te verbinden, daarna de andere boomstructuur en vervolgens terug de eerste boomstructuur enz.

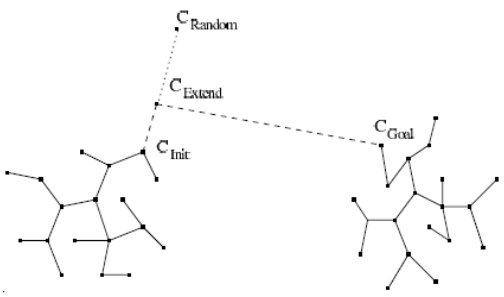
De moeilijkheid van RRT-Connect is dat het algoritmeervoormoet zorgen dat de twee boomstructuren naar elkaar toegroeien en dat ze verbinding maken met elkaar zodat er een pad als resultaat is. Bij het genereren van een nieuw monster probeert het algoritme, net als bij RRT, een verbinding te maken met de boomstructuur die op dat moment actief is. Het algoritme probeert na een geldige verbinding deze *vertex* ook te verbinden met de andere boomstructuur. Als deze verbinding geldig is, zijn beide boomstructuren verbonden met elkaar en is er een pad gevonden. Als dit niet het geval is, gaat het algoritme gewoon verder met het bemonsteren.

Figuur 32 laat een toepassing zien waar het gebruik van meerdere boomstructuren een groot voordeel oplevert. De groene boomstructuur vertrekt vanaf de begintoestand en de rode boomstructuur vertrekt vanaf de eindtoestand. Het is makkelijker voor het algoritme om het obstakel binnen te geraken dan eruit te geraken waardoor het gebruik van een tweede boomstructuur, vertrekkende vanaf de eindtoestand, een groot voordeel oplevert.

Figuur 32: Voorbeeld RRT-Connect [2]

RRT-Connect is dus vooral nuttig om te gebruiken voor toepassingen waar er een obstakel ligt tussen of in de buurt van de begin- en eindtoestand. Nochtans is het algoritme ook bruikbaar in toepassingen zonder obstakels, maar in die toepassingen is RRT ook voldoende aangezien het gemiddeld gezien even snel een oplossing zal vinden. RRT heeft namelijk geen berekeningstijd nodig om de verbinding tussen de twee boomstructuren te maken. In de praktijk bestaan er ook algoritmen die meer dan twee boomstructuren gebruiken. De extra complexiteit om de verschillende bomen met elkaar te verbinden zorgt ervoor dat algoritmen zelden gebruikmaken van meer dan twee boomstructuren wegens een te hoge berekeningstijd.

In figuur 33 staat een voorbeeld van RRT-Connect afgebeeld. C_{random} is het willekeurige, nieuwe monster dat toegevoegd is. Het algoritme probeert dat monster te verbinden met de boomstructuur die op dat moment actief is, wat in dit geval de boomstructuur is vertrekkende van de begintoestand. Aangezien deze verbinding te lang is, induceert het algoritme een nieuwe sample (C_{extend}) op een ingestelde maximale afstand. Het algoritme voegt deze C_{extend} toe aan de structuur als *vertex* en voegt de verbinding van C_{init} tot C_{extend} als *edge* toe aan de structuur. Vervolgens probeert het algoritme een verbinding te maken tussen de twee boomstructuren door C_{extend} te verbinden met een *vertex* van de boomstructuur vertrekkende van de eindtoestand. De *vertex* waarmee C_{extend} probeert te verbinden is willekeurig gekozen, maar hoe dichter de *vertex* bij C_{extend} ligt, hoe meer kans dat het algoritme deze *vertex* neemt. [19] [20]



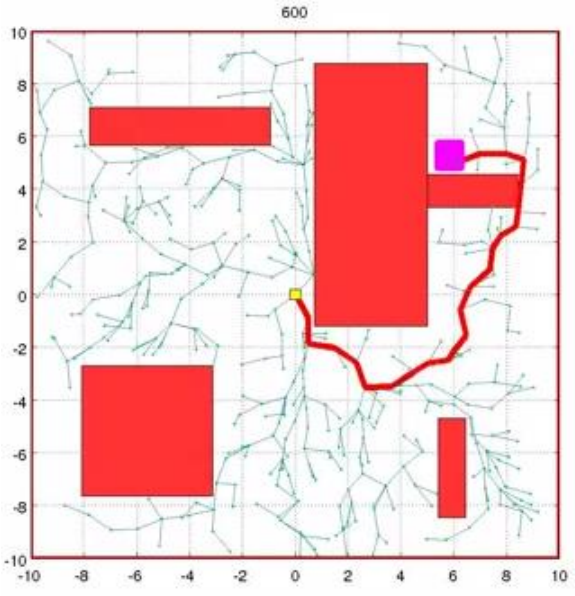
Figuur 33: Voorbeeld RRT-Connect: het verbinden van de twee boomstructuren [20]

3.2.3 RRT*

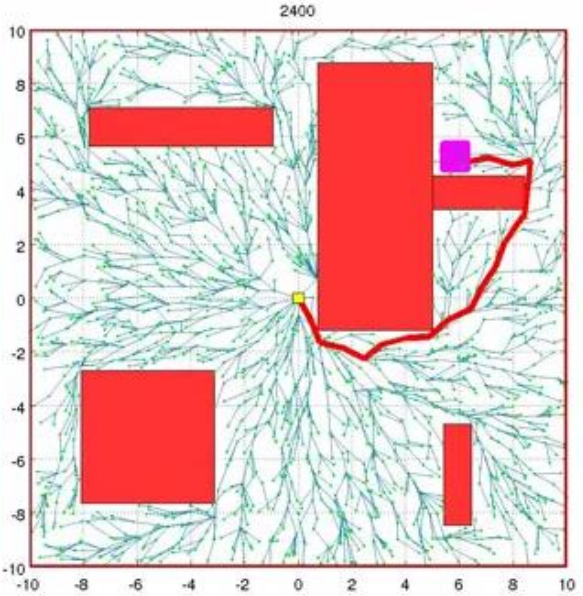
RRT* of *Rapidly exploring Random Tree Star* of *Optimal Rapidly exploring Random Tree* is een aanpassing van RRT en geeft als resultaat een optimaal pad terug. Net als bij PRM* staat de * (ster) voor optimaal. Het algoritme bekomt een optimaal pad omdat het algoritme niet zo snel mogelijk een pad als resultaat teruggeeft. Het algoritme probeert daarentegen na het vinden van een pad, dat pad te optimaliseren. RRT* volgt dezelfde stappen als RRT tot een *vertex* en *edge* is toegevoegd aan de structuur. Op dat moment genereert het algoritme niet het volgende monster, maar gaat de pas toegevoegde *vertex* nog andere verbindingen proberen te maken. De nieuwe *vertex* zal net als bij PRM verbinding proberen te maken met zijn k (parameter) dichtstbijzijnde buren of met alle *vertices* die binnen een bepaalde radius (parameter) liggen. Het algoritme houdt uiteindelijk maar één van die verbindingen over, namelijk diegene die het meest optimale is op dat moment. Deze verbinding behoort vanaf dan als *edge* bij de boomstructuur terwijl de andere verbindingen verworpen worden.

Wanneer is een verbinding de meest optimale verbinding van dat moment? Indien het pad van de begintoestand tot de nieuwe *vertex* inclusief de verbinding een lagere kost heeft dan alle andere paden van begintoestand tot de andere nieuwe *vertices*, dan is die verbinding het meest optimaal van dat moment . De kost van een pad is simpelweg de kosten van iedere verbinding opgeteld. De kost van een verbinding is proportioneel met de lengte van die verbinding. Soms geeft het algoritme voor iedere verbinding dezelfde kost waardoor de totale kost overeenkomt met het aantal verbindingen. Die methode is een eenvoudigere oplossing die iets minder goede resultaten geeft, maar die ook minder rekentijd nodig heeft. Door dit toe te passen werkt het algoritme enkel verder op paden die op dat moment het meest optimaal zijn.

Een tweede verschil van RRT* ten opzichte van RRT is dat als het algoritme een geldig pad heeft gevonden, het algoritme niet stopt met zoeken. Het algoritme zal daarentegen verder blijven bemonsteren tot de maximale tijd verstreken is. Op dat moment geeft het algoritme het meest optimale pad terug als resultaat. Als er een nieuw monster wordt toegevoegd dat voor een optimaler pad zorgt dan een bestaande *vertex* van in de buurt, dan verdwijnt de verbinding met die *vertex* en verbindt het algoritme het nieuwe monster met de rest van de boomstructuur. Dit nieuwe pad is iets optimaler dan het vorige pad. Deze stap blijft zich herhalen tot de maximale tijd verstreken is. Op die manier kan het gevonden pad steeds in kleine stappen verbeteren. Ook de rest van de omgeving krijgt steeds betere paden tot de buitenste *vertex*, ook al is deze buitenste *vertex* niet de eindtoestand. Een voorbeeld van RRT* is hieronder afgebeeld met in figuur 34 een geldig pad en in figuur 35 een optimaler pad. [20] [21]

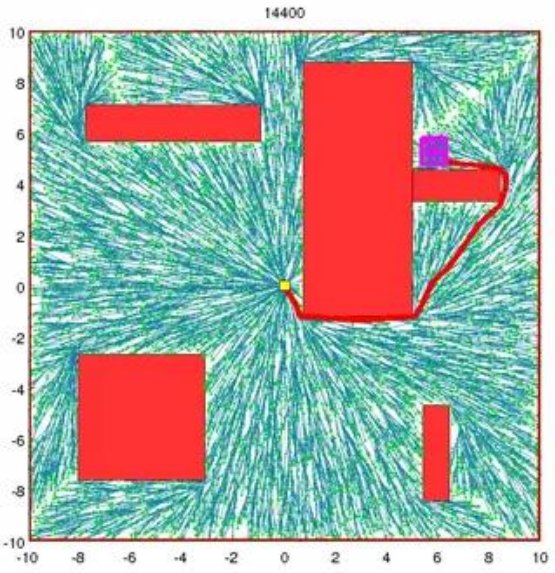


Figuur 34: Een gevonden pad na 0,6 seconden [22]



Figuur 35: Een verbeterd pad gevonden na 2,4 seconden [22]

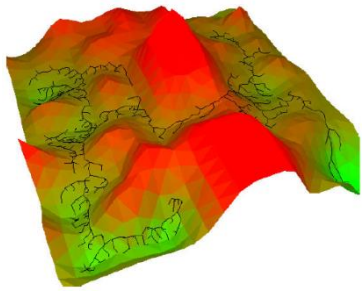
Zoals op de figuren te zien is, heeft het algoritme na 0,6 seconden al een pad gevonden die van de begintoestand (geel vierkantje) tot de eindtoestand (roos vierkantje) loopt. Het algoritme geeft dit pad echter niet als resultaat terug, maar zoekt nog door om het pad optimaler te maken. Na 2,4 seconden heeft het algoritme al een beduidend optimaler pad gevonden. Een belangrijke parameter bij dit algoritme is de maximale tijd die het algoritme mag spenderen aan het zoeken van een optimaal pad. Het algoritme zal namelijk al zijn tijd gebruiken terwijl het pad eventueel al na enkele seconden een optimaal pad heeft gevonden. De tijd die het algoritme daarna nog spendeert aan het verbeteren van het pad is overbodig. Een voorbeeld van deze *overprocessing* is te zien in figuur 36. In die afbeelding is na 14,4 seconden het meest optimale pad gevonden. Het pad is nog een verbetering ten opzichte van het pad van figuur 35, maar het algoritme neemt 7 keer zo veel tijd in beslag terwijl de verbetering relatief gezien niet hoog is.



Figuur 36: Overprocessing van RRT [22]*

3.2.4 T-RRT

T-RRT of *Transition-based Rapidly exploring Random Tree* is een aanpassing op RRT. T-RRT bevat enkele verbeteringen ten opzichte van RRT door het snel verkennen (RRT) te combineren met een kostenfunctie. Ook maakt het algoritme gebruik van een zelfaanpassende parameter die het verkennen nog verbetert. T-RRT bekomt ook optimale paden door het gebruik van de kostenfunctie. Het verschil met RRT* is dat het mogelijk is om T-RRT in hogere dimensies te gebruiken doordat de kost via een kostenfunctie aangeduid is. Een voorbeeld in 3D van T-RRT is in figuur 37 weergegeven.



Figuur 37: Voorbeeld T-RRT in 3D [21]

De kost van een pad kan verschillende betekenissen hebben. Zo kan de kost terugslaan op het aantal verbindingen, de totale lengte van het pad, enz. De definitie van de kost is instelbaar via een parameter. In het voorbeeld van figuur 37 is de kost evenredig met de hoogte van het landschap. Zo zullen bergen (rode gebieden) een hoge kost hebben en valleien (groene gebieden) een lage kost. Op die manier probeert het algoritmeervoorte zorgen dat bijvoorbeeld een bestuurder van een auto niet door de bergen moet rijden, maar dat hij via de valleien van de begintoestand tot de eindtoestand kan geraken.

Zoals eerder vermeld gebruikt het algoritme een kostenfunctie om een pad met lage kost te vinden in hogere dimensies. Deze kostenfunctie combineert verschillende definities van een kost. Een mogelijk voorbeeld (figuur 37) is dat iedere *vertex* een bepaalde kost heeft die zowel positief als negatief kan zijn. *Vertices* in het rode gebied hebben een hoge kost en hebben daarom een positieve kost. Dat wil zeggen dat het algoritme een kracht induceert tegen de richting van de robot in zodat de robot energie verliest. *Vertices* in het groene gebied hebben daarentegen een lage kost en hebben daarom een negatieve kost. Een negatieve kost zorgt voor een kracht in de richting van de robot waardoor deze energie bijkrijgt. Hoe positiever of negatiever de *vertices* zijn, hoe groter de kracht is die het algoritme genereert. Uiteindelijk is het de bedoeling dat de robot op het einde van het pad zo veel mogelijk energie overhoudt. Op die manier kiest het algoritme het pad met de laagste kost.

Om een onderscheid te maken tussen de paden die op de vooropgestelde manier evenveel energie overhouden, voegt het algoritme in dit voorbeeld ook nog een positieve kost toe (tegengestelde kracht en dus energieverlies) proportioneel met de lengte van de verbinding. Korte paden krijgen op die manier voorrang op langere paden bij paden die initieel dezelfde energie overhielden.

Het opbouwen van de boomstructuur is volledig analoog als RRT ondanks dat het algoritme een kostenfunctie bijhoudt. Het algoritme vertrekt van de begintoestand en voegt een willekeurig monster toe. De botsingsdetectie controleert dat monster op zijn geldigheid. Als dat monster geldig is dan verbindt het algoritme deze met de boomstructuur op dezelfde manier als bij RRT. Dat monster en de overeenkomstige verbinding behoren pas tot de structuur indien ze bijdragen aan een pad met een minimale kost.

Overgangstest

Het bijdragen aan een pad met een minimale kost gebeurt pas indien het monster en zijn verbinding voldoen aan de zogenaamde overgangstest. De overgangstest verwerpt monsters en verbindingen met een te hoge kost. Als het nieuwe monster een kost heeft die boven een instelbare maximale waarde ligt dan verwerpt de overgangstest het monster en de verbinding meteen. Als het nieuwe monster daarentegen een lagere kost heeft dan zijn *parent* (de *vertex* waarmee het monster verbonden is), dan voegt de overgangstest de verbinding en het monster toe aan de structuur als *edge* en *vertex*.

Als de kost van het monster daarentegen hoger is dan zijn *parent* dan gaat het pad in dit voorbeeld omhoog (bergop). In dat geval beschrijft de overgangstest de kans op toevoeging, van de verbinding en het monster, als een formule die hieronder vermeld staat.

$$p_{ij} = \begin{cases} exp(-\frac{\Delta c^*_{ij}}{K*T}) & \text{if } \Delta c^*_{ij} > 0 \\ 1 & \text{otherwise.} \end{cases}$$

met: - $\Delta c^*_{ij} = \frac{c_j - c_i}{d_{ij}}$

- c_j = kost nieuwe sample
- c_i = kost ouder
- d_{ij} = afstand tussen de 2
- $K = \frac{Kost(begintoestand) - Kost(eindtoestand)}{2}$

De kans op een overgang is dus 100% als de kost van het nieuwe monster lager is dan de kost van de *parent* en is gelijk aan bovenstaande uitdrukking als de kost van het nieuwe monster hoger ligt dan de kost van de *parent*. De factor K dient om de kostvariatie van de hellingsgraad (teller) te normaliseren. T is een instelbare factor die de moeilijkheidsgraad van de overgang weergeeft. Bij lage waarden van T laat de test lage hellingsgraden toe en bij hoge waarden van T laat de test hogere hellingsgraden ook toe. Doorheen het proces laat het algoritme daarom de factor T gradueel stijgen zodat steeds hogere hellingsgraden mogelijk zijn. Dat wil zeggen dat het algoritme eerst probeert om een verbinding te maken met een lage hellingsgraad en indien dit niet mogelijk is, verhoogt het algoritme T steeds totdat een verbinding gevonden is. Algemeen gezien is de kans dat de test een verbinding accepteert steeds kleiner naar mate de hellingsgraad hoger is.

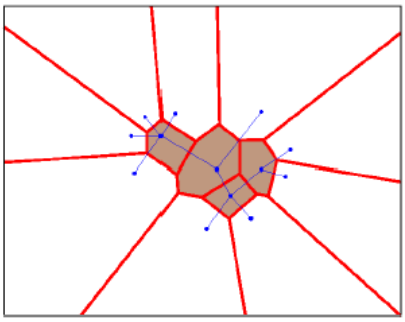
De parameter T start dus met een lage waarde. Iedere keer als een monster niet geaccepteerd wordt door de overgangstest en dus verworpen wordt, verhoogt de overgangstest nFail met één. Wanneer nFail zijn maximale waarde bereikt, dat een instelbare parameter is, vermenigvuldigt de test T met twee zodat hogere hellingsgraden mogelijk zijn. Indien de test een geldig monster vindt, deelt de test de waarde T terug door twee. Op die manier zal het volgende monster opnieuw maar een lage hellingsgraad mogen hebben. Indien het monster geldig is en het algoritme dat monster en zijn verbinding toevoegt als *vertex* en *edge* aan de structuur, begint het algoritme weer met het toevoegen van een nieuw monster. De stappen die hiervoor beschreven zijn, blijven zich herhalen totdat een eindvoorwaarde bereikt is. Een eindvoorwaarde is bereikt als er een geldig pad gevonden is. Dat gevonden pad is ook meteen een optimaal pad.

Minimale uitbreidingstest

Een tweede test die het algoritme uitvoert naast de overgangstest is de minimale uitbreidingstest. Deze test zorgt ervoor dat het algoritme niet blijft steken in een lokaal minimum (zie figuur 8 in ‘2.3.1 Combinatorische- en bemonsteringsalgoritmen’). Door de parameter T zal de verkenning namelijk vrij traag gaan en zal het bemonsteren steeds plaatsvinden in de buurt van de vorige *vertices*. De minimale uitbreidingstest probeert dit te beperken door het algoritme af en toe te verplichten om in grote niet verkende gebieden te bemonsteren.

Een omgeving bestaat uit grenspunten en verfijningspunten. Grenspunten zijn punten die niet volledig omsloten zijn door een *Voronoi*-gebied (‘2.3.2 Het toevoegen van monsters’). Een voorbeeld van grenspunten is in figuur 38 te zien. De witte gebieden zijn *Voronoi*-gebieden waar grenspunten in liggen en de rode gebieden zijn *Voronoi*-gebieden waar verfijningspunten in liggen. De waarde T zalervoorzorgen dat er vooral monsters bijkomen in de verfijningsgebieden aangezien de parameterervoorzorgt dat er enkel monsters met een lage kost bijkomen. Monsters

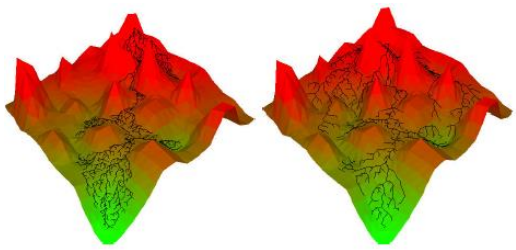
die vlakbij een ander monster liggen hebben een kleine afstand en ook vaak een kleine hellingsgraad waardoor de overgangstest deze monsters vaak accepteert. Zo een monster zal dus snel als *vertex* deel uitmaken van de structuur. Hierdoor halveert de parameter T weer waardoor het algoritme blijft steken in de verfijningsgebieden terwijl het de bedoeling is om ook te verkennen in de grensgebieden. De minimale uitbreidingstest zorgt dat het algoritme ook andere delen van de omgeving verkent.



Figuur 38: Grenspunten en verfijningspunten in Voronoi-gebieden [21]

- Witte Voronoi-gebieden: grenspunten
- Rode Voronoi-gebieden: verfijningspunten

Deze test beheerst de verhouding van ontdekken op verfijnen. De test gaat er van uit dat grenspunten een groot *Voronoi*-gebied hebben en dat verfijningspunten een klein *Voronoi*-gebied hebben. Op die manier kan het algoritme schatten of een nieuw monster een verfijningspunt of een grenspunt is. Hoe groter de afstand is met de naburige *vertex*, hoe meer kans dat het nieuwe monster een grenspunt is. Algemeen beschouwt het algoritme een nieuw monster als verfijningspunt indien de afstand tot de dichstbijzijnde *vertex* kleiner is dan een instelbare parameter δ. Het nieuwe monster behoort tot de categorie van grenspunten of ontdekkingspunten indien de afstand groter is dan δ. De minimale uitbreidingstest houdt beide aantallen bij en indien de verhouding ontdekkingspunten/verfijningspunten te laag is, verwijdert deze test monsters die door de overgangstest geaccepteerd werden, maar tot de verfijningspunten behoorden. Daardoor stijgt T zodat het algoritme nieuwe monsters toelaat die een hogere hellingsgraad hebben en/of een grotere afstand hebben tot de structuur. In onderstaande afbeelding (figuur 39) is duidelijk het verschil te zien tussen het algoritme zonder de minimale uitbreidingstest (linkse afbeelding) en het algoritme met gebruik van de test (rechtse afbeelding). [21]



Figuur 39: T-RRT zonder (links) en met (rechts) gebruik van de minimale uitbreidingstest [21]

3.2.5 EST

Het algoritme EST (*Expansive Space Trees*) is ook een algoritme dat gebruikmaakt van een boomstructuur. Het algoritme gaat eerst de gebieden zoeken waar weinig verkend is via een rooster. Daarna probeert het de boomstructuur vooral in deze gebieden uit te breiden. Hierbij probeert het algoritme vooral rechte paden te bekomen met een lage kost. EST maakt net als RRT-Connect gebruik van twee boomstructuren die vertrekken van de begin- en eindtoestand. De manier waarop het algoritme de boomstructuur opbouwt en verbinding probeert te maken is echter anders.

In de eerste stap, het uitbreiden van de boomstructuren, wisselt het algoritme niet van boomstructuur zoals bij RRT-Connect. EST zal de twee boomstructuren daarentegen simultaan opbouwen. Bij EST krijg iedere *vertex* een gewicht (w) toegekend met een index. Deze index verklaart tot welke structuur deze *vertex* behoort (w_x en w_y). Het gewicht van een *vertex* is gelijk aan het aantal *vertices* van die boomstructuur die binnen een straal d (parameter) van de *vertex* ligt. Stel bijvoorbeeld dat boomstructuur y een *vertex* bevat waar 10 *vertices* langs liggen binnen een straal d die allemaal tot diezelfde structuur y behoren. Die *vertex* krijgt dan een gewicht van 10 ($w_y = 10$).

Aangezien het algoritme beide boomstructuren simultaan opbouwt, is de werking voor beide structuren hetzelfde. De onderstaande uitleg beschrijft de werking van één boomstructuur, maar deze werking voert het algoritme twee keer uit aangezien er twee boomstructuren zijn. EST selecteert een willekeurige *vertex* van boomstructuur x. De kans waarop het algoritme een *vertex* kiest is omgekeerd evenredig met het gewicht van die *vertex* ($P = 1/w_x$). *Vertices* met een laag gewicht hebben dus meer kans om gekozen te worden. Daardoor is er meer kans dat het algoritme verder gaat ontdekken in gebieden waar nog niet veel *vertices* liggen en waar nog niet veel verkend is.

Vervolgens genereert het algoritme rond deze gekozen *vertex* K nieuwe monsters met K een instelbare parameter. Al deze gegeneerde monsters moeten binnen een bepaalde straal d (instelbare parameter) liggen van de *vertex*. Als het algoritme dit voltooid heeft, kent het algoritme aan ieder gegenereerd monster een gewicht toe op dezelfde manier als hierboven beschreven. Het algoritme kiest dan een willekeurig monster uit de gegenereerde monsters. De kans dat het algoritme een monster kiest, is omgekeerd evenredig met het gewicht van dat monster ($P = 1/w(\text{monster})$). Indien het algoritme het gekozen monster geldig kan verbinden met de *vertex* via een rechte lijn, voegt het algoritme het monster en de verbinding toe als *vertex* en *edge* in de figuur.

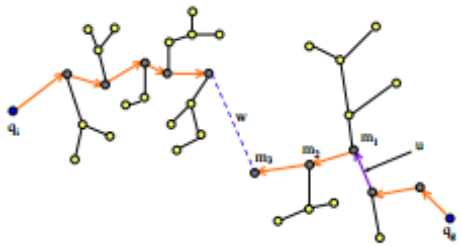
In de tweede stap probeert het algoritme de verschillende boomstructuren met elkaar te verbinden zodat er een pad vormt. Het algoritme berekent de afstand van iedere *vertex* van structuur x tot iedere *vertex* van structuur y. Voor alle afstanden lager dan 'l', wat een instelbare parameter is, probeer het algoritme een verbinding te maken. Indien er een geldige verbinding tot stand komt zijn beide structuren met elkaar verbonden en is er een pad gevonden. Het algoritme blijft de twee besproken stappen (uitbreiden en verbinden) uitvoeren tot een pad gevonden is of tot het maximaal aantal samples is bereikt. [23]

3.2.6 SBL

SBL staat voor *Single-query Bi-directional Lazy collision checking planner* en maakt net als RRT-Connect en EST gebruik van twee boomstructuren die vertrekken van de begin- en eindtoestand. Het algoritme probeert zo weinig mogelijk te verkennen en toch een pad te vinden. De twee stappen (uitbreiden en verbinden) zijn volledig analoog als bij EST waar het algoritme verder bouwt op een willekeurige *vertex* (P = 1/w(x of y)) van een boomstructuur en daarna verbinding probeert te maken met die *vertex*.

Een groot verschil met de andere technieken is het '*lazy collision checking*' karakter of de luie botsingsdetectie. Het algoritme voert bij dit algoritme geen botsingsdetectie uit tot dit echt nodig is, wat betekent dat het algoritme enkel een botsingsdetectie uitvoert als het algoritme een pad heeft gevonden. Als dat pad ongeldige stukken bevat, verwijdert het algoritme deze stukken en begint het algoritme opnieuw te bemonsteren tot er een nieuw pad gevonden is. De botsingsdetectie controleert opnieuw de verschillende delen van het gevonden pad en verwijdert ongeldige stukken tot er een pad gevonden is dat volledig geldig is. Op dat moment stopt het algoritme en geeft het als resultaat het geldige pad terug.

De reden voor deze aanpak is de tijdsbesparing. Bij andere algoritmen zoals PRM neemt de botsingsdetectie ongeveer 90% van de tijd in beslag. Ook duurt de botsingsdetectie op verbindingen het langst indien de verbindingen geldig zijn ('2.2.2 Werking van botsingsdetectie'). Het principe van de luie botsingsdetectie heeft als voordeel dat het algoritme zeer snel een pad vindt. De botsingsdetectie moet slechts de verbindingen controleren die deel uitmaken van het resultante pad en deze eventueel verwijderen indien ze ongeldig zijn. Dat zorgtervoordat de berekeningstijd van SBL lager ligt. Daarnaast controleren andere algoritmen ook verbindingen op hun geldigheid die niet eens in het uiteindelijke pad voorkomen, wat overeenkomt met een verspilling van tijd. Door deze verminderde rekentijd kan SBL meer tijd spenderen aan het oplossen van complexere problemen. Op onderstaande afbeelding (figuur 40) is zowel de uitbreidings- als de verbindingsstap te zien die analoog zijn als bij EST.



Figuur 40: Voorbeeld SBL [24]

In het voorbeeld is de rechtse boomstructuur op dat moment actief. Het algoritme heeft als nieuwe monster m_3 gegenereerd. Het algoritme probeert vervolgens dat monster te verbinden met de *vertex* m_2 die gekozen was. Vervolgens probeert het algoritme in de tweede fase de twee boomstructuren met elkaar te verbinden. De willekeurige keuze van de *vertex* van de boomstructuur die niet actief was, ligt op een afstand w van de actieve boomstructuur wat in dit geval een te grote afstand is. Het algoritme begint daarom terug bij de eerste fase. Indien de afstand w wel kleiner is dan de maximale afstand geeft, dan controleert het algoritme dit pad op zijn geldigheid. Is het pad volledig geldig dan geeft het algoritme dat pad als resultaat terug. Bevat het pad ongeldige stukken dan verwijdert het algoritme deze stukken terug en begint het algoritme terug bij de eerste fase, namelijk de uitbreidingsfase. [24]

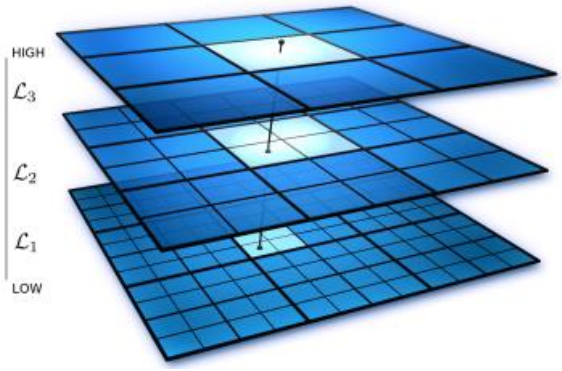
3.2.7 KPIECE

KPIECE staat voor *Kinematic Planning by Interior-Exterior Cell Exploration* en maakt ook gebruik van een boomstructuur. Het algoritme is vooral bruikbaar bij complexe, dynamische problemen. Het grote verschil tussen KPIECE en andere algoritmen is dat KPIECE geen gebruikmaakt van een afstandsmeting en dat KPIECE ook geen toestanden bemonstert. Het algoritme vertrekt daarentegen van een initiële toestand en voert vervolgens een actie uit in de vorm van een beweging gedurende een bepaalde tijd. Deze beweging vervangt het genereren en verbinden van monsters. Daardoor is KPIECE vooral bruikbaar voor complexe problemen beschreven door een fysisch model in plaats van problemen beschreven door bewegingsvergelijkingen.

Het algoritme is ook bruikbaar voor problemen waar de eindtoestand niet bekend is tot deze bereikt is. Door gebruik te maken van een fysisch model, zoals een robot, in plaats van bewegingsvergelijkingen is de nauwkeurigheid hoger aangezien een fysisch model meer rekening houdt met de dynamische eigenschappen van de robot zoals wrijvingskracht, zwaartekracht,… . Het fysisch model heeft echter wel een nadeel, namelijk de tijdsduur om te simuleren ligt hoger aangezien integratie (gebruik van bewegingsvergelijkingen) sneller gaat. De voordelen zoals hogere nauwkeurigheid en het gebruik in complexe omgevingen heffen dit nadeel op. Bij hoge complexiteit is het zelfs een noodzaak om over te gaan naar een fysisch model.

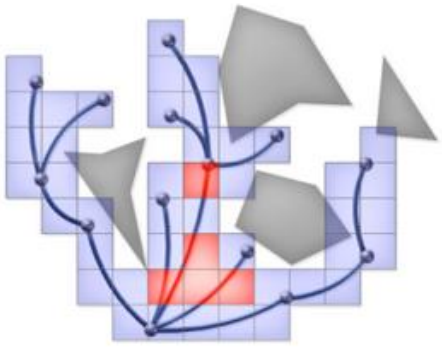
Discretisatie en projectie van toestandsruimte

Het algoritme discretiseert de toestandsruimte zodat het algoritme gemakkelijker gebieden kan terugvinden waar nog niet veel verkend is zodat het algoritme zich kan focussen op deze gebieden. Het is namelijk de bedoeling dat in ieder gebied zo snel mogelijk iets verkend is. Het algoritme doet dat door gebruik te maken van een rooster. Dat rooster ontstaat door de toestandsruimte te projecteren indien de toestandsruimte uit hogere dimensies (twee of meer) bestaat. Het discretiseren van het rooster gebeurt in een aantal niveaus k waar de grootte van een cel (vak in een rooster) verschillend is voor ieder niveau zoals in figuur 41 te zien is dat uit drie niveaus bestaat (k=3). Het aantal niveaus en de celgrootte zijn parameters van dit algoritme.



Figuur 41: Verschillende niveaus van gediscretiseerde toestandsruimte [25]

Voor iedere actie die de robot uitvoert, registreert het algoritme, voor ieder niveau, door welke cel de actie van de robot gegaan is. Dat is ook te zien op bovenstaande figuur 41, waar een actie door drie cellen gaat op verschillende niveaus. Het hoogste niveau heeft de hoogste celgrootte en naarmate het niveau lager ligt, is de celgrootte ook lager. Indien een actie door meerdere cellen van het zelfde niveau gaat, splitst het algoritme deze actie op in deelacties tot iedere deelactie maar door maximaal één cel per niveau gaat. De reden waarom KPIECE dat doet is zodat het algoritme iedere actie maar één keer verwerkt binnen hetzelfde niveau. Het algoritme discretiseert niet alle delen van het rooster, maar enkel de delen van het rooster waar effectief een actie doorloopt. Daardoor genereert het algoritme enkel de cellen die echt nodig zijn. Door dat toe te passen projecteert het algoritme enkel de cellen die een projectie weergeven van de boomstructuur, wat het gebruik van geheugen verlaagd. Onderstaande afbeelding (figuur 42) illustreert dat enkel de cellen die het algoritme effectief gebruikt deel zijn van het rooster. De grijze geometrische vormen zijn obstakels, de punten zijn toestanden en de lijnen zijn acties die uitgevoerd zijn om van toestand te veranderen.



Figuur 42: Een rooster met enkel de nuttige cellen [25]

KPIECE deelt het rooster op in interieure en exterieure cellen. Exterieure cellen zijn cellen die minder dan 2n buren hebben. Een cel x is een buur van cel y indien cel x geprojecteerd is, wat betekent dat er een actie door deze cel loopt, en indien de cel x niet diagonaal ligt ten opzichte van de cel y. De n geeft het aantal dimensies van de toestandsruimte weer. Interieure cellen zijn daarentegen cellen die 2n niet-diagonale buren hebben die deel uitmaken van de projectie van de toestandsruimte. In het voorbeeld van figuur 42 zijn er twee dimensies. Dat wil zeggen dat er een maximaal aantal van 4 niet-diagonale buren mogelijk zijn (interieure cel). De interieure cellen zijn in de figuur in het rood gemarkeerd. Hoe langer het algoritme aan het verkennen is, hoe meer exterieure cellen veranderen in interieure cellen aangezien er door steeds meer cellen een actie loopt. Voor heel hoge dimensies maakt het algoritme een uitzondering aangezien de kans op interieure cellen dan heel laag is. Het algoritme bevat een aanpasbare parameter die de waarde kan verlagen tot lager dan 2n om tot een interieure cel gerekend te worden.

Uitvoering van algoritme

Het doel van de discretisatie is om na te gaan welke stukken van het rooster voldoende verkend zijn en welke stukken weinig tot niet verkend zijn door de boomstructuur. De hoeveelheid dat een cel ontdekt is, is de som van de tijdsduren van de robot zijn acties in die cel, wat de bedekking van die cel noemt. Als het algoritme een cel vindt waar het nog niet veel heeft verkend, gaat KPIECE de boomstructuur richting die cel sturen om daar te verkennen. Naarmate de boomstructuur groter is, zal het zoeken naar een niet verkende cel steeds moeilijker zijn.

Het algoritme vertrekt van een begintoestand en voert als eerste actie een null-operatie (‘Begrippenlijst’) uit gedurende 0 seconden. Deze operatie zorgt ervoor dat ieder niveau van de discretisatie één exterieure cel krijgt doordat deze operatie gewoon een punt is. Al deze cellen op de verschillende niveaus vormen samen een ketting van cellen voor die actie. In de volgende stappen of iteraties voegt het algoritme iedere keer zo een nieuwe ketting van cellen toe. Het maken van een ketting van cellen gebeurt als volgt:

1) het algoritme beslist eerst of de structuur verder zal gaan op een exterieure of interieure cel. De kans dat het een exterieure cel is, is hoger dan een interieure cel (±75% kans op een exterieure cel);

2) vervolgens kiest KPIECE een reeds geprojecteerde cel (exterieur of interieur afhankelijk van het resultaat van stap 1 deterministisch op basis van hun belang. Belangrijke cellen krijgen de voorkeur op minder belangrijke cellen. Een formule beschrijft de belangrijkheid van een cel als volgt:

$$Importance(p) = \frac{\log(\mathcal{I}) \cdot \texttt{score}}{\mathcal{S} \cdot \mathcal{N} \cdot \mathcal{C}}$$

Met: - $\mathcal{I}$ = bij de hoeveelste iteratie deze cel gemaakt is
- score = 1
- $\mathcal{S}$ = aantal keren dat deze cel al gekozen is
- $\mathcal{N}$ = aantal reeds geprojecteerde cellen als buur van deze cel
- $\mathcal{C}$ = berekende bedekking van de cel

3) na de keuze van een cel kan het algoritme meer in detail kijken naar de belangrijkheid van cellen door naar een lager niveau te gaan indien deze cel niet op het onderste niveau ligt. De cellen op het vorige niveau worden in het nieuwe niveau opgedeeld in kleinere cellen aangezien de celgrootte lager is op een lager niveau. Op dezelfde manier als beschreven in stappen 1 en 2, kiest het algoritme een cel op een lager niveau. Het algoritme blijft dit herhalen tot het algoritme een cel vindt op het laagste niveau;

4) op dat moment heeft het algoritme een ketting van cellen gevonden aangezien het algoritme door de iteraties in stap 3 op ieder niveau een cel heeft gekozen. Het algoritme pakt op het laagste niveau een reeds toegepaste actie uit de structuur. De keuze van de actie is volgens een half-normale verdeling. De reden dat het algoritme gebruikmaakt van een half-normale verdeling is om het feit dat de volgorde waarop het algoritme de acties uitvoert belangrijk is aangezien het algoritme deze volgorde bewaart. KPIECE verkiest namelijk om door te gaan op acties die recent zijn uitgevoerd in plaats van op acties door te gaan die het algoritme helemaal op het begin heeft uitgevoerd;

5) het algoritme kiest willekeurig een toestand dat gelegen is op de actie gekozen in stap 4. Herinner dat een actie uit een begintoestand, een beweging en een tijdsduur bestaat. Het algoritme kiest dus een willekeurig punt op deze lijn;

6) de boomstructuur, wat bij KPIECE vaak bewegingsstructuur wordt genoemd, bouwt verder op de gekozen toestand van stap 5 door een actie op de toestand uit te voeren. Het algoritme kiest de actie willekeurig wat wil zeggen dat het soort beweging en de tijdsduur willekeurig zijn;

7) indien het algoritme de actie van stap 6 met succes uitvoert, breidt de bewegingsstructuur uit. Indien de nieuwe actie door cellen gaat die nog niet geprojecteerd zijn, voegt het algoritme deze cellen toe via een projectie. Indien de actie door de cel van de eindtoestand loopt, is er een oplossing gevonden waarop het algoritme als resultaat het pad teruggeeft;

Door het gebruik van een fysisch model en door gebruik te maken van een projectie van de toestandsruimte zakt de rekentijd van het algoritme sterk ten opzichte van andere algoritmen. Ook de hoeveelheid geheugen die nodig is bij KPIECE ligt lager dan andere algoritmen. Daar tegenover staat wel dat het algoritme gebruik moet maken van een fysisch model en dat het de toestandsruimte moet projecteren, wat complex is en veel tijd in beslag neemt. Hierdoor zijn andere algoritmen vaak sneller bij eenvoudige, laag dimensionale problemen. Dit heeft als gevolg dat KPIECE in de praktijk enkel bruikbaar is voor zeer complexe problemen met hoge dimensies wat in dit eindwerk niet het geval is. [25] [26]

3.2.8 BKPIECE

BKPIECE staat voor *Bi-directional Kinematic Planning by Interior-Exterior Cell Exploration* en is een variant van KPIECE. Het volledige algoritme komt overeen met KPIECE op één element na. BKPIECE maakt namelijk gebruik van twee bewegingsstructuren die verbinding met elkaar proberen te maken. De manier waarop twee structuren naar elkaar groeien en verbinding proberen te maken is analoog als bij RRT-Connect (‘3.2.2 RRT-Connect’), EST (‘3.2.5 EST’) en SBL (‘3.2.6 SBL’). Belangrijk om te weten blijft het feit dat BKPIECE net als KPIECE niet gebruikmaakt van monsters maar van acties die bestaan uit een begintoestand, een beweging en een tijdsduur. Volgende opsomming van de verschillende stappen die BKPIECE uitvoert illustreren de combinatie van het bi-directionele karakter en het KPIECE karakter.

