

En undersøgelse af algoritmer til løsning af generalized movers problem i 3D

Master's Thesis af
**Michael Nielsen (20021059) og
Jonas Maturana Larsen (20023690)**

Vejledt af
Gerth Stølting Brodal



Datalogisk Institut
Århus Universitet
Danmark
Marts 2008

Abstract

I dette speciale har vi beskrevet, implementeret og testet fire samplingsbase-rede planlægningsalgoritmer der alle kan bruges til at løse *generalized movers problem*. De fire algoritmer er: *Expansive-Spaces Trees*, *Rapidly-exploring Random Trees*, *Probabilistic Roadmaps* og *Sampling-based Roadmap of Trees*. I forbindelse med implementationen af algoritmerne, har vi konstrueret effektive data-strukturer til at løse ikke-trivielle delproblemer som algoritmerne giver anledning til. I specialet er algoritmerne blevet undersøgt og sammenlignet for at finde ud af hvordan de klarer forskellige typer probleminstanser.

Vores sammenligninger af algoritmerne har vist at det er afgørende for deres effektivitet af de rigtige parametre bliver brugt. Desuden kan probleminstansen være afgørende for hvilken algoritme der er bedst.

English abstract

In this thesis we have described, implemented and tested four sampling based planning algorithms which can be used for solving generalized movers problem. The algorithms are: *Expansive-Spaces Trees*, *Rapidly-exploring Random Trees*, *Probabilistic Roadmaps* and *Sampling-based Roadmap of Trees*. During the implementation of the algorithms we have constructed efficient data structures for solving non trivial sub problems. The algorithms has been investigated and compared to find out how they perform on different kinds of problem instances.

The comparison has shown that it is important for the efficiency of the algorithms that the right parameters are used. Further more the testing shows that the problem instance is a deciding factor for how well the algorithms perform.

Erkendelser

Vi vil gerne takke vores vejleder Gerth Stølting Brodal for at have guidet os under arbejdet med specialet og for at have givet gode råd og hints undervejs. Derudover vil vi gerne takke Thomas Greve Kristensen for uvurderlig feedback på specialerapporten.

Indhold

Abstract	i
Erkendelser	ii
Indhold	iii
I Planlægningsalgoritmer	1
1 Introduktion	2
2 Historisk baggrund	7
2.1 Potential-field methods	7
2.2 Heuristic planners	10
3 Teknisk baggrund	11
3.1 Rotation	11
3.2 Robot	13
3.3 Scene	14
3.4 Konfigurationer	14
3.5 Lokal planlægger (LP)	16
3.6 Roadmap	17
4 Enkel-forespørgsels-algoritmer	19
4.1 Expansive-Spaces Trees (EST)	20
4.2 Rapidly-exploring Random Trees (RRT)	22
5 Multi-forespørgsels-algoritmer	25
5.1 Probabilistic Roadmaps (PRM)	26
5.2 Sampling-based Roadmap of Trees (SRT)	29
6 Hjelpeværktøjer	32
6.1 Graf	32
6.2 Trekanter	34
6.3 Biased Bingo Bag	34
6.4 Naboer i C	37
6.5 Kollisionstest	42
7 Status for implementationen	47

<i>Indhold</i>	iv
II Test og sammenligning	51
8 Test	52
8.1 Testscener	52
8.2 Parametre	55
8.3 Sammenligning af algoritmerne	65
9 Konklusion	67
Litteratur	68
A Resultater for EST-parametertertests	70
B Resultater for RRT-parametertertests	75
C Resultater for PRM-parametertertests	78
D Resultater for SRT-parametertertests	86
Indeks	93

Del I

Planlægningsalgoritmer

1 Introduktion

“Støvsug kælderen”, “Smid skraldeposen ud” eller “Isæt bolten B3 i hullet H1 i pladen P7” er eksempler på opgaver man kunne forestille sig at give en robot. De første to opgaver kunne være opgaver for en hus-robot, der kunne tage sig af nogle af de kedelige opgaver i dagligdagen. Den tredje opgave er derimod mere passende for en tonstung industrirobot, der står ved siden af et samlebånd og bolter Toyotaer sammen i døgndrift. Gældende for alle tre opgaver er dog at de kræver en del planlægning førend de kan løses. Udover at skulle fortolke opgaveformuleringen, hvilket vi ikke vil beskæftige os med i dette speciale, er det nødvendigt at vide hvornår opgaven er løst.

Den første opgave er løst, når hele kælderen er støvsuget. Det kan f.eks. være, når der ikke er mere skidt på gulvet eller robottens bevægelser har dækket hele gulvet. De sidste to opgaver er først løst, når robotten befinder sig i en konfiguration¹, hvor skraldeposen er placeret i skraldespanden eller B3 er placeret i H1.



Figur 1.1: En typisk industrirobot som denne består af en række led der hver især kan bøjes eller roteres på forskellig vis.

En underforstået del af alle opgaver er, at robotten ikke bevæger sig ind i møbler, vægge eller andre forhindringer, der måtte være i vejen.

¹“Konfiguration” bruges om en robot, der er placeret i en bestemt position med ledene på en bestemt måde. Begrebet er defineret mere formelt i 3.4

Der er to måder, at sørge for at robotten ikke støder ind i forhindringer. Den ene metode baserer sig på sensorer, som tillader, at robotten hele tiden kan "se", om den er ved at kollideres med en forhindring. Den anden metode benytter sig af et kort over scenen, der omgiver robotten. For at robotten kan bevæge sig rundt og stole 100% på et kort, er det vigtigt at alle forhindringerne er tegnet ind og at de ikke ændrer sig. De to metoder har hver deres fordele og mange robotter benytter sig da også af en blanding. F.eks. kan industrirobotten benytte sig af et kort hvis bilerne altid er placeret ens på samlebåndet, men vil også benytte sig af sensorer til at korrigere for små unøjagtigheder.