1) projecteer de toestandsruimte en discretiseer deze projectie tot een rooster. Er ontstaan verschillende niveaus met verschillende celgroottes. De celgrootte op eenzelfde niveau is wel constant;

2) vertrek de eerste keer bij de begin- of eindtoestand (afwisselend) en voer een null-operatie uit. (Deze stap dus enkel uitvoeren bij de allereerste iteratie voor beide structuren);

3) selecteer een cel op basis van de belangrijkheid van de cellen en doe dit voor ieder niveau zoals in stap 3 van KPIECE;

4) neem een willekeurige actie bij de willekeurig gekozen cel op het onderste niveau zoals in stap 4 van KPIECE;

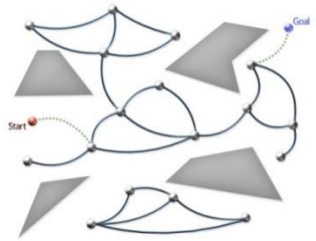
5) neem een willekeurige toestand op de actie die gekozen is in stap 4;

6) voer een willekeurige actie uit vertrekkende van het willekeurige punt gevonden in stap 5, analoog als stap 6 van KPIECE;

7) voeg de actie toe aan de bewegingsstructuur die op dat moment actief is indien de actie met succes is uitgevoerd;

8) indien de actie door een cel komt waar de andere bewegingsstructuur zich bevindt, is er een pad gevonden en stopt het algoritme met verder te zoeken. Indien dat niet het geval is wisselt BKPIECE van bewegingsstructuur en begint het algoritme opnieuw bij stap 3;

Het voordeel van BKPIECE ten opzichte van KPIECE is analoog als de voordelen bij RRT-Connect, SBL en EST. Het algoritme vindt namelijk sneller een oplossing bij complexere problemen. Ook heeft BKPIECE een voordeel ten opzichte van KPIECE indien er obstakels liggen tussen en/of in de buurt van de begin- en eindtoestand. Als er veel grote of complexe obstakels zijn, is het daarom verstandig om voor BKPIECE te kiezen in plaats van KPIECE. Bevat het probleem echter niet al te veel grote en complexe obstakels of zelfs geen obstakels, dan is de keuze niet zo belangrijk al valt dan KPIECE. Dat omdat het algoritme minder complex is aangezien de structuren geen verbinding met elkaar moeten maken terwijl de rekentijd ongeveer gelijk is. BKPIECE neemt bovendien meer geheugen in beslag het algoritme moet bijhouden bij welke structuur iedere actie hoort. Een voorbeeld van BKPIECE is in onderstaande figuur te zien (figuur 43). Net als KPIECE is BKPIECE ten opzichte van andere algoritmen enkel bruikbaar in complexe problemen met hoge dimensies. [26] [27]



Figuur 43: Voorbeeld BKPIECE [24]

3.2.9 LBKPIECE

LBKPIECE of *Lazy Bi-directional Kinematic Planning by Interior-Exterior Cell Exploration* is een variant op de bi-directionele KPIECE die hiervoor besproken is. Aangezien dit algoritme volledig analoog is aan BKPIECE met toevoeging van een element van SBL (het luie karakter van de botsingsdetectie) bespreekt dit deel niet de volledige werking van het algoritme, maar enkel de aanpassing op BKPIECE. Alle stappen van BKPIECE ('3.2.8 BKPIECE') zijn analoog op stappen 7 en 8 na. In stap 7 voegt LBKPIECE namelijk de actie altijd toe aan de structuur aangezien het algoritme de actie hier nog niet controleert op zijn geldigheid. In stap 8 doet het algoritme daarentegen iets extra ten opzichte van BKPIECE indien er een verbinding optreedt tussen de twee structuren zodat er een pad gevonden is. Op dat moment controleert de botsingsdetectie het pad op zijn geldigheid. Indien het pad geldig is, geeft het algoritme dit als resultaat terug. Indien het niet geldig is, verwijdert de botsingsdetectie de ongeldige acties van het pad en hervat het algoritme terug bij stap 3.

De voordelen van LBKPIECE ten opzichte van BKPIECE en KPIECE zijn vergelijkbaar met de voordelen van SBL. Algoritmen spenderen vaak tijd aan het uitvoeren van een botsingsdetectie op verbindingen, wat in dit geval acties zijn, die uiteindelijk niet in het resultante pad zitten. Om dat te voorkomen controleert LBKPIECE enkel acties op hun geldigheid als dat noodzakelijk is waardoor de rekentijd verlaagt. Ook hier is het toepassingsgebied van het algoritme gebaseerd op problemen met een hoge complexiteit in hoge dimensies. [26] [27]

4 Software

Het gebruik van de juiste software is een belangrijk element om een probleemstelling goed te kunnen simuleren. Om in simulatie een bak met willekeurig geplaatste werkstukken leeg te maken is er een visualisatie- en een simulatieprogramma nodig. ROS en OMPL zijn de twee belangrijkste programma's die daarvoor nodig. Dit zijn bekende programma's in de roboticawereld aangezien hier al verschillende projecten mee gerealiseerd zijn. Dergelijke software belooft enkel in bepaalde Unix besturingssystemen een goede werking. Bij deze masterproef is geopteerd om in Linux Ubuntu 12.04 *Precise* te werken omdat deze compatibel is met bijna iedere software die nodig is voor deze masterproef. Enkel voor *SolidWorks* is Microsoft nodig. Besturingssystemen zoals Microsoft Windows zijn nog in een testfase voor de ROS *software*. Vandaar dat softwareontwikkelaars geen optimale werking garanderen voor de *software* in een Windows besturingssysteem.

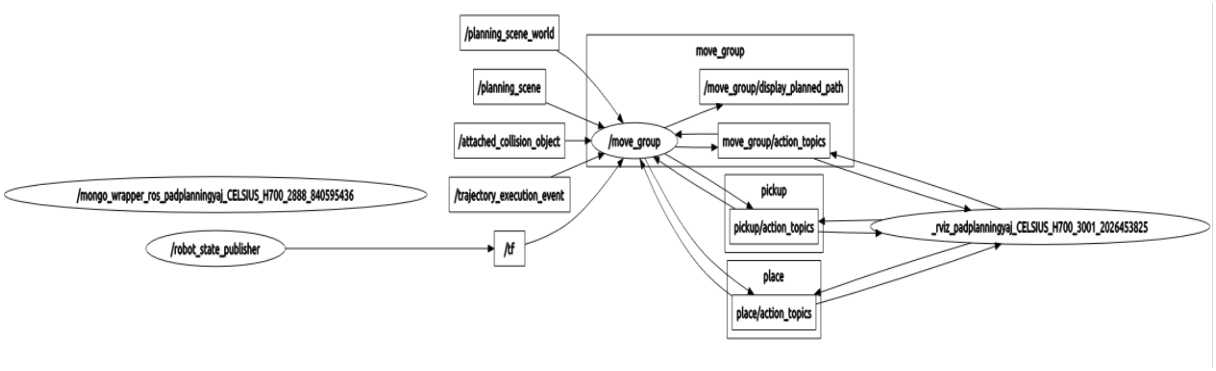
4.1 ROS

Het *Robot Operating System* is door Willow Garage opgericht in 2007. Ondertussen zijn er al vijf versies van ROS op de markt, wat wijst op een continue innovatie van deze software. Problemen die voor een mens heel simpel lijken, zijn vaak moeilijk over te brengen naar een robot. ROS is ontwikkeld om de communicatie tussen mens en robot en de implementatie van ideeën te vereenvoudigen. Dat gebeurt allemaal in eenzelfde programmeertaal waardoor het samenwerken tussen bedrijven met verschillende specialisaties mogelijk is. ROS is de basisbouwsteen van alle programma's die in deze masterproef gebruikt zijn. Het *Robot Operating System* doet dienst als programmeerplatform voor de verschillende *software*. Dankzij dat platform is het communiceren tussen *software* en bibliotheken, die geïmplementeerd zijn in ROS, mogelijk. Deze bibliotheken bevatten padplanningsalgoritmen en robotmodellen waar de programmeur gebruik van kan maken. ROS bevat ook conventies waar de programmeur zich aan moet houden. [28]

Zogenaamde *nodes* zorgenervoordat het uitvoeren van berekeningen in ROS mogelijk is. De software in ROS roept deze *nodes* op, wat regels in documenten zijn. Deze documenten kunnen zowel *launch-files* als *config-files* als *URDF-files* zijn. De *nodes* zijn regels in zo een document, die zorgen voor het starten van berekeningen. Ze zorgen ook voor het opslaan en opvragen van informatie, gegenereerd door de *software*. Het communiceren tussen actieve *software* gebeurt ook aan de hand van deze *nodes*. *Software* is actief indien de gebruiker dat programma heeft geopend. *Nodes* zorgen voor de communicatie tussen programma's door de resultaten van de berekeningen te publiceren als *messages*. Vervolgens schrijven de *nodes* deze *messages* weg naar Ros-*topics*. De *nodes* kunnen deze weggeschreven resultaten terug opvragen, in de vorm van *messages*, om hier nieuwe berekeningen op uit te voeren. Een voorbeeld van een *node* is een camera die *live* beelden levert zodat andere *nodes* deze beelden kunnen opvragen om er

bewerkingen op uit te voeren. Een voorbeeld van zo een bewerking is het omzetten van de omgeving naar *octomappen*, zodat de beelden visualiseerbaar zijn in het simulatieprogramma Rviz. [29] [30]

Het is op ieder moment mogelijk om via de *terminal* commando's uit te voeren zoals *rosnode*, *rosmsg* en *rqt-graph*. Het commando r*osnode* somt als resultaat de verschillende actieve *nodes* op in de *terminal*. Het commando *rosmsg* geeft daarentegen als resultaat de uitgewisselde *messages* weer in de terminal. *Rqt-graph* geeft als resultaat zowel de *nodes* als de *messages* visueel weer in een nieuw venster zoals in figuur 44 te zien is. In dat voorbeeld zijn de omcirkelde elementen de *nodes* en de regels die daar tussen staan, zijn de uitgewisselde *messages*. De *messages* zijn met een rechthoek omkadert.

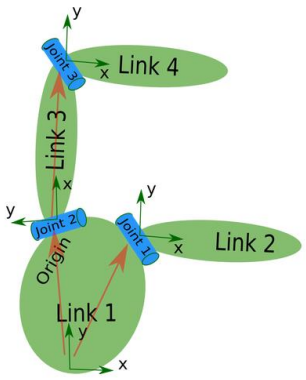


Figuur 44: Voorbeeld resultaat rqt-graph commando

4.1.1 URDF

URDF staat voor *Unified Robot Description Format* en is een XML gebaseerd bestand. Zo een XML-bestand representeert een robot. Omdat XML in verschillende bestandstypen voorkomt, maar niet ieder bestandstype voldoet aan de nodige specificaties om een robot te reprensteren, is de ontwikkeling van het URDF-bestandstype bij Willow Garage gestart. Vanwege deze conventie is de chaos van veel verschillende bestandstypes vermeden. Deze conventie zorgtervoordat de gebruiker de garantie krijgt dat het URDF-bestand compatibel is met alle ROS *plugin software*. [31]

Een URDF-bestand bestaat uit enkele delen die steeds terugkomen. Deze delen moeten verplicht aanwezig zijn zodat het URDF-bestand geldig is. Onderstaand voorbeeld laat de verschillende delen van een URDF-bestand zien aan de hand van de code van het URDF-bestand. In figuur 45 is als voorbeeld een algemene structuur te zien van een robot. Een robot bestaat uit zogenaamde *links* en *joints*. Hierbij zijn de *links* de onderdelen of verbindingen van de robot en de *joints* de scharnierende elementen tussen deze onderdelen.



Figuur 45: eenvoudige voorstelling industriële robot [31]

Het URDF-bestand van de robot uit figuur 45 bevat de volgende delen code:

```
1  <robot name="test_robot">
2    <link name="link1" />
3    <link name="link2" />
4    <link name="link3" />
5    <link name="link4" />
```

De code van hierboven vormt het eerste deel van het URDF-bestand dat verplicht aanwezig moet zijn. In dit deel staat de naam van het URDF-bestand, de naam van de robot en de naam van iedere link die de robot bevat. De naam van het URDF-bestand moet overeenkomen met de naam van de robot.

```
7    <joint name="joint1" type="continuous">
8      <parent link="link1"/>
9      <child link="link2"/>
10     <origin xyz="5 3 0" rpy="0 0 0" />
11     <axis xyz="-0.9 0.15 0" />
12   </joint>
```

Bovenstaande code vormt het tweede deel van het URDF-bestand dat verplicht aanwezig moet zijn in het bestand. Dit deel van de code bevat alle attributen van het element '*joint*'. Als eerste is in regel 7 de naam en het type van de *joint* gegeven. Hierna zijn in regel 8 en 9 de twee *links* gegeven waartussen dit scharnier ($joint_1$) ligt, wat overeenkomt met $link_1$ en $link_2$. Vervolgens is in regel 10 de oorsprong van dit scharnier gegeven in cartesische coördinaten inclusief de rotatie die uitgedrukt is in rpy. Rpy staat voor *roll, pitch* en *yaw* wat respectievelijk overeenkomt met de rotatie over de x-, y- en z-as en is uitgedrukt in radialen. Een positieve *roll, pitch* of *yaw*-waarde komt overeen met een rotatie tegenwijzerzin ten opzichte van de bijhorende as. De cartesische coördinaten zijn relatief uitgedrukt ten opzichte van de basis van $link_1$, wat overeenkomt met de oorsprong van het bijhorende assenstelsel. Ten slotte specificeert regel 11 de as waarrond de *links*, verbonden met dit scharnier, ten opzichte van elkaar draaien.

In regel 14 tot en met 27 van het voorbeeld zijn $joint_2$ en $joint_3$ op dezelfde manier gespecificeerd als $joint_1$ dat hierboven besproken is.

```
14   <joint name="joint2" type="continuous">
15     <parent link="link1"/>
16     <child link="link3"/>
17     <origin xyz="-2 5 0" rpy="0 0 1.57" />
18     <axis xyz="-0.707 0.707 0" />
19   </joint>

21   <joint name="joint3" type="continuous">
22     <parent link="link3"/>
23     <child link="link4"/>
24     <origin xyz="5 0 0" rpy="0 0 -1.57" />
25     <axis xyz="0.707 -0.707 0" />
26   </joint>
27 </robot>
```

Zoals eerder vermeld zijn dit de delen van het bestand die verplicht aanwezig moeten zijn om een geldig URDF-bestand te zijn. Deze code kan nog aangevuld worden met enkele specificaties over de robot zoals het uiterlijk van de robot (*mesh*), de fysische eigenschappen van de robot (*inertia*),… . Een bespreking van het URDF-bestand van de Epson C3 robot volgt in '5.1.2 Opbouw URDF-bestand van Epson-robot'.

4.1.2 ROS en PR2

De PR2 is een zeer gekende robot binnen het ROS-platform. Ook de PR2-robot is ontwikkeld door Willow Garage en kwam in 2010 op de markt. De reden dat de PR2 gekend is bij ROS-gebruikers, is dat de PR2 in bijna alle handleidingen over ROS wordt gebruikt. De PR2 is een tweearmige robot die uitgerust is met een vijf megapixel optische camera en een laser. Zowel de robot als de sensoren zijn geïntegreerd in ROS zodat de PR2 volledig simuleerbaar is op de computer. [33]

4.2 Rviz

Rviz is een 3D-visualisatieprogramma in ROS. De visualisatie van objecten in de omgeving van de robot gebeurt op basis van *octomappen*. Rviz bevat meerdere *plugin*s die in enkele gevallen door gebruikers zelf zijn geprogrammeerd. Het is mogelijk om zelf gegenereerde URDF-bestanden met Rviz te testen. Dat is noodzakelijk omdat URDF-bestanden regelmatig compileerbaar zijn zonder een fout weer te geven terwijl er fouten aanwezig zijn in het bestand. Hierdoor staan verschillende onderdelen van de robot fout ten opzichte van elkaar en bewegen scharnierende elementen fout. Door het URDF-bestand te testen vallen deze fouten meteen op waarop de gebruiker het bestand kan aanpassen totdat de robot werkt zoals het moet. [34]

Rviz is niet alleen handig om een URDF-bestand te testen, maar is ook handig om te gebruiken voor andere aspecten. Dat komt door de verschillende *plugin*s die beschikbaar zijn bij Rviz. Na het doorlopen van de 'MoveIt!' *software* ('4.4 MoveIt!') verschijnt er in Rviz een *plugin* van 'MoveIt!' die het mogelijk maakt om padplanningsalgoritmen toe te passen op de robot. Deze *plugin* noemt de *MotionPlanner plugin* en deze heeft toegang tot enkele padplanningsalgoritmen die aanwezig zijn in OMPL ('4.3 OMPL').

4.2.1 *Octomappen*

Octomapping is ontwikkeld om vier kenmerken samen te brengen, die nog niet bestonden in deze samenhang. Zo moet er als eerste een volledig 3D-model gemaakt worden van de omgeving. Hiervoor moeten alle gebieden in de *map* een declaratie meekrijgen zoals bezet, leeg, onbekend of *inner node*. Een tweede vereiste kenmerk is dat de *map* op ieder moment moet kunnen *updaten*. Op ieder moment moeten dus nieuwe sensormetingen en nieuwe informatie de *map* kunnen bijwerken. Een voorwaarde hierbij is dat er sensormetingen van evenveel sensoren nodig zijn als dat er benodigd waren om het gebied te maken. Een derde kenmerk is dat de *map* flexibel moet zijn. Dat wil zeggen dat er op voorhand geen grenzen van de *map* worden meegegeven. Deze grenzen worden pas gedefinieerd bij compressie als de map wordt opgeslagen of als ze wordt verwerkt. Het vierde en laatste kenmerk is dat de *map* compact moet kunnen opgeslagen worden zodat het weinig geheugen verbruikt bij opslag. Op die manier kan de *map* ook worden doorgestuurd zonder de bandbreedte te overschrijden. [35]

Zogenaamde *octrees* onderzoeken de werkingsruimte. Dat gebeurt door de werkingsruimte voor te stellen door een kubus met vaste afmetingen, ook wel een *voxel* genoemd. Als deze kubus de status 'bezet' heeft, wordt deze verder gesplitst in acht kleinere kubussen. Een kubus is bezet indien een (deel van een) object binnen de kubus ligt. Dat proces blijft zich herhalen tot de kubus de minimale *voxel*grootte bereikt. Deze minimale *voxel*grootte bepaalt ook de maximale resolutie

van de *octomap.* Een kubus kan ook vroegtijdig stoppen met opsplitsen in kleinere kubussen indien de kubus leeg is.

Dankzij de *octree* methode is de *map* multi-resolutioneel. De *map* bezit dus informatie over verschillende resoluties. De maximale resolutie is, zoals hierboven al vermeld, bepaald door de minimale *voxel*grootte. Een lagere resolutie ontstaat wanneer kubussen vanaf een bepaalde grootte niet meer splitsen, dit kan dankzij de hiërarchische structuur van de *map*. In volgende afbeelding (figuur 46) wordt eenzelfde omgeving afgebeeld, maar steeds met een lagere resolutie.

Figuur 46: Octree met verschillende resoluties [34]

Elke kubus wordt onderzocht op de waarschijnlijkheid dat het bezet is. Deze waarschijnlijkheid is bepaald door de *logOdds*-waarde die uit een formule berekend is. Het systeem vergelijkt deze waarde met een op voorhand bepaalde *treshold*-waarde om te bepalen of het gebied bezet is of niet. Indien de *logOdds*-waarde hoger is dan de bovengrens van de *treshold* dan beschouwt de *octree* de kubus als bezet waarop het de kubus gaat opsplitsen in acht kleinere kubussen. Indien de *logOdds*-waarde daarentegen lager is dan de ondergrens van de *treshold* dan beschouwt de *octree* de kubus als leeg. Alle waarden die tussen de boven- en ondergrens liggen van de *treshold* worden beschouwd als onbekend.

Hoe dichter deze *treshold* zijn boven- en ondergrens bij hun uiterste waarde liggen, namelijk 0 voor de ondergrens en 100 voor de bovengrens, hoe geloofwaardiger de *map* is. Hiertegenover neemt de *map* meer geheugen in beslag na compressie. De grenzen kunnen dus best een beetje van deze uiterste waarden afwijken zodat het ingenomen geheugen verkleint. De geloofwaardigheid kan toch hoog blijven door de *map* enkele keren te *updaten* en meerdere keren de berekening uit te voeren.

4.3 OMPL

OMPL of *Open Motion Planning Library* is een bibliotheek die padplanningsalgoritmen bevat. Het is een *open source* bibliotheek die volledig onafhankelijk van andere programma's kan werken dankzij de OMPL.app GUI. Enkele padplanningsalgoritmen ('Hoofdstuk 3: Padplanningsalgoritmen in de theorie') van deze bibliotheek zijn ook geïmplementeerd in ROS zodat hiervan gebruik gemaakt kan worden. De padplanningsalgoritmen die terug te vinden zijn in de *Open Motion Planning Library* zijn:

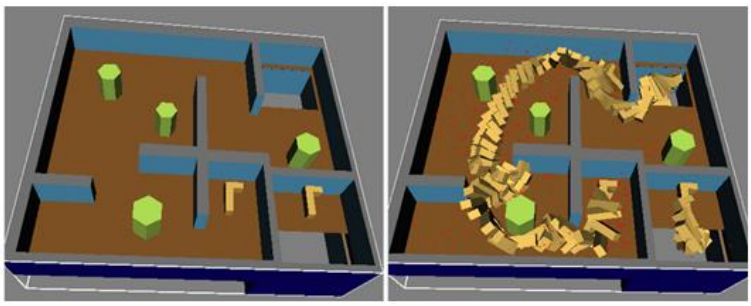
- KPIECE: *Kinematic Planning by Interior-Exterior Cell Exploration*
- BKPIECE: *Bi-directional* KPIECE
- LBKPIECE: *Lazy Bi-directional* KPIECE
- SBL: *Single-query Bi-directional Lazy collision checking planner*
- pSBL: *Parallel Single-query Bi-directional Lazy collision checking planner*
- EST: *Expansive Space of Trees*
- STRIDE: *Search Tree with Resolution Independent Density Estimation*
- PDST: *Path-Directed Subdivision Trees*
- RRT: *Rapidly exploring Random Trees*
- RRT-Connect
- pRRT: *Parallel RRT*
- LazyRRT: *Lazy RRT*
- PRM: *Probabilistic RoadMaps*
- PRM*
- LazyPRM
- SPARS
- SPARS2
- RRT*
- LBTRRT: *Lower Bound Tree RRT*
- T-RRT: *Transition-based RRT*

Bij de OMPL.app GUI is het mogelijk om parameters van padplanningsalgoritmen aan te passen. Hierbij zijn er enkele algemene parameters die voor ieder padplanningsalgoritme hetzelfde zijn zoals:

- de maximale rekentijd die het algoritme mag gebruiken;
- de botsingsdetectieresolutie;
- de begin- en eindtoestand van de robot;
- de keuze in welke dimensie er naar een pad gezocht wordt (2D of 3D);
- de afmetingen van de werkingsruimte.

Naast deze algemene parameters zijn er ook nog parameters die specifiek zijn voor de padplanningsalgoritmen. Deze specifieke parameters zijn opgesomd en besproken in '6.2 Specifieke parameters'.

Het nadeel van het onafhankelijk werkende OMPL.app GUI-programma is dat het programma geen gebruik kan maken van industriële robots. Hierdoor is het testen en evalueren van padplanningsalgoritmen op een robot onmogelijk. Het is daarentegen wel mogelijk om in de grafische *interface* van OMPL voorbeeldsituaties te simuleren. Zo een situatie bestaat uit een omgeving, met eventuele hindernissen, waarin een object van een begintoestand tot een eindtoestand moeten geraken met behulp van een padplanningsalgoritme. Een object moet tot deze eindtoestand geraken zonder in botsing te treden met een obstakel. Een mogelijk voorbeeld van een situatie in de OMPL.app Gui staat in de linkse afbeelding van figuur 47 afgebeeld. De oplossing voor deze situatie staat in de rechtse afbeelding in figuur 47 afgebeeld. [36]



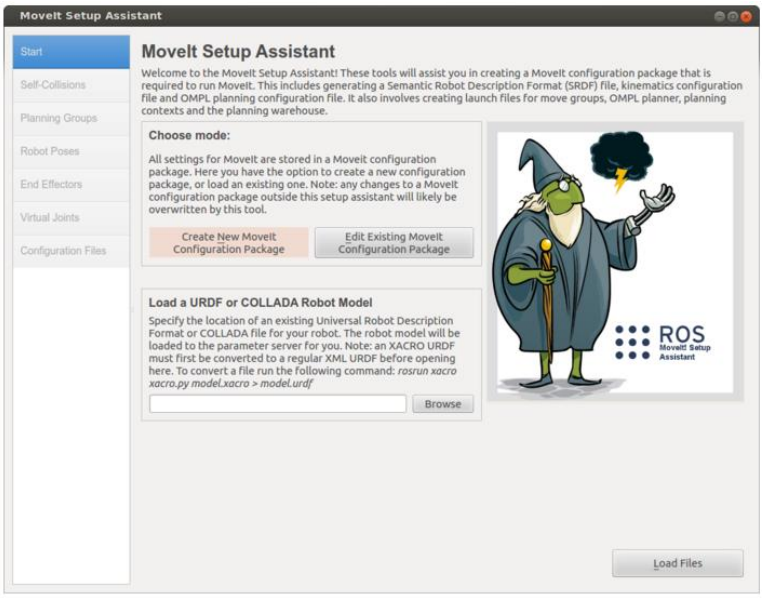
Figuur 47: Een voorbeeldsituatie van een padplanningsprobleem (links) met de oplossing ervan (rechts) [35]

Een pad berekenen voor enkel de eindeffector van de robot levert geen betrouwbaar resultaat op in de OMPL.app GUI aangezien er geen rekening gehouden wordt met:

- het werkingsgebied van de robotarm,
- de rotatiebeperkingen van de scharnieren,
- de mogelijke botsing van de rest van de robot met een obstakel.

4.4 MoveIt!

'MoveIt!' zorgtervoordat een deel van de padplanningsalgoritmen van OMPL op een industriële robot toegepast kunnen worden in Rviz. MoveIt! Maakt het namelijk mogelijk om een robot te gebruiken binnen Rviz. De *software* is, net als de meeste programma's van dit eindwerk, geïmplementeerd in ROS. Dat wil zeggen dat robots die in het ROS-platform zijn opgeslagen als URDF-bestand ('4.1.1 URDF'), herkend en geopend kunnen worden via de MoveIt! *setup assistant*. De MoveIt! *setup assistant* is een grafische *interface* dieervoorzorgt dat robots eenvoudig en manueel te configureren zijn. Figuur 48 laat een afbeelding zien van het programma. [37] [38] [39]



Figuur 48: de MoveIt! Setup Assistant

In 'MoveIt!' is het mogelijk om een robotbestand te openen en hiervan de planningsgroepen te bepalen, de botsingsmatrix op te stellen en de eindeffector(en) aan te duiden. Dit werkt niet enkel met URDF-bestanden, maar ook met bestanden van het COLLADA-type (*collaborative design activity*).

5 Creëren van simulatieomgeving

5.1 Genereren van URDF-bestand

Om met robotmodellen te kunnen werken in ROS moeten de robotmodellen gespecificeerd zijn met URDF-bestanden (‘4.1.1 URDF’). Aangezien de probleemstelling van de masterproef stelt dat een Epson C3-A601ST robot een bak met willekeurig geplaatste werkstukken moet leegmaken, is een URDF-bestand nodig die deze Epson-robot beschrijft. De site van Epson bevat echter alleen de volgende bestandsformaten van de robot:

- 3D-*SolidWorks* bestanden met extensie .SLDPRT;
- 3D-STEP bestanden met extensie .STEP;
- 2D-DXF bestanden met extensie .dxf.

De eerste twee bestandsformaten zijn CAD-tekeningen die de texturen van een model weergeven. Het laatste bestandsformaat geeft 2D-tekeningen met de afmetingen van alle onderdelen van de robot. Het is niet mogelijk om met deze bestandstypen rechtstreeks een URDF-bestand te genereren. Hiervoor zijn enkele aanpassingen nodig.

Het is ook mogelijk om een URDF-bestand van de robot handmatig te maken. Om dit te kunnen voltooien bevat de site van ROS handleidingen die stap voor stap uitleggen hoe de gebruiker een URDF-bestand kan opbouwen. De handleidingen starten eerst met een algemene uitleg over de basis van ROS. Als die basis gekend is, geven de volgende stappen van de handleidingen aan hoe de gebruiker een klein URDF-bestand handmatig kan maken. Vervolgens volgt een bespreking van het URDF-bestand van de PR2-robot zodat de gebruiker inzicht krijgt in de dingen die mogelijk zijn met een URDF-bestand. Vervolgens is de gebruiker in staat om zelf een URDF-bestand handmatig te maken. Het handmatig maken van een URDF-bestand is echter gevoelig voor typ- en compileerfouten waardoor het een zeer tijdrovende en moeilijke procedure is bij een zesassige robot. [32]

Omdat deze procedure zo omslachtig is, bestaat er een lijst van robots waarvan de URDF-bestanden al van gegenereerd zijn en waarvan de hardware al compatibel is met de ROS-*software*. Die lijst van robots is vrij toegankelijk waardoor nieuwe gebruikers eerst in die lijst kunnen kijken of hun gewenste robot er tussen staat zodat ze het URDF-bestand van de robot niet meer zelf moeten genereren. Aangezien de Epson C3 robot niet in de deze lijst staat, is er een alternatief nodig om het URDF-bestand van de robot te bekomen.

5.1.1 *SolidWorks to URDF exporter*

Indien de gebruiker een URDF-bestand nodig heeft voor een grote robot, kan deze beroep doen op een speciaal programma. Dat programma is de *SolidWorks to URDF exporter plugin* die in opdracht van Willow Garage ontwikkeld is. Het programma zet een *assembly* of een samenstelling van een robot, die getekend is in het tekenprogramma *SolidWorks*, om naar een URDF-bestand. Om het programma te kunnen gebruiken is een professionele *SolidWorks* 2012 versie nodig. [40]

De *SolidWorks assembly* van de Epson-robot is in de toestand die op het internet staat niet volledig genoeg om te kunnen converteren naar een URDF-bestand. Dat komt doordat de *assembly* bestaat uit de texturen van de robot. Enkel het uiterlijk van de robot is niet genoeg om een volledig en geldig URDF-bestand te maken van de robot. De verschillende stappen die nodig zijn om de *assembly* van *SolidWorks* om te zetten naar het URDF-bestand, met alle aanpassingen die nodig zijn, staan in bijlage vermeld (Bijlage A). In de bijlage staan ook de belangrijkste elementen waarmee de gebruiker rekening moet houden. Samengevat bestaan de stappen, om een *assembly* van *SolidWorks* te exporteren naar een URDF-bestand, uit:

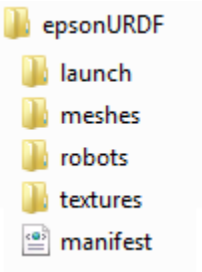
- het *downloaden* van de noodzakelijke CAD-bestanden van de Epson-site;
- het converteren van ieder onderdeel naar een massief onderdeel;
- het configureren van de oriëntatie van de assenstelsels van ieder onderdeel;
- het configureren van de positie van de rotatieassen van de scharnierende elementen;
- het beperken van de hoekrotaties van scharnierende elementen tot de juiste waarden.

Ieder onderdeel van de robot moet massief zijn in *SolidWorks* zodat het converteerbaar is naar een STL-bestand. Indien een onderdeel niet massief is in *SolidWorks*, visualiseert Rviz dat onderdeel niet. Ook is het mogelijk om de massa aan een massief onderdeel mee te geven zodat *SolidWorks* de bijhorende inertie genereert.

De oriëntatie van de assenstelsels van ieder onderdeel is van groot belang. De algemene conventie in ROS is om de positie van het assenstelsel gelijk te leggen met de as van het voorgaande scharnier. Hierbij moet de z-as altijd omhoog wijzen en de x-as vooruit waardoor ook de richting en zin van de y-as bepaald zijn.

De laatste stap beperkt de hoekrotaties van ieder scharnierend element. De hoekrotatie van de onderdelen zijn terug te vinden in de *datasheets* van de robot of in bijlage A van deze scriptie.

De resulterende omzetting, na het volgen van de stappen uit de bijlage, zorgt voor meer bestanden dan enkel het URDF-bestand. Het URDF-bestand staat in een map die de structuur heeft van figuur 49.



Figuur 49: Hiërarchie van een map die een URDF-bestand bevat.

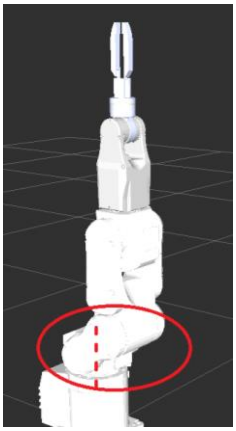
In de map *launch* staan twee bestanden: *display.launch* en *Gazebo.launch.* Met deze bestanden is het mogelijk om het URDF-bestand respectievelijk in Rviz of in Gazebo te openen en te visualiseren. In de map *meshes* staan alle onderdelen van de robot in STL-formaat. Dat is de reden waarom het niet mogelijk is om met een studentenversie van *SolidWorks* de omzetting uit te voeren aangezien het in de studentenversie van *SolidWorks* onmogelijk is om een onderdeel op te slaan als STL-bestand. Het visualiseren van de onderdelen in Rviz gebeurt dankzij de STL-bestanden. Verderop in de scriptie volgt een toelichting van hoe een URDF-bestand opgeroepen kan worden. In de map *robots* staat het URDF-bestand dat dezelfde naam heeft als de map dat in dit geval overeenkomt met epsonURDF. De map *textures* is leeg en wordt nergens gebruikt. Het laatste bestand in de map is het *manifest.xml*-bestand wat algemene informatie bevat over de map. In het *manifest*-bestand staat de naam van het document, de auteur en de licentie.

Het exporteren van de *SolidWorks assembly* naar een geldig URDF-bestand is een iteratief proces. Het kan namelijk zijn dat de gebruiker een kleine fout maakt waardoor het resulterende URDF-bestand ook een fout bevat. Rviz visualiseert het bestand waardoor te zien is dat er een fout zit in de robot. Op deze manier is het eenvoudig om de oorzaak van eventuele fouten op te sporen en onmiddellijk te corrigeren.

De overzichtelijke structuur van het URDF-bestand is te danken aan het gebruik van de XML-programmeertaal. Hierdoor is het ook mogelijk om enkele handmatige aanpassingen te doen aan het bestand nadat dit geëxporteerd is via de *SolidWorks plugin*. Dit zijn enkel aanpassingen die niet toepasbaar zijn op een CAD-bestand. Het is namelijk vaak nodig om een virtuele *link* aan te maken met de naam '*world*' die geen visueel aspect bevat noch een afmeting. Deze *link* dient enkel om de robot permanent te verbinden met het *world frame* dat zowel in visualisatieprogramma's zoals Rviz als in simulatieprogramma's zoals Gazebo het basis*frame* is.

5.1.2 Opbouw URDF-bestand van Epson-robot

Het gegenereerde URDF-bestand bevat altijd dezelfde structuur (zie ook '4.1.1 URDF'). Eerst is er een beschrijving van een *link,* daarna volgt de opbouw van de *joint* die deze *link* verbindt met de vorige *link*. Hieronder staat een voorbeeld van de code voor $link_1$ en $joint_1$ die in figuur 50 ook zijn aangeduid.



Figuur 50: Epson-robot met $link_1$ omcirkeld en stippellijn overeenkomstig met de as van $joint_1$

```
<link
   name="link1">
   <inertial>
     <origin
       xyz="0.0262414878025176 0.00346699232845504
            0.0775022969941165"
       rpy="0 0 0" />
     <mass
       value="0.902877509632839" />
     <inertia
       ixx="0.0057648550330030 7"
       ixy="-0.000290767116334956"
       ixz="0.00124260404590737"
       iyy="0.00541806922654225"
       iyz="-0.000145329861017593"
       izz="0.00692060485538743" />
   </inertial>
   <visual>
     <origin
       xyz="0 0 0"
       rpy="0 0 0" />
     <geometry>
       <mesh
         filename="package://epson_gripper/meshes/link1.STL" />
```

```
      </geometry>
      <material
        name="">
        <color
          rgba="1 1 1 1" />
      </material>
    </visual>
    <collision>
      <origin
        xyz="0 0 0"
        rpy="0 0 0" />
      <geometry>
        <mesh
          filename="package://epson_gripper/meshes/link1.STL" />
      </geometry>
    </collision>
  </link>
```

Het eerste element van de code definieert de naam van het onderdeel, namelijk *link1.* Het tweede element van de code is nieuw ten opzichte van het basis URDF-bestand, dat eerder besproken is in '4.1.1 URDF' en is de *inertial*. Dit deel van de code is het zogenaamde inertie-element dat automatisch gegenereerd is door de *SolidWorks exporter*. Onderdelen van dit element zijn de oorsprong van het assenstelsel van de *link* (*origin*), de massa van de *link* (*mass*), die uitgedrukt is in kg, en de massatraagheidsmomenten over de zes assen van het assenstelsel (*inertia*). Het programma bepaalt de waarden van deze traagheidsmomenten aan de hand van de massa van de *link*.

Het derde element is het *visual* element. De belangrijkste onderdelen hiervan zijn de *geometry* en het *material.* Bij *geometry* wordt de naam van, en het pad naar het STL-bestand van *link1* meegegeven. Het tweede belangrijke onderdeel van het *visual* element is het materiaal. Hiermee is het mogelijk om de materiaalsoort van de *link* te definiëren en ook een kleur te geven aan de *link*.

Het vierde en laatste element van een *link* is het *collision* element. Dezelfde *mesh* die gebruikt is bij het visualiseren van de *link* wordt hier ook gebruikt voor het opstellen van de botsingsmatrix. Deze *mesh* heeft een zeer grote resolutie aangezien het getekend is met behulp van driehoeken. Hoe groter de resolutie van een *mesh* is, hoe kleiner de driehoeken zijn en hoe meer driehoeken nodig zijn om een *link* te vormen. Een hele grote resolutie is echter niet nodig voor de geometrie van een *mesh* die gebruikt wordt bij het *collision* element. De resolutie van een *mesh* is aan te passen met het tekenprogramma *blender*. Dit levert dezelfde *collision matrix* op, maar de rekentijd verlaagt hierdoor. Dat is een voordeel indien het over een grote robot gaat. Bij de Epson-robot is dit niet het geval aangezien het een vrij kleine robotarm is.

```
<joint
  name="joint1"
  type="revolute">
  <origin
       xyz="0.22362 -0.81207 0.14526"
       rpy="6.123E-17 0 3.1416" />
  <parent
       link="base_link" />
  <child
       link="link1" />
  <axis
       xyz="0 0 -1" />
  <limit
       lower="-2.96"
       upper="2.96"
       effort="0"
       velocity="0" />
</joint>
```

Het eerste element, van de specificering van een *joint*, geeft een naam aan de *joint*. Vervolgens definieert het eerste element het type van de *joint.* De meeste *joints* bij de Epson-robot zijn van het type *revolute*. *Revolute joints* kunnen niet blijven ronddraaien, maar hebben een beperkt bereik. De grenzen van de hoekrotaties moeten later ergens ingevuld worden. Het tweede element (*origin)* bepaalt de ligging van de *joint*. Het derde element bepaalt welke *link* de *parent link* is en welke *link* het *child link* is. Dit zijn de twee *links* waar de *joint* als verbinding dient. In dit voorbeeld ligt de *joint* tussen de *base_link* en *link1*. Daarna volgt als vierde element het *axis* element waar de aanduiding plaatsvindt van de as waarrond de rotatie gebeurt. Ten slotte drukt het *limit* element zowel de minimum- als de maximumhoek van de *joint* uit in radialen.

5.1.3 Controle URDF-bestand

Het controleren van het uiteindelijke URDF-bestand gebeurt voornamelijk visueel. Deze visuele controle gebeurt door het bestand in Rviz te openen. Zonder een actie of beweging uit te voeren, is al zichtbaar of alle onderdelen aanwezig zijn en of ze juist gepositioneerd zijn. Rviz heeft een interface ter beschikking waarmee het aansturen van de scharnierende elementen mogelijk is. Het onderzoeken van hoekrotaties gebeurt eveneens visueel.

5.2 Robotconfiguratie via MoveIt!

Om de robot in MoveIt! te kunnen gebruiken, is het eerst noodzakelijk om de *MoveIt! setup assistant* te volgen. Die *setup assistant* wordt toegepast op het URDF-bestand dat resulteert uit de *SolidWorks to URDF exporter plugin*. Het resultaat van de *MoveIt! setup assistant* is een map waarin *launch files* en configuratiebestanden zitten. Deze bestanden zijn nodig voor:

- het manipuleren van de robot,
- het toepassen van padplanningsalgoritmen op de robot,
- het inlezen van sensorbeelden van camera's en lasers,
- de controle en navigatie van de robot.

De inhoud van de configuratiebestanden is afhankelijk van de informatie die het URDF-bestand bevat. De relevante gegevens voor het configuratiebestand zijn de traagheden van de onderdelen, de krachten van de motoren, het type van de camera en de maximale snelheden of versnellingen van de *joints*. Na het configureren van het URDF-bestand vindt de automatische invulling van het configuratiebestand plaats. Indien er gegevens van een parameter ontbreken dan is de waarde onbekend en is dat deel van het bestand leeg of irrelevant. Indien de robot later een ongekende parameter wil gebruiken dan is het vereist om deze handmatig in te vullen.