I forbindelse med dette speciale har vi implementeret fire planlægningsalgoritmer, som kan løse planlægningsproblemer i 3-dimensionelle rum ud fra en komplet beskrivelse af en robot, en scene og scenens forhindringer. Nærmere bestemt kan de løse en gruppe af problemer kendt som *generalized movers problems*. Hvilke problemer denne gruppe omfatter vender vi tilbage til i næste afsnit, hvor vi ligeledes gennemgår en række andre problemgrupper. Målet med specialet er at undersøge de fire algoritmers styrker og svagheder og sammenligne deres evner på en række planlægningsproblemer. Som oplæg til undersøgelserne vil vi præsentere teorien bag algoritmerne og begreberne der bruges på feltet. Vi vil ligeledes introducere de underliggende datastrukturer og hjælpealgoritmer, vi har måttet implementere for at opnå effektive egenskaber for algoritmerne.

Eksempler på planlægningsproblemer

De følgende problemer er alle af denne karakter, men indeholder forskellige udfordringer. Vores implementationer kan ikke løse alle problemerne. De er er listet her for at give et indblik i de forskellige muligheder.

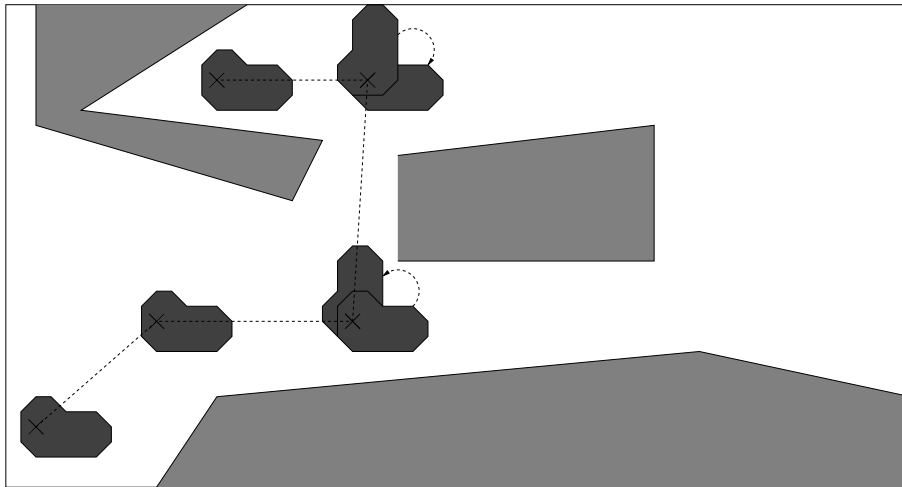
Piano movers problem Det klassiske planlægningsproblem, hvor et stift og "frit-flyvende" 3-dimensionelt legeme (robotten) skal bevæges gennem et rum, fra en startposition til en slutposition, med kendte stationære forhindringer uden at kollideres med forhindringerne. "Frit-flyvende" vil sige, at robotten kan placeres hvor som helst i rummets tre dimension, og yderligere at den kan være roteret på alle tænkelige måder omkring sig selv. Se figur 1.2.

Generalized movers problem er en mere generel problemstilling, hvor robotten kan bestå af en mængde af legemsdele, som er forbundet via led, således at delene kan bevæges i forhold til hinanden. Et oplagt eksempel er robot-arme, som vi kender dem fra utallige produktionsanlæg. Dette er et af de mest centrale/mest bearbejdede af problemerne, og det er sådanne problemer vores implementationer kan løse. Se figur 1.3

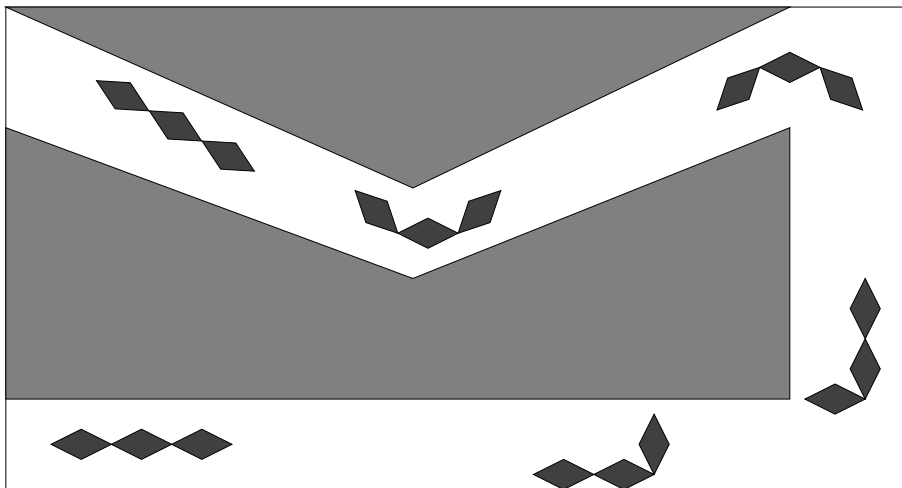
Multiple movers problem Dette problem omfatter ganske enkelt flere robotter samtidig. Disse skal både undgå forhindringer og hinanden.

Kinodynamic problems Udover de geometriske begrænsninger, omfatter/modellerer disse problemer også dynamikbegrænsninger, så som accelerationsevne eller friktion.

Manipulation planning tilføjer begrebet om flytbare objekter. En given opgave kan således gå ud på at en robot-arm, først skal nå frem til et flyt-



Figur 1.2: Ved at rotere robotten, er det muligt at bevæge den igennem den smalle åbning. Der er i alt vist 7 konfigurationer.



Figur 1.3: I generalized movers problem har robotten en række led der kan bøjes. Desuden kan hele robotten roteres og forskydes.

bart objekt og gribe fat i det, for derefter at flytte det til en ny position i rummet.

Assembly planning handler om at planlægge samlingen af komponenter til en helhed eller om at skille helheden ad. Det kan som sådan blot opfattes som en række piano/generalized movers problemer, hvor komponenten er robotten og resten af helheden er en forhindring.

Flexible objects omfatter problemer hvor formen af objekter kan manipuleres - F.eks. hvis en plade kan bøjes.

Løsningsmetoder

Overordnet, kan planlægningsalgoritmerne, der forsøger at finde stier for en robot, deles op i to grupper:

1. Metoder baseret på repræsentation af konfigurationsrummet².
2. Metoder baseret på sampling i konfigurationsrummet.

Metoder baseret på repræsentation af konfigurationsrummet er tilstrækkeligt for *piano movers problem* og instanser af *generalized movers problem*, hvor robotten er relativt simpel. Hver ekstra bevægelig legemsdel på en robot giver dog anledning til en ekstra dimension i konfigurationsrummet, og det bliver derfor hurtigt upraktisk at skulle repræsentere det.

Den anden (og relativt nye) kategori af metoder gør derfor ikke brug af en repræsentation af konfigurationsrummet, men sampler³ i stedet konfigurationer i rummet og forsøger at opbygge et kort over, hvor robotten kan bevæge sig, som er detaljeret nok til at løse det konkrete problem. Et udvalg af disse metoder vil blive beskrevet i kapitel 4 og 5 og er fokusområdet for specialet. Det bør dog allerede nu bemærkes at ikke alle samplings-algoritmerne forsøger at bygge et kort. Da det for nogle problemer blot er nødvendigt at finde en enkelt sti fra en konfiguration til en anden, kan det være spildt arbejde at udforske hele konfigurationsrummet. En del af samplings-algoritmerne forsøger derfor kun at bygge et kort over de interessante områder af konfigurationsrummet og bygger kun kortet indtil den første sti er fundet.

Det bør bemærkes at mange planlægningsproblemer er PSpace-complete, dvs. mindst lige så svære som NP-hårde problemer, og at det kan derfor være nødvendigt at bruge heuristikker til at finde eventuelle løsninger.

Opbygningen af specialet

Specialet vil altså kort fortalt handle om løsning af *generalized movers problem* ved hjælp af samplingsbaserede planlægningsalgoritmer, herunder en undersøgelse af hvilke styrker og svagheder de enkelte algoritmer har i forhold til hinanden. Specialet er inddelt i to dele.

Del I er en teoretisk gennemgang af planlægningsalgoritmerne. Kapitel to giver et historisk perspektiv og beskriver nogle af de repræsentationsbaserede algoritmer, der gik forud for de samplingsbaserede algoritmer, som vi beskriver i kapitel fire og fem. I kapitel tre lægges der op til kapitel fire med en gennemgang af koncepter og nogle tekniske detaljer, som anvendes i algoritmerne. Efter præsentationen af algoritmerne, i kapitel fire og fem, går vi i kapitel seks i dybden med flere tekniske detaljer, herunder en række datastrukturer, som anvendes i algoritmerne. Kapitel syv redegør vi kort for praktiske detaljer omkring vores implementationer.

Del II indeholder vores test af implementationerne. I kapitel otte beskriver vi først en række testscener. Herefter undersøger vi hvordan ændringer af algoritmernes parametre påvirker deres evne til at løse problemer i scenerne.

²Konfigurationsrummet er mængden af alle konfigurationer, se afsnit 3.4 for definition.

³Af mangel på et dansk enkelt-ord, benytter vi "at sample" eller "en sample", som hhv. "at vælge en tilfældig konfiguration" og "en tilfældigt valgt konfiguration", hvor "tilfældig(t)" vil være defineret i konteksten.

Til sidst sammenligner vi algoritmerne med nogle af de parametersæt vi fandt frem til. Kapitel ni afrunder specialet med en konklusion.

I figur 1.4 ses en oversigt over de samplingsbaserede algoritmer vi har beskæftiget os med i specialet.

	Navn	Type	Årstal
EST	Expansive-Spaces Trees	Single	1999 [8]
RRT	Rapidly-exploring Random Trees	Single	1999 [11]
PRM	Probabilistic Roadmaps	Multi	1994 [9]
SRT	Sampling-based Roadmap of Trees	Multi	2003 [4]

Figur 1.4: Oversigt over samplingsbaserede planlægningsalgoritmer.

2 Historisk baggrund

Forud for de samplingsbaserede algoritmer, som idag er den metode, der arbejdes med, har man forsøgt sig med en række andre forskellige løsninger på planlægningsproblemerne. Tidligst med de tilgange der, som nævnt, benytter en repræsentation af konfigurationsrummet, som f.eks. *Cell Decomposition*. Komplexiteten af problemerne flyttede sidenhen fokus over på metoder, der kunne bruges i praksis. Disse praktisk anvendelige metoder kan inddeles i fire kategorier:

1. Løsning af en delmængde af problemerne.
2. Approksimation af det frie konfigurationsrum.
3. Heuristiske metoder.
4. Metoder, der er komplette i forhold til en svagere definition af komplet-hed.¹

Vi vil kigge lidt nærmere på metoder i de to sidstnævnte kategorier, først RPP, som tilhører fjerde kategori og dernæst en heuristisk metode. Vi vil ikke gå helt i dybden med de to algoritmer, men blot give et indblik i hvilke ideer, man arbejdede med inden de samplingsbaserede algoritmer.

2.1 Potential-field methods

Potential-field methods benytter sig af såkaldte potentialefunktioner, til at lede en robot mod et mål og samtidig udenom forhindringer. En potentiale funktion er en differentiabel funktion $U : \mathbb{R}^m \rightarrow \mathbb{R}$.

I den simpleste udgave af potential-field methods, som kaldes Attractive/Repulsive Potential, består funktionen af en sum af hhv. en tiltrækkende og en frastødende funktion. Den tiltrækkende funktion er baseret på afstanden mellem robot og mål-konfigurationen og har til formål at lede robotten mod målet. Den frastødende funktion er baseret på afstandene mellem robotten og de omkringliggende forhindringer, og dens formål er at skubbe robotten væk fra forhindringerne. Den samlede funktion har minimum i mål-konfigurationen, og maksima i forhindringer. Således kan robotten, via gradientstigning, ledes mod målet og udenom forhindringer.

¹Herunder de samplingsbaserede algoritmer.