5.2.1 *MoveIt! Setup Assistant*

Het configureren van het URDF-bestand gebeurt in meerdere stappen. De eerste stap is het kiezen van de robot. Nadat de robot is ingeladen, is deze zichtbaar in de gebruikers*interface*, hierin is het mogelijk om nog eens na te gaan of alle onderdelen er zijn en juist gepositioneerd zijn. Daarna volgt in de tweede stap de eerste operatie namelijk het genereren van een eigen *collisie*matrix of botsingsmatrix. Dat gebeurt met de *self-collision matrix generator*. Deze *generator* zoekt combinaties van twee verschillende onderdelen die hij kan uitschakelen. De botsingsdetectie moet in de *MoveIt! MotionPlanner plugin* in Rviz geen rekening meer houden met de uitgeschakelde combinaties. Combinaties dienen uitgeschakeld te zijn omdat:

- ze altijd in botsing treden,
- ze nooit in botsing treden,
- ze in botsing treden indien de robot zich in de *home*toestand bevindt;
- het met elkaar verbonden onderdelen zijn.

De resolutie van de *generator* is instelbaar tussen 1000 en 100 000. Standaard staat deze waarde ingesteld op 10 000, wat wil zeggen dat de generator 10 000 willekeurige robottoestanden controleert. De resolutie van de generator is omgekeerd evenredig met de rekentijd die nodig is om deze toestanden te bepalen. Het is aan te raden om de resolutie op zijn maximum te zetten bij een industriële robot. Het duurt dan heel lang om de botsingsmatrix te berekenen, maar de betrouwbaarheid verhoogt hierdoor beduidend. De kans bestaat immers dat bij een lage resolutie niet alle uitschakelbare combinaties aan de lijst zijn toegevoegd terwijl er combinaties in staan die er niet in horen te staan. Na het overlopen van alle robottoestanden, komen alle mogelijke combinaties in een lijst te staan die zichtbaar is in de *setup asisstant.* In deze lijst staat bij iedere combinatie of hij uitgeschakeld is of niet. Indien de combinatie is uitgeschakeld, staat er een reden bij waarom.

Een verhoogde rekentijd kan ook te wijten zijn aan de resolutie van de gebruikte *mesh* die dient om een onderdeel te visualiseren. Indien er geen extra informatie wordt meegegeven in het URDF-bestand over *collision checking,* dan wordt deze *mesh* ook gebruikt om de botsingsmatrix op te stellen. Bij zeer complexe robots is het daarom aangeraden om deze *meshes* te vereenvoudigen omdat een hoge resolutie van de robot niet nodig is. Het aanpassen van de *meshes* is mogelijk in tekenprogramma's zoals Blender. Bij de Epson-robot is dit echter niet nodig waardoor het ook niet uitgevoerd is.

In stap drie moet een virtuele *joint* of scharnier toegevoegd worden aan de robot. Deze *joint* verbindt de Epson-robot met het wereld*frame*. De conventionele manier om dat te doen is door de *parent link* te verbinden met de wereld. In het geval van de Epson-robot is dat de *base_link*. Bij de keuze van het type *joint* zijn er opnieuw zes mogelijkheden, net zoals bij de *joints* van een URDF-bestand. Deze zes type *joints* zijn: *revolute, continuous, prismatic, fixed, floating en planar.* In tegenstelling tot het voorbeeld van de PR2-robot kan de Epson-robot niet rondrijden. Hier valt dus de keuze op *joint* type *fixed*. De naam van de virtuele *joint* is zoals aangeraden in de *tutorial*: *virtual_joint*.

Stap vier van de *MoveIt! setup assistant* dient om *planning groups* toe te voegen. Planningsgroepen beschrijven de onderdelen van een robot die samen horen bij het uitvoeren van een bepaalde beweging. De Epson-robot bevat twee planningsgroepen. De eerste planningsgroep is de arm, dat is de volledige Epson-robot zoals die zich in de geleverde toestand bevindt. Het toevoegen van de arm gebeurt door deze als een kinematische ketting te beschrijven en dan als kinematische ketting toe te voegen. Dat heeft als voordeel dat dan hierop de *KDLKinematicsPlugin* toepasbaar is. Bij de tweede planningsgroep, de eindeffector, gebeurt het toevoegen anders. Hier is bij de Epson-robot gekozen voor het toevoegen van de *links* apart, dus de onderdelen van de eindeffector worden apart toegevoegd hier. Het toepassen van bewegingen is nu mogelijk op deze twee planningsgroepen. Dat een planningsgroep kan bewegen is te danken aan de *kdl_kinematics_plugin/ KDLKinematicsPlugin*. Deze *plugin* past inverse kinematica toe op de robot zodat de robot kan bewegen. De *plugin* zorgt dus voor de aansturing van de aparte *links* van de robot indien er een gewenste begin- en eindtoestand gekend is. Iedere planningsgroep maakt gebruik van de kinematische *solver*. De kinematische

solver heeft enkele instelbare parameters. Bij de Epson-robot staan deze parameters op hun standaardwaarden. De *kinematics_solver_search_resolution* komt overeen met 0.005 en de *kinematics_solver_timeout* staat op 0.05 seconden. Zoals de naam doet vermoeden is de *kinematics_solver_search_resolution* de resolutie waarop de *solver* zoekt naar mogelijke toestanden van de *links*. De *kinematics_solver_timeout* is daarentegen uitgedrukt in seconden en geeft de tijd weer die de *solver* krijgt voor iedere interne iteratieve berekening. [41]

Stap vijf is geen verplichte stap. In deze stap is het mogelijk om robottoestanden toe te voegen indien dit nodig is. Dat is bijvoorbeeld handig indien de robot, beschreven in het URDF-bestand, initieel niet in zijn *home*positie staat. De Epson-robot stond initieel al in zijn *home*toestand waardoor deze stap overbodig is. In deze stap is het ook mogelijk om alle componenten van de robot te bewegen. Dat levert een extra visuele controle op het URDF-bestand, aangezien te zien is of de onderdelen correct bewegen of niet.

Stap zes maakt het mogelijk om de eindeffector te benoemen. In stap vier is de eindeffector reeds gemaakt als planningsgroep. Als eerste moet de eindeffector een naam toegewezen krijgen. Vervolgens kiest men de planningsgroep die overeenkomt met de eindeffector. Deze informatie is al genoeg voor de *setup assistant*. Voor de Epson-robot zijn de optionele vakken ook ingevuld om eventuele fouten zeker te vermijden. Zo is als *parent link* $link_6$ gekozen en als *parent group* de volledige Epson-arm.

Stap zeven is het toevoegen van passieve *joints*. Dat zijn onderdelen die tot een robot behoren, maar niet tot een planningsgroep. Deze stap is optioneel en is niet van toepassing voor de Epson-robot.

De achtste en laatste stap slaat alles op en genereert automatisch de configuratiebestanden. De definitieve versie van de map waarin alle configuratiebestanden van de Epson-robot staan, heet '*epson_moveit_generated*'. Dit is, in combinatie met het URDF-bestand, de belangrijkste map van deze masterproef.

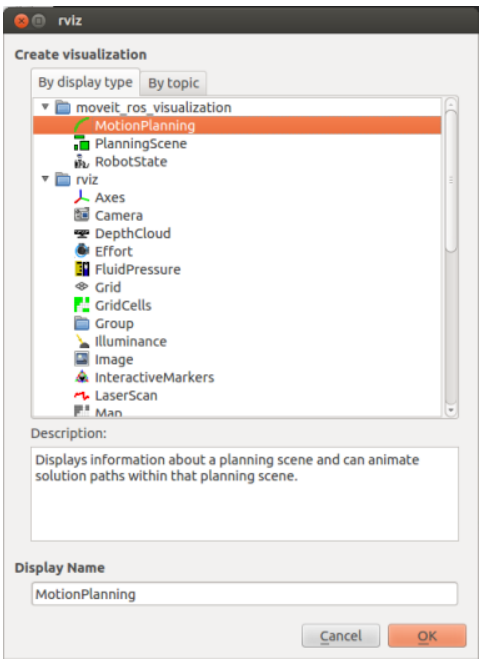
5.3 MoveIt! Rviz *plugin*

Het visualisatieprogramma van ROS, Rviz, bevat een *plugin* van MoveIt!. In Rviz bestaan er heel wat verschillende *plugin*s die voor heel veel verschillende toepassingen gebruikt kunnen worden. De belangrijkste *plugin* van Rviz die nodig is voor het oplossen van de probleemstelling van dit eindwerk, is de *MotionPlanning plugin*. Elke *plugin* is in Rviz in of uit te schakelen via de *global options tab*.

Voordat de *MotionPlanner plugin* te gebruiken is op de robot, moeten eerst enkele gegevens van de robot aan de *plugin* gekoppeld worden zoals:

- *robot description*: robot_description,
- *planning scene topic*: /move_group/monitored_planning_scene,
- *planning group*: complete_arm,
- *trajectory topic*: /move_group/display_planned_path.

Vanaf nu is het duidelijk waarom de configuratie in de *MoveIt! setup assistant* zo noodzakelijk is. De *plugin* gebruikt onder andere de planningsgroep die met de *setup assistant* aangemaakt is. De *trajectory topic* is de map waarin de *messages* van ROS zijn opgeslagen. Door deze *messages* is communicatie mogelijk tussen verschillende programma's die tegelijk actief zijn in ROS. *Plugins* toevoegen is mogelijk door op *add* te klikken in de *display* tab van Rviz. Daarna opent het venster dat te zien is in figuur 51. In dat venster staan alle *plugins* die in Rviz bestaan.



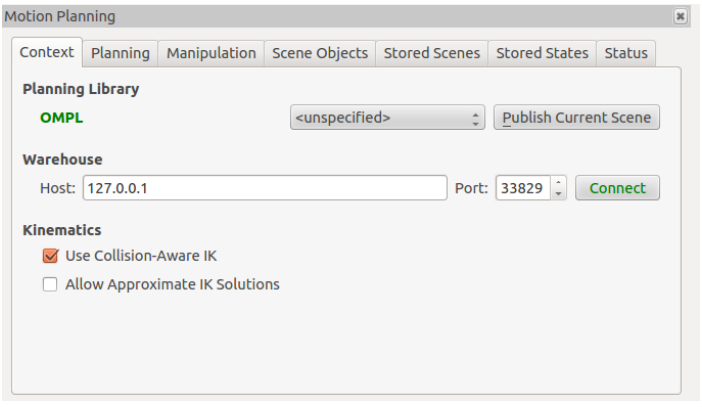
Figuur 51: Lijst van plugins van Rviz

Als alle besproken gegevens juist ingesteld zijn, is het mogelijk om de *MotionPlanner plugin* toe te passen op de Epson C3 robot. Deze *plugin* maakt communicatie tussen MoveIt! en bibliotheken die padplanningsalgoritmen bevatten mogelijk. De standaardbibliotheek voor deze toepassing is de *Open Motion Planning Library*, of OMPL. Niet alle algoritmen die in de *interface* van de *omplapp* aanwezig zijn, zijn ook te gebruiken in Rviz. De lijst van padplanningsalgoritmen die wel te gebruiken zijn in Rviz zijn:

- BKPIECE
- EST
- KPIECE
- LBKPIECE
- PRM
- PRM*
- RRT
- RRT-Connect
- RRT*
- SBL
- T-RRT

De algoritmen van OMPL zijn in eerste instantie niet bruikbaar voor toepassingen in de robotica. Deze algoritmen zijn eigenlijk abstract, maar MoveIt! heeft de algoritmen configureerbaar gemaakt zodat het toch mogelijk is om ze te gebruiken op eender welke robot. De enige voorwaarde is dat de robot juist geconfigureerd moet zijn.

Ieder padplanningsalgoritme zoekt een pad als oplossing om van de begintoestand naar de eindtoestand te geraken. De keuze van het padplanningsalgoritme is in te stellen in het tabblad 'context' van de *MotionPlanner plugin,* dat te zien is in figuur 52, op de plaats waar <*unspecified*> staat.



Figuur 52: Keuze padplanningsalgoritme bij de MotionPlanning plugin in Rviz

Indien er geen keuze wordt gemaakt voor een padplanningsalgoritme, zoals dat in figuur 52 het geval is, kan de robot toch een beweging uitvoeren. In dat geval kiest Rviz automatisch het standaardalgoritme, LBKPIECE. In het tabblad *context* zijn er nog andere parameters die in te stellen zijn, maar die ook optioneel zijn. In het tabblad staat ook de knop '*publish current scene*'. Nadat er veranderingen zijn aangebracht in de omgeving, zoals het verplaatsen, het toevoegen of het verwijderen van objecten, moet deze knop ingedrukt worden zodat Rviz vanaf dan berekeningen uitvoert op de nieuwe omgeving in plaats van de vorige omgeving.

'*Warehouse'* is het volgende, belangrijke onderdeel van het *context*tabblad. Met de knop *connect* is het mogelijk om een verbinding te maken met een *database,* die opgeslagen is op de computer. Het standaard poortnummer van de database is 33829 en hoeft niet veranderd te worden, tenzij deze poort al bezet is. Indien de poort bezet is, moet de gebruiker een andere poort kiezen en moet de gebruiker de poort in de *database* zelf ook veranderen zodat een verbinding mogelijk is. Dat komt echter zelden voor in de praktijk.

Het is mogelijk om *scenes* en *queries* op te slaan in een *database*. *Scenes* zijn omgevingen die uit objecten bestaan. Bij het opslaan van zo een *scene* onthoudt de *database* welke objecten allemaal aanwezig waren in de omgeving en welke positie en oriëntatie ze hadden. Bij het invoegen van de *scene* voegt Rviz automatisch alle bijhorende objecten toe op hun juiste positie. Op die manier is het niet nodig deze objecten iedere keer opnieuw te importeren en te verslepen tot ze op de juiste plaats staan. *Queries* zijn opgeslagen begin- en eindtoestanden van de robot. Een enkele *query* komt overeen met een combinatie van een begin- en eindtoestand. Op die manier moet niet iedere keer handmatig de begin- en eindtoestand ingesteld worden. Rviz kan een *query* invoegen vanuit de *database* zodat experimenten altijd op dezelfde *query's* uitgevoerd kunnen worden. Dankzij de *database* is het dus mogelijk om verschillende algoritmen en verschillende parameters op dezelfde situatie toe te passen zodat de algoritmen rechtstreeks te vergelijken zijn met elkaar. Het vergelijken van padplanningsalgoritmen op dezelfde situatie noemt men ook wel *benchmarking*. Zo een *benchmarking tool* bestaat al voor de OMPLapp. Deze *tool* leest gegevens in, die in een configuratiebestand zijn gespecificeerd. Dat configuratiebestand is aangemaakt door de gebruiker. Het bestand beschrijft de algoritmen, de *scene*, de *queries* en het aantal testen die de gebruiker wil uitvoeren. Rviz voert dan deze testen uit met de instellingen die zijn meegegeven in het bestand. De resultaten van deze testen zijn dan terug te vinden in een *log*-bestand. Deze resultaten kunnen ook gevisualiseerd worden in grafieken in een pdf-bestand.

De *benchmarking tool* voor de *MotionPlanner plugin* is echter nog in ontwikkeling, hiervan is dus geen gebruik gemaakt in de masterproef. Het simuleren en vergelijken van de verschillende situaties gebeurt daarom handmatig wat omslachtig en tijdrovend is ('5.3.3 Uitlezen van resultaten'). Bij het toepassen van een *benchmarking tool* is er daarentegen geen visuele controle van het pad die de robot volgt. Het voordeel van het handmatig simuleren is dat het gevolgde pad van de robot zichtbaar is in Rviz. Ieder pad wordt beoordeeld op zijn vloeiendheid of zijn benadering op het optimale pad. Ook wordt ieder pad beoordeeld op zijn betrouwbaarheid. De betrouwbaarheid hangt af van het percentage dat het algoritme een geldig pad vindt en dus niet *failed* als resultaat geeft. Het is opgevallen dat de *MotionPlanner* soms als resultaat een geldig

pad teruggeeft terwijl duidelijk te zien is dat er een botsing optreedt tussen de robot en een obstakel. Deze slechte resultaten van een algoritme zijn alleen visueel waarneembaar en zijn niet uit de *benchmarking* resultaten te halen. De manier waarop de resultaten uitgelezen worden, is in '5.3.3 Uitlezen van resultaten' besproken en de resultaten van de experimenten staan in 'Hoofdstuk 6: Resultaten' vermeld.

Een laatste onderdeel van het '*context'* tabblad is de kinematics. Hier is het mogelijk om te kiezen voor *collision-aware IK* of niet. Deze parameter moet aangevinkt blijven want anders registreert de robot geen botsingen met de omgeving. Indien de parameter aangevinkt is en er een botsing optreedt, markeert de *MotionPlanner* de onderdelen die in botsing treden in het rood.

Het tweede tabblad van de *MotionPlanner plugin* is het '*planning'* tabblad. Het tabblad bevat parameters die nog niet volledig operationeel zijn, dat wil zeggen dat ze er wel al staan, maar dat ze nog geen werking of invloed hebben. Een onderdeel van dit tabblad heeft als titel *options*. Hier zijn enkele aanpassingen mogelijk zoals het aanvinken van *allow replanning*. Het effect van de functie is echter niet duidelijk. Het verwachte resultaat zou zijn dat, als er een foutief pad is gevonden en er nog rekentijd over is, dat het algoritme een nieuwe poging zou doen om een geldig pad te vinden. Er zijn echter geen zichtbare gevolgen bij het variëren van deze parameter. Ook is er op het internet geen documentatie te vinden over deze parameter. Vermoedelijk zijn deze knoppen dus nog niet *software*matig geïntegreerd, maar dienen ze als mogelijke uitbreiding van de *MotionPlanner plugin* die immers nog niet volledig is afgewerkt. Een parameter die in het tabblad '*planning'* staat en wel effectief werkt, is de maximale rekentijd die een padplanningsalgoritme mag gebruiken. Indien de maximale rekentijd bereikt is en het algoritme nog geen pad heeft gevonden, is het resultaat van dat pad *failed* wegens een te hoge rekentijd. In het *planner* tabblad zijn ook de instellingen van de werkingsruimte van de robot aan te passen. Zowel de afmetingen als het centrum van de werkingsruimte is instelbaar. Het belangrijkste van het tabblad '*planning'* is het onderdeel om te plannen. Het onderdeel bevat drie knoppen: *plan, execute* en *plan and execute*. De eerste knop is de belangrijkste aangezien dezeervoorzorgt dat een pad wordt berekend voor de ingestelde *query* in de ingestelde *scene* door middel van het ingestelde padplanningsalgoritme. De *execute* knop test nog eens of het pad wel degelijk geldig is. De *plan and execute* knop berekent het pad met als begintoestand de *home*toestand van de robot. De eindtoestand is nog steeds de ingestelde toestand.

Het derde tabblad is het '*manipulation'* tabblad. Dit tabblad is van belang indien de eindeffector moet interageren met de omgeving zoals het opnemen van een object. De functie werkt met behulp van de *Object Recognition Kitchen*. Het algemene principe is om, in c++ code, objecten te herkennen in de omgeving van de robot zodat deze de objecten kan opnemen en verplaatsen. De functie is niet gebruikt in de masterproef en wordt dus niet verder besproken.

Het vierde tabblad bevat de *scene objects*. Hier staat een lijst van alle objecten die op dat moment in de omgeving aanwezig zijn. Twee verschillende bestandsformaten van *scenes* zijn toelaatbaar om in te voegen in de omgeving. De eerste zijn STL-bestanden en de andere zijn tekst gebaseerde bestanden met de extensie '*scene'*. Op het moment dat objecten ingeladen zijn, is het mogelijk om de grootte, positie en oriëntatie ervan aan te passen met behulp van de *interactive markers*. Nadat op de knop '*publish current scene*' is gedrukt, houdt Rviz rekening met deze objecten als obstakel.

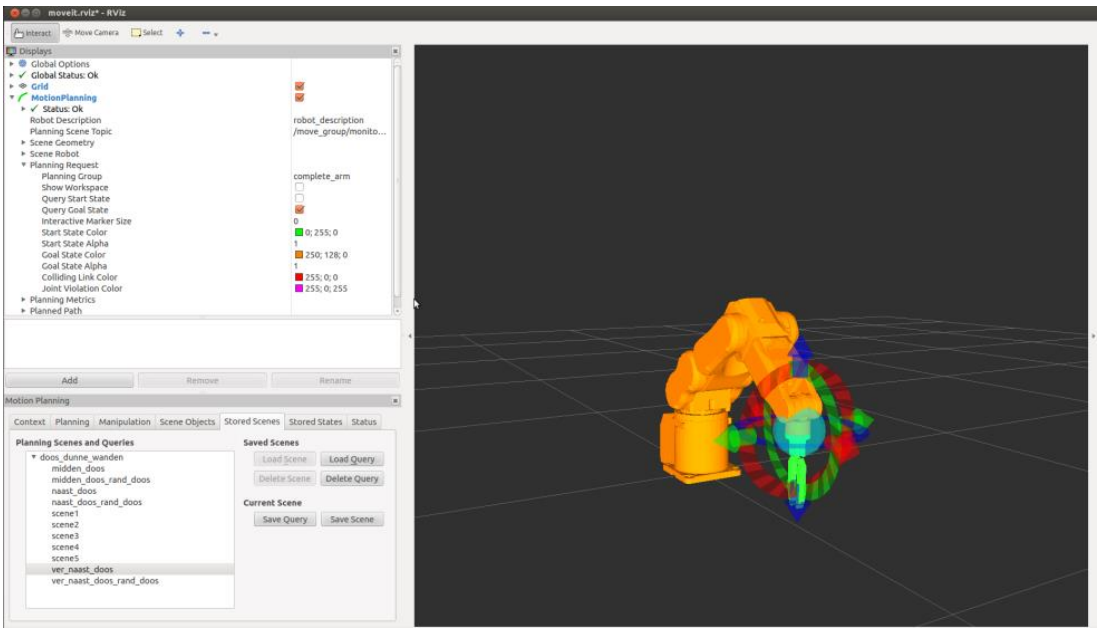
Het volgende tabblad, *stored scenes,* bevat de opgeslagen *scenes* en *query's*. Het tabblad is enkel nuttig indien er een verbinding is gemaakt met de *database*. Eens dat die verbinding tot stand is gebracht, zijn alle *scenes* en *queries* die in de *database* zitten zichtbaar in dit vak. De knop '*publish current scene*' moet na het inladen van een *scene* weer ingedrukt worden zodat de algoritmen hier rekening mee houden.

Ook is het mogelijk om verschillende *states* of toestanden van de robot op te slaan. De opgeslagen toestanden kunnen dan als begin- of als eindtoestand ingesteld worden. Dit gebeurt in het '*stored states'* tabblad.

Het laatste tabblad, '*status',* geeft een lijst van activiteiten die in Rviz uitgevoerd zijn in chronologische volgorde. Een voorbeeld van zo een activiteit is het aanpassen van de begintoestand.

5.3.1 Interactie met de robot

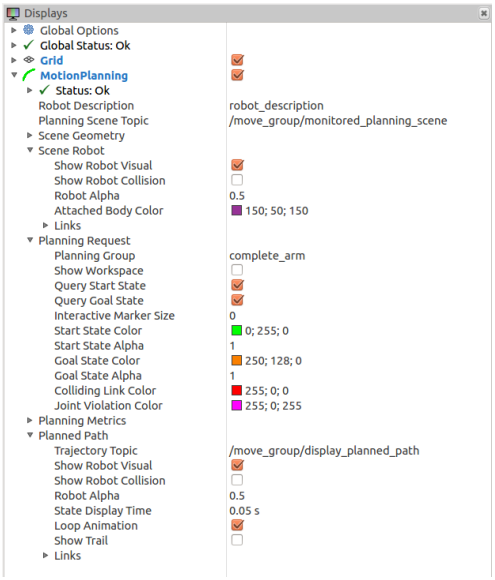
Als de *interact*-knop is ingeduwd in Rviz, links bovenaan het scherm, is het mogelijk om de gevisualiseerde robot te bewegen. Een interactieve *marker* is dan zichtbaar rond de eindeffector van de robot indien deze juist is geconfigureerd met de *MoveIt! setup assistant*. Met deze *marker* is het mogelijk om de eindeffector te verslepen naar een andere positie en/of rotatie. Zolang de eindeffector binnen het werkingsgebied van de robot blijft, passen de *links* van de robot zich aan zodat de eindeffector de *marker* kan volgen. De posities en rotaties van de *links* kunnen dus niet door de gebruiker aangepast worden. De begin- en eindtoestand zijn in te stellen met de interactieve *markers*. De begintoestand wordt standaard voorgesteld als een groene robot, terwijl de eindtoestand als een oranje robot wordt voorgesteld. In figuur 53 is de Rviz *interface* te zien waar de Epson-robot is gevisualiseerd in een eindtoestand. Ook is de interactieve *marker* te zien waarmee de positie en rotatie van de eindeffector van de robot aan te passen is.



Figuur 53: Epson-robot in eindtoestand met interactieve marker in Rviz

Het opslaan van toestanden van de robot zorgt voor enkele voordelen. Ten eerste wordt vermeden dat de *joints* van de robot anders staan bij dezelfde positie en oriëntatie van de eindeffector. Ten tweede kunnen alle opgeslagen standen van de robot als begin- of als eindtoestand ingesteld worden. Daardoor is het mogelijk om na het uitvoeren van een pad de vorige eindtoestand als huidige begintoestand te nemen zodat de robot zijn pad vervolgt. Dit is nodig omdat er nog geen mogelijkheid is in Rviz om een robot een bepaald pad te laten volgen, zoals een rechte lijn indien de toepassing een lasrobot is. Ook is het genereren van tussentoestanden, waar een pad door moet lopen, onmogelijk in Rviz.

In het *display* tabblad staan enkele opties die de interactie met de robot vereenvoudigen. Bij *scene robot* is het mogelijk om de *home*toestand van de robot aan of uit te vinken. Indien deze uit gevinkt is, is de witte robotarm verborgen. Het is ook mogelijk om de kleur van de robotarm te veranderen of enkele *links* onzichtbaar te maken. Bij *planning request* is het mogelijk om de begin- en eindtoestand aan of uit te vinken. Dit is nodig indien bijvoorbeeld enkel de beweging belangrijk is om te zien. De begin- en eindtoestand belemmeren het zicht op het pad dan niet. De kleuren van de begin- en eindtoestand zijn ook aanpasbaar. Standaard staat de begintoestand in het groen en de eindtoestand in het oranje. Rode onderdelen van de robot zijn in botsing met een ander object. Het laatste belangrijke element van het *display* tabblad is '*planned path*'. Zowel de *loop animation* als de *show trail*-functie zijn hier aan en uit te schakelen. Door *loop animation* aan te vinken wordt het pad steeds opnieuw gevisualiseerd. De *show trail* functie toont daarentegen verschillende toestanden waardoor de robot gaat bij het uitvoeren van een pad. Figuur 54 geeft een overzicht van het *displays* tabblad.



Figuur 54: Het display tabblad van Rviz

5.3.2 Aanpassen parameters van algoritmen

De parameters van de padplanningsalgoritmen in Rviz hebben allemaal een standaardwaarde ingesteld. Er wordt niet van de gebruiker verwacht dat hij een parameter gaat aanpassen. Er is daarom geen *interface* aanwezig die het mogelijk maakt om parameters eenvoudig aan te passen. De parameters moeten daarentegen aangepast worden voor het opstarten van Rviz. Dat komt doordat de instellingen van de parameters enkel aangepast worden op het moment dat Rviz opstart. Vanaf het moment dat Rviz geopend is, is het niet meer mogelijk om de parameters aan te passen. [42]

Het aanpassen van parameters gebeurt in het *ompl_planning.yaml*-bestand, dat zich in de map *config* bevindt. Dit bestand is automatisch aangemaakt tijdens het configureren van de robot met behulp van de *MoveIt! setup assistant*. Elk algoritme van OMPL dat aanwezig is in Rviz wordt in het *config* bestand gedeclareerd. De declaratie van een algoritme gebeurt als volgt, met als voorbeeld het RRT-algoritme:

```
Planner_configs:
  RRTkConfigDefault:
    type: geometric::RRT
```

Het gedeelte dat volgt op *type* verwijst naar de *.cpp*-bestanden en de bijhorende *.h*-bestanden waarin de declaratie plaatsvindt van de parameters van padplanningsalgoritmen. Om een waarde van een parameter zoals *Range* of *GoalBias* aan te passen, moet de volgende code toegevoegd worden:

```
Planner_configs:
  RRTkConfigDefault:
    type: geometric::RRT
    range: 0.2
    goal_bias: 0.05
```

De juiste benamingen van de parameters zijn terug te vinden in de *.cpp* bestanden van het bijhorende padplanningsalgoritme. Een voorbeeld hiervan is in figuur 55 te zien, voor het RRT-algoritme. In dit bestand is ook de minimale en de maximale waarde terug te vinden van iedere parameter. Bij RRT staat de standaardwaarde van *GoalBias* op 0.05 en de standaardwaarde van de *range* staat op 0.0. De waarde van de *range* wil zeggen dat er geen beperking is gesteld op de maximale lengte van een verbinding. De *range* moet tussen 0 en 10 000 liggen en de *goalBias* is een procentuele waarde en ligt dus tussen 0 en 1.

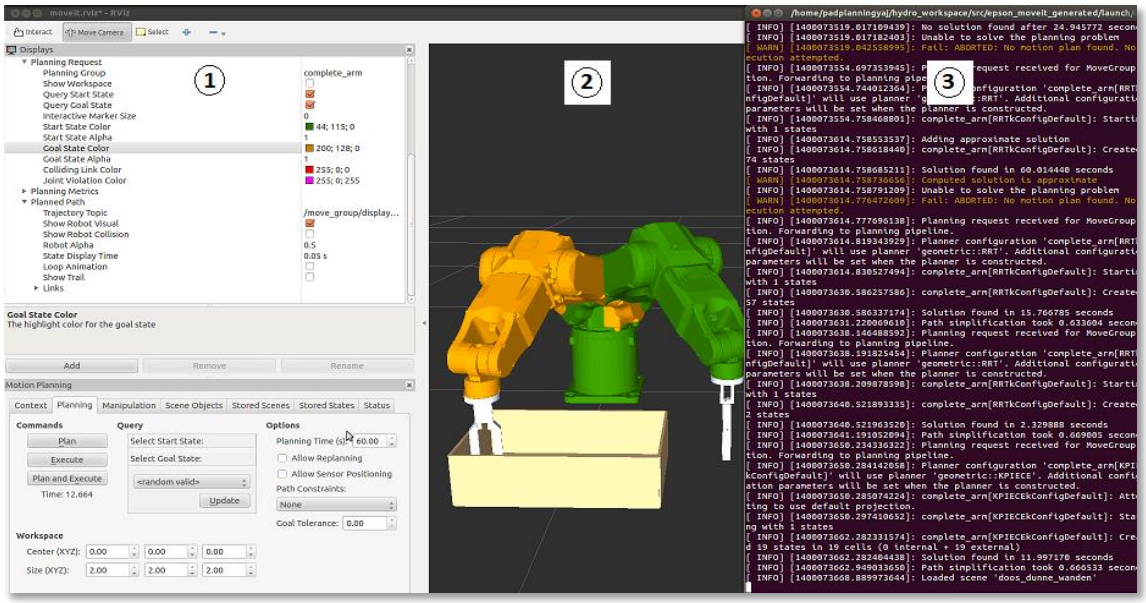
```
ompl::geometric::RRT::RRT(const base::SpaceInformationPtr &si) : base::Planner(si, "RRT")
{
    specs_.approximateSolutions = true;
    specs_.directed = true;

    goalBias_ = 0.05;
    maxDistance_ = 0.0;
    lastGoalMotion_ = NULL;

    Planner::declareParam<double>("range", this, &RRT::setRange, &RRT::getRange, "0.:1.:10000.");
    Planner::declareParam<double>("goal_bias", this, &RRT::setGoalBias, &RRT::getGoalBias, "0.:.05:1.");
}
```

Figuur 55: Standaardinstellingen RRT-parameters in het rrt.cpp bestand van OMPL

5.3.3 Uitlezen van resultaten

Het simuleren gebeurt dus in Rviz waarervoorgetest kan worden met verschillende padplanningsalgoritmen. De resultaten van de simulaties staan in 'Hoofdstuk 6: Resultaten' besproken. Dit deel verklaart hoe de gebruiker aan de resultaten kan geraken. In het onafhankelijk werkende OMPL-programma is het mogelijk om resultaten van algoritmen te bekijken in een *log window*. Hierin staat de rekentijd die het algoritme nodig heeft om een pad te vinden. Ook resultaten die afhankelijk zijn van het gebruikte algoritme staan vermeld in het venster.

Rviz bevat niet zo een *log window,* maar de resultaten zijn wel zichtbaar in de *terminal* van waaruit Rviz is opgestart. In figuur 56 is de volledige simulatieomgeving te zien (1 en 2) met rechts (3) de terminal waarin een deel van de resultaten zichtbaar zijn. Zo komt er na het vinden van een (geldig pad) de benodigde rekentijd te staan en het aantal toestanden (en cellen) waardoor het resultante pad loopt.

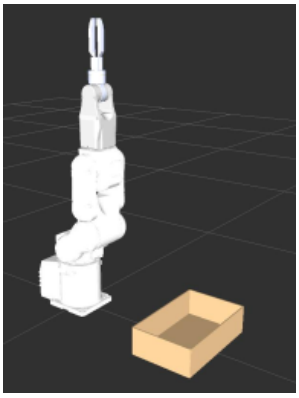


Figuur 56: Simulatieomgeving van Rviz inclusief de terminal

Een opmerking hierbij is dat de benodigde rekentijd afhankelijk is van de computer. De behaalde resultaten die in het volgende hoofdstuk besproken zijn, zijn behaald met een fujitsu-CELSIUS-H700 laptop, de specificaties hiervan staan in bijlage 2.

6 Resultaten

Dit hoofdstuk geeft, analyseert en evalueert de resultaten van de padplanningsalgoritmen toegepast op de probleemstelling. De robot moet een bak met willekeurig geplaatste werkstukken leegmaken door middel van een zesassige robot die gemodelleerd is ('Hoofdstuk 5: creëren van simulatieomgeving'). Zoals eerder vermeld is de omgeving nagemaakt door een bak te modelleren. Door middel van de gemodelleerde robot en bak is de probleemstelling in simulatie te onderzoeken. Figuur 57 toont deze modellering van de robot en bak in een werkingsgebied binnen Rviz. Om een bak optimaal leeg te maken is er een onderzoek nodig die een realistische situatie neemt. In de simulatie is er daarom voor gekozen om de begintoestand zo te kiezen dat deze verschillende posities aanneemt. De eindtoestand zal daarentegen in de bak liggen wat de simulering van het opnemen van een werkstuk nabootst. Om de optimale werking voor deze probleemstelling te vinden zijn er verschillende experimenten uitgevoerd. Zo test het eerste experiment wat de invloed is van een algemene parameter op het gevonden pad. Een tweede experiment gaat de invloed na van de specifieke parameters van de padplanningsalgoritmen. Een derde experiment gaat na welk padplanningsalgoritme het beste resultaat geeft voor de probleemstelling van dit eindwerk. Een vierde en laatste experiment test een alternatieve methode om de bak leeg te maken door middel van padplanningsalgoritmen. Bij de bespreking van deze alternatieve methode wordt dit resultaat vergeleken met het beste padplanningsalgoritme dat gevonden is in experiment drie.



Figuur 57: Gemodelleerde omgeving

Enkele instellingen blijven doorheen de experimenten hetzelfde. De maximale rekentijd staat ingesteld op 60 seconden. Indien het algoritme langer dan een minuut nodig heeft om een pad te berekenen, dan is dat in de praktijk niet bruikbaar. Daarom stopt ieder algoritme na 60 seconden zoeken en geeft het algoritme als resultaat terug dat het geen pad kon vinden. Ook de bak is in (bijna) alle experimenten een vaste parameter. De wanden van de bak hebben een dikte van 0,005m. Deze dikte is vrij laag, maar in de praktijk vaak een waargenomen waarde aangezien kartonnen dozen ongeveer deze dikte hebben. De lengte, breedte en hoogte van de bak zijn respectievelijk 0,35m, 0,22m en 0,10m. Deze afmetingen zijn vrij laag voor in de industrie, maar hier is toch voor gekozen omdat het grijpen van werkstukken in een kleine bak moeilijker is dan in een grote bak. Dit komt doordat de robot minder ruimte heeft in de bak om

te bewegen waardoor er een hogere kans is op een botsing. Hierdoor hebben algoritmen het moeilijker om een pad te zoeken tot in een kleine bak. Indien het algoritme goede resultaten geeft voor een kleine bak, dan geeft het algoritme ook goede resultaten bij een grotere bak waar meer ruimte is.

Bij ieder experiment wordt de performantie van een algoritme beoordeeld. De beoordeling op de performantie gebeurt op basis van:

- de gemiddelde benadering van het pad tot het optimale pad;
- de gemiddelde rekentijd die nodig is om een geldig pad te vinden;
- de betrouwbaarheid van het algoritme;

De eerste parameter van de beoordeling, de gemiddelde benadering van het pad tot het optimale pad, geeft een beeld van de vloeiendheid van een pad. Het kan namelijk zo zijn dat een algoritme iedere keer snel een pad vindt, maar dat de paden hiervan zeer lang zijn en veel omweg maken. Om deze parameter in rekening te brengen wordt ieder pad beoordeeld op basis van zijn benadering tot het optimale pad. Een pad kan bij de beoordeling van de benadering enkele discrete waarden aannemen namelijk:

- 6: zeer goed,
- 5: goed,
- 4: redelijk,
- 3: matig,
- 2: zwak,
- 1: zeer zwak,
- 0: *failed* of een rekentijd hoger dan 60 seconden,
- -5: foutieve oplossing.

Een pad is *failed* indien het algoritme als resultaat teruggeeft dat het geen pad kan vinden. Dat kan voorvallen indien er een eindvoorwaarde is bereikt en het algoritme nog geen pad heeft gevonden of een pad dat door een obstakel gaat. Indien het algoritme meer als 60 seconden nodig heeft om een pad te vinden, geeft het algoritme ook *failed* terug als resultaat, wat een waarde van 0 oplevert.

Het algoritme kan ook een foutieve oplossing geven wat erger is dan *failed* als resultaat teruggeven. Een pad is foutief indien het algoritme denkt dat het een geldig pad is, terwijl visueel te zien is dat de robot een obstakel raakt. Aangezien deze situatie absoluut niet mag voorvallen in de praktijk, houden de tabellen deze waarden bij en krijgt de vloeiendheid van een pad de waarde -5 toegewezen.

De laatste parameter van de beoordeling, de betrouwbaarheid van het algoritme, bestaat uit enkele parameters. Algemeen gezien kan men het algoritme als volledig betrouwbaar beschouwen indien het algoritme 100% van de keren een geldig pad vindt. Vandaar dat de betrouwbaarheid van een pad sterk overeenkomt met het percentage van geldige paden dat het algoritme gevonden heeft. Zoals eerder vermeld kan een niet gevonden pad te wijten zijn aan een te hoge rekentijd, een *failed* pad of een foutief pad. Bij het bespreken van de betrouwbaarheid van een algoritme, hebben deze elementen ook een aandeel aangezien een algoritme dat vaak foutieve paden berekent minder betrouwbaar is dan een algoritme dat vaak te lang zoekt of *failed* geeft als resultaat. De evaluatie, van de betrouwbaarheid van een algoritme, bespreekt daarom:

- het percentage van gevonden paden die geldig zijn;
- het percentage dat het algoritme *failed* als resultaat teruggeeft;
- het percentage dat het algoritme meer dan 60 seconden nodig heeft om te berekenen;
- het percentage van gevonden paden die foutief zijn;

Aangezien dit totaal altijd met 100% overeenkomt, gebruikt ieder experiment een '100% gestapelde kolom' grafiek om de verschillen in betrouwbaarheid duidelijk te maken. Bovendien komt ieder resultaat dat in een grafiek of tabel staat overeen met een gemiddelde waarde van een aantal metingen. Die metingen zijn allemaal op dezelfde manier uitgevoerd en hebben allemaal dezelfde parameters.

6.1 Algemene parameter

Algemene parameters zijn onafhankelijk van het gekozen padplanningsalgoritme. Een voorbeeld van een algemene parameter is de botsingsdetectieresolutie. Een andere algemene parameter is de maximale rekentijd dat het algoritme mag gebruiken. Deze parameter is hiervoor al besproken en wordt niet aangepast aangezien het de werking van de algoritmen niet verandert.