Afstande

En udfordring ved metoden er at beregne afstandene til forhindringerne, så de kan anvendes til den frastødende funktion. Brushfire algoritmen er én løsning. Den approksimerer afstandene ved hjælp af en grid-opdeling af konfigurationsrummet. Hver grid-enhed angiver, med 0 og 1, hvorvidt enheden er fri hhv. helt/delvist optaget af en forhindring. Dette grid opdateres efterfølgende iterativt på følgende måde. I hver iteration sættes alle felter med en nul-værdi, der samtidig har en nabo med ikke-nul-værdi, til naboens værdi plus een. Algoritmen terminerer, når alle nul-værdier er fjernet. Hver felt har nu en værdi, der angiver en approksimeret afstand til den nærmest forhindring. Denne fremgangsmåde giver naturligvis problemer med konfigurationsrum af høje dimensioner.

Lokale minima

Potential-field methods har endnu en udfordring i det faktum at potentiale funktionen kan have lokale minima, som ikke svarer til mål-konfigurationen. Bliver algoritmen fanget i et sådan, må den altså gøre noget andet end gradient nedstigning for at komme videre. Alternativt kunne man forsøge at konstruere en potentiale funktion, der ikke har lokale minima. Men det er et problem i sig selv:

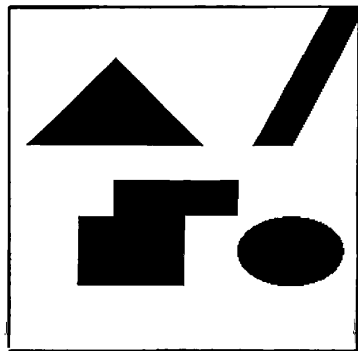
[...] computing local-minima-free potentials in large dimensional spaces is a difficult problem (Koditschek 1987) that is at least as hard as path planning itself. [2][s. 764]

Randomized Path Planner, RPP

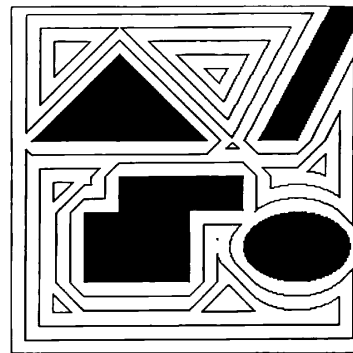
RPP er en potential-field metode, der prøver at få bugt med de lokale minima og samtidig prøver at løse problemer med relativt mange dimensioner, i forhold til andre potential-field metoder. Vi vil beskrive algoritmen, som den er fremlagt i [3].

Potentialefunktionen U i RPP er en kombination af potentiale-funktioner V_{p_i} for hver dimension i konfigurationsrummet. Det er implementeret ved at have tre kontrolpunkter for hver legemsdel på robotten, som angiver positionen i den givne dimension.

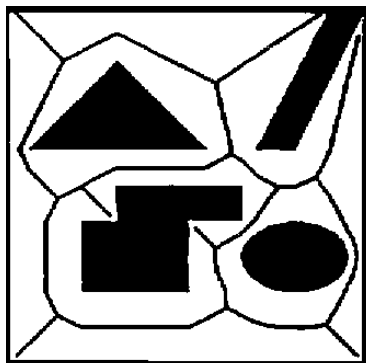
Vi vil nu overordnet gennemgå hvordan V_{p_i} funktionerne beregnes, og illustrere det med figurer fra [3]. For at beregne V_{p_i} funktionerne, beregnes der først afstande til nærmeste forhindring for alle punkter i det frie rum, ligesom i brushfire algoritmen. I forhold til scenen vist i figur 2.1, giver det et konturkort som vist i figur 2.1. Dernæst udføres en bølgefronts funktion, a la brushfire, fra kanterne af det frie rum. Fremfor at gemme afstande, gemmer denne funktion i stedet de punkter hvor bølgefronterne mødes rundt omkring i scenen. Det giver grafen, eller skelettet, som det kaldes i artiklen, som er vist i figur 2.1. Skelettet angiver, i en hvis forstand, stier rundt i rummet, som ligger længst muligt fra forhindringerne. Skelettet og afstandskortet kombineres til sidst til en potentialefunktion, der holder gradientnedstigningsruter længst muligt fra forhindringerne og som kun har minima i mål-konfigurationen. Det opnåes kort fortalt ved først at forbinde målkonfigurationen til skelettet. Herefter propageres afstande fra målkonfigurationen ud langs skelettets grene.



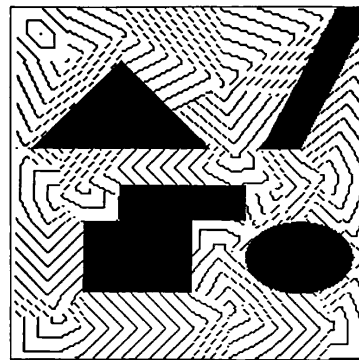
Figur 2.1: Scenen



Figur 2.2: Konturkort for afstande til forhindringerne



Figur 2.3: Skelettet



Figur 2.4: Konturkort for potentialefunktionen

Af de felter, der står for skud mht. propageringen, vælges et felt med størst afstand til forhindringerne først. Når alle felter i skelettet er markeret med en afstand, udføres en brushfire algoritme med udgangspunkt i skelettet. Resultatet kan illustreres med et konturkort, som ses i figur 2.1

Den endelige potentiale funktion for hele konfigurationsrummet er en kombination af V_{p_i} funktionerne. Ifølge artiklen havde man tidligere oftest brugt en lineær kombination af funktionerne, men to andre funktioner gav dem bedre resultater. Selvom V_{p_i} funktionerne kun har et enkelt minima, kan der opstå mange ved kombinationen af dem. Hvordan de kombineres har indflydelse på antallet af lokale minima, der opstår i funktionen, samt størrelsen af dem. Det er på de punkter artiklens valgte kombinationsfunktioner skulle være bedre end de tidligere lineære kombinationer. De to funktioner returnerer følgende:

1. Den mindste værdi blandt V_{p_i} funktionerne, plus den største vægtet med ϵ .
2. Den største værdi blandt V_{p_i} funktionerne.