6.1.1 Botsingsdetectieresolutie

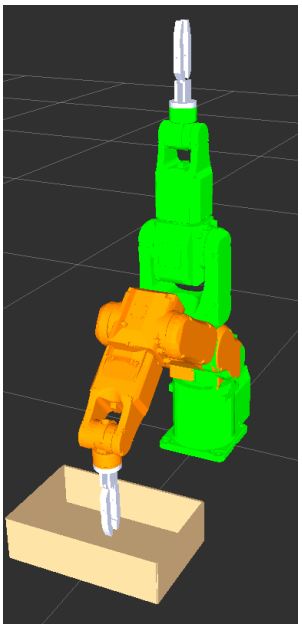
Dit experiment onderzoekt de invloed van de botsingsdetectieresolutie, wat een zeer belangrijke parameter is, op het pad. De botsingsdetectieresolutie is als '*longest valid segment fraction'* aan te passen in het 'ompl_*planning*.yaml'-bestand (figuur 58). De waarde van het *longest valid segment fraction* is uitgedrukt in meters. Het *longest valid segment fraction* komt overeen met de afstand tussen twee deel*vertices* bij het uitvoeren van een botsingsdetectie op een verbinding ('2.2.1 Geldigheid van *vertices* en *edges'*). Een lage waarde van het *longest valid segment fraction* komt overeen met een hoge resolutie. De resolutie verhoogt namelijk als de botsingsdetectie onderscheid kan maken tussen kleinere afstanden.

```
    - PRMstarkConfigDefault
  projection_evaluator: joints(joint1,joint2)
  longest_valid_segment_fraction: 0.05
gripper:
  planner_configs:
```

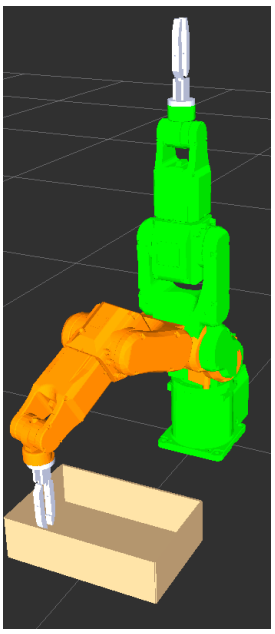
Figuur 58: Aanpassing botsingsdetectieresolutie

Het experiment is uitgevoerd op verschillende *query's* en twee padplanningsalgoritmen. Aangezien de algemene parameter onafhankelijk is van het algoritme wordt het experiment uitgevoerd op PRM en RRT, wat de twee meest algemene padplanningsalgoritmen zijn in hun categorie. PRM behoort tot de wegenkaartalgoritmen en RRT behoort tot de boomstructuuralgoritmen. Het *longest valid segment fraction* wordt de eerste keer op 0,05m ingesteld en de tweede keer op 0,005m. De *query's* zijn zo gekozen dat er eenvoudige problemen tussen zitten en dat er moeilijkere problemen tussen zitten. Het experiment is uitgevoerd op de volgende *query's*:

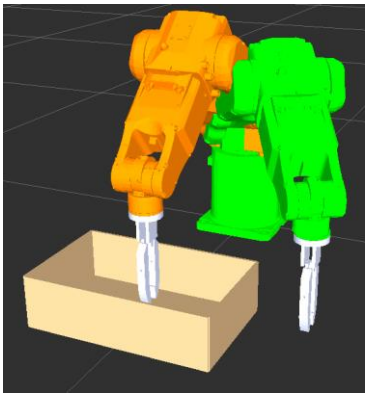
- van de *home*toestand naar het midden van de doos (figuur 59),
- van de *home*toestand naar de rand van de doos (figuur 60),
- van vlak langs de doos naar het midden van de doos (figuur 61),
- van vlak langs de doos naar de rand van de doos (figuur 62),
- van ver langs de doos naar het midden van de doos (figuur 63),
- van ver langs de doos naar de rand van de doos (figuur 64).



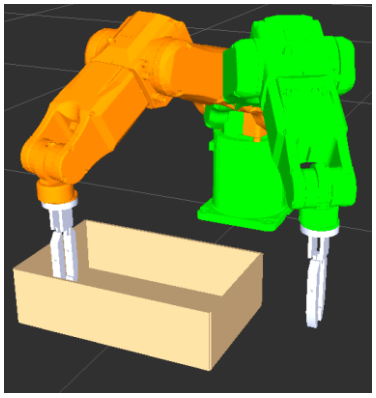
Figuur 59: Home naar midden doos



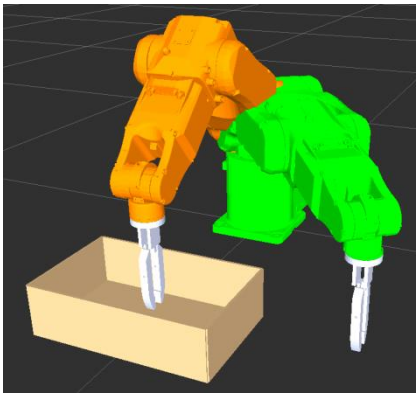
Figuur 60: Home naar rand doos



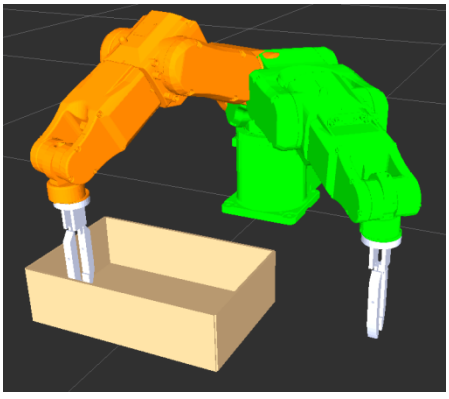
Figuur 61: Langs doos naar midden doos



Figuur 62: langs doos naar rand doos



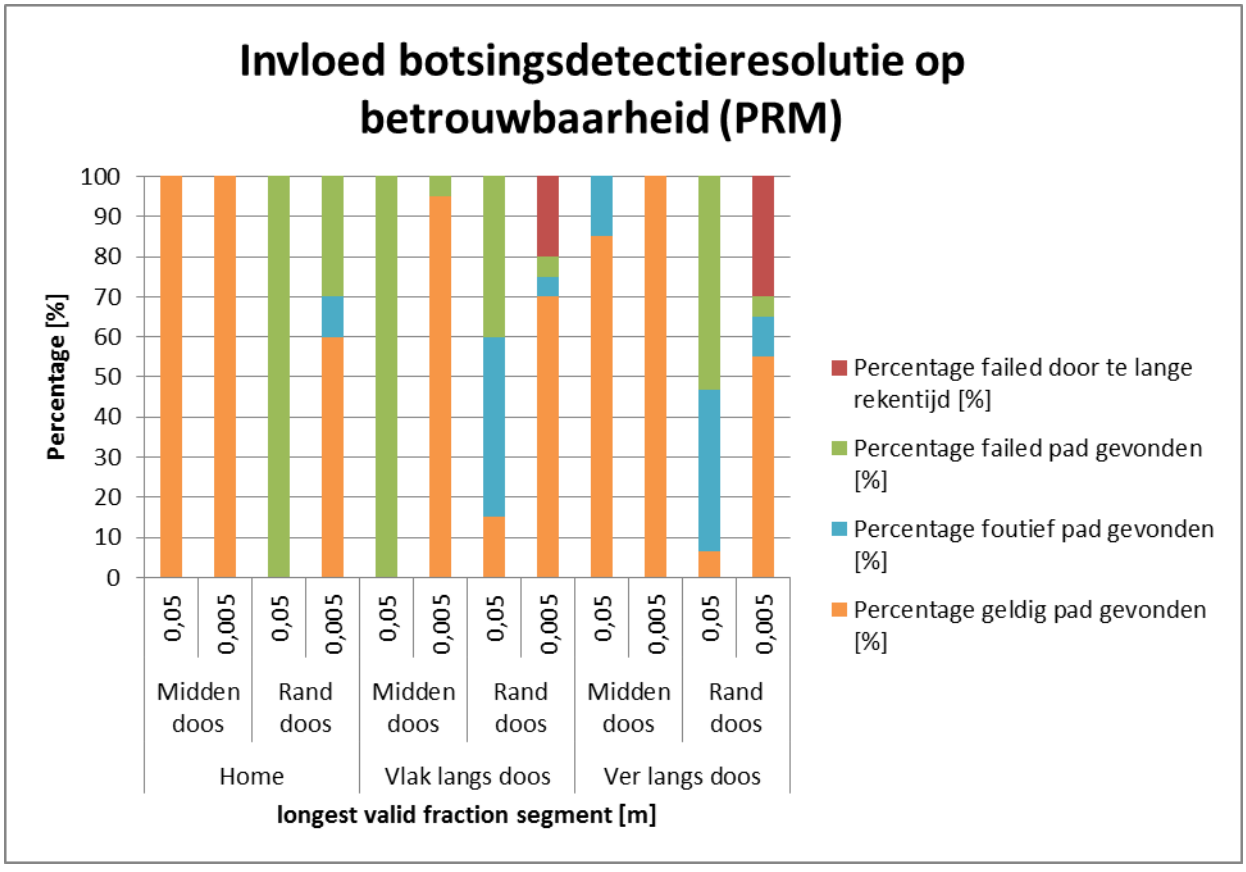
Figuur 63: Ver langs doos naar midden doos



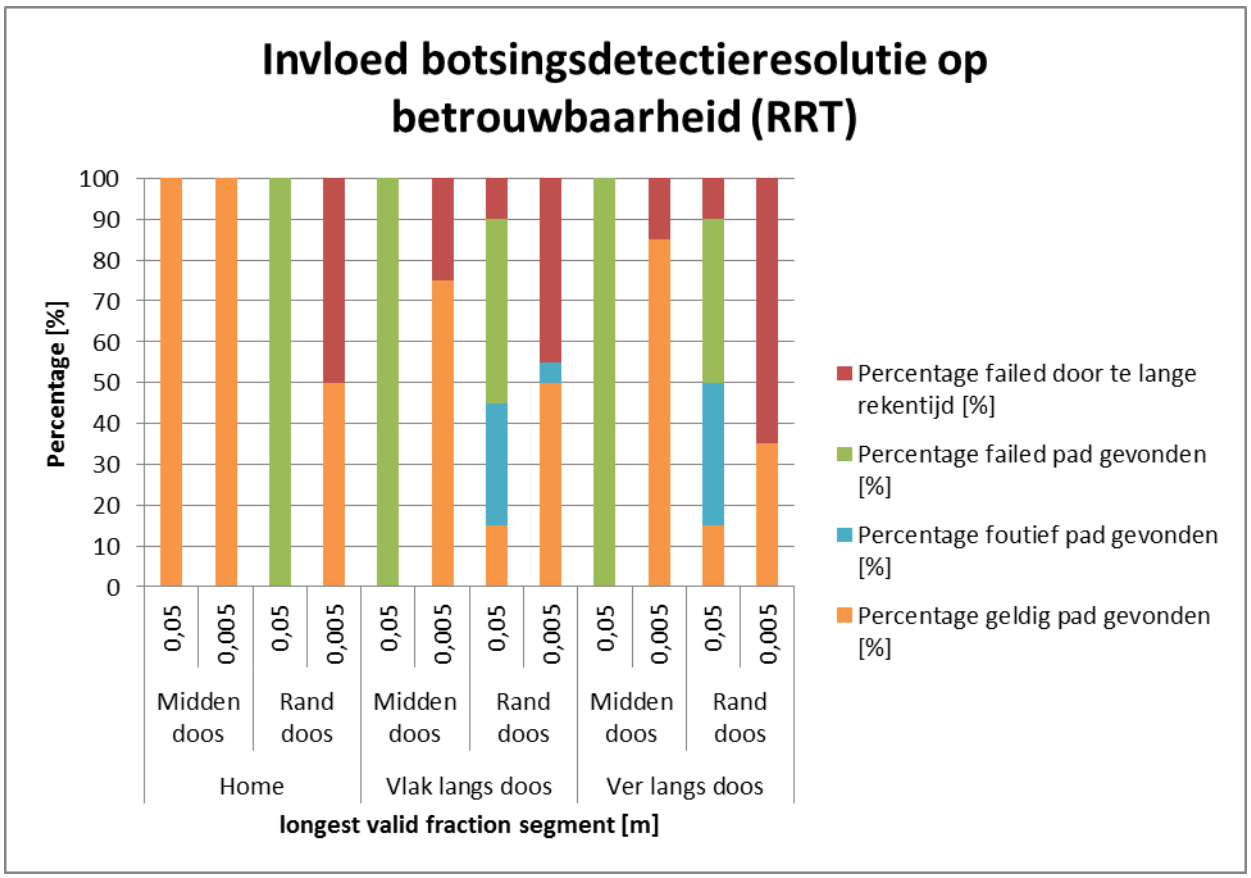
Figuur 64: Ver langs doos naar rand doos

De resultaten van het experiment zijn voor PRM terug te vinden in tabel 1, achteraan dit hoofdstuk ('6.1.2 Wanddikte doos'). Hetzelfde experiment is uitgevoerd met RRT, deze resultaten staan in tabel 2 achteraan dit hoofdstuk ('6.1.2 Wanddikte doos') weergegeven.

Om een beter zicht te krijgen van de invloed van de botsingsdetectieresolutie op een algoritme, zijn de metingen in enkele grafieken gezet. Dit experiment onderzoekt de verandering in betrouwbaarheid van een algoritme indien de botsingsdetectieresolutie varieert. Dit experiment is getest voor verschillende *query's* en voor iedere *query* zijn de metingen twee keer uitgevoerd. Een eerste keer met een lage botsingsdetectieresolutie (*longest valid segment fraction* = 0,05m) en een tweede keer met een hoge botsingsdetectieresolutie (*longest valid segment fraction* = 0,005m). Zoals in 'Hoofdstuk 6: Resultaten' vermeld, gebeurt het beoordelen van de betrouwbaarheid van een algoritme op basis van enkele parameters. Deze parameters staan ook in de legende vermeld. Figuur 65 illustreert de resultaten die bekomen zijn met het PRM-algoritme. Figuur 66 geeft de resultaten van hetzelfde experiment weer, maar deze keer is het uitgevoerd met het RRT-algoritme.

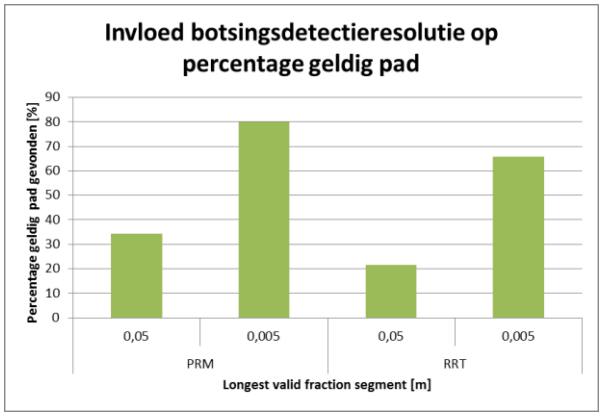


Figuur 65: Invloed botsingsdetectieresolutie op betrouwbaarheid PRM

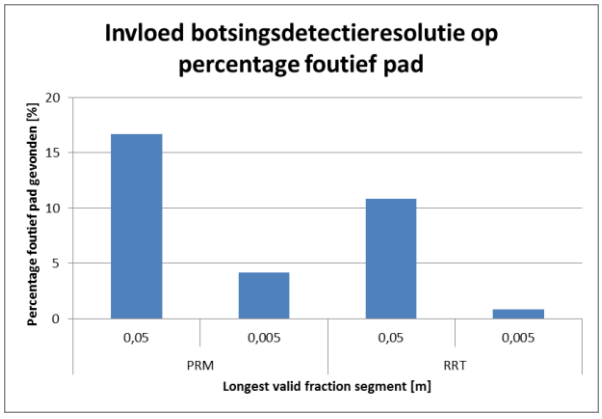


Figuur 66: Invloed botsingsdetectieresolutie op betrouwbaarheid RRT

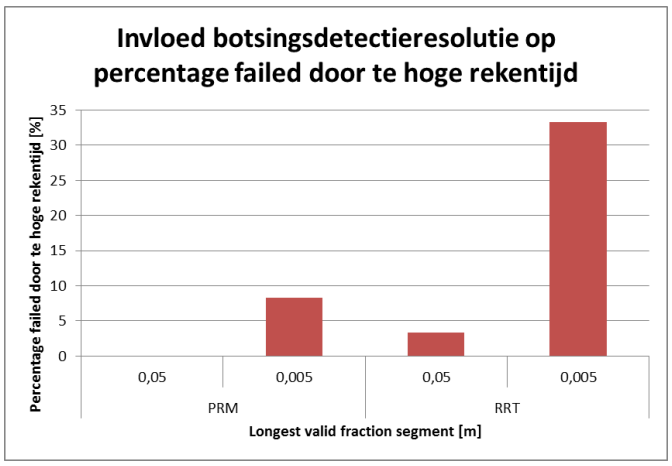
De grafieken illustreren dat de betrouwbaarheid van een algoritme hoger ligt indien de botsingsdetectieresolutie hoger is (of het *longest valid segment fraction* lager). Bij een hogere botsingsdetectieresolutie stijgt het percentage van gevonden paden die geldig zijn sterk (figuur 67) en daalt het percentage van gevonden paden die foutief zijn significant (figuur 68). Het percentage dat het algoritme *failed* geeft als resultaat, door een te hoge rekentijd, stijgt daarentegen, wat een negatief gevolg is (figuur 69).



Figuur 67: Botsingsdetectieresolutie i.f.v. percentage geldig pad gevonden



Figuur 68: Botsingsdetectieresolutie i.f.v. percentage foutief pad gevonden



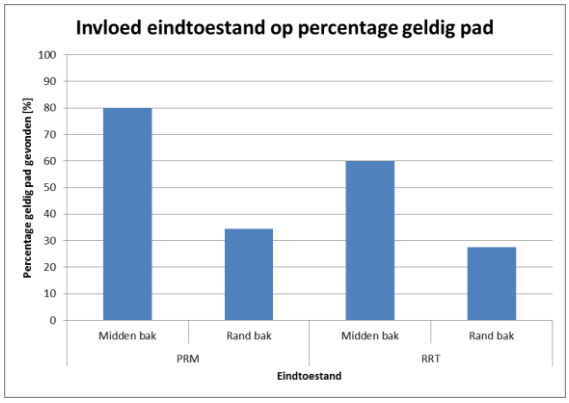
Figuur 69: Botsingsdetectieresolutie i.f.v. percentage failed als resultaat door te hoge rekentijd

Uit figuur 67 is te besluiten dat het percentage dat het algoritme een geldig pad vindt sterk stijgt als de botsingsdetectieresolutie verhoogt (of *longst valid fraction segment*) daalt. Het percentage, dat het algoritme een geldig pad vindt, is bij PRM met een factor 2,3 gestegen en bij RRT met een factor 3. In figuur 68 is te zien dat het percentage van foutief gevonden paden sterk daalt wat, een zeer groot en belangrijk voordeel is. Bij PRM verlaagt dit percentage met en factor 4 en bij RRT zelfs met een factor 13. Figuur 69 demonstreert echter een nadelig gevolg van het verhogen van de botsingsdetectieresolutie. Het komt namelijk vaker voor dat het algoritme geen pad kan vinden wegens een te hoge rekentijd. Bij PRM stijgt het percentage namelijk van 0 naar 8 en bij RRT stijgt het percentage van 3 naar 33. De voordelen van een verhoogde botsingsdetectieresolutie heffen dit nadeel op aangezien het erger is om een foutief pad te hebben dan een *failed* pad wegens een te hoge rekentijd. De botsingsdetectieresolutie heeft daarom een positieve invloed op de betrouwbaarheid van een pad.

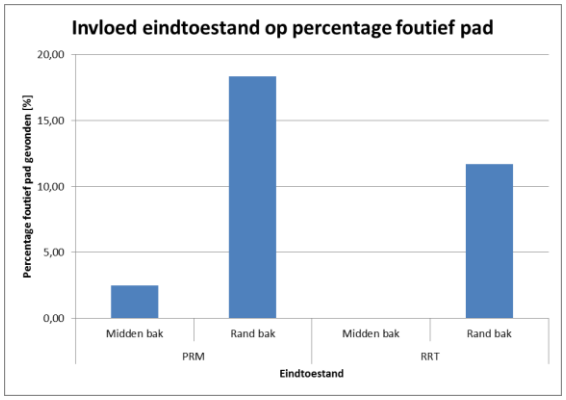
De botsingsdetectieresolutie of het *longest valid segment fraction* definieert, zoals eerder vermeld in dit hoofdstuk, de afstand tussen de deel*vertices* als de botsingsdetectie een verbinding gaat controleren op zijn geldigheid (‘2.2.1 Geldigheid van *vertices* en *edges*’). Het percentage foutief gevonden paden daalt omdat het algoritme de wand van de bak nu vaker registreert als obstakel. Als het *longest valid segment fraction* een waarde van 0,05m heeft en de dikte van de wanden overeenkomt met 0,005m, bestaat de kans dat de botsingsdetectie geen deeltoestand heeft liggen in de bakwand. Het kan namelijk voorvallen dat wanneer een verbinding, die door de bakwand loopt, gecontroleerd wordt op zijn geldigheid, dat er een deeltoestand op 0,0225m van ieder uiteinde van de wand ligt. Daardoor ligt er geen deeltoestand in het obstakel waardoor de botsingsdetectie de verbinding als geldig beschouwt. Dat is de reden waarom een algoritme een foutief pad als resultaat terugkrijgt. Indien het *longest valid segment fraction* echter op de waarde 0,005m staat en de wanddikte ook, dan is de kans zeer klein dat er een foutief pad optreedt. Dat komt doordat een verbinding, die door een obstakel loopt, opgedeeld wordt in deeltoestanden met een maximale afstand van 0,005m tussen de deeltoestanden. Hierdoor zal er altijd een deeltoestand binnen het obstakel liggen wat ervoor zorgt dat de verbinding ongeldig is.

De reden waarom het percentage, *failed* door een te hoge rekentijd, stijgt is omwille van het feit dat de botsingsdetectie 10 keer zo veel deeltoestanden moet controleren op hun geldigheid. Het controleren op de geldigheid van een verbinding duurt daardoor zeer lang wat deze stijging verklaart.

In figuren 65 en 66 valt ook op dat padplanningsalgoritmen het moeilijker hebben om werkstukken aan de rand van de bak te grijpen. Dit is te zien aan het percentage geldige paden dat gevonden is door het algoritme. Dat percentage ligt namelijk beduidend hoger bij de *query's* die als eindtoestand het midden van de bak hebben (figuur 70). Ook ligt het percentage van foutief gevonden paden veel lager bij de resultaten die als eindtoestand het midden van de bak hebben (figuur 71). Uit het onderzoek is dus ook te concluderen dat werkstukken aan de rand van de bak moeilijker te grijpen zijn. Dat resultaat is te verklaren door het feit dat de eindeffector meer ruimte heeft in het midden van de doos dan aan de rand van de doos.



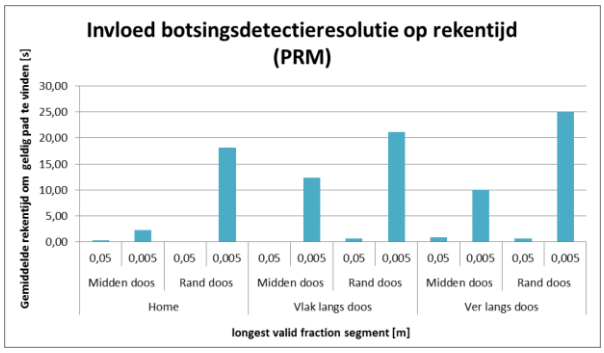
Figuur 70: Eindtoestand i.f.v. percentage geldig pad gevonden



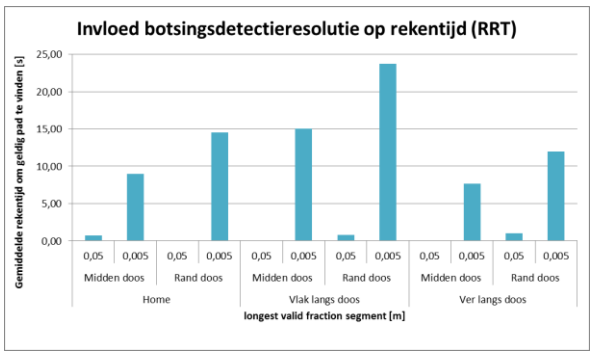
Figuur 71: Eindtoestand i.f.v. percentage foutief pad gevonden

Figuren 65 en 66 illustreren ook dat de betrouwbaarheid van een algoritme afhangt van de begintoestand. Aangezien er enorm veel begintoestanden mogelijk zijn, is het niet relevant om deze in detail met elkaar te vergelijken. Er valt wel op dat *home* als begintoestand het meest betrouwbare resultaat oplevert. Dat komt doordat de robot van de *home*toestand tot de doos geen obstakel rechtstreeks tegenkomt. De andere twee begintoestanden zijn significant minder betrouwbaar, maar onderling is er geen beduidend verschil te zien.

De volgende figuren illustreren de invloed van de botsingsdetectieresolutie op de rekentijd. In deze grafiek staan enkel de rekentijden van een geldig pad en niet van foutieve of '*failed*' paden aangezien deze de resultaten negatief zouden beïnvloeden. Het is namelijk belangrijk om te weten hoeveel tijd een algoritme gemiddeld nodig heeft om een geldig pad te berekenen. De resultaten van de betrouwbaarheid van het algoritme houden immers al rekening met de paden die foutief of *failed* zijn. Figuur 72 toont voor verschillende *query's* de invloed van de botsingsdetectieresolutie op de rekentijd van het algoritme bij het gebruik van PRM. Figuur 73 illustreert hetzelfde experiment uitgevoerd met het RRT-algoritme.



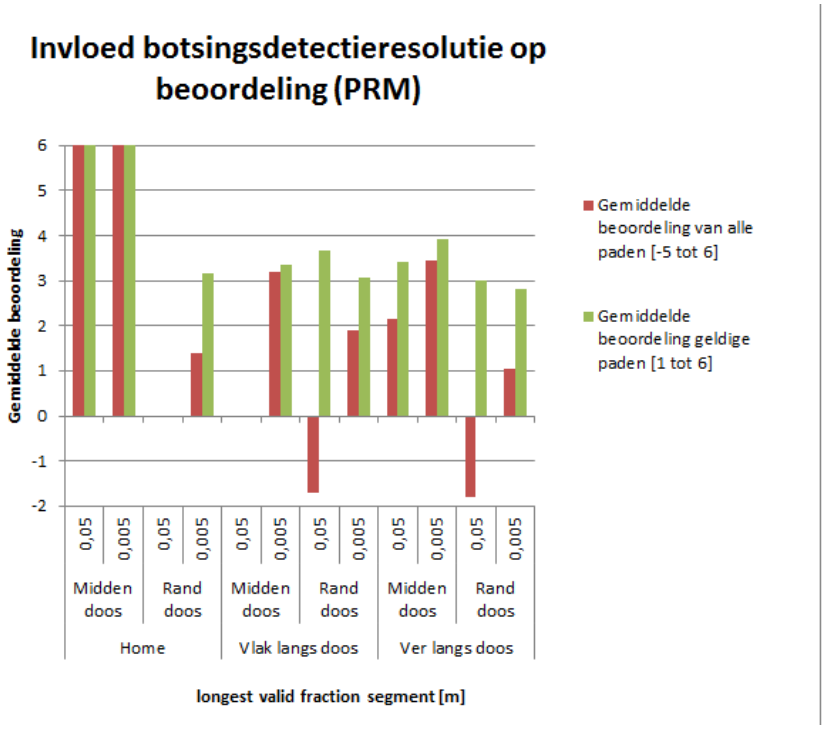
Figuur 72: Botsingsdetectieresolutie i.f.v. gemiddelde rekentijd om geldig pad te vinden voor PRM



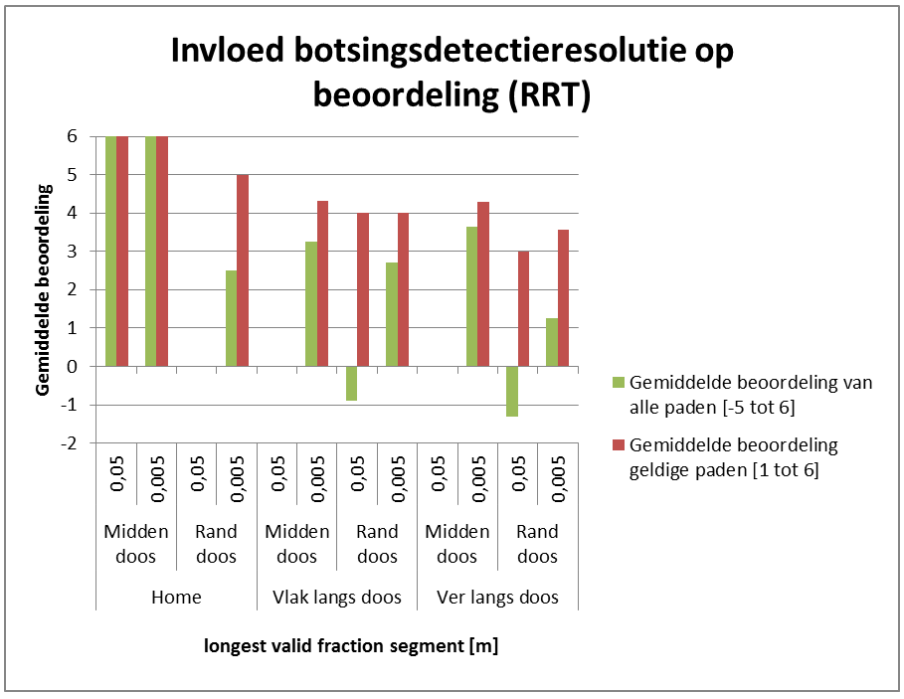
Figuur 73: Botsingsdetectieresolutie i.f.v. gemiddelde rekentijd om geldig pad te vinden voor RRT

Uit figuren 72 en 73 valt meteen op dat een hogere botsingsdetectieresolutie veel meer tijd vraagt. De gemiddelde rekentijd om een geldig pad te vinden ligt bij beide algoritmen voor iedere *query* beduidend hoger. De verklaring hiervan is analoog als de verklaring van de stijging van het percentage *failed* door een te hoge rekentijd. De botsingsdetectie moet immers veel meer deeltoestanden testen waardoor de rekentijd hoger ligt.

De laatste twee figuren van het experiment beschouwen de invloed van de botsingsdetectieresolutie op de vloeiendheid van het pad. Zoals eerder vermeld (‘Hoofdstuk 6: Resultaten’: beoordeling vloeiendheid van een pad) wordt er aan ieder pad een waarde gegeven op basis van zijn benadering van het optimale pad. Deze beoordeling is één keer uitgevoerd op alle paden, dus inclusief de foutieve en *failed* paden, en één keer is het gemiddelde genomen van de geldige paden. Hierdoor is het ook mogelijk om conclusies te trekken uit de vloeiendheid van enkel de geldige paden. In figuur 74 staan de resultaten van de vloeiendheid van een pad visueel opgesteld voor verschillende *query’s* en een veranderlijke botsingsdetectieresolutie uitgevoerd met PRM. Figuur 75 toont dezelfde parameters, maar hier is het experiment uitgevoerd met RRT.



Figuur 74: Botsingsdetectieresolutie i.f.v. gemiddelde beoordeling bij PRM



Figuur 75: Botsingsdetectieresolutie i.f.v. de gemiddelde beoordeling bij RRT

Een opmerking bij figuren 74 en 75 is dat de rode kolommen altijd even hoog of hoger liggen dan de groene kolommen aangezien de rode kolommen enkel het gemiddelde neemt van de geldige paden. Dat wil zeggen dat hier de waarden 0, van een *failed* pad en een *failed* pad wegens een te hoge rekentijd, en -5, van een foutief pad, niet aan worden toegevoegd. Uit de figuren 74 en 75 valt op dat de gemiddelde beoordeling bij een hogere botsingsdetectie ook hoger ligt. Bij

sommige *query's* hebben beide kolommen een waarde 0. Deze waarde komt voor indien alle metingen van die situatie *failed* waren.

Uit dit experiment zijn dus enkele belangrijke conclusies te trekken. Een hogere botsingsdetectieresolutie of een lagere *longest valid segment fraction* zorgt namelijk voor:

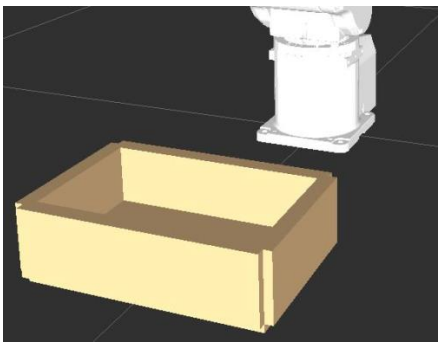
- een hogere betrouwbaarheid,
- een hogere rekentijd,
- een hogere beoordeling.

Vooral de hogere betrouwbaarheid, met de sterke daling van het percentage foutieve paden, is een zeer sterk voordeel en een belangrijke conclusie. Aangezien een hoge botsingsdetectieresolutie (*longest valid segment fraction* = 0,005m) overeenkomt met een betere betrouwbaarheid, wordt de waarde in de volgende onderzoeken van dit hoofdstuk altijd gelijk genomen aan 0,005m wat overeenkomt met de dikte van de bak zijn wanden.

6.1.2 Bak met dikke wanden

In '6.1.1 Botsingsdetectieresolutie' is geconcludeerd dat de botsingsdetectieresolutie een grote invloed heeft op de performantie van een algoritme. Een lagere waarde voor het *longest valid segment fraction* komt overeen met een hogere botsingsdetectieresolutie. Zoals eerder vermeld is de kans groter dat een deel*vertex,* van een verbinding die gecontroleerd wordt op zijn geldigheid, binnen een obstakel valt indien de botsingsdetectieresolutie hoger is (of het *longest valid segment fraction* lager). Wat gebeurt er dan als de wanddikte van de bak groter is? Er zou dan ook een verhoogde kans moeten zijn dat een deel*vertex*, van een verbinding die gecontroleerd wordt op zijn geldigheid, binnen het obstakel valt aangezien het obstakel dikker is.

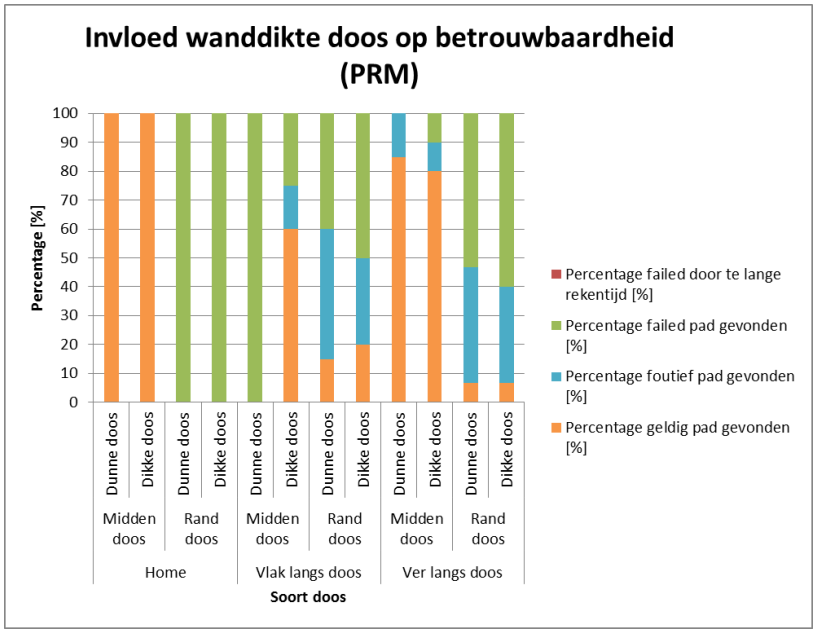
Om de invloed van de wanddikte van een bak op de performantie van een algoritme te testen, zijn er net als bij de botsingsdetectieresolutie enkele experimenten uitgevoerd. De experimenten zijn op dezelfde *query's* uitgevoerd als bij de botsingsdetectieresolutie om resultaten te krijgen die met elkaar te vergelijken zijn. De botsingsdetectieresolutie is ingesteld op 0,05m zoals dat standaard het geval is. De wanddikte van de bak is aangepast van 5mm naar 25mm om de invloed ervan te onderzoeken (figuur 76).



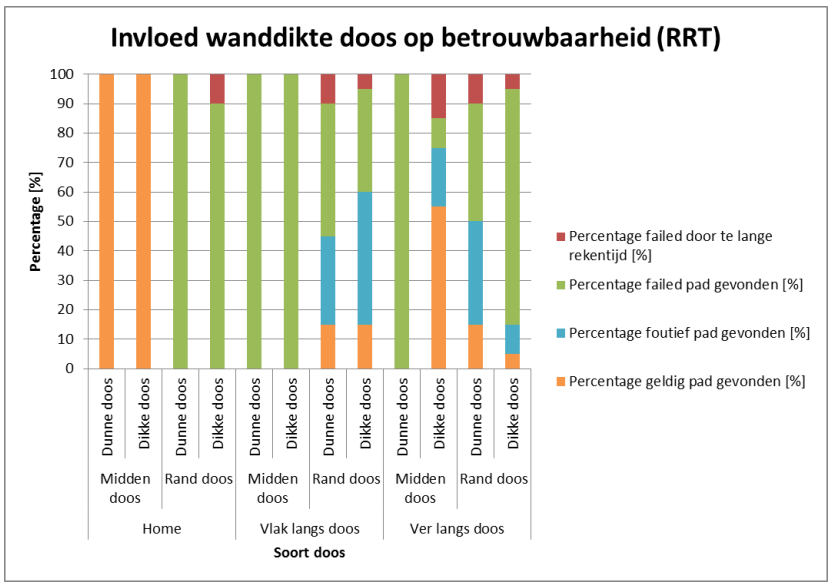
Figuur 76: Aangepaste bak met wanddikte 25mm

De resultaten van het experiment zijn terug te vinden op het einde van dit deel in tabel 1 voor PRM. Hetzelfde experiment is ook uitgevoerd met het RRT-algoritme, deze resultaten zijn in tabel 2 terug te vinden op het einde van dit deel.

De belangrijkste resultaten van de metingen zijn in enkele grafieken weergegeven om de verschillen visueel voor te stellen. Figuur 77 illustreert de invloed van de wanddikte van de doos op de betrouwbaarheid van PRM. Figuur 78 geeft de resultaten van hetzelfde experiment weer, maar deze keer uitgevoerd met RRT.

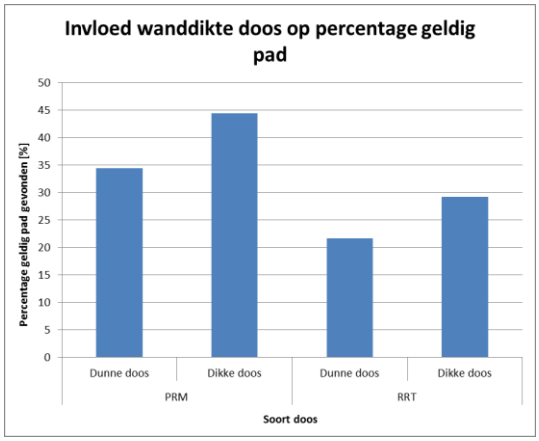


Figuur 77: Wanddikte doos i.f.v. betrouwbaarheid PRM

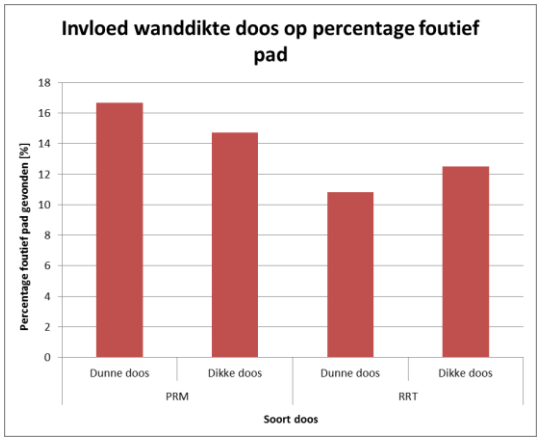


Figuur 78: Wanddikte doos i.f.v. betrouwbaarheid RRT

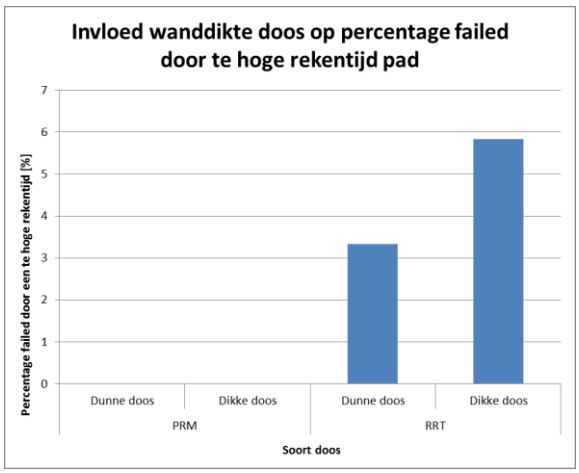
Net zoals bij een verhoogde botsingsdetectieresolutie is er bij een hogere wanddikte vaker een geldig pad gevonden waardoor de betrouwbaarheid stijgt. De verschillen tussen een dunne en een dikke doos is echter niet heel groot. Onderstaande figuren maken de verschillen duidelijker. Figuur 79 toont de invloed van de wanddikte op de geldigheid van een pad. Figuur 80 illustreert de invloed van de wanddikte op het percentage dat een algoritme een foutief pad vindt. Figuur 81 beeld de invloed van de wanddikte af op het percentage dat een *failed* pad is gevonden door een te hoge rekentijd.



Figuur 79: Wanddikte doos i.f.v. percentage foutief pad gevonden



Figuur 80: Wanddikte doos i.f.v. percentage foutief pad gevonden

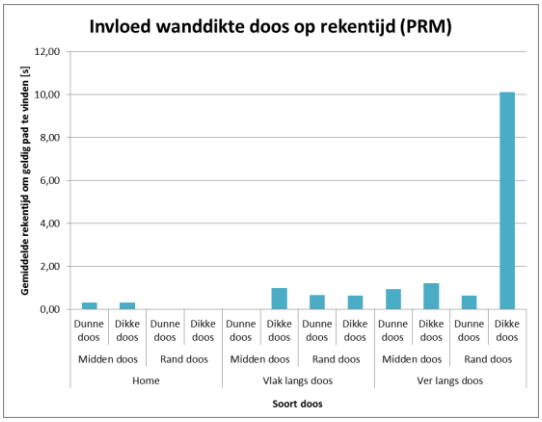


Figuur 81: Wanddikte doos i.f.v. percentage failed pad gevonden wegens te hoge rekentijd

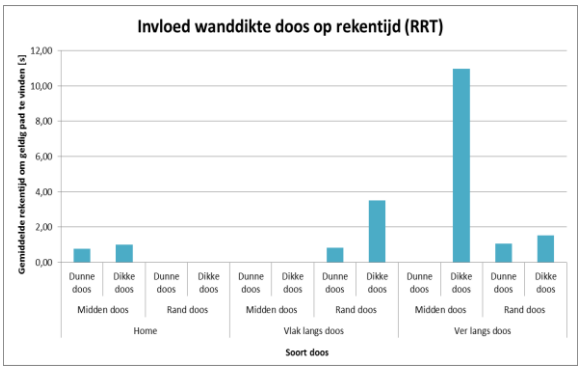
Uit bovenstaande grafieken is ongeveer dezelfde tendens te zien als bij de botsingsdetectieresolutie. De betrouwbaarheid stijgt indien de wanddikte van de doos verhoogt aangezien het percentage dat de algoritmen een geldig pad vinden stijgt. De verschillen zijn echter niet zo significant als bij een verhoging van de botsingsdetectieresolutie. Figuur 79 toont aan dat indien de wanddikte van de doos hoger is, algoritmen vaker een geldig pad vinden. Voor PRM als RRT stijgt dit percentage met een factor 1,3. Er is dus een stijging waarneembaar, maar een sterke stijging is het niet. In figuur 80 is te zien dat bij PRM het percentage van een foutief pad daalt als de wanddikte stijgt, namelijk met een factor 1,1. Bij RRT stijgt het percentage echter als de wanddikte stijgt met een factor 1,2 wat een onverwacht resultaat is. Er is in ieder geval geen beduidend verschil te zien voor het percentage foutief gevonden paden indien de wanddikte verandert. Theoretisch gezien zou het percentage echter moeten zakken, maar dat is hier niet het geval. In figuur 81 is te zien dat het percentage *failed* wegens een te hoge rekentijd bij PRM 0 blijft terwijl bij RRT het percentage 2,5% stijgt. Dit is echter geen significant verschil waardoor hier geen conclusie uit getrokken wordt.

De reden waarom de betrouwbaarheid dezelfde tendens volgt als de botsingsdetectieresolutie is om het feit dat ze er beiden proberen voor te zorgen dat de kans verhoogt dat een deeltoestand in het obstakel valt. Hierdoor stijgt het percentage dat een geldig pad wordt gevonden en daalt het percentage dat een foutief pad gevonden wordt. Bij dit experiment is de wand echter maar half zo dik als de lengte tussen twee deeltoestanden die gecontroleerd worden op hun geldigheid (‘2.2.1 Geldigheid *vertices* en *edges*’). Hierdoor kan het nog altijd voorvallen dat een deeltoestand niet binnen het obstakel ligt, wat verklaart waarom de tendens niet zo fel als bij de botsingsdetectieresolutie. De dikte van de wand is niet gelijk aan 0,05m genomen aangezien in de praktijk de bakwanden niet aangepast kunnen worden. Daarom is hier enkel besproken wat de invloed van de wanddikte is en is niet de optimale waarde voor de wanddikte gekozen. De reden waarom *failed* wegens een te lange rekentijd niet stijgt zoals bij de botsingsdetectieresolutie is logisch aangezien er geen verandering is aangebracht in de botsingsdetectieresolutie. Het berekenen van een pad gebeurt met andere woorden volledig analoog indien enkel de wanddikte van de bak verandert.

De volgende twee grafieken illustreren de invloed van de wanddikte op de rekentijd. In deze grafiek staan enkel de rekentijden van een geldig pad en niet van foutieve of '*failed*' paden aangezien deze de resultaten negatief zouden beïnvloeden. Het is namelijk belangrijk om te weten hoeveel tijd een algoritme gemiddeld nodig heeft om een geldig pad te berekenen. De resultaten van de betrouwbaarheid van het algoritme houden immers al rekening met de paden die foutief of *failed* zijn. Figuur 82 toont voor verschillende *query's* de invloed van de wanddikte van de bak op de rekentijd van PRM. Figuur 83 illustreert hetzelfde experiment uitgevoerd met RRT.



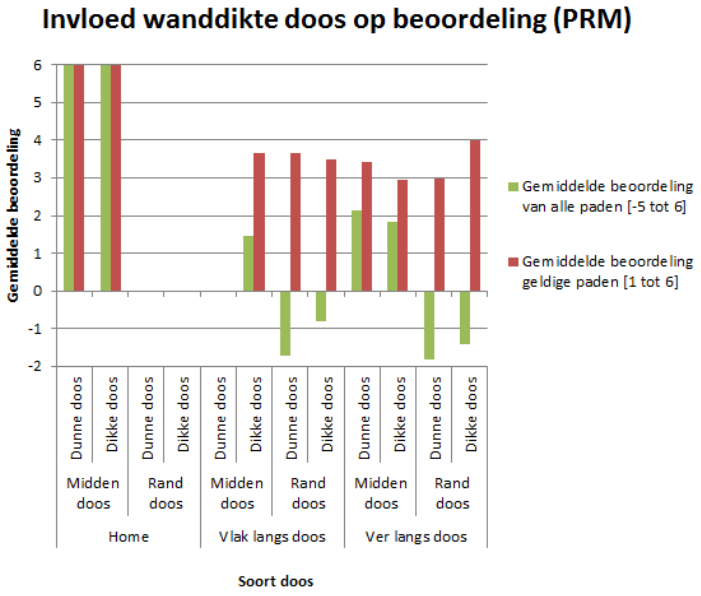
Figuur 83: Wanddikte doos i.f.v. de gemiddelde rekentijd om een geldig pad te vinden bij PRM



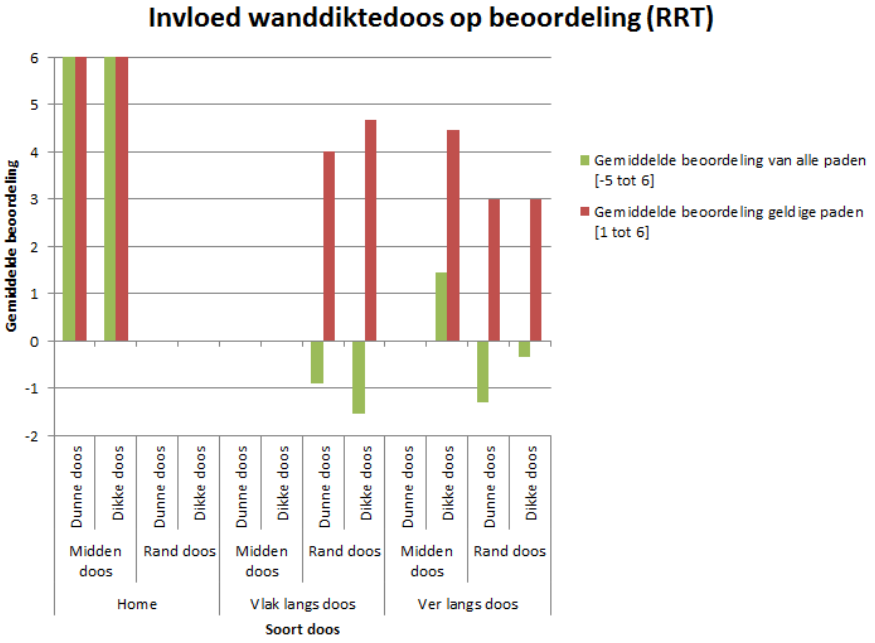
Figuur 82: Wanddikte doos i.f.v. de gemiddelde rekentijd om een geldig pad te vinden bij RRT

Zoals verwacht heeft het veranderen van de wanddikte van de doos weinig invloed op de rekentijd om een geldig pad te vinden. PRM bevat één *query* waar een stijging te zien is en RRT bevat twee *query's* waar een duidelijke stijging te zien is. Dat komt doordat deze *query's* maar een zeer beperkt percentage aan geldige paden hebben zoals in figuren 77 en 78 te zien is. Hierdoor zijn de waarden niet representatief genoeg om zinvol te zijn. De resultaten van deze *query's* staan ook, zoals eerder vermeld, in tabellen 1 en 2 achteraan dit deel van het hoofdstuk. De *query* 'ver langs doos tot rand doos' heeft bij een dikke doos bij PRM immers maar een percentage van 6,67 aan geldige paden. Bij RRT heeft de *query* 'vlak langs doos tot rand doos' bij een dikke doos een percentage van 15 aan geldige paden en heeft de *query* 'ver langs doos tot midden doos' bij een dunne doos een percentage van 0. Hierdoor zijn deze resultaten niet zinvol. De conclusie is dus dat de wanddikte geen invloed heeft op de gemiddelde rekentijd die nodig is om een geldig pad te vinden. De verklaring hiervoor is dezelfde als die van de invloed van de wanddikte op het percentage *failed* wegens een te hoge rekentijd.

De laatste twee figuren van het experiment beschouwen de invloed van de wanddikte op de vloeiendheid van het pad. De beoordeling is één keer uitgevoerd op alle paden, dus inclusief de foutieve en *failed* paden, en één keer is het gemiddelde genomen van de geldige paden. Figuur 84 illustreert de resultaten van de vloeiendheid van een pad voor verschillende *query's* en een variërende wanddikte, uitgevoerd met PRM. Figuur 85 toont dezelfde parameters, maar het experiment is hier uitgevoerd met het RRT-algoritme.



Figuur 84: Wanddikte doos i.f.v. gemiddelde beoordeling pad bij PRM



Figuur 85: Wanddikte doos i.f.v. gemiddelde beoordeling pad bij RRT

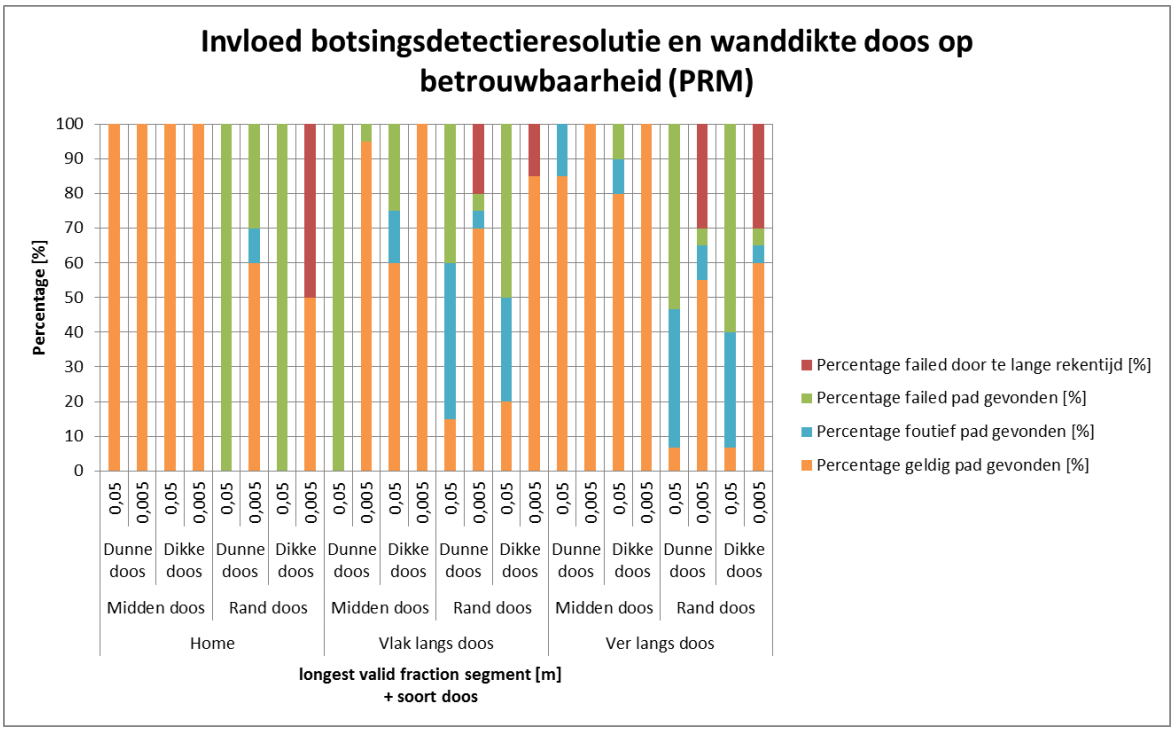
Bovenstaande grafieken tonen aan dat er geen significant verschil is in de vloeiendheid van de paden indien de wanddikte van een doos varieert. Er is bij één *query* een groot verschil bij PRM en bij één *query* een groot verschil bij RRT in beoordeling. Dit is weer te verklaren door het feit dat het percentage aan geldig gevonden paden bij die eindtoestanden gelijk is aan 0 voor een dunne doos. Uit deze resultaten is dus te besluiten dat de wanddikte geen invloed heeft op de vloeiendheid van een pad.

De onderzoeken van dit hoofdstuk tonen aan dat een hogere botsingsdetectieresolutie en een hogere wanddikte van de bak overeenkomt met een hogere betrouwbaarheid. Hierdoor is de performantie van het algoritme beter, ondanks dat de rekentijd stijgt bij een verhoogde botsingsdetectieresolutie. De eis van een lage rekentijd is echter niet zo belangrijk als de eis van een hoge betrouwbaarheid, waardoor dit nadeel niet zwaar doorweegt in de beoordeling. Het belangrijkste is dat het percentage van foutieve paden laag ligt.

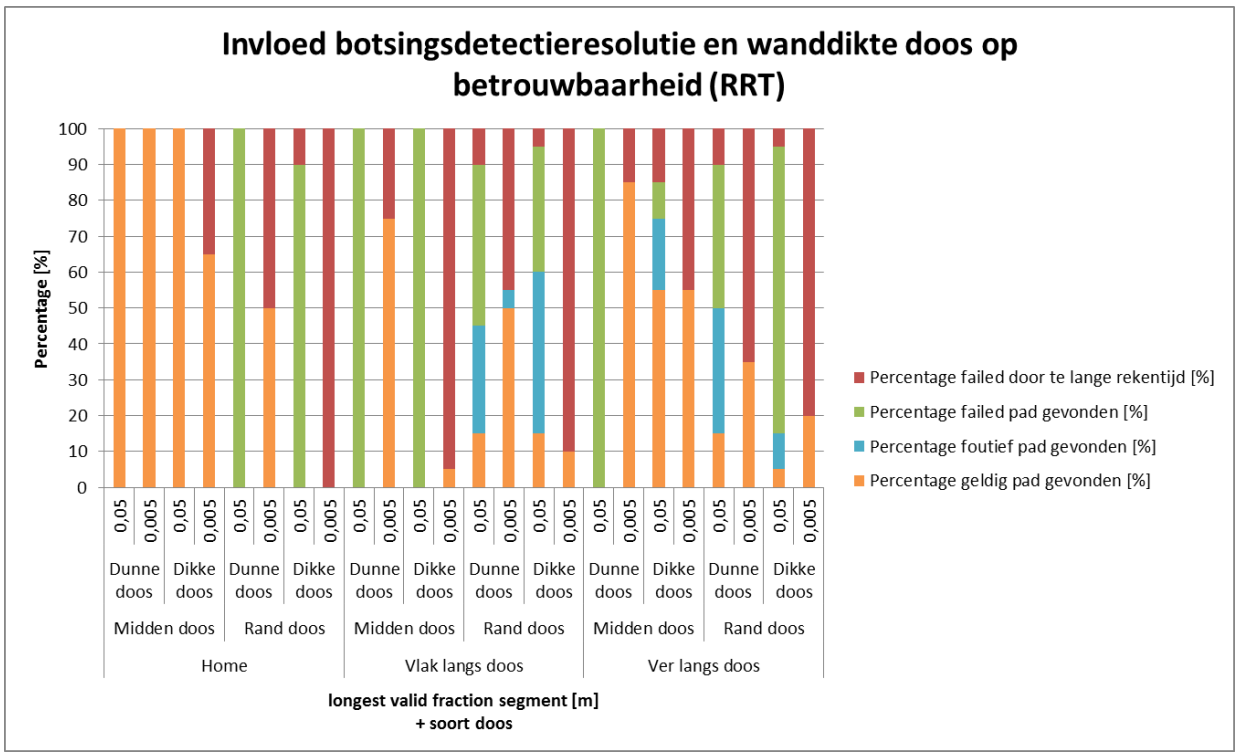
Maar welke aanpassing van de twee geeft de beste resultaten, en wat gebeurt er indien beide aanpassingen gecombineerd worden? Om dit te onderzoeken is er bij dezelfde *query's* een experiment uitgevoerd met als instellingen een *longest valid segment fraction* van 0,005m en een bak met dikke wanden. Deze resultaten worden vergeleken met alle andere resultaten die al bekomen zijn. Deze reeds bekomen resultaten hebben als instellingen een:

- *longest valid segment fraction* van 0,05m en een bak met dunne wanden;
- *longest valid segment fraction* van 0,005m en een bak met dunne wanden;
- *longest valid segment fraction* van 0,05m en een bak met dikke wanden.

De bekomen resultaten staan in tabel 1, voor de metingen uitgevoerd met PRM, en in tabel 2, voor de metingen uitgevoerd met RRT. Om de verschillen duidelijker te maken zijn van deze resultaten ook enkele grafieken gemaakt die hierna besproken zijn. Figuur 86 illustreert de invloed van zowel de wanddikte van de doos als de botsingsdetectieresolutie op de betrouwbaarheid van PRM. Figuur 87 geeft de resultaten weer van hetzelfde experiment, maar deze keer uitgevoerd met RRT.

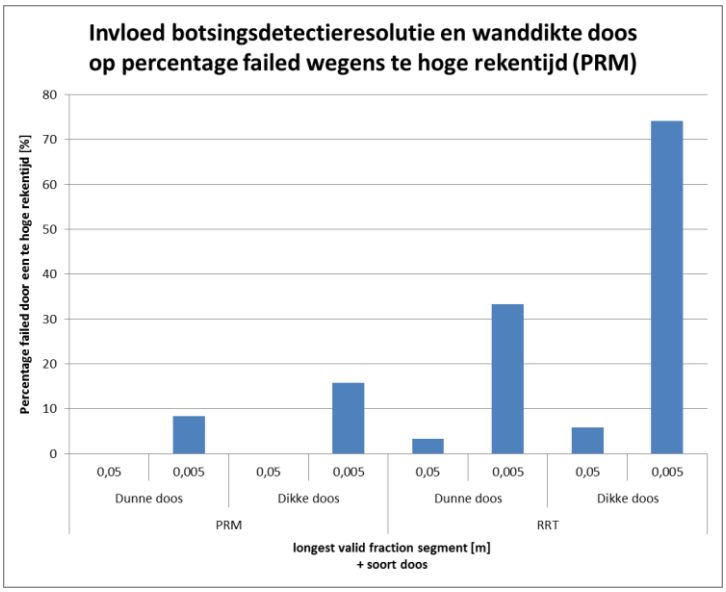


Figuur 86: Botsingsdetectieresolutie en wanddikte doos i.f.v. betrouwbaarheid pad bij PRM



Figuur 87: Botsingsdetectieresolutie en wanddikte doos i.f.v. de betrouwbaarheid van een pad bij RRT

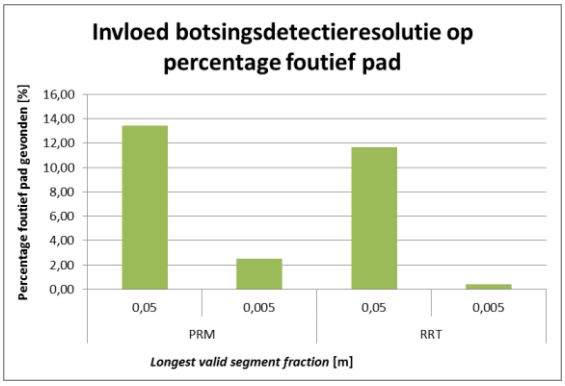
Figuren 86 en 87 bevatten alle resultaten van tabellen 1 en 2 die nodig zijn om de invloed, van de botsingsdetectieresolutie en de wanddikte van de doos, op de betrouwbaarheid te evalueren. Het eerste wat opvalt in de grafieken is dat er een veel groter gedeelte van de kolommen rood is. Rood komt overeen met het percentage *failed* door een te lange rekentijd. Dat komt vooral voor bij de situaties die gebruikmaken van zowel een hoge botsingsdetectieresolutie als een bak met dikke wanden. Deze vaststelling is nog eens uitvergroot door middel van figuur 88.



Figuur 88: Botsingsdetectieresolutie en wanddikte doos i.f.v. gemiddeld percentage failed wegens te hoge rekentijd

Uit figuur 88 is duidelijk af te leiden dat het percentage *failed* wegens een te hoge rekentijd drastisch stijgt indien de situatie zowel een bak met dikke wanden als een hoge botsingsdetectieresolutie bevat. Indien de botsingsdetectieresolutie hoog staat ingesteld (*longest valid segment fraction*) en de wanddikte van de doos verandert van dunne wanden naar dikke wanden, dan verdubbelt het percentage van *failed* paden wegens een te hoge rekentijd. Indien de wanddikte op dikke wanden staat ingesteld en de botsingsdetectieresolutie verandert van laag naar hoog, dan verhoogt het percentage van *failed* paden wegens een te hoge rekentijd enorm. Uit dit experiment is te concluderen dat het absoluut niet voordelig is om het *longest valid segment fraction* lager van waarde in te stellen dan de wanddikte van de doos aangezien dit de rekentijd onnodig verhoogt. Het gemiddelde percentage van *failed* paden wegens een te hoge rekentijd is bij RRT zelfs 74% wat een zeer extreme waarde is. Het enige voordeel, dat optreedt bij het combineren van een hoge wanddikte met een hoge botsingsdetectieresolutie, is dat het gemiddelde percentage van foutieve paden zeer laag is. Maar het percentage van foutieve paden is immers al zeer laag indien het *longest valid segment fraction* dezelfde waarde aanneemt als de wanddikte van de bak. Het nemen van een *longest valid segment fraction* waarde die lager ligt dan de wanddikte van de bak is dus uitgesloten. Zoals eerder al vermeld kan in de praktijk de wanddikte van de doos niet aangepast worden. Enkel het aanpassen van de botsingsdetectieresolutie is dan nog mogelijk.

Uit het onderzoek blijkt dat het optimaal is om de botsingsdetectieresolutie af te stemmen op de wanddikte. Dat wil zeggen dat indien er een bak is met een wanddikte van 0,005m, het *longest valid segment fraction* ook best ingesteld staat op 0,005m. Door dit uit te voeren blijft de hoeveelheid van foutieve paden zeer beperkt (figuur 89).



Figuur 89: Botsingsdetectieresolutie i.f.v. percentage foutief pad gevonden

In figuur 89 is een beduidend verschil te zien. Bij een hogere botsingsdetectieresolutie (lagere *longest valid segment fraction*) berekenen de algoritme beduidend minder foutieve paden wat het grootste voordeel is aan het afstemmen van de botsingsdetectieresolutie op de wanddikte van de bak. Uit de vorige experimenten was ook al geconcludeerd dat het afstemmen van de botsingsdetectieresolutie op de wanddikte zorgt voor een hogere betrouwbaarheid en een hogere rekentijd. Het nadeel van de verhoogde rekentijd weegt echter niet op tegen het voordeel van de verhoogde betrouwbaarheid aangezien een hoge rekentijd niet zo erg is als een foutief pad. Vandaar dat voor de volgende experimenten steeds is gekozen voor een bak met dunne wanden (0,005m) en een *longest valid segment fraction* van 0,005m.

Tabel 1 laat alle meetresultaten zien die ontstaan door de botsingsdetectieresolutie en het soort doos te variëren, uitgevoerd met PRM.

Tabel 1: Invloed botsingsdetectieresolutie en wanddikte bak bij PRM

Positie begintoestand	Positie eindtoestand	Het soort doos	Botsingsdetectie-resolutie	Percentage geldig pad gevonden [%]	Percentage foutief pad gevonden [%]	Percentage failed pad gevonden [%]	Percentage failed door te lange rekentijd [%]	Gemiddelde berekeningstijd voor goed pad [s]	Gemiddeld aantal vertices dat een goed pad bevat	Gemiddelde beoordeling van alle paden [-5 tot 6]	Gemiddelde beoordeling geldige paden [1 tot 6]
Home	Midden doos	Dunne doos	0,05	100	0	0	0	0,32	2	6	6
			0,005	100	0	0	0	2,33	2	6	6
		Dikke doos	0,05	100	0	0	0	0,32	2	6	6
			0,005	100	0	0	0	2,39	2,45	6	6
	Rand doos	Dunne doos	0,05	0	0	100	0	/	/	0	/
			0,005	60	10	30	0	18,19	14,25	1,4	3,17
		Dikke doos	0,05	0	0	100	0	/	/	0	/
			0,005	50	0	0	50	28,91	19,67	1,3	0,82
Vlak langs doos	Midden doos	Dunne doos	0,05	0	0	100	0	/	/	0	/
			0,005	95	0	5	0	12,33	10,21	3,2	3,37
		Dikke doos	0,05	60	15	25	0	1	7,25	1,45	3,67
			0,005	100	0	0	0	15,04	10,73	2,75	3
	Rand doos	Dunne doos	0,05	15	45	40	0	0,67	4,00	-1,7	3,67
			0,005	70	5	5	20	21,14	15,86	1,9	3,07
		Dikke doos	0,05	20	30	50	0	0,65	4	-0,80	3,50
			0,005	85	0	0	15	30,40	19	2,6	3,06
Ver langs doos	Midden doos	Dunne doos	0,05	85	15	0	0	0,94	5,94	2,15	3,41
			0,005	100	0	0	0	10,02	10,73	3,45	3,91
		Dikke doos	0,05	80	10	10	0	1,23	8,31	1,85	2,94
			0,005	100	0	0	0	16,55	14,73	2,8	3
	Rand doos	Dunne doos	0,05	6,67	40	53,33	0	0,64	5	-1,8	3
			0,005	55	10	5	30	24,92	15,91	1,05	2,82
		Dikke doos	0,05	6,67	33,33	60	0	10,12	7,3	-1,4	4
			0,005	60	5	5	30	28,01	18,67	1,7	1,45

Tabel 2 laat alle meetresultaten zien die ontstaan door de botsingsdetectieresolutie en het soort doos te variëren, uitgevoerd met RRT.

Tabel 2: Invloed botsingsdetectieresolutie en wanddikte bak bij RRT

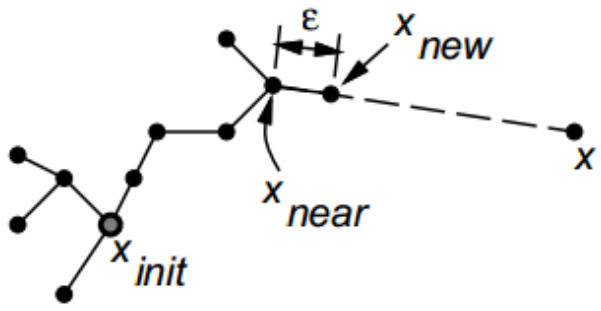
Positie begintoestand	Positie eindtoestand	Het soort doos	Botsingsdetectie-resolutie	Percentage geldig pad gevonden [%]	Percentage foutief pad gevonden [%]	Percentage failed pad gevonden [%]	Percentage failed door te lange rekentijd [%]	Gemiddelde berekeningstijd voor goed pad [s]	Gemiddeld aantal vertices dat een goed pad bevat	Gemiddelde beoordeling van alle paden [-5 tot 6]	Gemiddelde beoordeling geldige paden [1 tot 6]
Home	Midden doos	Dunne doos	0,05	100	0	0	0	0,75	67,45	6	6
			0,005	100	0	0	0	8,99	29,15	6	6
		Dikke doos	0,05	100	0	0	0	0,99	90,05	6	6
			0,005	65	0	0	35	8,18	79,62	3,9	6
	Rand doos	Dunne doos	0,05	0	0	100	0	/	/	0	/
			0,005	50	0	0	50	14,50	47,10	2,5	5
		Dikke doos	0,05	0	0	90	10	/	/	0	/
			0,005	0	0	0	100	/	/	0	/
Vlak langs doos	Midden doos	Dunne doos	0,05	0	0	100	0	/	/	0	/
			0,005	75	0	0	25	15,02	54,40	3,25	4,33
		Dikke doos	0,05	0	0	100	0	/	/	0	/
			0,005	5	0	0	95	1,09	11,00	0,3	6
	Rand doos	Dunne doos	0,05	15	30	45	10	0,83	74,33	-0,9	4
			0,005	50	5	0	45	23,76	81,45	2,7	4
		Dikke doos	0,05	15	45	35	5	3,51	295,67	-1,55	4,67
			0,005	10	0	0	90	13,30	127,50	0,5	5
Ver langs doos	Midden doos	Dunne doos	0,05	0	0	100	0	/	/	0	/
			0,005	85	0	0	15	7,68	26,29	3,65	4,29
		Dikke doos	0,05	55	20	10	15	10,98	848,27	1,45	4,45
			0,005	55	0	0	45	19,80	189,91	2,75	5
	Rand doos	Dunne doos	0,05	15	35	40	10	1,05	96,33	-1,3	3
			0,005	35	0	0	65	11,97	37,86	1,25	3,57
		Dikke doos	0,05	5	10	80	5	1,52	132,00	-0,35	3
			0,005	20	0	0	80	13,49	131,00	1,15	5,75

6.2 Specifieke parameters

De specifieke parameters zijn afhankelijk van het gebruikte padplanningsalgoritme. De meeste algoritmen bevatten zo een specifieke parameter en sommige algoritmen bevatten zelfs meerdere specifieke parameters. Er zijn vier verschillende parameters die specifiek zijn voor bepaalde algoritmen. Deze parameters zijn de *Range*, de *GoalBias,* de *Nearest Neighbours* en de *Border Fraction*. Dit deel van het hoofdstuk onderzoekt deze parameters hun invloed op het algoritme waarbij ze horen.

6.2.1 Range

De *range* is een belangrijke parameter die door meerdere algoritmen gebruikt wordt. Deze parameter beschrijft de maximale lengte van een verbinding. De standaardwaarde van deze parameters is 0. Dat wil niet zeggen dat de maximale lengte van de verbinding 0 is, want de waarde 0 is een uitzondering bij deze parameter. De waarde 0 komt overeen met een oneindige range. Dit houdt in dat er geen beperking staat op de maximale lengte van een verbinding. De *range* moet een waarde zijn die gelegen is tussen 0 en 10 000. Het aanpassen van de *range* gebeurt in het *ompl_planning.yaml* bestand, zoals dat in '5.3.2 Aanpassen parameters van algoritmen' al is uitgelegd. Het *yaml* bestand wordt opgeroepen door de *MotionPlanner plugin* tijdens het opstarten van Rviz waarop de algoritmen en hun parameters geïnitialiseerd worden. Figuur 319031 toont een boomstructuur van het RRT-algoritme dat een nieuw monster *x* wil toevoegen aan de boomstructuur. De afstand van het nieuwe monster tot de boomstructuur is echter te groot. Een nieuw bemonsterd punt x_{new}, dat wordt toegevoegd aan de boomstructuur, bevindt zich op de verbinding van monster *x* en de boomstructuur, maar ligt op de maximale afstand (ε=*range*) van de boomstructuur. Het algoritme verwijdert het monster x terug uit zijn omgeving. Het principe is uitgebreider uitgelegd in '3.2.1 RRT'.



Figuur 3190: Maximale range bij RRT [16]

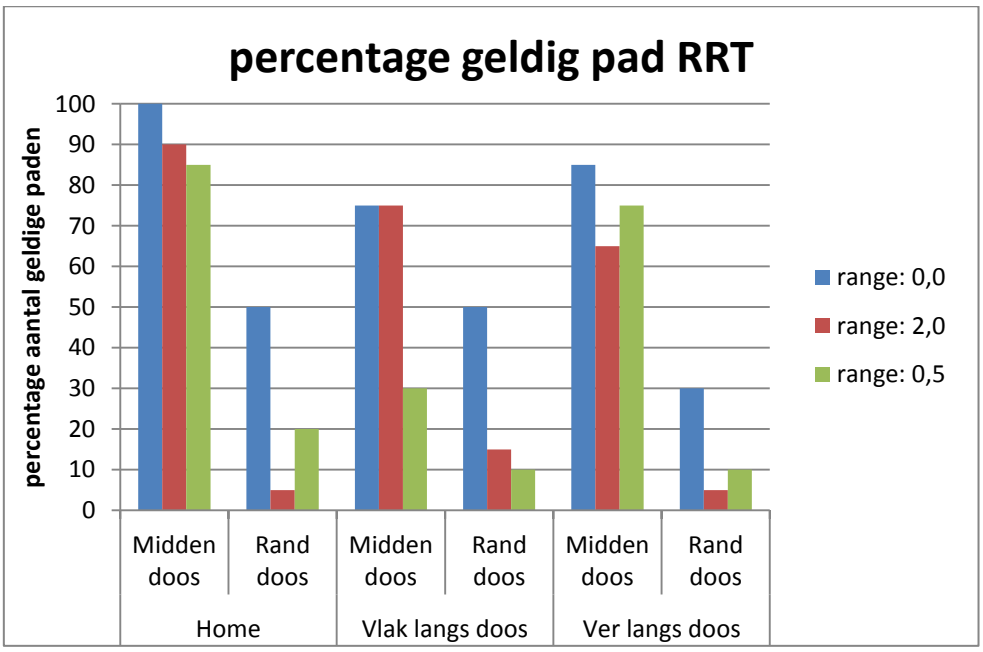
Het effect van een verhoogde *range* bij een algoritme heeft zowel voor- als nadelen. Het grote voordeel is dat, hoe kleiner de range is, hoe optimaler het pad is. Een ideaal pad is een pad waarbij geen onnodige omwegen gemaakt worden en waar de eindeffector het kortste pad volgt van begin- naar eindtoestand. Het grote nadeel hierbij is dat de rekentijd omgekeerd evenredig is de *range*. Dat wil zeggen dat als de *range* verkleint, de rekentijd stijgt. De algoritmen die gebruik maken van de *range* parameters zijn de volgende:

- LBKPIECE
- BKPIECE
- KPIECE
- EST
- RRT
- RRT*

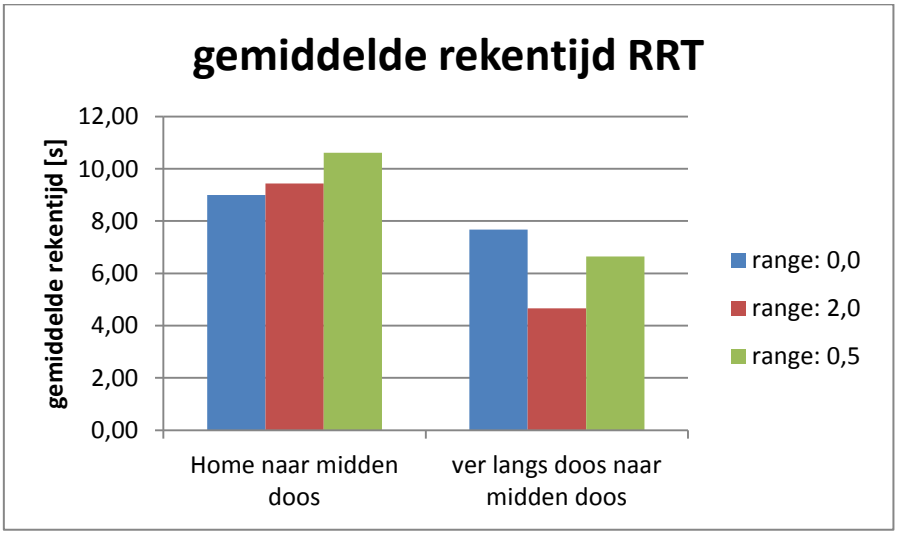
In tabel 3, die aan het einde van dit paragraaf staat, worden steeds alle *query's* doorlopen met drie verschillende instellingen van de *range* parameter. De instellingen van de andere parameters staan hierbij op een constante waarde: *longest_valid_segment_fraction* op 0.005= en de *GoalBias* op 0.05 (5%). De maximale rekentijd is ingesteld op 60 seconden, indien deze wordt overschreden, wordt er niet meer verder gezocht en is er geen geldig pad gevonden. De *range* instelling is in het eerste geval 0.0 (de standaardwaarde). Hierin is voor elke query, met uitzondering van de laatste query, in minstens 50% van de gevallen een pad gevonden. Voor de laatste query is dit slechts in 30% van de gevallen. Als de range is ingesteld op 2.0 dan daalt het aantal succesvolle paden tot onder 50% voor de volgende drie *query's*: van de *home* positie naar de rand van de doos, van langs de doos naar de rand van de doos en van ver langs de doos naar de rand van de doos. In de meeste gevallen faalt het vinden van het pad omdat er een te hoge rekentijd benodigd is. In het geval dat de *range* daalt tot 0.5 zijn er maar twee *query's* waar het percentage van een gevonden pad hoger ligt dan 50%: van de *home* positie naar de midden van de doos en van ver langs de doos naar het midden van de doos.

De invloed van de *range* op de simulatietijd en het aantal monsters is zichtbaar door alle resultaten van de eerste *query* met elkaar te vergelijken, dat is de situatie met als begintoestand de *home* positie en als eindtoestand het midden van de doos ('**Error! Reference source not found.**59 in 6.1.1 botsingsdetectieresolutie'). Daarin is zichtbaar dat bij de laagste *range* 0.5 de simulatietijd de hoogste is, maar ook dat het aantal *vertices* die toebehoren aan de boomstructuur beduidend toeneemt. Ook is de kans groter dat er geen pad wordt gevonden wegens een te lange rekentijd bij een kleinere *range.* Als de *range* onbeperkt is wordt er in 100% van de gevallen een pad gevonden, bij een *range* van 2.0 is dit in 90% van de gevallen en uiteindelijk bij een *range* van 0.5 is dat in 85% van de gevallen.

Figuur 91 toont de resultaten van verschillende parameter instellingen van de *range* bij het RRT-algoritme. Hierin zijn op de y-as het aantal geldige paden aangeduid in procenten en staan op de x-as de verschillende bewegingen met de verschillende begin- en eindtoestanden. Figuur 92 geeft een grafiek weer waarin twee verschillende *query's* met elkaar worden vergeleken op basis van de gemiddelde rekentijd. Deze twee *query's* zijn de enige twee waarvan, voor elke instelling van de *range,* meer dan 50% van de paden succesvol zijn. De andere *query's* hebben een te lange rekentijd (60 seconden) en vormen bijgevolg geen geldig pad.



Figuur 91: procentueel aantal van de gevonden paden met verschillende range instellingen bij RRT



Figuur 92: gemiddelde rekentijd in twee verschillende scenario's van het RRT-algoritme met verschillende range instellingen

In de *query* van de beweging van vlak langs de doos naar het midden van de doos (Figuur 61 in '**Error! Reference source not found.**') is het aantal succesvolle paden, van het algoritme met een *range* van 0.5, beduidend kleiner dan die van de andere twee instellingen zoals te zien is in figuur 91. Dit is te wijten aan een te hoge rekentijd. Bij de tweede en de vijfde *query* (respectievelijk figuur 60 en figuur 62 in '6.1.1 botsingsdetectieresolutie') vindt dit algoritme daarentegen meer succesvolle paden dan het algoritme met een ingestelde *range* van 2.0. Bij deze *query's* liggen de begin- en de eindtoestand relatief dicht bij elkaar. Indien deze afstand niet erg groot is, maar het pad wel redelijk complex, is de kans groter dat de instelling met een heel nauwkeurige instelling vaker een goed pad vindt. Er zijn net iets meer goede resultaten gevonden bij een *range* instelling van 0.5 (zie figuur 91 en 92**Error! Reference source not found.**), waarvan de rekentijd ook lager ligt.

Tabel 3 toont de resultaten van verschillende *query's* die gesimuleerd zijn met RRT waarbij de *range* eerst is ingesteld op 0.0, dus onbeperkt en daarna zijn de *ranges* verkleind naar 2.0 en 0.5.

Tabel 3: invloed van de range op RRT

Positie begintoestand	Positie eindtoestand	parameters:	Percentage geldig pad gevonden [%]	Percentage foutief pad gevonden [%]	Percentage failed pad gevonden [%]	Geen oplossing wegens te hoge rekentijd (%)	Gemiddelde berekeningstijd voor goed pad [s]	Gemiddeld aantal vertices dat een goed pad bevat	Beoordeling van het pad [-5 tot 6] (inclusief gefaalde berekeninge)	beoordeling van de geldige paden
Home	Midden doos	range: 0,0	100	0	0	0	8,99	29,15	6,00	6,00
	Rand doos		50	0	0	50	14,50	47,10	2,50	5,00
Vlak langs doos	Midden doos		75	0	0	25	15,02	54,40	3,25	4,33
	Rand doos		50	5	0	45	23,76	81,45	2,20	4,90
Ver langs doos	Midden doos		85	0	0	15	7,68	26,29	3,65	4,29
	Rand doos		30	5	0	65	/	/	/	/
Home	Midden doos	range: 2,0	90	0	0	10	9,43	90,06	5,40	6,00
	Rand doos		5	5	0	90	/	/	/	/
Vlak langs doos	Midden doos		75	0	0	25	5,83	56,73	3,60	4,80
	Rand doos		15	0	0	85	/	/	/	/
Ver langs doos	Midden doos		65	0	0	35	4,66	45,54	3,10	4,77
	Rand doos		5	5	0	90	/	/	/	/
Home	Midden doos	range: 0,5	85	0	0	15	10,61	336,47	5,10	6,00
	Rand doos		20	5	5	70	/	/	/	/
Vlak langs doos	Midden doos		30	0	5	65	/	/	/	/
	Rand doos		10	0	0	90	/	/	/	/
Ver langs doos	Midden doos		75	5	10	10	6,65	209,17	4,23	5,93
	Rand doos		10	10	0	80	/	/	/	/

6.2.2 *GoalBias*

De *GoalBias* is een procentuele waarde die instelbaar is voor meerdere planners. Deze parameter komt enkel voor bij padplanningsalgoritmen die gebaseerd zijn op een boomstructuur. De *GoalBias* is de kans dat het volgende willekeurige monster, de eindtoestand zal zijn. Bij een boomstructuuralgoritme zijn immers de begin- en de eindtoestand gekend, maar zijn die niet met elkaar verbonden.