Den første siges at være god til robotter med relativt få led, mens den anden er bedre i forbindelse med mere komplekse robotter.

RPP algoritmen forløber således: Der udvælges uniformt en tilfældig nabo fra den nuværende konfigurations nabolag. Dvs. en nabo, som i en eller flere dimensioner ligger én enhed fra konfigurationen, og i alle andre dimensioner har samme position. Hvis denne udvalgte nabo har mindre potentiale og ligger i det frie konfigurationsrum, fortsættes der til denne, ellers forsøges nye naboer indtil en grænse afhængig af n . Findes der ingen nabo, opfattes konfigurationen som et lokalt minima.

Når et lokalt minima er nået, udføres en række tilfældige bevægelser i et forsøg på at undslippe minimaet. Såfremt en af disse tilfældige stier rammer et punkt, der ligger under det lokale minima, fortsætter algoritmen fra dette punkt. Findes der ingen lavere punkter går algoritmen tilbage til et tidligere punkt på den sti, der har ført den til sin nuværende position. Mere præcist vælger den uniformt blandt punkter på stien, som stammer fra de segmenter, som blev tilføjet af de tilfældige bevægelser, der som sagt udføres, når algoritmen lander i et lokalt minima. Da disse er skabt på baggrund af tilfældighed, og altså ikke på baggrund af potentialefunktionen, er der mulighed for at nedstigning derfra vil lande i et nyt lokalt minima, som ikke lå på stien før.

2.2 Heuristic planners

RPP leverede imponerende resultater i sin tid. Andre udforskede dog alligevel problemet i en anden retning for til dels at søge at undgå de varierende udførselstider, der følger med RPP's randomisering og dels fordi scenarier kan opstilles, hvor det ikke lykkes for RPP at finde en løsning. Et alternativt bud på en planlægger er beskrevet i [7].

Det er en heuristisk algoritme, der givet en planlægningsopgave med en robotarm med flere led, løser algoritmen opgaven ét led ad gangen, fra den ene ende af armen til den anden. At planlægge for et enkelt segment af armen er et planlægnings-problem med få frihedsgrader, som kan løses nemt. Når bevægelsen af første segment er planlagt, er positionen for den ene ende af næste segment dikteret. Næste delopgaver er så at planlægge rotationen af næste segment i forhold til den dikterede position af segmentets ene ende.

Algoritmen deler altså ét problem med mange frihedsgrader op i mange problemer med få frihedsgrader, og undgår dermed tidskompleksiteten der ellers er forbundet med konfigurationsrum med mange dimensioner.

Metoden kan i praksis løse vanskelige problemer, men algoritmen er ikke komplet, så der vil forekomme problemer, den ikke kan løse. For at udvide algoritmens potentiale, har man tilføjet backtracking til den. Således kan en planlægning af et tidligere led gøres om, hvis det viser sig umuligt at planlægge en bevægelse for efterfølgende led, givet planlægningen for de foregående led. Det ekstra potentiale ved ubegrænset backtracking giver anledning til eksponentielle worst-case udførselstider, men eksperimenter har vist begrænset backtracking som en effektiv udvidelse af algoritmen.

3 Teknisk baggrund

Dette kapitel beskriver et udvalg af teoretiske begreber som vi mener er nødvendige for at få fuldt udbytte af kapitel fire og fem. For ikke at gennemgå alt for meget teori inden vi kommer til planlægningsalgoritmerne, har vi valgt at flytte den mere generelle teori til kapitel seks hvortil vi vil referere når vi mener det kan hjælpe på forståelsen.

3.1 Rotation

En rotation i rummet kan repræsenteres på forskellige måder. Vi har på baggrund af [10] valgt at repræsentere en rotation ved hjælp af en såkaldt enhedsquaternion, da der i forbindelse med vores planlægningsproblemer er en række fordele ved denne repræsentation. Inden vi kigger på fordelene vil vi kort forklare hvad en enhedsquaternion er. En enhedsquaternion er en fire-dimensionel vektor, $q_\phi = [x, y, z, w]$, af længde én. For at få en intuition om quaternioners¹ forhold til rotationer, kan man se på deres relation til akse-vinkel-par. Et akse-vinkel-par angiver en akse i tre dimensioner, $v = [v_x, v_y, v_z]$, samt en rotationsvinkel ϕ omkring aksen. Disse par kan repræsentere samtlige rotationer i 3D. Den tilsvarende quaternion er givet ved følgende:

$$q_\phi = [v_x \sin(\frac{\phi}{2}), v_y \sin(\frac{\phi}{2}), v_z \sin(\frac{\phi}{2}), \cos(\frac{\phi}{2})]$$

For ethvert akse-vinkel-par findes et andet par, som repræsenterer samme rotation, givet ved den modsatte akse og den modsatte rotationsvinkel. Denne dualitet er også tilstede ved quaternionerne.

Som sagt er der en række fordele ved at benytte quaternioner. [10] sammenligner quaternionerne med rotationsmatricer og Euler vinkler på følgende kvalitative punkter:

1. Effektivitet ved beregninger.
2. Pladsforbrug.
3. Numerisk stabilitet.
4. Effektivitet ved uniform sampling.
5. Effektivitet ved interpolation mellem rotationer.

¹Vi vil i resten af specialet benytte "quaternion(er)". I alle tilfælde er der tale om enhedsquaternioner.

6. Mulighederne for at definere en meningsfyldt afstandsfunktion.

Vi har opsummeret artiklens vurdering af hvordan de tre repræsentationer egner sig til planlægningsproblemer udfra ovennævnte parametre.

Rotationsmatricer er relativt ineffektive pladsmæssigt, og de er ikke numerisk stabile ved beregninger. Det er uklart hvordan der interpoleres mellem matricer og hvordan en afstandsfunktion kan defineres.