Als de *GoalBias* op 100% staat wil dit zeggen dat het algoritme de eindtoestand als volgende monster aan zijn boomstructuur wil toevoegen. Dit werkt slechts in enkele gevallen. Indien de begin- en de eindtoestand bijvoorbeeld verder uit elkaar liggen dan de waarde die bij de *range* is ingegeven, kan de verbinding niet rechtstreeks tot stand komen. De planner vindt uiteindelijk wel een pad als de maximale rekentijd hoog genoeg is en als er zich geen hindernis tussen de toestanden bevindt. Dit komt omdat, zoals in '3.2.1 RRT' is uitgelegd, de planner het punt wel wil toevoegen, maar zijn *range* klein is. Hierdoor voegt de planner een *vertex* toe dat op een rechte lijn ligt naar de eindtoestand op een afstand die gelijk is aan de *range*. Indien de begin- en eindtoestand 10 keer verder uit elkaar liggen dan de waarde van de *range*, worden 10 monsters toegevoegd voordat er een pad is gevonden. Als er nu een hindernis ligt tussen de begin- en de eindtoestand, kan het algoritme nooit een pad vinden als de *GoalBias* is ingesteld op een waarde van 100% aangezien het algoritme altijd een rechte verbinding probeert te maken. Er worden geen andere monsters toegevoegd waardoor het onmogelijk is om rond het obstakel te geraken. Indien de *GoalBias* daarentegen de waarde 0 zou hebben, dan betekent dit dat het algoritme de eindtoestand nooit toevoegt als volgende monster en zal de planning van het pad bijna altijd falen. Er kan alleen een pad gevonden worden indien een willekeurig monster toevallig net dezelfde toestand is als de eindtoestand. Deze kans is extreem laag waardoor het algoritme nooit binnen de tijd een pad vindt.

Experimenteel zijn beide gevallen getest waarbij de *GoalBias* zowel op 0% als op 100% is ingesteld. Hierbij zijn de verwachtte resultaten experimentele bevestigd. Een ideale instelling van de *GoalBias* is 5% of 0.05. Deze waarde is aangeraden als vaste parameter door ontwikkelaars van OMPL [43]. Alle simulaties, waarbij deze parameter nodig was, zijn ook met een waarde van 0.05 uitgevoerd. De planners die gebruikmaken van de *GoalBias* parameter zijn de volgende:

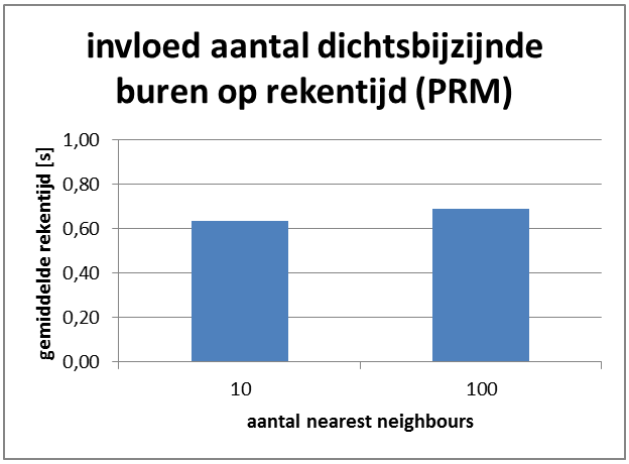
- KPIECE
- EST
- RRT
- RRT*
- SBL

6.2.3 *Nearest neighbours*

De volgende parameter die aanpasbaar is, is het aantal buren waarmee een *vertex* maximaal een verbinding mee mag proberen te maken. In de *cpp*-bestanden heet deze parameter *max_nearest_neighbours*. Van alle padplanningsalgoritmen in OMPL is PRM de enige planner die hier gebruik van maakt. De werking van PRM en de parameter is terug te vinden in '3.1.1 PRM' van deze scriptie.

De waarde van de parameter is in OMPL beperkt tot een minimale waarde van 8 en een maximale waarde van 1000. Indien een waarde wordt ingegeven die kleiner is dan de minimale waarde wordt deze afgerond naar 8. Omgekeerd werkt dit ook zo bij het overschrijden van de maximale waarde. De standaardwaarde van deze parameter in de *MotionPlanner plugin* is 10.

In figuur 93 staan de gemiddelde rekentijden voor de simulaties van meerder *query's* met verschillende instellingen van de parameter *nearest neighbours*. De instellingen van de andere parameters blijven constant: *longest_valid_segment_fraction* staat ingesteld op 0.005 en de maximale rekentijd staat op 60 seconden. Het aantal *nearest neighbours* staat eerst ingesteld op 10 en daarna op 100.



Figuur 93: invloed parameter aantal dichtstbijzijnde buren op de rekentijd bij PRM

Bij de simulaties zijn geen noemenswaardige verschillen opgevallen door het aanpassen van deze parameter. In tabel 4**Error! Reference source not found.** zijn de resultaten te zien van deze parameter die in het eerste geval is ingesteld op 10 en in het tweede geval op 100. De gemiddelde simulatietijden van de succesvolle paden liggen zeer dicht bij elkaar, voor het geval dat de parameters is ingesteld op 10 is dit 0.63 en voor 100 is dat 0.69 seconden. In figuur 93 zijn die resultaten in grafiek gebracht. Ook het gemiddelde aantal *vertices*, het aantal buren, ligt zeer dicht bij elkaar. Het aantal buren overschrijdt nooit de waarde 28, bijgevolg kan een parameter die ingesteld is op 100 nog steeds maar 28 buren verbinden. Een nog hogere instelling van het aantal *nearest neighbours* dan 100 zou dus geen verschil maken. Bonvendien is het verschil tussen 10 en 100 zo klein dat hieruit het besluit volgt dat het aanpassen van deze parameter amper een effect heeft op de resultaten.

Tabel 4 de resultaten van verschillende *query's* weer die gesimuleerd zijn met PRM waarbij de parameter *max_nearest_neighbours* in het eerste geval is ingesteld op 10 en daarna op 100.

Tabel 4: resultaten van de PRM planner met max_nearest_neighbours waarden van 10 en 100

Positie begintoestand	Positie eindtoestand	Instelbare parameter: aantal dichtstbijzijnde buren	Percentage geldig pad gevonden [%]	Percentage foutief pad gevonden [%]	Percentage failed pad gevonden [%]	Percentage failed door te lange rekentijd [%]	Gemiddelde berekeningstijd voor goed pad [s]	Gemiddeld aantal vertices dat een goed pad	Gemiddelde beoordeling van alle paden [-5 tot 6]	Gemiddelde beoordeling geldige paden [1 tot 6]
Home	Midden doos	10	100	0	0	0	0,32	2	6	6
	Rand doos		0	0	100	0	/	/	0	/
Vlak langs doos	Midden doos		0	0	100	0	/	/	0	/
	Rand doos		15	15	25	0	/	/	-1,7	3,67
Ver langs doos	Midden doos		85	15	0	0	0,94	5,94	2,15	3,41
	Rand doos		6,67	40	53,33	0	/	/	-1,80	3
Home	Midden doos	100	100	0	0	0	0,31	2	6	6
	Rand doos		0	0	100	0	/	/	0	/
Vlak langs doos	Midden doos		0	0	100	0	/	/	0	/
	Rand doos		30	35	35	0	/	/	-0,8	3,33
Ver langs doos	Midden doos		80	5	15	0	1,07	7,19	2,50	3,44
	Rand doos		10	55	35	0	/	/	-2,45	3

6.3.4 *Border fraction*

De *border fraction* parameter is enkel relevant voor de verschillende KPIECE padplanningsalgoritmen, namelijk KPIECE, BKPIECE en LBKPIECE. Om de instelling van de parameter volledig te begrijpen moet eerst de werking van KPIECE gekend zijn. Deze is in het eindwerk te vinden in '3.2.7 KPIECE: stap 1 uitvoering algoritme'. De waarde van de *border fraction* is uitgedrukt als een procentuele waarde, gelegen tussen 0 en 1. De ingevulde waarde komt overeen met de kans dat de boomstructuur verder verkent op een exterieure cel. De ingevulde *border fraction* is enkel een minimale waarde. Indien de structuur een percentage, van exterieure cellen t.o.v. de totale cellen, heeft dat hoger ligt dan de *border fraction,* dan neemt de parameter de waarde over van het percentage exterieure cellen in de structuur. Het totale aantal cellen is de som van de interieure en de exterieure cellen. Er volgt nu een voorbeeld om dit te verduidelijken.

Stel dat de *border fraction* gelijk is aan 80%. De kans dat de volgende willekeurig gekozen cel een exterieure cel is, is gelijk aan 80%. Indien 90% van alle cellen echter exterieure cellen zijn, verandert deze instelling van de parameter naar 90%.

In tabel 5 zijn de resultaten te zien van het KPIECE-algoritme dat in alle *query's* is getest met twee verschillende instellingen van de *range*. De parameter *border fraction* staat steeds op zijn standaardwaarde van 90%. De laatste kolom toont het gemiddelde percentage van de *border fraction* voor de volledige structuur. In het uiteindelijke resultaat zijn amper interieure cellen aanwezig waardoor dat percentage een hoge waarde aanneemt, meestal zelfs 100%. Een verandering van de *border fraction* parameter heeft daarom geen invloed op het resultaat omdat deze altijd overschreven wordt. De simulaties van KPIECE zijn dus uitgevoerd met een constante waarde van de *border fraction*, namelijk de standaardwaarde 0.9 zoals in tabel 5 te zien is.

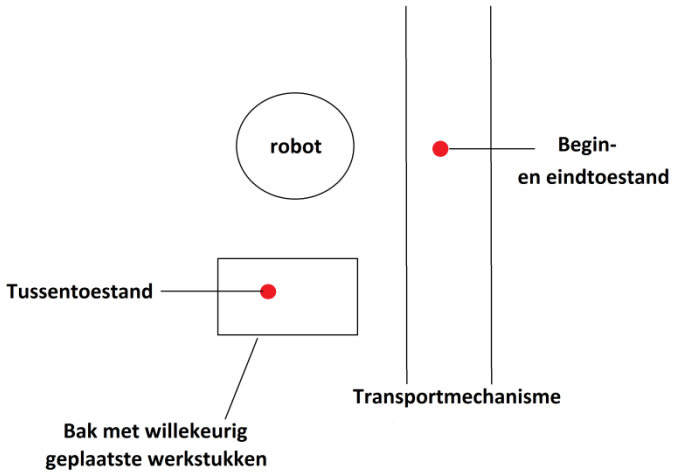
Tabel 5 enkel de relevante resultaten om het de gemiddelde *border fraction* aan te tonen bij KPIECE.

Tabel 5: border fraction resultaten bij verschillende instellingen van de range met KPIECE

Positie begintoestand	Positie eindtoestand	parameters:	Percentage geldig pad gevonden [%]	Cells		gemiddelde border fraction
				internal	external	
Home	Midden doos	range: 0,0 border_fraction: 0,9	95	0,00	27,47	100,00
	Rand doos		70	0,00	36,73	100,00
Vlak langs doos	Midden doos		90	0,00	27,50	100,00
	Rand doos		55	0,09	53,18	99,83
Ver langs doos	Midden doos		100	0,00	26,33	100,00
	Rand doos		60	0,00	49,17	100,00
Home	Midden doos	range: 20,0 border_fraction: 0,9	95	0,00	27,79	100,00
	Rand doos		60	0,00	40,09	100,00
Vlak langs doos	Midden doos		90	0,00	27,19	100,00
	Rand doos		55	0,00	18,00	100,00
Ver langs doos	Midden doos		80	0,00	17,69	100,00
	Rand doos		35	0,00	20,83	100,00

6.3 Vergelijking padplanningsalgoritmen

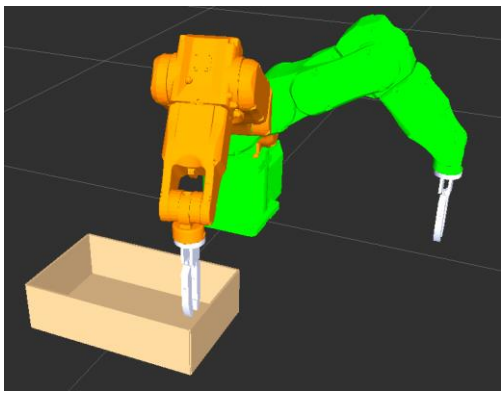
In de vorige delen van dit hoofdstuk zijn de verschillende parameters besproken. Ook hun invloed op de performantie van een algoritme is besproken hiervoor. Maar welk padplanningsalgoritme is het beste om toe te passen in de vooropgestelde probleemstelling? Om dit te onderzoeken zijn er twee *query's* gemaakt die een realistische nabootsing zijn van een industriële robot die een bak moet leegmaken. In de praktijk komt het vaak voor dat een robot de werkstukken op een transportband, of een ander soort transportmechanisme, moet leggen. Deze mechanismen staan bijna altijd langs de bak die de robot moet leegmaken. Daarom iservoorgekozen om de robot zijn begin- en eindtoestand 90° uit elkaar te leggen uit bovenaanzicht. Een vaak voorkomende opstelling van een robot die een bak moet leegmaken en op een transportmechanisme moet plaatsen is in figuur 94 te zien.



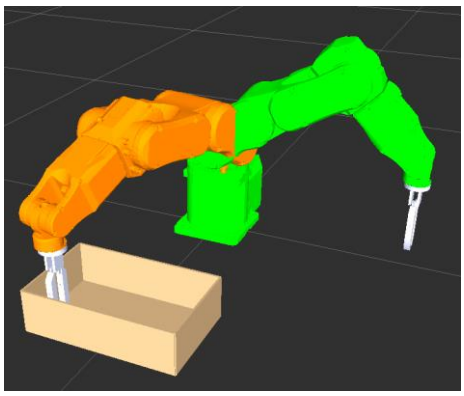
Figuur 94: Plattegrond praktijkopstelling robot, bak en transportmechanisme

De robot start aan de begintoestand, dat gelegen is aan het transportmechanisme. Vervolgens gaat de robot naar de bak om daar een werkstuk te grijpen. De plaats en oriëntatie die nodig is om een werkstuk te grijpen, is de tussentoestand. Vervolgens moet de robot dit werkstuk op het transportmechanisme plaatsen, wat overeenkomt met de eindtoestand. Om deze situatie te simuleren iservoorgekozen om de begintoestand in de buurt van het transportmechanisme te leggen en de eindtoestand in de bak. Aangezien verschillende posities in de bak overeenkomen met een verschillende moeilijkheidsgraad iservoorgekozen om twee *query's* te simuleren met een verschil in moeilijkheidsgraad. Het experiment is uitgevoerd op de volgende *query's*:

- *query* 1: van 90° verwijdert van de bak tot binnenkant bak (figuur 95);
- *query* 2: van 90° verwijdert van bak tot buitenkant bak (figuur 96).



Figuur 95: Query 2: van 90° verwijdert van bak tot binnenkant bak



Figuur 96: Query 1: van 90° verwijdert van bak tot buitenkant bak

De parameters die specifiek zijn voor het algoritme staan allemaal op hun standaard waarde ingesteld. Dit is gedaan om de invloed van de parameters uit te sluiten zodat enkel de padplanningsalgoritmen effectief met elkaar vergeleken worden. De simulatie maakt voor dit onderzoek gebruik van een bak met dunne wanden en een maximale rekentijd van 60 seconden zoals dat altijd het geval is. Om realistische waarden te bekomen, is een goede instelling van de botsingsdetectieresolutie gemaakt. Het *longest valid segment fraction* is namelijk gelijk genomen aan de wanddikte van de bak (0,005m) zodat het percentage aan foutieve paden beperkt blijft. Al de resultaten van het experiment zijn terug te vinden in tabel 6. Deze tabel is achteraan dit deel van het hoofdstuk terug te vinden.

De bedoeling is dus om de meest optimale algoritmen, voor deze probleemstelling, te selecteren. Dit zijn valzelfsprekend de algoritmen die:

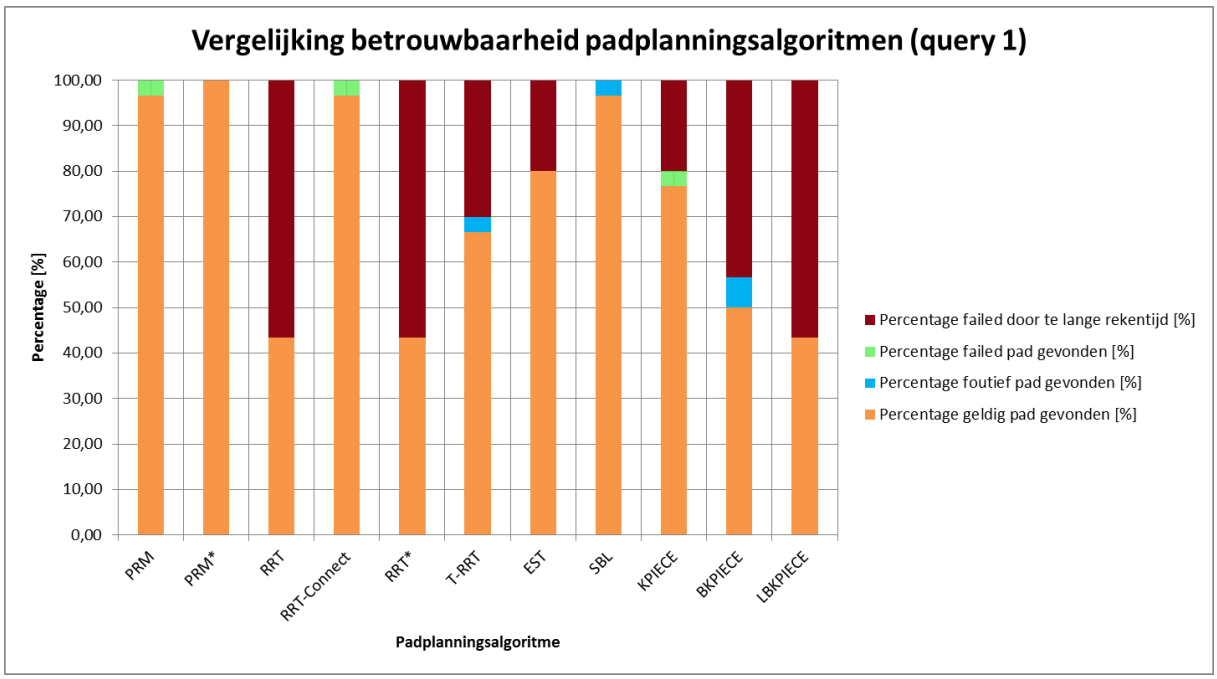
- een hoog percentage hebben voor 'geldige paden gevonden';
- een laag percentage hebben voor '*failed* wegen een te hoge rekentijd';
- een zeer laag percentage hebben voor 'foutieve paden gevonden';
- een lage gemiddelde rekentijd hebben;
- geen slechte beoordeling krijgen voor hun benadering op het optimale pad.

Alvorens een padplanningsalgoritme als voldoende wordt bestempeld om in de praktijk te gebruiken, moet het padplanningsalgoritme aan enkele eisen minimaal voldoen. Het padplanningsalgoritme:

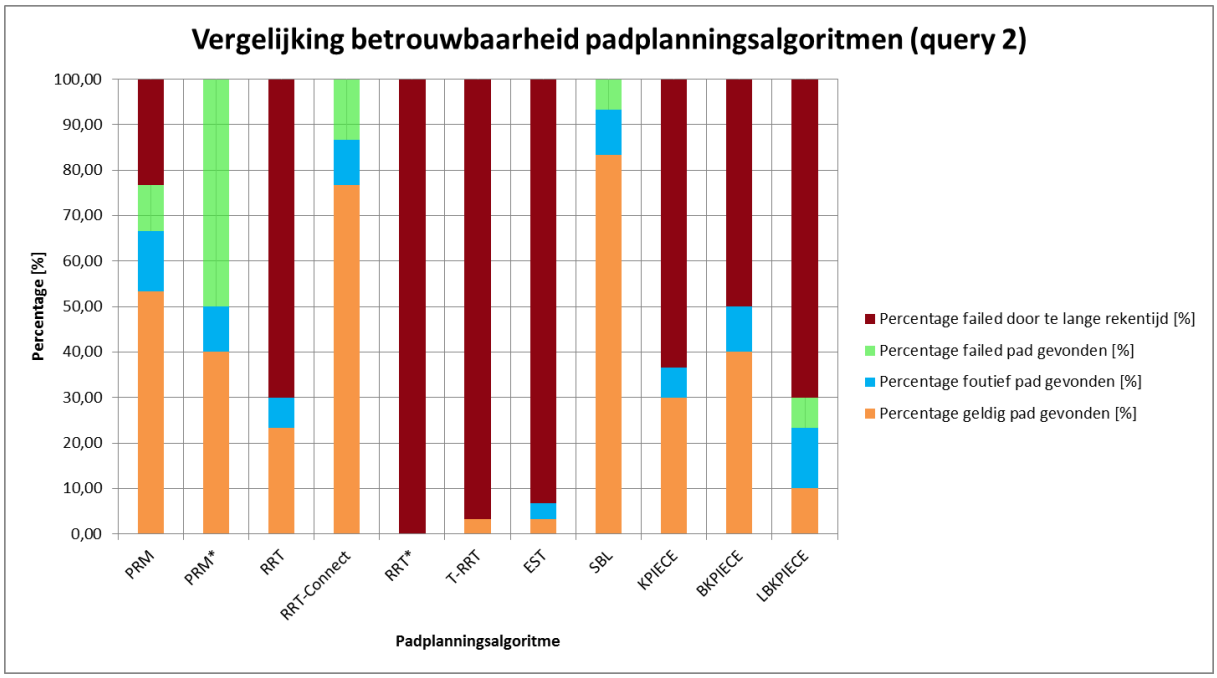
- moet minstens 95% van de keren een geldig pad als resultaat geven;
- moet bijna 0% van de keren een foutief pad als resultaat geven;
- mag een maximale rekentijd van 5 seconden hebben;
- moet een gemiddelde beoordeling van 3 of meer halen voor alle paden.

De laatste eis is dus de gemiddelde beoordeling van alle paden (tussen -5 en 6) en niet de gemiddelde beoordeling van de geldige paden (tussen 1 en 6).

Om de resultaten beter te kunnen beoordelen zijn de belangrijkste gegevens uit tabel 6 verzameld in enkele grafieken. De twee grafieken die volgen illustreren de betrouwbaarheid van ieder padplanningsalgoritme. Figuur 97 laat de betrouwbaarheid van de verschillende padplanningsalgoritmen zien voor *query* 1 (figuur 95). Figuur 98 laat hetzelfde zien, maar deze keer getest voor *query* 2 (figuur 96).



Figuur 97: Padplanningsalgoritmen i.f.v. de betrouwbaarheid van het algoritme, toegepast op query 1



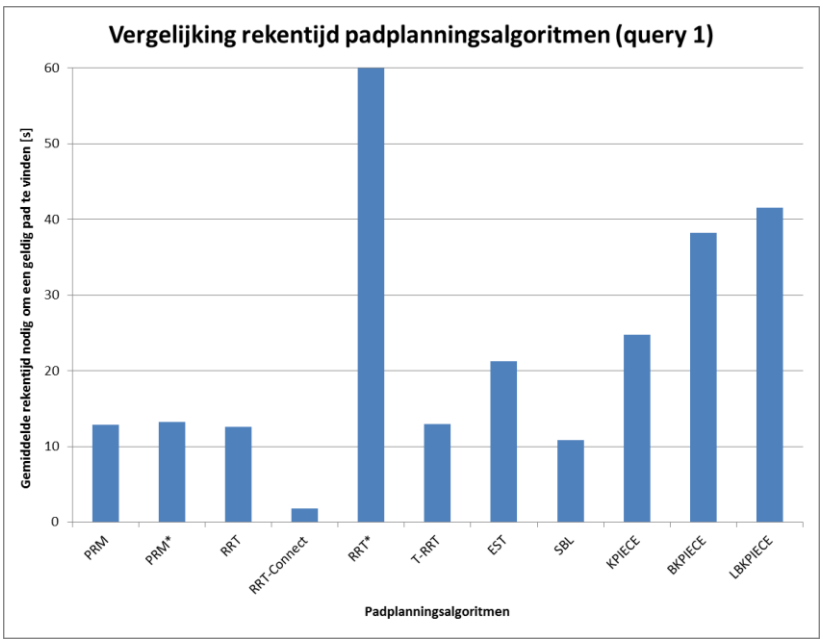
Figuur 98: Padplanningsalgoritmen i.f.v. de betrouwbaarheid van het algoritme, toegepast op query 2

Uit de grafieken blijkt meteen dat de betrouwbaarheid voor *query 2* (buitenkant doos) lager ligt dan die van *query* 1 (binnenkant doos). Dat komt doordat de wand, die het verst verwijdert is van de robot, bij *query* 2 voor moeilijkheden zorgt. Dit omwille van het feit dat deze wand altijd in de weg ligt om tot een eindtoestand te geraken die in de buurt van deze wand ligt.

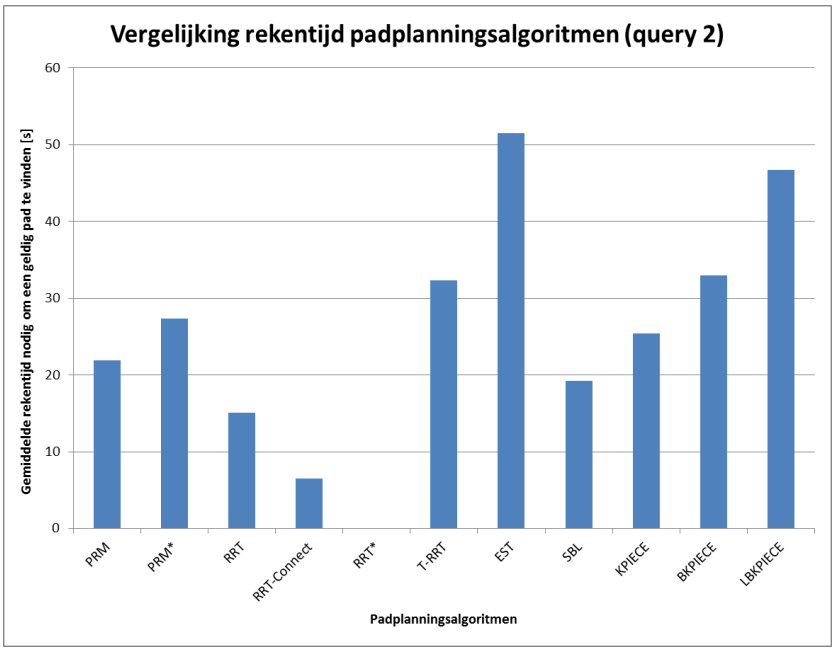
Uit de figuren 97 en 98 is ook af te leiden dat er een heel deel padplanningsalgoritmen zijn die, vooral voor *query* 2, een heel hoog percentage *failed* wegens een te hoge rekentijd halen. Het grootste deel van de padplanningsalgoritmen hebben bij *query* 2 zelfs meer dan 50% van de keren *failed* als resultaat wegens een te hoge rekentijd. Deze padplanningsalgoritmen zijn dus al niet aan te raden voor deze probleemstelling. Het gaat over de algoritmen RRT, RRT*, T-RRT, EST, KPIECE, BKPIECE en LBKPIECE. De algoritmen die gebruik maken van KPIECE (KPIECE, BKPIECE en LBKPIECE) hebben te veel tijd nodig om de toestandsruimte te projecteren ('Hoofdstuk 3: Padplanningsalgoritmen in de theorie'). Deze algoritmen zijn enkel bruikbaar in probleemstellingen die zeer complex en hoog dimensionaal zijn. RRT, RRT* en EST zijn algoritmen die gebruikmaken van lange, rechte verbindingen. Aangezien er een obstakel ligt tussen de begin- en eindtoestand gaan de verbindingen een omweg maken rond het obstakel. De verbindingen zijn vaak lang waarop de botsingsdetectie heel lang moet controleren of de verbinding geldig is. Hierdoor zit het algoritme snel aan 60 seconden. Deze algoritmen hebben het dus moeilijk indien er een grote afstand overbrugd moet worden en er een obstakel tussen de begin- en eindtoestand ligt. T-RRT maakt gebruik van een kostenfunctie waar het algoritme veel tijd aan spendeert. Het zoeken van een pad met een optimale kost neemt veel tijd in beslag, maar doordat iedere kost berekend moet worden aan de hand van een kostenfunctie, is het algoritme veel te lang aan het rekenen.

De algoritmen die minder als 50% van de keren een *failed,* wegens te hoge rekentijd, als resultaat krijgen zijn PRM, PRM*, RRT-Connect en SBL. Alle vier de algoritmen tonen gelijkaardige, goede resultaten voor *query* 1. Het percentage dat een geldig pad wordt gevonden ligt voor al deze algoritmen hoger dan 95% wat een goed resultaat is. Voor *query* 2 hebben deze algoritmen ook minder goede resultaten. Vooral PRM en PRM* halen minder goede resultaten dan RRT-Connect en SBL aangezien het percentage, dat ze een geldig pad als resultaat terugkrijgen, zakt tot ongeveer 50%. De algemene conclusie van deze twee grafieken is dat PRM, PRM* en vooral RRT-Connect en SBL de padplanningsalgoritmen zijn met de hoogste betrouwbaarheid. De analyse of één van deze algoritmen goed genoeg is om aan de eisen te voldoen, staat verderop in dit deel van het hoofdstuk.

De volgende twee grafieken maken het mogelijk om te bepalen welke padplanningsalgoritmen de laagste rekentijd nodig hebben om een geldig pad te vinden. Figuur 99 geeft voor alle padplanningsalgoritmen de gemiddelde rekentijd weer, dat een algoritme nodig heeft om een geldig pad te vinden, toegepast op *query* 1. Figuur 100 laat hetzelfde zien, maar deze keer toegepast op *query 2*.



Figuur 99: Padplanningsalgoritmen i.f.v. gemiddelde rekentijd nodig om een geldig pad te vinden, toegepast op query 1



Figuur 100: Padplanningsalgoritmen i.f.v. gemiddelde rekentijd nodig om een geldig pad te vinden, toegepast op query 2

Uit figuren 99 en 100 valt meteen op dat RRT* een vreemd gedrag vertoont. RRT* heeft bij *query* 1 namelijk een gemiddelde rekentijd van 60 seconden nodig om een pad te vinden terwijl het bij *query* 2 precies 0 seconden nodig heeft. Deze twee extreme waarden zijn zeer eenvoudig te verklaren. *Query* 2 staat op 0 seconden omdat hier 100% van de keren *failed*, door een te hoge rekentijd, is geweest. Daardoor is er geen enkel geldig pad gevonden en is er dus ook geen gemiddelde rekentijd om een geldig pad te vinden. *Query* 1 heeft een gemiddelde van 60 seconden, wat overeenkomt met de maximale toegelaten waarde. Deze waarde ontstaat doordat RRT* niet stopt met zoeken indien het een pad heeft gevonden (‘Hoofdstuk 3:

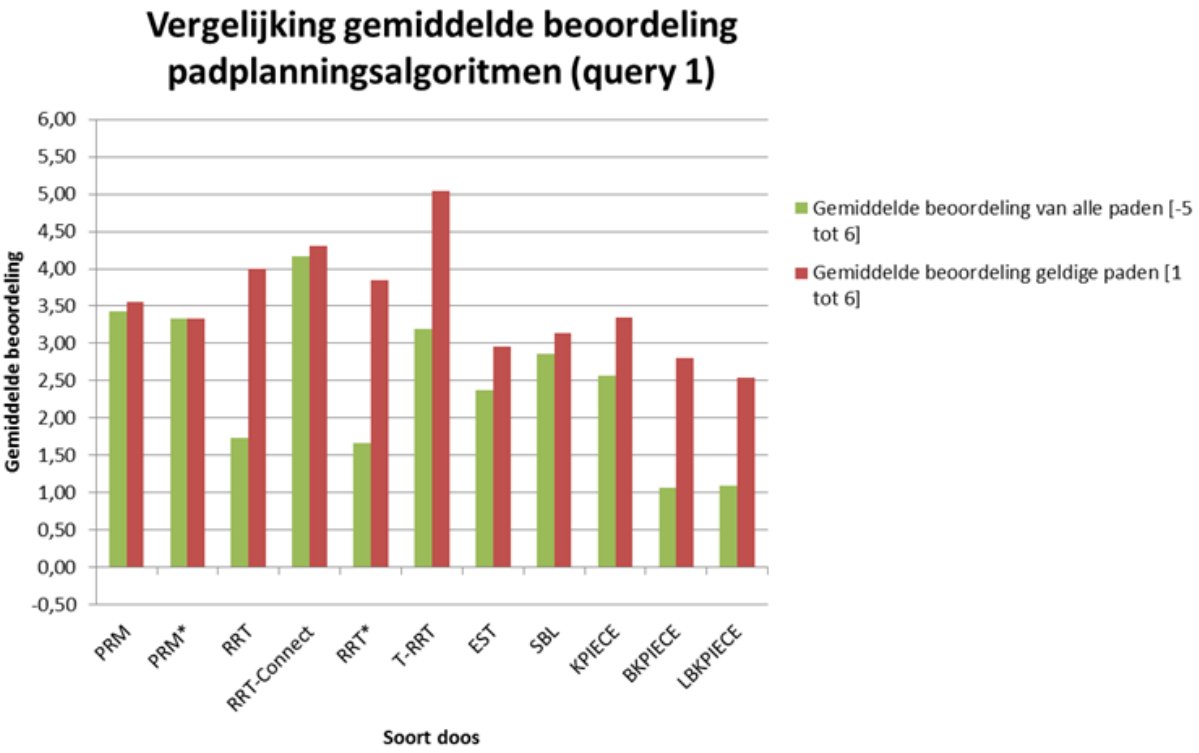
Padplanningsalgoritmen in de theorie'). RRT* blijft daarentegen doorzoeken naar een optimaler pad, tot de maximale rekentijd bereikt is. Op dat moment geeft het algoritme het meest optimale pad terug als resultaat.

Ook hier valt weer op dat de algoritmen die gebruik maken van KPIECE (KPIECE, BKPIECE en LBKPIECE) veel tijd nodig hebben om een pad te vinden. De verklaring is dezelfde als degene die hiervoor al gezegd is, het algoritme moet namelijk iedere keer opnieuw de toestandsruimte projecteren wat redelijk wat tijd in beslag neemt. Ook hier komt EST terug als een algoritme dat veel tijd nodig heeft om te berekenen. RRT en T-RRT vallen hier niet op als algoritmen die veel tijd nodig hebben om een geldig pad te vinden. Nochtans hadden ze een lage betrouwbaarheid doordat ze vaak *failed* als resultaat hadden wegens een te hoge rekentijd. RRT (en dus ook T-RRT) heeft als karakter dat het zeer snel in alle richtingen verkend. Indien het algoritme geluk heeft, vindt het snel een geldig pad. Hiertegenover kan het ook voorvallen dat RRT zich focust op de verkeerde richting waardoor het algoritme heel veel tijd nodig heeft of zelfs niet binnen de maximale rekentijd blijft.

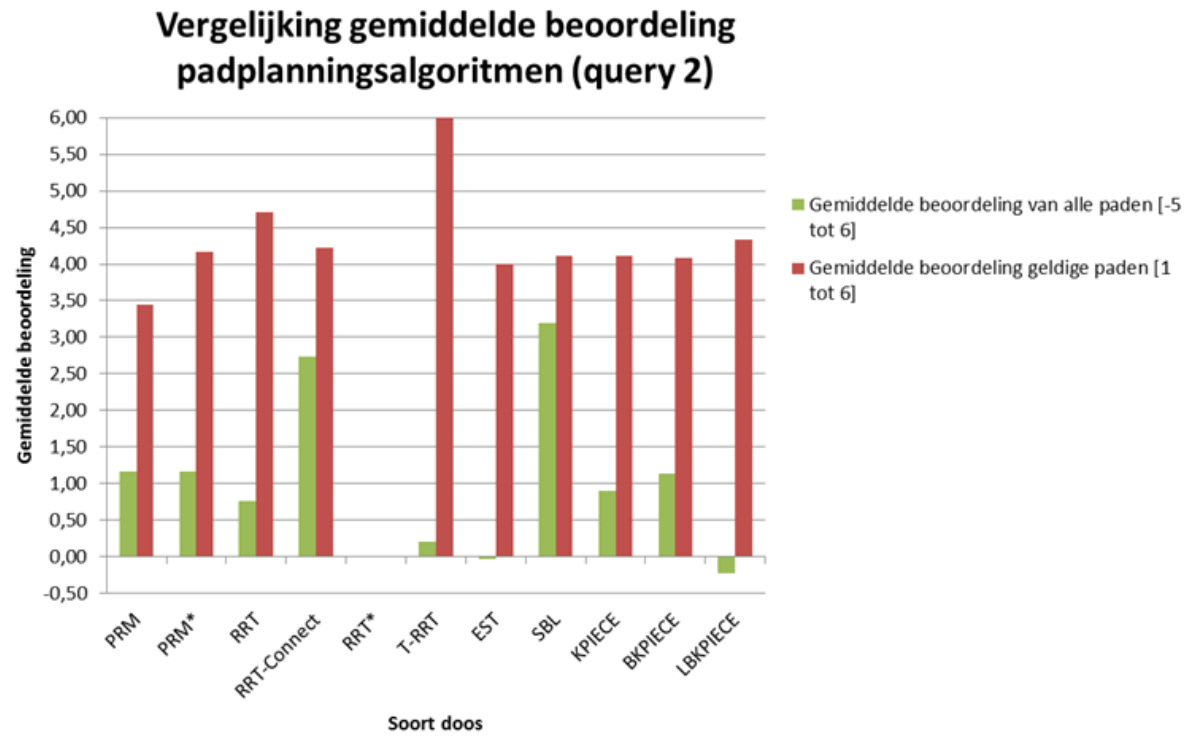
Wat ook opvalt uit de grafieken is dat RRT-Connect bijzonder goed scoort. Het algoritme heeft bij beide *query's* veel minder tijd nodig om een geldig pad te vinden. Dat komt doordat RRT-Connect twee voordelen combineert, namelijk het snel ontdekkingskarakter van RRT en de twee boomstructuren. De boomstructuur, die vertrekt vanuit de eindtoestand, zorgtervoordat de boomstructuur, vertrekkende vanuit de begintoestand, geen pad meer moet zoeken om in de bak te geraken. Deze bak is namelijk een obstakel waarin de eindeffector moet komen. Het moeilijkste voor de padplanningsalgoritmen is om in de bak te geraken zonder de obstakels te geraken. Doordat er een structuur vertrekt vanuit de eindpositie is dit moeilijke aspect niet meer nodig en moet het algoritme enkel nog proberen een verbinding te maken tussen de twee structuren. Het snelle ontdekkingskarakter van RRT is ideaal hiervoor om dat in korte tijd te doen. RRT laat namelijk toe om verre afstanden af te leggen in vele richtingen waardoor de bomen elkaar al snel kruisen. Op dat moment vindt het algoritme al een pad.

De conclusie die te trekken is uit figuren 99 en 100 is dat RRT-Connect duidelijk het minste rekentijd nodig heeft. KPIECE, BKPIECE, LBKPIECE, EST en RRT* schieten duidelijk te kort volgens de grafieken. De waarden van PRM, PRM*, RRT, T-RRT en SBL zijn acceptabel, maar zijn zeker niet goed. De waarden liggen rond de 12 seconden bij *query* 1 en rond de 20 seconden bij *query*. Deze rekentijden zijn vrij hoog om te kunnen gebruiken in de praktijk. De analyse, of er een algoritme is dat voldoet aan de eisen, volgt verderop in dit deel van het hoofdstuk.

De twee volgende grafieken beschouwen de beoordeling van het gemiddelde pad voor verschillende padplanningsalgoritmen bij beide *query's*. De grafieken bevatten zowel de gemiddelde beoordeling van een geldig pad als de gemiddelde beoordeling van alle paden. Figuur 101 illustreert de gemiddelde beoordelingen van ieder algoritme bij *query* 1 en figuur 102 toont hetzelfde, maar deze keer toegepast op *query* 2.



Figuur 101: Padplanningsalgoritmen i.f.v. de gemiddelde beoordeling van een pad, toegepast op query 1



Figuur 102: Padplanningsalgoritmen i.f.v. de gemiddelde beoordeling van een pad, toegepast op query 2

Bij *query* 2 heeft RRT* een gemiddelde beoordeling van 0 wat weer te wijten is aan het feit dat er geen enkel geldig pad is gevonden bij de metingen. T-RRT lijkt zeer goed te scoren bij *query*2 voor zijn geldige paden, maar dat komt omdat er maar 3,33% van de metingen geldig was. Daardoor is deze meting onbetrouwbaar. Wat ook opvalt in de figuren 101 en 102 is dat de padplanningsalgoritmen die niet goed scoorden op de betrouwbaarheid, hier ook niet hoog scoren op de gemiddelde beoordeling.