Euler vinkler er meget kompakte og relativ effektive til beregninger, hvor de også er numerisk stabile. Det er simpelt og effektivt at sample uniformt. En rotation r_1 kan repræsenteres af mange forskellige Euler vinkler, og afstandsfunktionen i artiklen kan give vidt forskellige svar for afstanden mellem r_1 og en anden rotation r_2 , alt efter hvilke Euler vinkler r_1 og r_2 er repræsenteret af. De mange ækvivalente repræsentationer kommer også i vejen for effektiv interpolation.

Quaternioner er relativt kompakte, effektive til beregninger og numerisk stabile, hvis de normaliseres. Der er simple og effektive metoder til både uniform sampling og interpolation, og der kan defineres meningsfyldte afstandsfunktioner.

Vi vil nu afdække hvordan sampling og interpolation udføres for quaternioner. Afstandsfunktionen, som vi har valgt at implementere for quaternionerne, kigger vi nærmere på i afsnit 6.4. Vi bruger quaternionerne til alle de algoritmer vi har implementeret, hvorfor vi også bruger samme afstandsfunktion i alle algoritmer, fremfor at benytte de funktioner, som blev anvendt i hver enkelt algoritme oprindeligt.

Uniform sampling

I forbindelse med sampling af quaternioner har vi implementeret metoden beskrevet i [10]. Pseudokoden fra artiklen er gengivet herunder.

Algoritme 3.1.1: QUATERNION.UNIFORMSAMPLE()

```

 $s \leftarrow \text{RAND}()$ 
 $\sigma_1 \leftarrow \sqrt{1 - s}$ 
 $\sigma_2 \leftarrow \sqrt{s}$ 
 $\theta_1 \leftarrow 2\pi \times \text{RAND}()$ 
 $\theta_2 \leftarrow 2\pi \times \text{RAND}()$ 
 $w \leftarrow \cos(\theta_2)\sigma_2$ 
 $x \leftarrow \sin(\theta_1)\sigma_1$ 
 $y \leftarrow \cos(\theta_1)\sigma_1$ 
 $z \leftarrow \sin(\theta_2)\sigma_2$ 
return (new Quaternion( $w, x, y, z$ ))

```

Metoden genererer en enhedsquaternion, så der er ikke behov for efterfølgende normalisering.

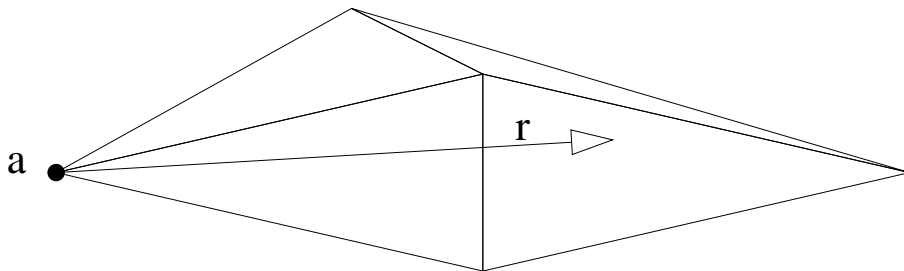
Interpolering

Quaternionerne ligger på en fire-dimensionel enhedssfære. Givet to quaternioner, q_1 og q_2 er der en storcirkel, på enhedssfæren, som går igennem dem begge. Punkterne langs buen, som forbinder q_1 og q_2 på denne storcirkel, repræsenterer den korteste kontinuerte overgang fra den ene rotation til den anden. Når vi skal finde quaternionen midt imellem q_1 og q_2 , er det altså midtpunktet på denne bue, vi skal finde. Dog skal man huske at tage højde for, at der som nævnt eksisterer to ækvivalente quaternioner for hver rotation. Holder vi q_1 fast, vælges den retter interpolation ved at beregne prikproduktet mellem q_1 og q_2 . Hvis prikproduktet er positivt, interpoleres imellem q_1 og q_2 ellers i mellem q_1 og q_2' .

Vi har i vores implementation benyttet os af *javax.vecmath.Quat4d* pakken fra Sun, som indeholder en metode, der, givet q_1 og q_2 , kan finde midtpunktet på den rigtige storcirkel-bue og dermed quaternionen, som ligger midt imellem q_1 og q_2 .

3.2 Robot

For at kunne modellere virkelige robotter, består vores robotter af en række lemmer der er sat sammen med led. Hvert lem er modeleret som et trekantsnet, med et ankerpunkt, a , og en retningsvektor, r , der bestemmer lemmets placering. Et eksempel kan ses i figur 3.1.



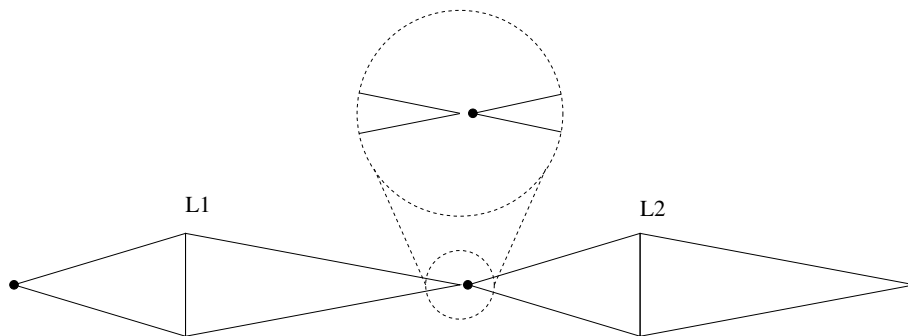
Figur 3.1: Et lem består af et trekantsnet og har defineret et ankerpunkt og en retningsvektor.

Led

Et led består af en quaternion som definerer vinklen mellem to lemmer. I 3.2, kan man se at L_2 er forskudt netop nok til at de to lemmer ikke rører hinanden. Hvis ledet bliver bøjet på en sådan måde at de to lemmer skærer hinanden, er der tale om en ulovlig konfiguration.