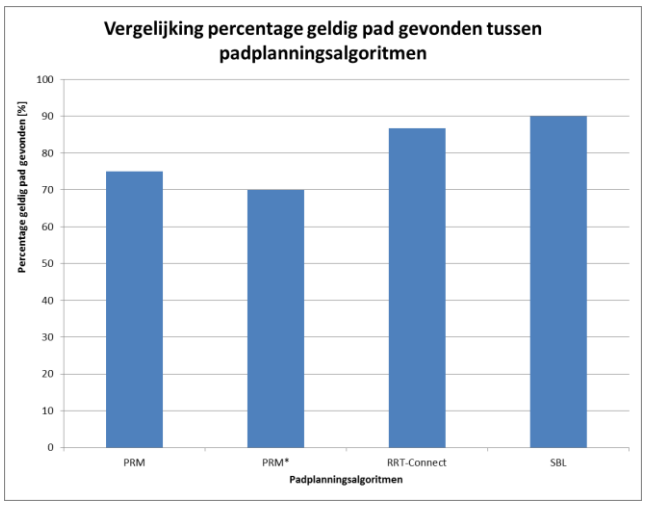
Om geen foutief beeld te krijgen, worden de gemiddelde beoordelingen voor alle paden geëvalueerd. Deze beoordelingen houden ook rekening met de foutief gevonden paden, de *failed* paden en de *failed* wegens een te hoge rekentijd. Bij *query* 1 behalen PRM, PRM*, T-RRT, SBL en vooral RRT-Connect een hoge beoordeling.

Bij *query* 2 behalen RRT-Connect en SBL een hele hoge score terwijl de rest een stuk lager scoort. PRM, PRM* en BKPIECE halen een gemiddelde beoordeling van alle paden die net boven 1 ligt. T-RRT zit niet meer bij de algoritmen met een goede beoordeling aangezien dit algoritme maar 3,33% van de keren een geldig pad vond.

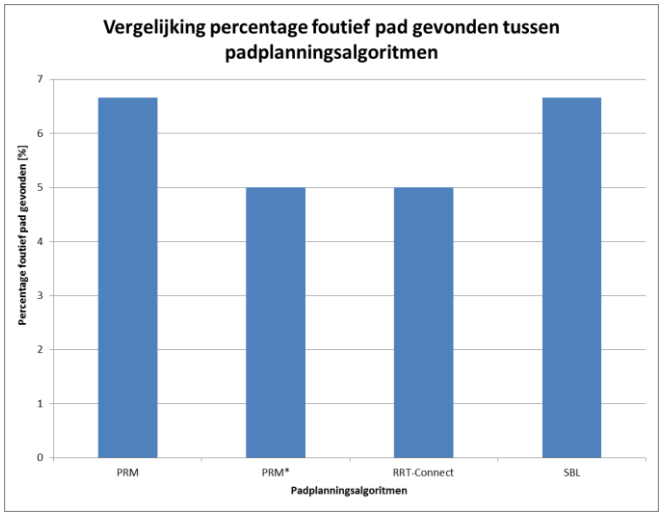
De algoritmen die voor beide *query's* een goed of een acceptabel resultaat behalen zijn PRM, PRM*, SBL en RRT-Connect. Deze vier algoritmen waren ook al de beste padplanningsalgoritmen bij de beoordeling van de betrouwbaarheid van een algoritme. Ook bij de gemiddelde rekentijd om een geldig pad te vinden behoorden deze padplanningsalgoritmen tot de beste algoritmen.

Uit dit onderzoek is dus te besluiten dat PRM, PRM*, SBL en RRT-Connect de beste performantie halen voor deze probleemstelling. Uit de grafieken blijkt dat SBL en vooral RRT-Connect gemiddeld het beste scoren. De andere algoritmen zijn zeker niet goed genoeg om te gebruiken in deze probleemstelling, maar de andere vier algoritmen zijn goed genoeg om verder in detail te analyseren. Om de verschillen tussen de vier beste algoritmen te accentueren, volgen hierna enkele grafieken die enkel toegepast zijn op deze vier algoritmen. Door middel van die figuren is ook na te gaan of één van de padplanningsalgoritmen voldoet aan de vooropgestelde eisen. De grafieken die volgen, nemen als resultaat het gemiddelde van de twee *query's*.

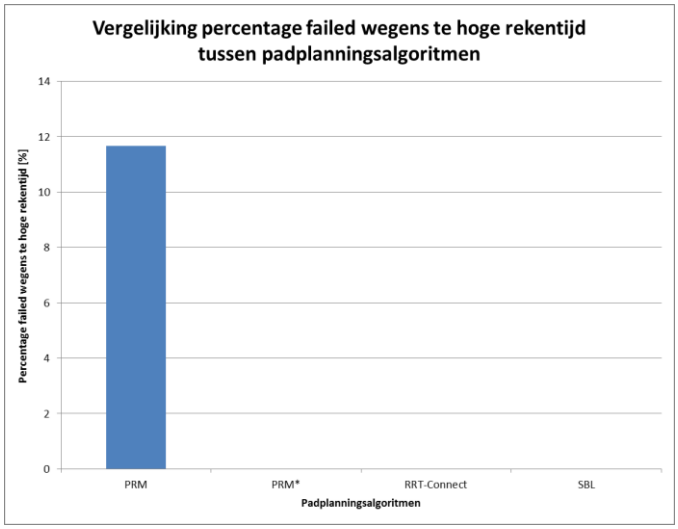
Eerst geeft Figuur 103 het gemiddeld percentage aan geldig gevonden paden voor de vier algoritmen. Vervolgens illustreert figuur 104 het gemiddeld percentage dat ieder algoritme een foutief pad als resultaat had. Daarna beschouwt figuur 105 het percentage dat de algoritmen een *failed* wegens een te hoge rekentijd bekomen. Figuur 106 maakt duidelijk hoeveel rekentijd de vier algoritmen gemiddeld nodig hebben om een geldig pad te vinden. De laatste afbeelding, figuur 107, vergelijkt voor de vier algoritmen de gemiddelde beoordeling van zowel alle paden als enkel de geldige paden.



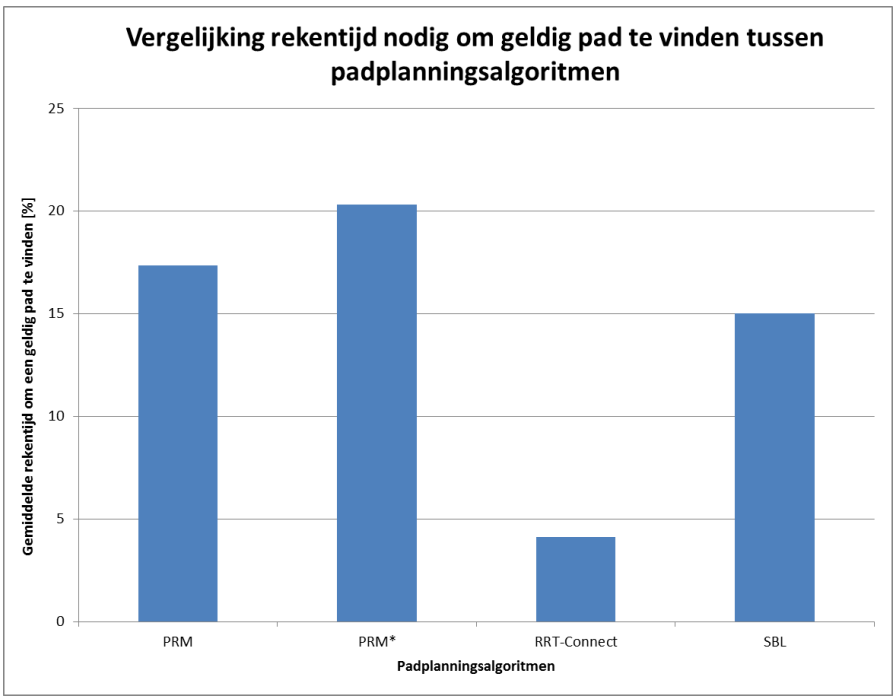
Figuur 103: Padplanningsalgoritmen i.f.v. percentage geldig pad gevonden



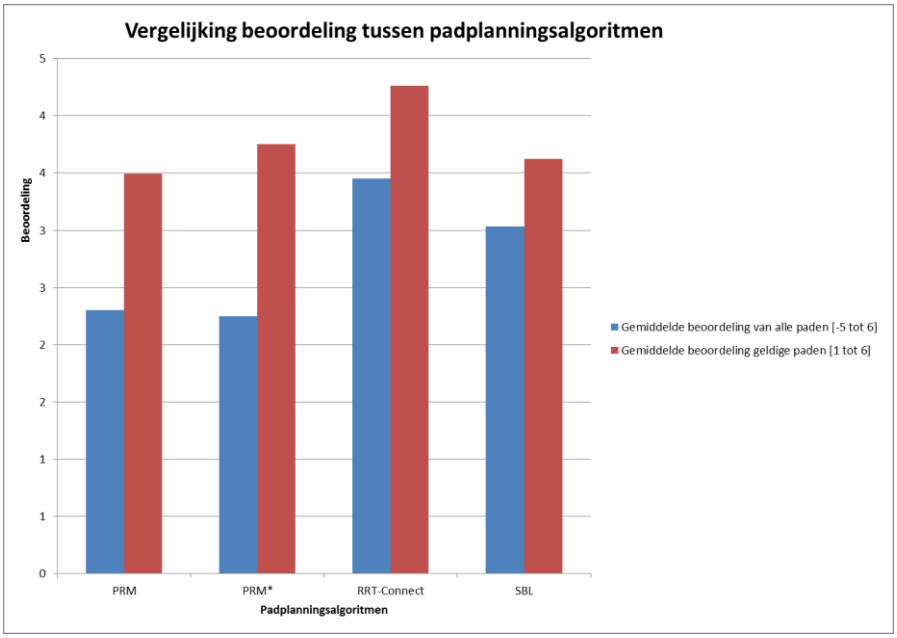
Figuur 104: Padplanningsalgoritmen i.f.v. percentage foutief pad gevonden



Figuur 105: Padplanningsalgoritmen i.f.v. percentage failed wegens een te hoge rekentijd



Figuur 106: Padplanningsalgoritmen i.f.v.gemiddelde rekentijd nodig om een geldig pad te vinden



Figuur 107: Padplanningsalgoritmen i.f.v. gemiddelde beoordeling

In figuur 103 is te zien dat SBL het vaakst een geldig pad vindt. Daarna volgen, in de juiste volgorde, RRT-Connect, PRM en PRM*. Figuur 104 illustreert dat PRM* en RRT-Connect het minst vaak een foutief pad als resultaat krijgen. PRM en SBL krijgen het vaakst een foutief pad als resultaat. Enkel PRM vindt soms geen pad wegens een te hoge rekentijd, zoals in figuur 105 te zien is. Om een pad te vinden heeft RRT-Connect gemiddeld gezien het minst tijd nodig (figuur 106). SBL, PRM en PRM* hebben, in die volgorde, een steeds hogere rekentijd nodig. Uit de grafieken blijkt dus dat RRT-Connect algemeen gezien het beste algoritme is voor deze

probleemstelling. SBL is het tweede beste algoritme om te gebruiken in deze toepassing, PRM en PRM* volgen hierna.

Nu het beste padplanningsalgoritme gevonden is, is het de vraag of dit padplanningsalgoritme voldoet aan de strenge eisen die opgesteld zijn om in de praktijk goed te kunnen werken. Deze eisen zijn eerder al eens vermeld, maar staan hieronder voor de duidelijkheid nog eens vermeld. Het padplanningsalgoritme:

- moet minstens 95% van de keren een geldig pad als resultaat geven;
- moet bijna 0% van de keren een foutief pad als resultaat geven;
- mag een maximale rekentijd van 5 seconden hebben;
- moet een gemiddelde beoordeling van 3 of meer halen voor alle paden.

RRT-Connect voldoet aan de volgende eisen:

- mag een maximale rekentijd van 5 seconden hebben (4,11 seconden);
- moet een gemiddelde beoordeling van 3 of meer halen voor alle paden (3,45);

RRT-Connect voldoet dus maar aan twee van de vier eisen, wat een teleurstellend resultaat is voor het beste algoritme in deze toepassing. SBL voldoet enkel aan de volgende eis:

- moet een gemiddelde beoordeling van 3 of meer halen voor alle paden (3,03);

SBL voldoet dus niet aan drie van de vier eisen. PRM en PRM* doen het nog slechter door aan geen enkele eis te voldoen. Uit dit onderzoek is te besluiten dat het beste padplanningsalgoritme RRT-Connect is. SBL volgt RRT-Connect als tweede beste padplanningsalgoritme voor deze probleemstelling. De verschillen tussen SBL en RRT-Connect zijn niet heel groot. PRM en PRM* volgen als derde en als vierde beste algoritme. Tussen deze twee algoritmen en de twee beste algoritmen zit wel een beduidend verschil in performantie. Toch voldoet geen enkel padplanningsalgoritmen aan de strenge eisen van de praktijk. Het is voor de algoritmen, met andere woorden, te moeilijk om van de begintoestand rechtstreeks naar de eindtoestand te gaan, die in de bak ligt.

Tabel 6 laat de meetresultaten zien van alle padplanningsalgoritmen die allemaal zijn uitgevoerd op twee verschillende *query's*.

Tabel 6: Vergelijking padplanningsalgoritmen voor twee query's

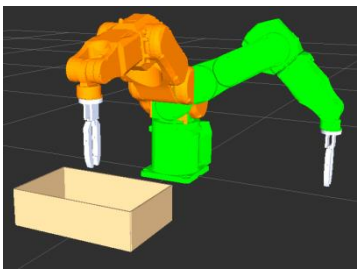
Algoritme	Query	Percentage geldig pad gevonden [%]	Percentage foutief pad gevonden [%]	Percentage failed pad gevonden [%]	Percentage failed door te lange rekentijd [%]	Gemiddelde berekeningstijd voor goed pad [s]	Gemiddeld aantal vertices dat een goed pad bevat	Gemiddelde beoordeling van alle paden [-5 tot 6]	Gemiddelde beoordeling geldige paden [1 tot 6]
PRM	1	96,67	0	3,33	0	12,84	14,45	3,43	3,55
	2	53,33	13,33	10	23,33	21,87	22,25	1,17	3,44
PRM*	1	100	0	0	0	13,27	15,47	3,33	3,33
	2	40	10	50	0	27,38	28	1,17	4,17
RRT	1	43,33	0	0	56,67	12,60	38,77	1,73	4
	2	23,33	6,67	0	70	15,06	49,57	0,77	4,71
RRT-Connect	1	96,67	0	3,33	0	1,77	5,97	4,17	4,31
	2	76,67	10	13,33	0	6,45	11,39	2,73	4,22
RRT*	1	43,33333333	0	0	56,67	60	91,54	1,67	3,85
	2	0	0	0	100	/	/	0	/
T-RRT	1	66,67	3,33	0	30	12,95	148,2	3,2	5,05
	2	3,33	0	0	96,67	32,28	369	0,2	6
EST	1	80	0	0	20	21,24	33,04	2,37	2,96
	2	3,33	3,33	0	93,33	51,45	82	-0,03	4
SBL	1	96,67	3,33	0	0	10,83	149,38	2,87	3,14
	2	83,33	10	6,67	0	19,19	246,37	3,2	4,11
KPIECE	1	76,67	0	3,33	20	24,75	43,57	2,57	3,35
	2	30	6,66	0	63,33	25,41	33,44	0,9	4,11
BKPIECE	1	50	6,67	0	43,33	38,25	68,47	1,07	2,8
	2	40	10	0	50	33,00	56,83	1,13	4,08
LBKPIECE	1	43,33	0	0	56,67	41,57	163,77	1,1	2,54
	2	10	13,33	6,67	70	46,71	153,67	-0,23	4,33

6.4 Alternatieve methode

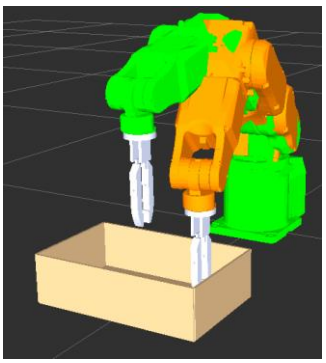
Uit het onderzoek van '6.3 Vergelijking padplanningsalgoritmen' blijkt dat zelfs de beste padplanningsalgoritmen voor deze probleemstelling af en toe een foutief pad geven of geen pad kunnen vinden. Aangezien men in de praktijk streeft naar een betrouwbaarheid van 100%, zijn voorgaande resultaten niet efficiënt genoeg. Een betrouwbaarheid van 100% betekent dat het algoritme in 100% van de gevallen een geldig pad moet vinden binnen een tijdsduur van 60 seconden. Deze 100% is een streefwaarde en moet zo goed mogelijk benaderd worden. Het percentage van aantal foutief gevonden paden moet de 0 benaderen. Aangezien de padplanningsalgoritmen niet verbeterd kunnen worden, op hun specifieke parameters na, is er een alternatief nodig om nog betere resultaten te behalen. Specifieke parameters kunnen algoritmen namelijk maar een beperkte hoeveelheid verbeteren. De enige variabele die nog aan te passen is, is het probleem. Natuurlijk mag er niet een willekeurig ander probleem gekozen worden.

Zoals in '6.3 Vergelijking padplanningsalgoritmen' geëvalueerd, geeft het uitvoeren van de probleemstelling in één *query* een beperkte betrouwbaarheid. Vandaar onderzoekt dit deel van het hoofdstuk de performantie van de padplanningalgoritmen bij het opdelen van de probleemstelling in meerdere *query's*. De eerste *query* (*query* 3) brengt de eindeffector vanuit zijn begintoestand, de plaats van het transportmechanisme, naar zijn eindtoestand, die vlak boven de opening van de bak is. Vervolgens brengt de tweede of derde *query* (*query* 4 of *query* 5) de eindeffector vanuit zijn begintoestand, vlak boven de doos, naar zijn eindtoestand, die in de doos is om een werkstuk te grijpen. Om deze alternatieve methode te kunnen vergelijken met de beste padplanningsalgoritmen van '6.3 Vergelijking padplanningsalgoritmen', iservoorgekozen om het experiment uit te voeren op twee eindtoestanden die in de bak liggen. Deze twee eindtoestanden zijn dezelfde als die van '6.3 Vergelijking padplanningsalgoritmen', namelijk aan de binnenkant van de bak en aan de buitenkant van de bak. Op die manier is de *query* van het vorige experiment opgedeeld in twee *deelquery's.* Het experiment bevat dus de volgende *query's*:

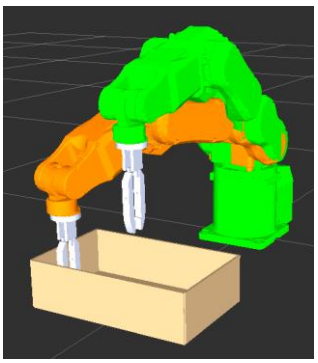
- *query* 3: van 90° verwijdert van de bak tot net boven de bak (figuur 108);
- *query* 4: van net boven de bak tot binnenkant bak (figuur 109);
- *query 5*: van net boven de bak tot buitenkant bak (figuur 110).



Figuur 108: Query 3: van 90° verwijdert van bak tot boven bak



Figuur 109: Query 5: van boven bak tot binnenkant bak

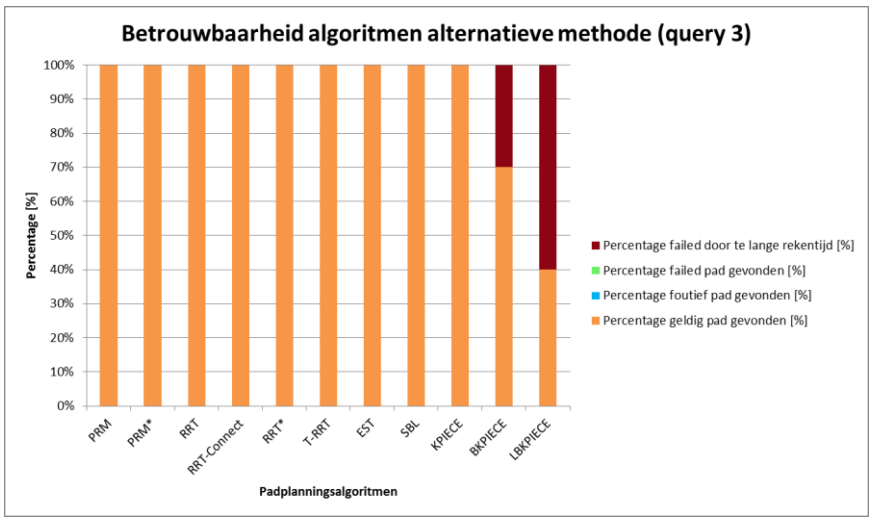


Figuur 110: Query 4: van boven bak tot buitenkant bak

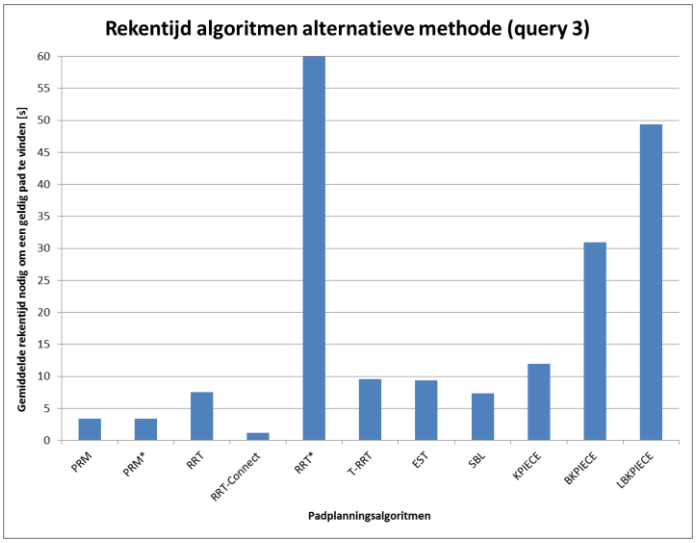
Ook hier staan alle specifieke parameters van de padplanningsalgoritmen op hun standaardwaarden om beter te kunnen vergelijken. Net als bij het vorige experiment maakt dit experiment gebruik van een bak met dunne wanden, een maximale rekentijd van 60 seconden en een *longest valid segment fraction* van 0,005 m. De resultaten van de metingen van het experiment zijn verzameld in tabel, dat zich achteraan dit deel van het hoofdstuk bevindt.

Zoals eerder vermeld moet er een combinatie worden gemaakt tussen *query 3* en *query* 4 of *query* 5. Om de resultaten beter te kunnen vergelijken iservoorgekozen om het gemiddelde te nemen van *query* 4 en *query* 5. Dat gemiddelde wordt apart beoordeeld van *query* 3 aangezien voor deze twee *query's* een ander padplanningsalgoritme gekozen mag worden. De probleemstelling is immers opgedeeld in deelproblemen. De robot kan bijvoorbeeld met behulp van KPIECE *query* 3 uitvoeren en vervolgens met SBL *query* 4 en 5 uitvoeren. Daarom wordt voor ieder deelprobleem apart bepaald wat het meest performante padplanningsalgoritme is voor dat deel. Vervolgens worden deze twee *query's* en dus ook de twee padplanningsalgoritmen gecombineerd om één actie uit te voeren, namelijk het grijpen van een werkstuk in de bak.

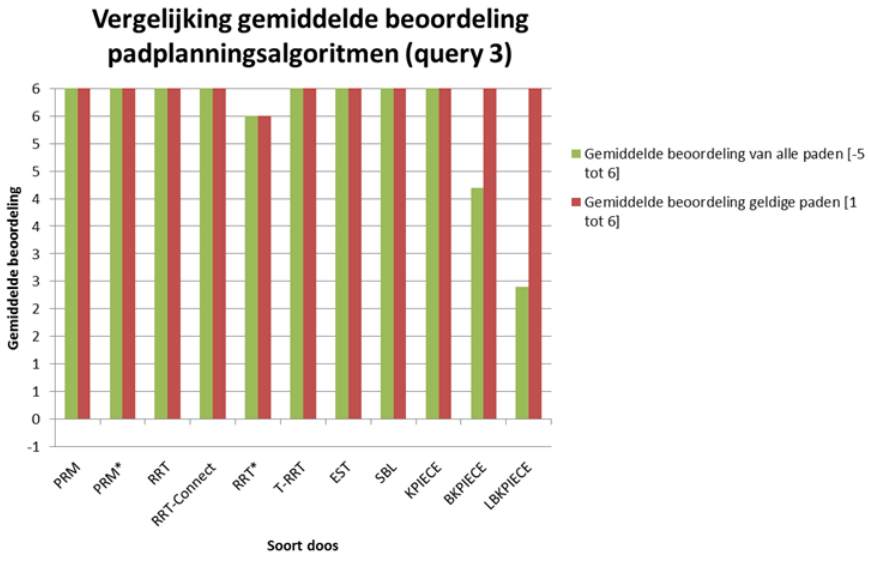
Eerst tracht dit onderzoek het meet performante padplanningsalgoritme te vinden voor *query* 3. De evaluatie gebeurt op dezelfde manier als bij de vorige onderzoeken. Onderstaande grafieken beschouwen de betrouwbaarheid van ieder algoritme (figuur 111), de rekentijd dat ieder algoritme gemiddeld nodig heeft om een geldig pad te vinden (figuur 112) en de gemiddelde beoordeling, op de benadering van het optimale pad, voor ieder algoritme (figuur 113).



Figuur 111: Padplanningsalgoritmen i.f.v. de betrouwbaarheid van een algoritme, toegepast op query 3



Figuur 112: Padplanningsalgoritmen i.f.v. de gemiddelde rekentijd nodig om een geldig pad te vinden, toegepast op query 3



Figuur 113: Padplanningsalgoritmen i.f.v. de gemiddelde beoordeling van een pad, toegepast op query 3

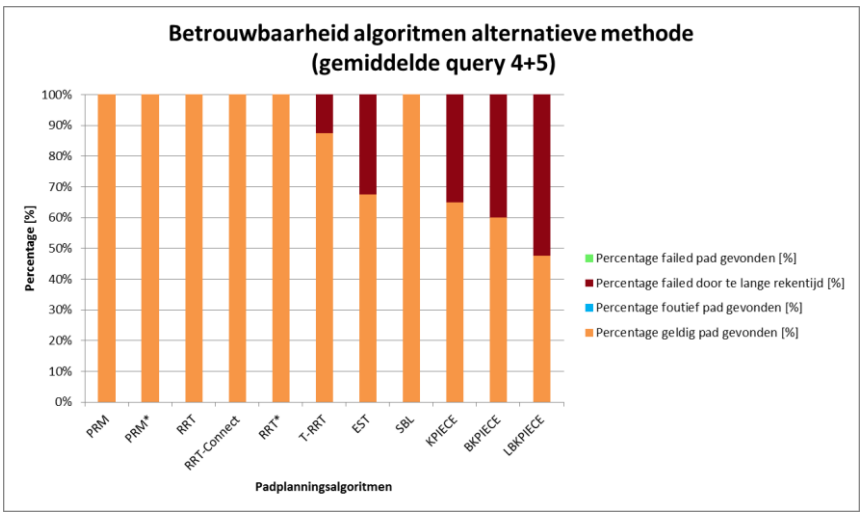
Uit de grafieken blijkt meteen dat dit deelprobleem veel makkelijker op te lossen is dan het volledige probleem in één keer. Zowel de betrouwbaarheid, als de gemiddelde rekentijd als de beoordeling ligt beduidend hoger dan in het onderzoek van '6.3 Vergelijking padplanningsalgoritmen'. Ieder algoritme vindt 100% van de keren een geldig pad, behalve BKPIECE en LBKPIECE die een behoorlijk percentage van de keren *failed* krijgt als resultaat (figuur 111). Dit is weer wegens een te hoge rekentijd wat veroorzaakt wordt door de projectie van de volledige toestandsruimte.

In figuur 112 is te zien dat enkele padplanningsalgoritmen veel te veel tijd nodig hebben om een geldig pad te vinden. RRT* gebruikt, zoals altijd, de volledige tijd die het beschikbaar krijgt. RRT* voldoet dus zeker niet aan de eisen. Ook BKPIECE en LBKPIECE hebben heel veel tijd nodig om een geldig pad te berekenen. Uit de vorige grafiek (figuur 111) bleek ook al dat deze twee algoritmen een lagere betrouwbaarheid hebben. BKPIECE en LBKPIECE zijn dus ook al twee algoritmen die niet bruikbaar zijn voor op deze *query* uit te voeren. De rest van de algoritmen doen er gemiddeld niet lang over om een geldig pad te vinden. RRT-Connect behaalt ook hier weer het beste resultaat aangezien het een veel lagere gemiddelde rekentijd heeft dan de rest. PRM en PRM* hebben ook weinig rekentijd nodig om een geldig pad te vinden.

De gemiddelde beoordeling van een pad ligt ook hoog in vergelijking met het onderzoek dat in '6.3 Vergelijking padplanningsalgoritmen' is uitgevoerd. Ieder algoritme behaalt de maximale beoordeling, behalve RRT*, BKPIECE en LBKPIECE. Deze drie algoritmen werden hiervoor reeds als niet bruikbaar beschouwd voor deze *query*. De andere algoritmen behalen allemaal het maximum van de beoordeling waardoor er geen onderscheid gemaakt kan worden.

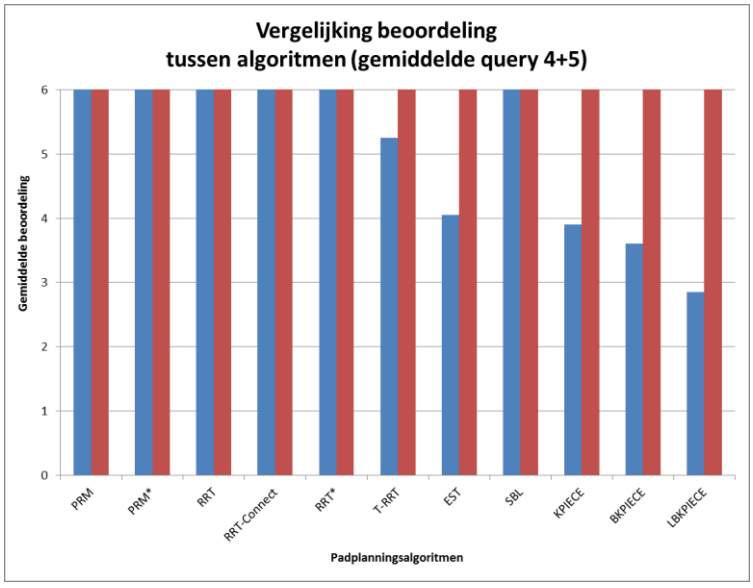
Het algoritme dat de beste performantie heeft voor deze *query* is RRT-Connect, gevolgd door PRM en PRM* en daarna SBL. Dit zijn weer de vier padplanningsalgoritmen die bij het vorige onderzoek ook het beste resultaat hadden.

De volgende grafieken trachten te achterhalen welk padplanningsalgoritme de beste performantie heeft voor *query* 4 en 5. Dit deel bepaalt dus welk algoritme het beste een pad kan berekenen dat vertrekt van net boven de bak en eindigt op verschillende plaatsen in de bak. Onderstaande grafieken beschouwen de betrouwbaarheid van ieder algoritme (figuur 114), de rekentijd dat ieder algoritme gemiddeld nodig heeft om een geldig pad te vinden (figuur 115) en de gemiddelde beoordeling, op de benadering van het optimale pad, voor ieder algoritme (figuur 116).



Figuur 114: Padplanningsalgoritmen i.f.v. de betrouwbaarheid van een algoritme, toegepast op query 4 en 5

Figuur 115: Padplanningsalgoritmen i.f.v. gemiddelde rekentijd nodig om een geldig pad te vinden, toegepast op query 4 en 5



Figuur 116: Padplanningsalgoritmen i.f.v. de gemiddelde beoordeling van een pad, toegepast op query 4 en 5

Figuur 114 illustreert dat de meeste algoritmen een heel hoge betrouwbaarheid hebben. T-RRT, EST, KPIECE, BKPIECE en LBKPIECE hebben een vrij groot percentage van de keren geen geldig pad als resultaat wegens een te hoge rekentijd. Deze algoritmen zijn daarom niet bruikbaar voor dit deelprobleem. Alle andere algoritmen vinden 100% van de keren een geldig pad.

In figuur 115 is te zien dat een deel van de algoritmen, die ook onbetrouwbaar zijn, slecht scoren voor de gemiddelde rekentijd die nodig is om een geldig pad te vinden. EST, KPIECE, BKPIECE en LBKPIECE zijn ook hier de algoritmen die als niet bruikbaar bevonden worden. RRT* gebruikt net als de andere keren de volledige 60 seconden. Indien de maximale rekentijd verlaagd wordt, haalt het algoritme slechtere resultaten dan RRT. RRT* krijgt dan namelijk niet de tijd om het pad te optimaliseren en het algoritme werkt trager dan RRT. Hierdoor is RRT* voor dit probleem nooit te verkiezen boven RRT. SBL heeft behoorlijk veel tijd nodig om een geldig pad te vinden. RRT, T-RRT en RRT-Connect hebben weinig tijd nodig om een geldig pad te vinden. De algoritmen die het snelst een geldig pad kunnen vinden, zijn PRM en PRM*.

De gemiddelde beoordeling is voor de meeste algoritmen zeer hoog (figuur 116). De algoritmen die onbetrouwbaar zijn, scoren ook hier slecht. Het gaat hier over EST, KPIECE, BKPIECE en LBKPIECE. T-RRT behaalt een iets lager resultaat dan de andere algoritmen, maar het is nog steeds hoog. Alle andere algoritmen behalen het maximum als beoordeling, wat wil zeggen dat ze iedere keer het optimale pad zeer goed benaderen.

De algoritmen die de hoogste performantie behalen voor het uitvoeren van dit deelprobleem, zijn PRM en PRM*. RRT-Connect behaalt de derde hoogste performantie en ligt dicht bij de performantie van PRM en PRM*. RRT is nog net bruikbaar om toe te passen in dit deelprobleem. SBL is net niet goed genoeg wegens een te hoge gemiddelde rekentijd die nodig is om een geldig pad te vinden.

Voor beide *query's* is onderzocht welke padplanningsalgoritmen het beste zijn. Maar voldoen deze algoritmen aan de vooropgestelde eisen? Aangezien de probleemstelling nu bestaat uit deelproblemen zijn de strenge eisen, om goed genoeg te zijn voor in de praktijk, anders. Indien de robot 95% van de keren een juist pad moet vinden vanaf het transportmechanisme tot een werkstuk in de bak, moeten niet beide *query's* een percentage van 95% halen. Wiskundig gezien is het namelijk zo, dat beide *query's* minstens een percentage halen van $\sqrt{0{,}95}$ = 97,5%. Dat wil zeggen dat een algoritme zowel bij *query* 3 als bij het gemiddelde van *query* 4 en 5, minstens 97,5% van de keren een geldig pad moet vinden om te voldoen aan de eisen. Deze eis is wel de enige die verandert. Beide *query's* moeten ongeveer 0% halen voor 'het percentage foutief pad gevonden'. De totale gemiddelde rekentijd moet onder de 5 seconden liggen. Hiervoor moet de gemiddelde rekentijd van beide *query's* opgeteld worden. Ook moet de gemiddelde beoordeling, op alle paden en niet enkel de geldige paden, weer minstens de waarde 3 bedragen. Om aan deze eis te voldoen moet het gemiddelde van de beoordelingen van *query* 3 en *query* 4+5, meer als 3 zijn.

De algoritmen die voor *query* 3 de hoogste performantie halen zijn (in de juiste volgorde):

- RRT-Connect,
- PRM*,
- PRM,
- SBL,
- RRT.

SBL en RRT hebben echter gemiddeld meer dan 5 seconden nodig om een geldig pad te vinden waardoor deze algoritmen sowieso niet aan de eisen voldoen. De algoritmen die voor *query* 4+5 de hoogste performantie halen zijn (in de juiste volgorde):

- PRM,
- PRM*,
- RRT-Connect,
- RRT,
- T-RRT.

RRT heeft gemiddeld meer dan 5 seconden nodig om een geldig pad te vinden waardoor dit algoritme niet voldoet aan de eisen. T-RRT vindt daarentegen maar 88% van de keren een geldig pad waardoor dit algoritme ook niet kan voldoen aan de eisen.

Het is dus mogelijk om een combinatie te maken tussen drie padplanningsalgoritmen (PRM, PRM* en RRT-Connect) voor beide deelproblemen. Aangezien alle drie de algoritmen 100% van de keren een geldig pad vinden en allemaal een maximale beoordeling hebben voor beide deelproblemen, moet er enkel getest worden of de totale gemiddelde rekentijd binnen de 5 seconden ligt. Tabel 7 illustreert de resultaten voor de totale rekentijd die nodig is om een pad te berekenen voor beide deelproblemen.

Tabel 7: Resultaten totale rekentijd nodig om probleemstelling op te lossen

Algoritmen		
Query 3	Query 4+5	Totale gemiddelde rekentijd nodig om een geldig pad te vinden [s]
PRM	PRM*	4,86
	PRM*	4,89
	RRT-Connect	6,10
PRM*	PRM	4,85
	PRM*	4,89
	RRT-Connect	6,10
RRT-Connect	PRM	2,66
	PRM*	2,69
	RRT-Connect	3,90

De resultaten van tabel 7 tonen aan dat het gebruik van RRT-Connect voor het eerste deelprobleem (figuur 108) en PRM voor het tweede deelprobleem (gemiddelde van de *query's* van figuren 109 en 110) het minste rekentijd nodig heeft. Dat wil ook zeggen dat deze combinatie voldoet aan alle eisen. De andere combinaties van tabel 7 voldoen ook aan de eisen op twee combinaties na.

Theoretisch gezien zou het beter zijn om voor *query* 3 PRM of PRM* te gebruiken en voor *query* 4+5 RRT-Connect. Dit zijn nochtans de combinaties van de tabel die als enige niet voldoen aan de eis om onder de 5 seconden te blijven. Deze combinatie is theoretisch gezien een logische combinatie aangezien *query* 3 vaak geen of weinig obstakels tegenkomt. En de obstakels die het tegenkomt zijn statisch en veranderen dus niet. PRM en PRM* hebben hier een groot voordeel aangezien het een statische omgeving is. De wegenkaart kan namelijk hergebruikt worden. In simulatie houdt het algoritme deze wegenkaart niet bij, waardoor PRM en PRM* een lagere rekentijd krijgen bij *query* 3. Het bijhouden van de wegenkaart in simulatie is daarom een mogelijke verbetering in deze simulatieomgeving ('7.2.4 Opslaan wegenkaart'). RRT-Connect is daarentegen een algoritme dat vooral bruikbaar is in dynamische omgevingen zoals de bak, aangezien de obstakels (de werkstukken) daar continu van plaats veranderen. Het gebruik van PRM en PRM* zou dus theoretisch gezien geen voordeel opleveren hier. RRT-Connect zoekt daarentegen sneller een oplossing. Dat het algoritme gebruikmaakt van twee boomstructuren is een extra voordeel aangezien er rond de obstakels in de bak al vanaf het begin een structuur is, namelijk de structuur die vertrekt van de eindtoestand. Aangezien deze experimenten onderzocht zijn in simulatie kan het voorvallen dat de metingen gedeeltelijk afwijken in de praktijk. Hetzelfde onderzoek uitvoeren op een robot, en niet in simulatie, is daarom een logisch vervolg op dit eindwerk ('7.2.1 Communicatie met testrobot').

Tabel 8 laat de meetresultaten zien van alle padplanningsalgoritmen voor het uitvoeren van de alternatieve methode dat uit drie *query's* bestaat. Om een werkstuk te grijpen moet de robot twee *query's* combineren namelijk *query* 3 en *query* 4 of *query* 3 en *query* 5.

Tabel 8: Vergelijking padplanningsalgoritmen alternatieve methode voor drie query's

Algoritme	Query	Percentage geldig pad gevonden [%]	Percentage foutief pad gevonden [%]	Percentage failed pad gevonden [%]	Percentage failed door te lange rekentijd [%]	Gemiddelde berekeningstijd voor goed pad [s]	Gemiddeld aantal vertices dat een goed pad bevat	Gemiddelde beoordeling van alle paden [-5 tot 6]	Gemiddelde beoordeling geldige paden [1 tot 6]
PRM	3	100	0	0	0	3,38	2	6	6
	4	100	0	0	0	1,73	2	6	6
	5	100	0	0	0	1,24	2	6	6
PRM*	3	100	0	0	0	3,37	2	6	6
	4	100	0	0	0	1,60	2	6	6
	5	100	0	0	0	1,43	2	6	6
RRT	3	100	0	0	0	7,53	22,90	6	6
	4	100	0	0	0	4,94	15,75	6	6
	5	100	0	0	0	5,55	16,80	6	6
RRT-Connect	3	100	0	0	0	1,18	5,1	6	6
	4	100	0	0	0	1,66	5,55	6	6
	5	100	0	0	0	3,79	8,35	6	6
RRT*	3	100	0	0	0	60,00	95,20	5,5	5,5
	4	100	0	0	0	60,00	90,1	6	6
	5	100	0	0	0	60,00	88,00	6	6
T-RRT	3	100	0	0	0	9,56	111,20	6	6
	4	95	0	0	5	3,27	39,21	5,7	6
	5	80	0	0	20	3,78	44,88	4,8	6
EST	3	100	0	0	0	9,39	16,55	6	6
	4	95	0	0	5	16,62	26,79	5,7	6
	5	40	0	0	60	30,71	49,25	2,4	6
SBL	3	100	0	0	0	7,33	111,75	6	6
	4	100	0	0	0	10,59	152,45	6	6
	5	100	0	0	0	13,96	182,15	6	6
KPIECE	3	100	0	0	0	11,90	21,1	6	6
	4	90	0	0	10	23,36	40,78	5,4	6
	5	40	0	0	60	20,44	35,63	2,4	6
BKPIECE	3	70	0	0	30	30,92	59,14	4,2	6
	4	65	0	0	35	36,93	69,23	3,9	6
	5	55	0	0	45	36,42	67,09	3,3	6
LBKPIECE	3	40	0	0	60	49,37	144,38	2,4	6
	4	50	0	0	50	33,95	141,5	3	6
	5	45	0	0	55	42,41	126	2,7	6

7 Conclusie en toekomstig werk

7.1 Conclusie

Uit het onderzoek is gebleken dat de botsingsdetectieresolutie een grote invloed heeft op de betrouwbaarheid van een algoritme. Een hogere botsingsdetectieresolutie en dus een lagere *longest valid segment fraction* zorgt ervoor dat een algoritme vaker een geldig pad als resultaat krijgt. Ook zorgt het verhogen van de botsingsdetectieresolutie ervoor dat het padplanningsalgoritme minder vaak een foutief pad berekent. Een foutief pad bekomen, is het slechtste wat kan voorvallen bij het zoeken naar een pad. Hiertegenover staat het nadeel van een hogere rekentijd. De voordelen heffen dit nadeel op. De experimenten bewijzen dit ook aangezien een hoge botsingsdetectieresolutie overeenkomt met betere resultaten.