Frihedsgrad

I litteraturen bruges udtrykket "Degrees of Freedom" (forkortet DoF) om antallet af uafhængige variable, der skal bruges for at bestemme en konfiguration. Vi har valgt at oversætte dette til "frihedsgrader", men bruger også "dof" eller "doffer" i flertal. Antallet af en robots frihedsgrader har stor betydning



Figur 3.2: To lemmer er placeret på en sådan måde at de ikke rører hinanden.

for sværhedsgraden af et planlægningsproblem, vi har dog ikke lavet en systematisk undersøgelse af hvordan antallet af frihedsgrader påvirker vores implementation.

3.3 Scene

Scenerne hvori planlægningsproblemerne udspiller sig består af en række forhindringer som er modelleret med trekantsnet. Trekantsnet er en meget brugt metode til at repræsentere objekter i rummet inden for computer grafik og tekniske tegneprogrammer. Vi har valgt at bruge denne repræsentation for at kunne arbejde med scener som er genereret af denne type værktøjer, vi har dog ikke prøvet dette i praksis.

3.4 Konfigurationer

En konfiguration (t, Q) består af en forskydning $t = [x, y, z]$ og en liste af quaternioner $Q = \{q_1, \dots, q_n\}$, én quaternion for robotens basislegeme og én for hvert led i robotten. For eksempel har konfigurationer der repræsenterer robotten fra figur 1.3 én forskydning og tre quaternioner.

Konfigurationsrum

Igennem specialet kommer vi til at referere til en række mængder af konfigurationer. De mest brugte vil blive introduceret her.

Rummet, $\mathbb{R}^3$ Alle planlægningsproblemerne, som vi arbejder med udspiller sig i 3D og mængden af punkter er derfor en delmængde af $\mathbb{R}^3$, der repræsenterer scenen. Vi misbruger notationen lidt og kalder denne delmængde for $\mathbb{R}^3$ da alle gængse regler stadig gælder.

Forhindringer I alle interessante planlægningsproblemer er en del af $\mathbb{R}^3$ optaget af andre objekter end robotten. Vi refererer generelt til denne delmængde som "forhindringerne".

Konfigurationsrummet kaldes C , og er mængden af alle konfigurationer. Det vil sige både konfigurationer, som giver anledning til at robotten kolliderer med forhindringer eller med sig selv og konfigurationer, som er kollisionsfrie.

Det frie rum, C_f "Det frie rum", C_f , er oversat fra *free space*, som bliver brugt i det meste af litteraturen. C_f er mængden af konfigurationer robotten kan placeres i uden at overlappende med forhindringerne eller sig selv.

Nabolag, $N_d(x)$ Et nabolag, $N_d(x)$, er mængden af konfigurationer i C , som har en afstand til x på højst d . Altså givet en afstandsfunktion for C , $dist_C$, er d -nabolaget for x :

$$N_d(x) = \{c \in C \mid dist_C(c, x) \leq d\}$$

Afstandsfunktioner

Ofte har vi brug for at kunne afgøre hvor tæt to konfigurationer er på hinanden. Derfor må vi have defineret en afstandsfunktion på C . Vi bruger konsekvent afstandsfunktionen $dist_C$, som gennemgås i afsnit 6.7, men nævner også $dist_{max}$ som en del af artiklerne bruger. $dist_{max}$ er defineret som følger: Lad c være en konfiguration og x et punkt på robotten. $r(c, x)$ er placeringen i rummet af x når robotten befinder sig i konfigurationen c og $dist_{max}$ er beregnet som:

$$dist_{max}(c, c') = \max_{x \in robot} ||r(c, x) - r(c', x)||$$

Sampling

For at sample en konfiguration uniformt tilfældigt i C , finder vi først en tilfældig værdi for hver dimension i $\mathbb{R}^3$, en værdi mellem 0 og scenens udbredelse i den givne dimension. Herefter samples de n quaternioner med den uniforme samplingsmetode, som vi beskrev i afsnit 3.1. Alt i alt giver det en uniformt tilfældig placering i C .

Interpolation

Den simple lokalplanlægger, som benyttes af alle planlægningsalgoritmerne til at forbinde konfigurationer, har brug for at kunne finde konfigurationen (t_b, Q_b) , der ligger midt på den lige linje i C mellem to givne konfigurationer, (t_a, Q_a) og (t_c, Q_c) . Denne interpolation af den mellemliggende konfiguration opnås ved først at udføre lineær interpolation mellem forskydningerne t_a og t_c , for at finde forskydningspunktet $t_b = (t_a + t_c)/2$ for den mellemliggende konfiguration. Herefter interpoleres quaternionerne, som beskrevet i 3.1, så $Q_b = \{q_{a_1} \# q_{c_1}, \dots, q_{a_n} \# q_{c_n}\}$ hvis vi bruger $\#$ som interpolationsoperator for quaternioner.

3.5 Lokal planlægger (LP)

Alle samplingsalgoritmerne benytter sig af en form for lokal planlægger, herfra kaldet LP. En LP, eller *local planner* som det kaldes i litteraturen, er en algoritme, der tager to konfigurationer, c og c' fra C_f og returnerer en kollisionsfri sti mellem disse, hvis algoritmen finder en. Finder algoritmen ikke en, returneres *null*. At algoritmen ikke returnerer en sti er ikke ensbetydende med at der ikke findes en, dog er det meget vigtigt at algoritmen ikke giver falske positiver. Bemærk alle planlægningsalgoritmer kan bruges som en LP, da de netop er konstrueret til at løse denne type opgaver. I [5] diskuteres to ekstremer af LPer:

En hurtig men ikke-grundig LP Denne type algoritmer forsøger at finde en kollisionsfri sti, f.eks. ved at følge en lige linje mellem c og c' , hvis ikke det lykkedes opgiver den.

En langsom men grundig LP Denne type algoritme bruger vilkårligt komplekse metoder til at forsøge at forbinde c og c' . Som eksempel, bruger SRT-algoritmen RRT-algoritmen som LP, se afsnit 5.2.