Indien het *longest valid segment fraction* een lagere waarde (hogere botsingsdetectieresolutie) aanneemt dan de wanddikte van een doos, dan treedt er geen verbetering meer op. De betrouwbaarheid stijgt verder, maar het stijgt heel miniem terwijl de gemiddelde rekentijd, om een geldig pad te vinden, zeer snel stijgt. Het aanpassen van de botsingsdetectieresolutie op de wanddikte van de bak is dus de optimale oplossing. Dat zo instellen is zeer belangrijk aangezien de betrouwbaarheid sterk stijgt en het percentage van foutieve paden sterk daalt.

Een tweede onderzoek heeft aangetoond welk padplanningsalgoritme de hoogste performantie heeft bij het oplossen van de probleemstelling. Deze probleemstelling is het leegmaken van een bak met willekeurig geplaatste werkstukken. RRT-Connect behaalt de hoogste performantie, wat overeenkomt met de hoogste betrouwbaarheid, de laagste rekentijd, en als beste het optimale pad benaderen. SBL, PRM en PRM* zijn de enige drie padplanningsalgoritmen die, naast RRT-Connect, ook een hoge performantie behalen bij deze probleemstelling. Nochtans voldoet geen enkel algoritme aan de vooropgestelde eisen. Om aan alle eisen te voldoen moet het padplanningsalgoritme:

- minstens 95% van de keren een geldig pad als resultaat geven;
- ongeveer 0% van de keren een foutief pad als resultaat geven;
- een gemiddelde rekentijd hebben die minder is dan 5 seconden;
- een gemiddelde beoordeling van 3 of meer halen voor alle paden.

Aangezien er geen enkel padplanningsalgoritme bij het tweede onderzoek voldoet aan de vooropgestelde eisen, iservoorhet derde onderzoek gezocht naar een alternatief. Bij dit alternatief is de probleemstelling opgedeeld in twee kleinere problemen. In plaats van rechtstreeks van een begintoestand naar een eindtoestand te gaan in de bak, deelt dit alternatief het pad op in twee delen. In het eerste deel moet een algoritme een pad berekenen van de begintoestand naar de eindtoestand die vlak boven de bak is. Vervolgens moet een

padplanningsalgoritme een pad berekenen die van vlak boven de bak tot in de bak gaat. Deze twee padplanningsalgoritmen moeten niet dezelfde zijn. Daardoor kan voor elk deel een algoritme gekozen worden dat voor dat deel goede resultaten behaalt. De alternatieve methode is makkelijker te berekenen aangezien er geen obstakel in de weg ligt tussen de begin- en de eindtoestand.

Uit het onderzoek blijkt dat het algoritme RRT-Connect de hoogste performantie heeft voor het eerste deelprobleem. PRM* en PRM zijn twee andere algoritmen die ook een hoge performantie halen voor dat deelprobleem. Voor het tweede deelprobleem hebben PRM en PRM* de hoogste performantie. RRT-Connect is het padplanningsalgoritme met de derde hoogste performantie voor dit deelprobleem. Deze drie algoritmen behaalden bij het vorige onderzoek ook de beste resultaten. Aangezien de volledige probleemstelling opgelost moet worden, moeten de deelproblemen gecombineerd worden tot een geheel.

Uit de experimenten blijkt dat de combinatie RRT-Connect en PRM de hoogste performantie behaalt. Het eerste deel van het probleem wordt berekend door RRT-Connect, terwijl het tweede deel berekend wordt door PRM. Deze combinatie voldoet vlot aan alle eisen. Er zijn nog acht andere mogelijke combinaties van algoritmen mogelijk. Zes van deze combinaties voldoen ook aan alle eisen. Enkel de combinatie PRM en vervolgens RRT-Connect en de combinatie PRM* en vervolgens RRT-Connect voldoen niet aan alle eisen.

Hieruit is te concluderen dat de alternatieve methode, namelijk het opdelen van de probleemstelling in deelproblemen, de beste methode is. Om de probleemstelling dus optimaal uit te voeren moet de botsingsdetectieresolutie aangepast zijn aan de wanddikte van de bak en moet het probleem opgedeeld worden in twee deelproblemen. Het eerste deelprobleem wordt het beste uitgevoerd door middel van RRT-Connect en het tweede deelprobleem wordt het beste uitgevoerd door middel van PRM.

7.2 Toekomstig werk

De intentie van de masterproef was om de padplanningsalgoritmen rechtsreeks op de Epson-robot toe te passen. Het streefdoel was om tegen het begin van het tweede semester de robot volledig in simulatie te hebben. Dat streefdoel is niet gehaald wegens een groot technische probleem. De programma's die de Epson-robot visualiseren en de padplanningsalgoritmen hierop toepassen, zijn te zwaar voor de laptops die ter beschikking waren op dat moment. Vooral de belasting van de grafische kaart is heel groot bij het visualiseren van de robot in Rviz.

De problemen met de grafische kaart zijn als eerste opgemerkt tijdens enkele testen in de grafische interface van OMPL. Hierin plannen de algoritmen een pad voor objecten die een soort hindernisparcour moeten afleggen. Doordat de grafische kaart van de aanwezige laptops onvoldoende sterk was, was het grootste deel van het eerste semester een literatuurstudie en kon nog geen praktisch werk van de masterproef gebeuren. Ook het onderdeel voor het aanmaken van een geldig URDF-bestand in *SolidWorks* was niet mogelijk omdat het de verificatie van het URDF-bestand moet gebeuren met het simulatieprogramma Rviz. Hierdoor was het onmogelijk de eerste deadline te halen en de robot volledig in simulatie te hebben tegen het begin van het tweede semester.

Doordat er een zeer uitgebreide literatuurstudie vooraf is gegaan aan de uiteindelijke masterproef, was de materie hiervan goed gekend. Ook is er al onderzoek gepleegd naar toekomstig werk dat voor deze masterproef. In de volgende paragrafen volgt een bespreking over mogelijke vervolgen van deze masterproef.

7.2.1 Communicatie met testrobot

Rviz dient enkel als visualisatieprogramma van de gesimuleerde robot. In Rviz gebeuren ook de berekeningen en de padplanningen indien deze nodig zijn. Echter moet de aansturing van de motoren ook gebeuren vanuit Rviz om zo de echte robot te bewegen. Het is nog niet mogelijk om de motorsturingen rechtstreeks uit Rviz op de echte robot toe te passen. Een tussenstap hiervoor is de simulatieomgeving gazebo.

In gazebo is het mogelijk eender welke omgeving te tekenen en weer te geven. Zo is het mogelijk om ook de omgeving van de Epson hierin na te bootsen. De Epson-robot is ook oproepbaar in deze omgeving indien er een SDF-bestand van deze robot bestaat. Een SDF-bestand is, ne t zoals het URDF-bestand, een XML gebaseerd bestand. Het is mogelijk om een SDF-bestand te genereren van een URDF-bestand. Dit komt omwille van het relatief klein verschil tussen URDF en SDF en omwille van de aanwezigheid van alle informatie in het URDF-bestand dat nodig is in

een SDF-bestand. Eens er een SDF-bestand van de robot gemaakt is, is het mogelijk deze te openen in gazebo. [44]

In gazebo bestaat een zogenoemde *fysics engine*. Deze bootst de werkelijke wereld zo nauwkeurig mogelijk na. Het belangrijkste onderdeel van zo een *fysics engine* is dat deze de zwaartekracht in rekening brengt. Dit wil zeggen dat Bij het oproepen van de Epson-robot in gazebo deze na enkele seconden in elkaar zakt vanwege de gesimuleerde zwaartekracht. Het is opgevallen dat de robot zelfs kan kantelen in die situaties, vanwege dat probleem is volgende regel aan het URDF-bestand toegevoegd:

```
<link name="world"/>
 <joint name="fixed" type="fixed">
  <parent link="world"/>
  <child link="base_link"/>
 </joint>
```

Dit is een virtuele link die de base_link van de robot verbindt met het *world frame*. Hierdoor kan de robot niet meer kantelen en is de basis vast verankerd met de grond in gazebo.

Om te vermijden dat de robot in elkaar zakt in gazebo moet er een connectie ontstaan tussen gazebo en Rviz. Deze connectie zorgt dat de robot in gazebo de bewegingen volgt van het robotmodel in Rviz. Dit gebeurt op de volgende manier: de uitgevoerde beweging van de robot in Rviz wordt opgeslagen als een *message* en wordt naar een bepaald topic geschreven. Dit gebeurt op basis van de *node* 'move_group'. In gazebo is het vervolgens mogelijk dit topic uit te lezen en deze toe te passen op de Epson-robot hierin. Er moeten wel nog enkele aanpassingen gebeuren in de *launch-files* van de Epson-robot, omdat in deze nog geen rekening is gehouden met belangrijke kenmerken van de robot:

- De maximale snelheid van de onderdelen;
- De maximale versnellingen van de onderdelen;
- De PID regelaars voor de aansturing van de motoren.

Het is echter nodig om ook de juiste *hardware* van de gebruikte motoren en eventuele sensoren toe te voegen in gazebo. Het aantal *drivers* voor deze motoren en sensoren die compatibel zijn in gazebo is nog redelijk beperkt en deze zijn terug te vinden op de site van gazebo of van ROS. [45] De ondersteunde camera's zijn tot nu toe enkel stereo camera's. De camera die nu op ACRO aanwezig is, staat nog niet tussen deze *drivers* waardoor er geen simulatie mogelijk is in gazebo. Hiervoor moet een gelijkaardige camera gebruikt worden in gazebo.

De communicatie met een camera in gazebo en Rviz is wederom mogelijk gemaakt door *ROS_messages*. Om precies te zijn, zijn dat de *sensor_msgs*. Via een *plugin* in Rviz is het mogelijk deze beelden uit te lezen indien het juiste *topic* gekozen is.

Om de volledige werking van gazebo met URDF-bestanden te kennen, is het aangeraden de handleidingen op de site van gazebo te overlopen.

7.2.2 Puntenwolk openen in Rviz

Een connectie met een camera is enkel mogelijk indien deze *drivers* compatibel zijn ROS. De beelden die van de camera in een *message* zijn opgeslagen worden achteraf gevisualiseerd in Rviz. Een mogelijk optie om al testen te kunnen uitvoeren in Rviz is om een manier te vinden om één enkele 3D-puntenwolk te openen naar Rviz. Deze 3D-puntenwolk is eventueel genereerbaar met het beeldverwerkingsprogramma halcon. Halcon genereert puntenwolken aan de hand van zes verschillende foto's met steeds een verschillende grijswaarde. Er moet een manier gevonden worden om deze zes beelden om te vormen tot één bestand dat compatibel is met ROS. Een lijst van bestandstypen en de manier om puntenwolken te publiceren met *messages* is terug te vinden op de site van ROS. [46]

7.2.3 Benchmarking

In paragraaf 6.1 en 6.2 van deze scriptie staan de invloeden van zowel de algemene alsook de specifieke parameters van planners vermeld. Hierin staat echter niet de ideale waarde van een parameter omdat deze afhankelijk is van de gesimuleerde situatie. Indien de *benchmarking tool* van ROS ontwikkeld is, tegen dat er veder onderzoek wordt gedaan op deze masterproef, kan er op deze manier veel eenvoudiger gecontroleerd worden wat de resultaten zijn van een bepaalde waarde van een parameter. Deze *benchmarking tool* is nog niet afgewerkt tegen het einde van de masterproef, waardoor het niet mogelijk was dat te testen. Met deze *benchmarking tool* is het mogelijk het aantal simulaties in te geven alsook de te testen planners en de begin- en eindtoestand. De resultaten daarvan kunnen automatisch uit het log-bestand omgezet worden naar grafieken zodat de resultaten meteen gevisualiseerd zijn. Op basis hiervan is het mogelijk om besluiten of conclusies te formuleren.

7.2.4 Opslaan wegenkaart

Theoretisch duurt de eerst padplanning met het wegenkaartalgoritme PRM het langste. De daarop volgende berekeningen in dezelfde omgeving zouden relatief kort moeten zijn in vergelijking met de eerste berekening. Dit komt omdat bij de eerst padplanning ook nog de volledige *roadmap* wordt opgeslagen. De volgende padplanningen maken gebruik van deze *roadmap* en moeten er dus zelf geen meer opstellen waardoor de simulatietijd daarvan lager is. Bij *MotionPlanner plugin* van Rviz, die gebruik maakt van OMPL, is dit echter niet het geval. In de code van de PRM planner is opgevallen dat steeds een nieuwe *roadmap* wordt aangemaakt bij een nieuwe padplanning, zelfs als de omgeving en de situatie dezelfde zijn. De performantie van PRM zal beduidend beter zijn indien deze *roadmap* kan opgeslagen worden voor bepaalde scenario's, zodat deze later weer kan opgeroepen worden. Het is dus aan te raden om de code te bekijken van de PRM planner. (Deze is te vinden in de volgende *directory*: Home → padplanningyaj → omplapp → ompl → src → ompl → geometric → planners → prm)

7.2.5 Planning pipeline

Uit de conclusie van de masterproef blijkt dat een combinatie van verschillende planners de ideale oplossing biedt. Dat is in verschillende stappen uitgevoerd in Rviz tijdens de simulaties. De bedoeling zou moeten zijn om maar één keer op de knop 'plan' te klikken zodat de volledige beweging dan wordt uitgevoerd. Dat is mogelijk zijn in ROS door een *planning pipeline* op te stellen die dan doorlopen wordt door Rviz. Zo een *planning pipeline* kan meerdere *query's* bevatten die worden uitgevoerd met verschillende planners. Eventueel kan het hierin ook mogelijk zijn om tussenpunten mee te geven die afgegaan moeten worden. De juiste benaming hiervan in ROS zijn de *trajectory* of *path constraints.*

Literatuurlijst

[1] Vision and Robotics, „EPSON komt met compacte 6-assige robot | Vision and Robotics," Vision and Robotics, 10 September 2009. [Online]. Available: http://visionandrobotics.nl/2009/09/10/epson-komt-met-compacte-6-assige-robot/#more-741. [Geopend 17 Mei 2014].

[2] S. M. LaValle, "Planning algorithms," Cambridge university press, University of Illinois, 2006.

[3] S. A. López, R. Zapata and M. A. O. Lama, "Sampling-Based Motion Planning: a survey," University of Puebla, 2008.

[4] E. Demeester, Artist, *Configuration space.* [Art]. KHLim, 2013.

[5] Kavraki Lab Rice University, "Open Motion Planning Library: A Primer," Rice University, Houston, 2013.

[6] J. P., T. F. en T. C., „Robot Motion Planning and Control," Centre National de la Recherche Scientifique, Barcelona, 1998.

[7] H. Faure, „Van der Corput sequences towards general (0,1)-sequences in base b," Journal de Théorie des Nombres de Bordeaux, Bordeaux, 2007.

[8] T.-T. Wong, W.-S. Luk en P.-A. Heng, „Sampling with Hammersley and Halton Points," Hong Kong.

[9] S. Karaman en E. Frazzoli, „Sampling-based Algorithms for Optimal Motion Planning," International Journal of Robotics Research, Cambridge.

[10] Wolfram MathWorld, „Voronoi Diagram," Wolfram Mathworld, 2014. [Online]. Available: http://mathworld.wolfram.com/VoronoiDiagram.html.

[11] A. Yershova, S. M. LaValle en J. C. Mitchell, „Generating Uniform Incremental Grids on SO(3) Using the Hopf Fibration," Illinois.

[12] S. R. Lindemann, A. Yershova en S. M. LaValle, „Incremental Grid Sampling Strategies in Robotics," Illinois.

[13] R. Geraerts en H. M. Overmars, „A Comparative Study of Probabilistic Roadmap Planners," Utrecht.

[14] L. E. Kavraki, P. Svetska, J.-C. Latombe en O. M.H., „Probabilistic roadmaps for path planning in high-dimensional configuration spaces," IEEE Trans. on Robotics and Automation, Stanford, 1996.

[15] M. S. LaValle, „Rapidly-Exploring Random Trees: A New Tool for Path Planning," Iowa.

[16] J. J. Kuffner en L. M. Steven, „Rapidly-Exploring Random Trees: Progress and Prospects," Iowa; Tokyo.

[17] S. Rodriguez, X. Tang, J.-M. Lien en M. N. Amato, „An Obstacle-Based Rapidly-Exploring Random Tree".

[18] M. Jordan en A. Perez, „Optimal Bidirectional Rapidly-Exploring Random Trees," Cambridge, 2013.

[19] J. J. Kuffner en L. S. M., „RRT-connect: An efficient approach to single-query path planning," Stanford.

[20] D. Brandt, „Comparison of A and RRT-Connect Motion Planning Techniques for Self-Reconfiguration Planning," IEEE, Beijing, 2006.

[21] L. Jaillet, C. Juan en T. Siméon, „Transition-based RRT for Path Planning in Continious Cos Spaces," IEEE/RSJ, Nice, 2008.

[22] S. Karaman, „RRT* algorithm illustrative example," [Online]. Available: http://www.youtube.com/watch?v=YKiQTJpPFkA. [Geopend 03 2014].

[23] D. Hsu, J.-C. Latombe en R. Motwani, „Path Planning in Expansive Configuration Spaces," World Scientific Publishing Company, 1999.

[24] G. Sanchez en J.-C. Latombe, „A Single-Query Bi-Directional Probabilistic Roadmap Planner with Lazy Collision Checking," Springer Berlin Heidelberg, Berlin, 2003.

[25] Kavraki Lab, „Kinodynamic Planning by Interior-Exterior Cell Exploration (KPIECE)," Houston.

[26] A. I. Sucan en E. L. Kavraki, „Kinodynamic Motion Planning by Interior-Exterior Cell Exploration".

[27] R. Bohlin en Kavraki, „Path planning using lazy PRM," IEEE, San Fransisco, 2000.

[28] „About Willow Garage," Willow Garage, 2014. [Online]. Available: http://www.willowgarage.com/pages/about-us.

[29] „Messages," Willow Garage, [Online]. Available: http://wiki.ros.org/Messages.

[30] „Nodes," Willow Garage, [Online]. Available: http://wiki.ros.org/Nodes.

[31] I. Sucan, „urdf," Willow Garage, februari 2014. [Online]. Available: http://wiki.ros.org/urdf.

[32] „Create your own urdf file," Willow Garage, januari 2014. [Online]. Available: http://wiki.ros.org/urdf/Tutorials/Create%20your%20own%20urdf%20file.

[33] „Robot for research and innovation," Willow Garage, 2014. [Online]. Available: http://www.willowgarage.com/pages/pr2/overview.

[34] D. Hershberger, D. Gossow en J. Faust, „rviz," Willow Garage, februari 2014. [Online]. Available: http://wiki.ros.org/rviz.

[35] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss en W. Burgard, „OctoMap: An Efficient Probabilistic 3D Mapping Framework Based," [Online]. Available: http://octomap.github.io/.

[36] I. A. Sucan, M. Moll en L. E. Kavraki, „The open Motion Planning Library," December 2012. [Online]. Available: http://ompl.kavrakilab.org.

[37] „Concepts," MoveIt!, [Online]. Available: http://moveit.ros.org/documentation/concepts/.

[38] „MoveIt setup assistant quick guide," MoveIt!, [Online]. Available: http://moveit.ros.org/wiki/PR2/Setup_Assistant/Quick_Start.

[39] I. A. Sucan en S. Chitta, „MoveIt!," [Online]. Available: http://moveit.ros.org.

[40] S. Brawner, „Solidworks to URDF exporter," 2014. [Online]. Available: http://wiki.ros.org/sw_urdf_exporter.

[41] „Kinematics/configuration," [Online]. Available: http://moveit.ros.org/wiki/Kinematics/Configuration.

[42] I. A. sucan, „non default OMPL params," juni 2013. [Online]. Available: https://groups.google.com/forum/#!topic/moveit-users/m89OnoMJiSk.

[43] I. A. Sucan, „Member Function Documentation," [Online]. Available: http://ompl.kavrakilab.org/classompl_1_1geometric_1_1RRT.html#gRRT. [Geopend 2014].

[44] „Using a URDF in gazebo," Gazebo, 2014. [Online]. Available: http://gazebosim.org/wiki/Tutorials/1.9/Using_A_URDF_In_Gazebo.

[45] „Sensors supported by ROS," Willow Garage, 2014. [Online]. Available: http://wiki.ros.org/Sensors.

[46] W. Woodall en J. Kammerl, „PCL overview," Point Cloud Library, Willow Garage, 2014. [Online]. Available: http://wiki.ros.org/pcl/Overview.

[47] „Epson C3 Compact 6-axis robots, Downloads," Epson, [Online]. Available: http://robots.epson.com/product-detail/10.

Bijlagen

Bijlage A: SolidWorks to URDF exporter

Urdf plugin downloaden

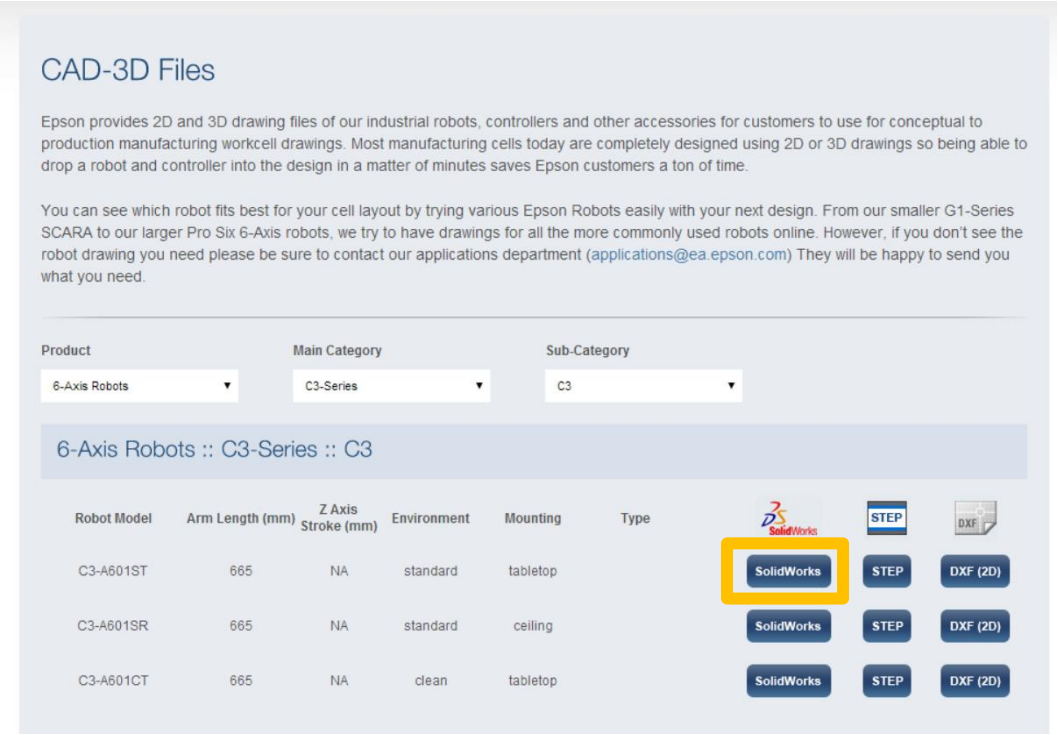
Als eerste moet de *plugin* zelf gedownload worden. Eens dat deze is geïnstalleerd zou deze automatisch aangevinkt moeten zijn als *plugin* van *SolidWorks*. [40]

SolidWorks

Professionele versie van *SolidWorks* is nodig(liefst 2012, maar met 2013 is het ook gelukt). Het werkt niet met de studentenversie omdat bij de studentenversie geen mogelijkheid bestaat om *solidparts* te converteren van bestanden van het type .stl. Deze bestandstypen worden gebruikt als *meshes* die worden opgeroepen in de URDF-bestanden om de robot te visualiseren.

CAD model Epson-robot

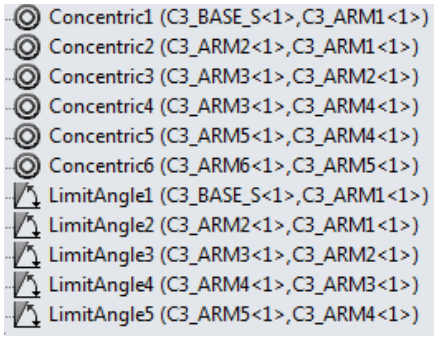
1. Download de *SolidWorks* bestanden van de Epson-site (figuur 117). [47]



Figuur 117: screenshot van de juiste locatie om de CAD-bestanden te downloaden

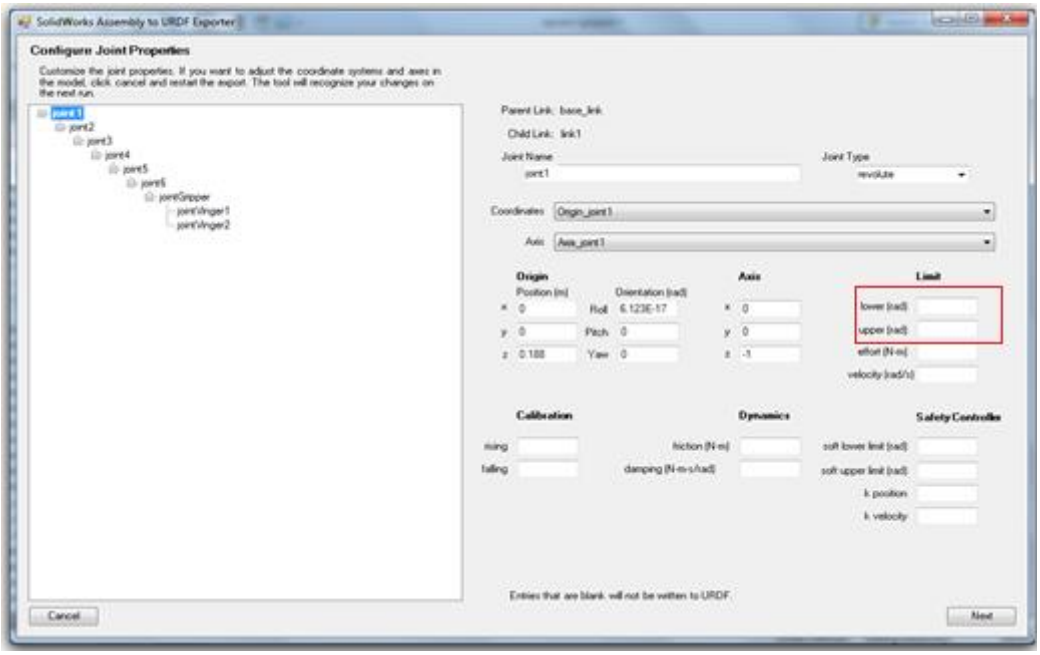
2. Dit model is niet volledig bruikbaar: op 1 onderdeel staat tekst, namelijk het onderdeel C3_arm4, dit onderdeel was gemodelleerd met oppervlakken in plaats van met volumes. Dit werd op een forum geplaatst: **https://forum.*SolidWorks*.com/message/412078**
3. Hierop kwam als antwoord een aangepast model gemodelleerd als *solid*.
4. Op dit moment zijn alle *mates* vast(*fixed*). Verwijder dit onderdeel uit de directory (C3_arm4) en vervang het door het nieuwe *solid* model dat exacte dezelfde naam moet hebben.
5. Neem voor het nieuwe onderdeel de *mates* over (concentriciteit, afstand tussen 2 vlakken – die laatste is uiteindelijk ook verwijderd). Ook de *limitAngle* is toegevoegd voor de eerste 5 links (de hoekrotaties opgeven). De laatste link kan 360 graden naar links en naar rechts roteren, daarvoor hoeft geen *limitAngle* toegevoegdte worden. Dit wordt ingegeven verderop in de *URDF export plugin* (bij aanduiden van assen als "*revolute*" en het instellen van de aslimieten).
    a. Geeft fouten in *mates* (rood) → best sommige onderdelen onzichtbaar maken om de *constraints* beter te zien; *constraints* verwijderen, juiste vlak aanduiden of cirkel aanduiden voor mate *constraints;*
    b. De blijven doen totdat er geen fouten meer optreden;
    c. URDF *plugin* een eerste maal laten lopen. Initieel is er nog geen assenstelsel gegeven. Met de URDF *exporter* genereert die assenstelsels;
    d. Beweegbaar maken: verwijder mates j1 tot j6;
    e. Robot in nul-positie zetten (dit gaat makkelijk dankzij de *limitAngle mates* waar je de gewenste hoek, dus 0°, kan ingeven);
    f. Alle *distance mates* verwijderen,
    g. Exporteren: aanduiden welke onderdelen welke *link* hebben en de *links* en *joints* benoemen (*reference frames* worden nog automatisch benoemd);
    h. *Limitangles* toevoegen (indien dit nog niet gebeurd is)

Dit (figuur 118) zijn de *mates* die je dan over hebt:



Figuur 118: de resterenden mates in SolidWorks

i. Opnieuw exporteren (alle *joint types* op *revolute* of *continuous* zetten)
j. Je moet, voordat je alles definitief exporteert, de *limitAngles* in radialen ingeven voor elke *revolute joint*. Dit (figuur 119) is de plaats waar je dit moet ingeven:



Figuur 119: SolidWorks exporter

<u>Voorwaarden CAD-modellen:</u>

- Het coördinaten stelsel van elk onderdeel moet samenvallen met de *joint* die het dichtste bij de basis van de robot is (base_link)
- Indien mogelijk, zo oriënteren dat als de robot in 'nul positie' is, de X-as voorwaarts en de z-as omhoog wijzen.
- Indien *SolidWorks* is gebruikt moeten deze coördinatenstelsels en de rotatie assen worden toegevoegd aan de *top-level assembly*, en niet aan elk onderdeel apart.
    - ➢ Alle coördinaten stelsels van de *links* zouden best in dezelfde richting wijzen (x-as vooruit en z-as naar boven)

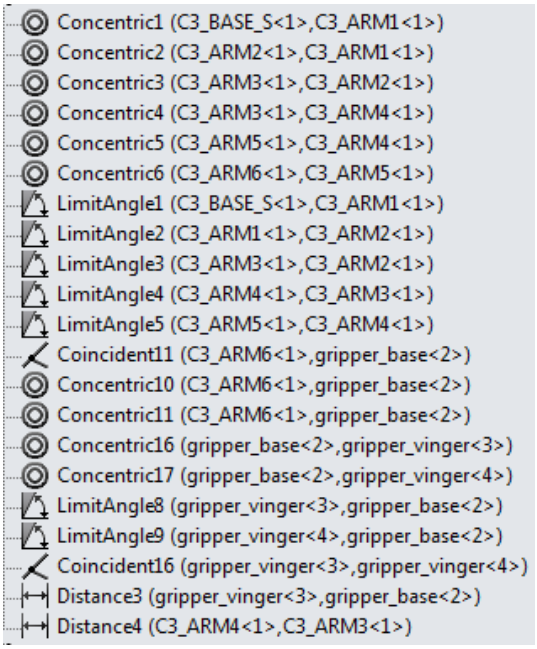
- Na exporteren, controleren of Origin_global juist is georiënteerd (x-as vooruit, z-as omhoog). Indien dit niet het geval is moet dit handmatig worden aangepast en moet erna de export opnieuw uitgevoerd worden.
- Opnieuw exporteren (niet met automatisch gegenereerde assenstelsels, maar de net gegenereerde assenstelsels gebruiken). En assen op *revolute* zetten en ook *lower* en *upper limit* zetten).

CAD-model met gripper

Wat als eerste moet gebeuren om de *gripper* toe te voegen, is om de *gripper* te tekenen indien er nog geen CAD-tekeningen van aanwezig zijn. Er hoeft geen *assembly* van de *gripper* gemaakt te worden, de aparte onderdelen zijn genoeg. Deze onderdelen worden dan één voor één tot de aanwezige *assembly* van de robot toegevoegd.

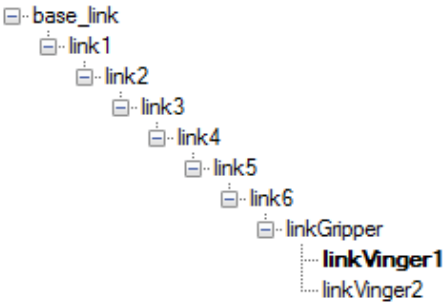
Gripper assembleren bij de robot

a. Open de robot *assembly* zonder *gripper* die je al succesvol hebt kunnen exporteren naar een URDF-bestand.
b. Voeg de nodige onderdelen (*parts*) van de *gripper* toe in deze *assembly*.
   (In dit geval zijn dat maar drie onderdelen, de basis van de *gripper* die op link 6 komt en de twee vingers die aan deze basis worden bevestigd)
c. Gebruik nu *mates* om deze basis eerst aan link 6 te koppelen, hiervoor zijn drie *mates* nodig:
   Twee concentrische *mates* zodat de basis altijd op dezelfde manier mee roteerd met link 6 en dus dezelfde as heeft, en één *distance mate*.
   LET OP: zorg ervoor dat de positie van de basis zich aanpast aan de positie van de robot en niet omgekeerd, want de robot mag nier verplaatsen! Dit zou ook invloeden hebben op de *origins* en de assen.
d. *Mate* nu de vingers aan de basis. Dit (figuur 120) zijn dan alle *mates* die er zijn:



Figuur 120: alle mates in SolidWorks, nadat de gripper is toegevoegd

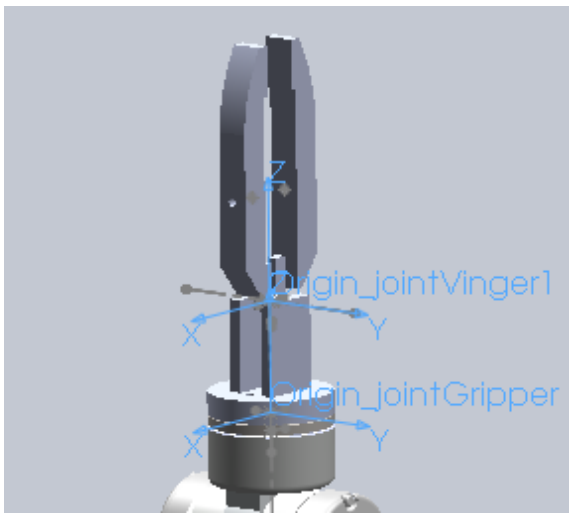
e. Vanaf hier is de procedure gelijk aan die van zonder de *gripper*. Je moet dus eerst weer 'export to URDF' gebruiken en de nieuwe *links* toevoegen als *childs*. Nu is er wel één onderdeel met twee *childs* zoals in figuur 121 te zien is.



Figuur 121: de kinematische volgorde van de links van de Epson-robot

f. Als je deze hebt toegevoegd stel je de gewone links (base_link tot link6) in zoals ze al stonden bij de vorige *export*, en linkGripper en de vingers zet je op '*automatically detect*'. Hierdoor worden weer de *origins* en assen gegenereerd. Indien de assen niet gegenereerd worden kan je deze rap zelf tekenen.

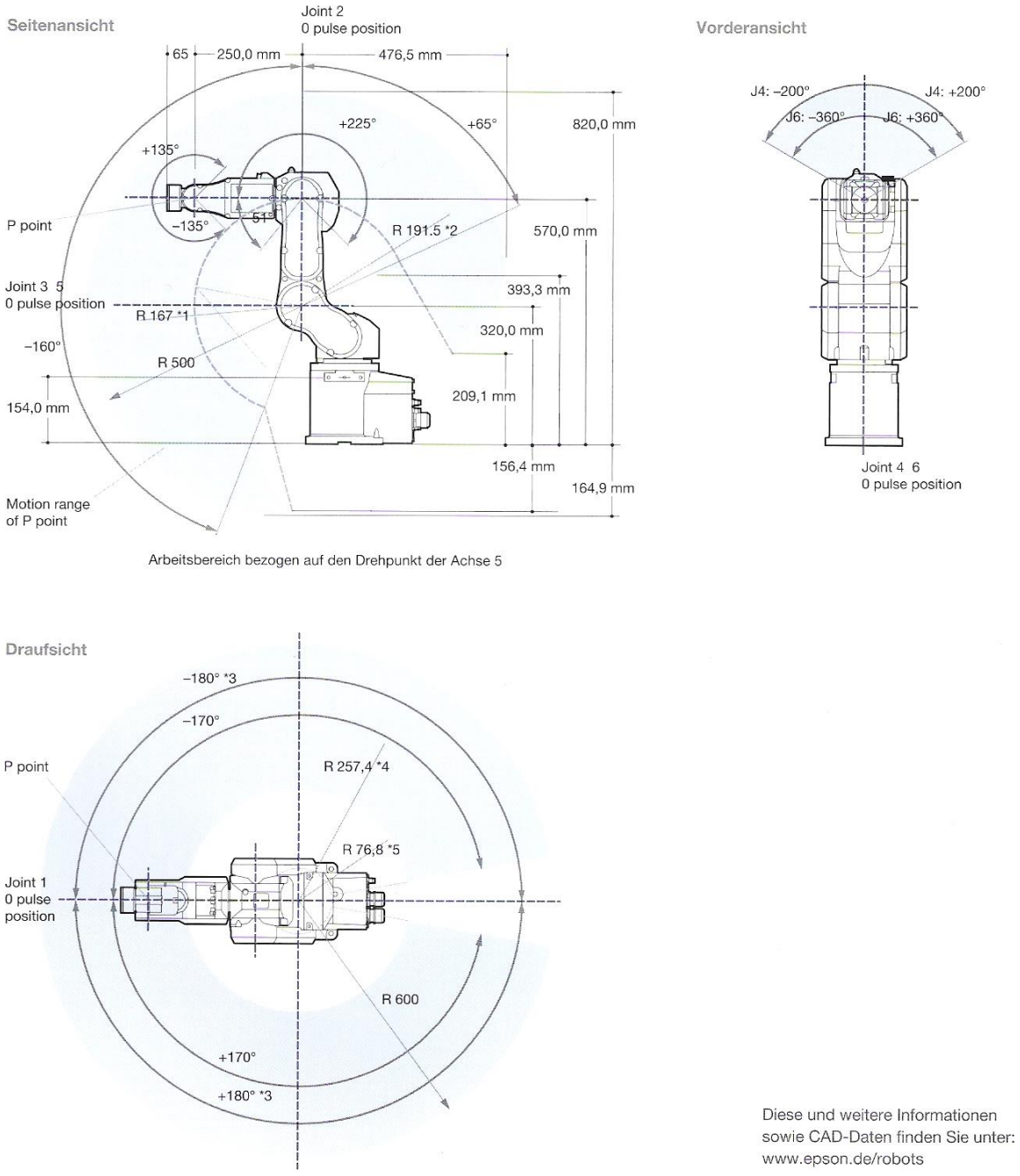
g. Controleer de *origins* zodat ze liggen volgens de conventionele manier (x-as omhoog en z-as naar beneden) en kijk dat de assen juist liggen. De *origins* van de vingers mogen in dit geval samen vallen zodat je hetzelfde *origin* voor beide vingers mag gebruiken. Dit staat afgebeeld in figuur 122



Figuur 122: de ligging van de assenstelsels van de onderdelen van de gripper

a. Hierna mag nogmaals de 'export as URDF' functie gebruikt worden, waarbij dan alles juist moet ingesteld zijn (juiste *origin* en as bij het juiste onderdeel) en de gripperLink moet als *fixed* ingesteld worden omdat deze niet kan bewegen ten opzichte van link 6, er wordt dus geen as gegenereerd voor gripperLink. De vingers worden ingesteld als *revolute*. Hiervoor moeten dus nog de upper en lower limits worden ingesteld in radialen. Alle limitAngles zijn op figuur 123 te zien. Hierop zijn ook alle benamingen van de *joints* aangeduid.

Als dit allemaal gebeurd is, mag de volledige robot + *gripper* worden geëxporteerd naar URDF.



Figuur 123: hoekrotatie's van de Epson-robot

Bijlage B: Specificaties laptop

Merk	Fujitsu
Type	Celsius H700
Processor	Intel® Core™ i7 CPU Q 840 @ 1,87GHz x 8
RAM	3,9 GiB
Operating system	Genuine Windows® 7 Professional 64-bit
	Ubuntu release 12.04 (precise) 32-bit
Grafische kaart	NVIDIA® Quadro® FX 880M
Toegewezen videogeheugen	1 GB (GDDR3 VRAM)

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Gerandomiseerde padplanningsalgoritmen voor opnemen van willekeurig geplaatste werkstukken met zesassige robot in simulatie

Richting: **master in de industriële wetenschappen: energie-automatisering**
Jaar: **2014**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Kenzeler, Yannik **Deneyer, Joren**

Datum: **6/06/2014**