Alt efter hvilken af disse to typer LP man bruger, vil antallet af kald til LP ændre sig. Jo mindre grundig LP er, jo flere gange vil den blive kaldt. I [5] påstås det at en hurtig LP er at foretrække, og alle de samplingsbaserede algoritmer på nær SRT bruger da også en simpel LP.

Vi har implementeret en lige-linje-LP der, som navnet siger, tester om den lige linje, der forbinder c og c' i C , er kollisionsfri. Følgende pseudokode er kernen i lige-linje-LP.

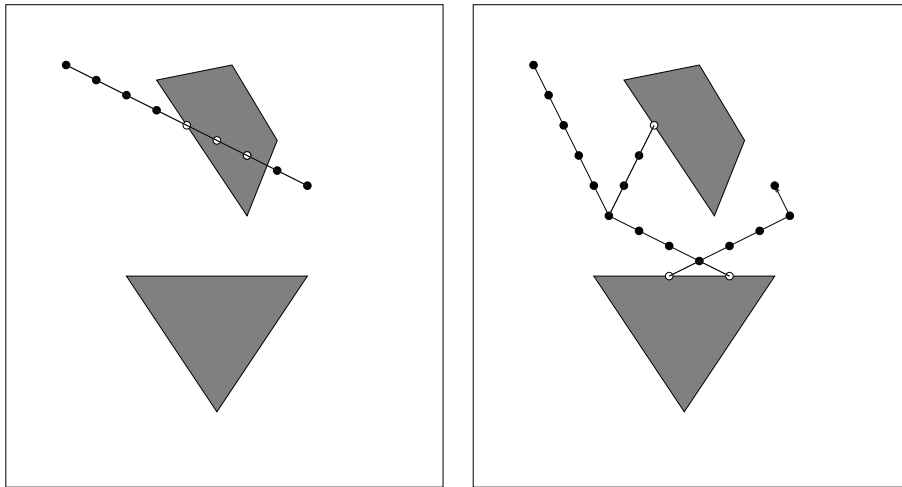
Algoritme 3.5.1: FREE(c, c')

```

 $Q \leftarrow \text{NEW QUEUE}()$ 
 $Q.\text{ADD}((c, c''))$ 
while not  $Q.\text{EMPTY}()$ 
     $(c, c') \leftarrow Q.\text{REMOVE}()$ 
    if  $\text{DIST}(c, c') \geq \text{step\_size}$ 
         $c'' \leftarrow \text{MIDPOINT}(c, c')$ 
        if  $c'' \in C_f$ 
             $Q.\text{ADD}((c, c''))$ 
             $Q.\text{ADD}((c'', c'))$ 
        else
            return ( false )
return ( true )

```

Pseudokoden antager at midtpunktet mellem to konfigurationer kan findes. Dette er en nem opgave, hvis det er muligt at interpolere mellem konfigurationer. At finde ud af om en konfiguration er i C_f kræver blot et kald til kollisionstesteren. Bemærk at funktionen ikke er rekursiv, som ellers ville være oplagt. Grunden til dette er at der er størst sandsynlighed for at et langt segment skærer en forhindring og vi udnytter dette til hurtigere at kunne afvise et segment.



Figur 3.3: To eksempler på en LP, der forsøger at finde vej mellem to konfigurationer.

Vores LP har bruger en fast *step_size*, men som det også nævnes i [5], findes der forskellige metoder til at arbejde med en dynamisk *step_size*. Problemet med at have et fast *step_size* er tveægget. Hvis det vælges for stort, er det ikke sikkert at en skæring bliver fundet. Vælges det derimod for lille, bliver der spildt en masse tid på at tjekke konfigurationer som er ligegyldige da de f.eks. ligger langt væk fra en forhindring.

I [9] luftes idéen om at beregne *swept volume*, SV , mellem c og c' for derefter at teste om SV kolliderer med forhindringerne. [9] fejler dog idéen af bordet igen med et argument, om at det er for langsomt at beregne SV , og de ender med at bruge lige-linje-LP. Vi har ikke prøvet andre LPs end lige-linje-LP, dog med den undtagelse at vi i implementationen af SRT bruger RRT som LP.

3.6 Roadmap

Ordet “roadmap”, er i litteraturen brugt om en række forskellige koncepter. [5] har en liste over nogle af disse. Vi vil udelukkende bruge ordet om det, der nogen steder i litteraturen bliver kaldt “visibility graphs”.

Idéen med et roadmap er at kortlægge så store dele af C_f som muligt. I stor lighed med et landkort, består et roadmap af nogle punkter, som repræsenterer konfigurationer. Konfigurationerne kan være forbundet med det, der svarer til veje på et landkort, hvis det er muligt at “se” fra en konfiguration til en anden konfiguration.

At kunne se fra en konfiguration til en anden, kan defineres på forskellige måder. Den simpleste måde er at sige at to konfigurationer kan se hinanden, hvis robotten kan bevæges langs en lige linje mellem dem i C uden at skære nogen forhindringer. Alternativt kunne man, sige at to konfigurationer kan se hinanden, hvis der findes en bue som forbinder dem, som ikke skærer nogen forhindringer. En tredje mulighed er lade to konfigurationer være naboer i roadmappet, hvis der kan bygges et træ, med rod i den ene konfiguration, som den anden konfiguration kan se med en ret linje. Dette træ kan være

begrænset i størrelse.

De forskellige planlægningsalgoritmer vi beskriver i det næste kapitel, prøver på forskellig vis at kortlægge C_f , som i de interessante tilfælde er et højdimensional ikke-euklidisk rum. Det er dog ikke noget problem at generalisere fra 2D eller 3D. Det eneste, der kræves, er at kunne interpolere mellem to konfigurationer, på en sådan måde at interpoleringen ikke kan passere forhindringer, uden at det opdages. Interpolering er forklaret i afsnit 3.4.

4 Enkel-forespørgsels-algoritmer

I dette kapitel introduceres to algoritmer der er lavet til at løse den samme type planlægningsproblemer. Problemet som vi kalder for enkel-forespørgsels, kaldes i litteraturen for *single query*, og forsøger at finde en kollisionsfri sti mellem to forud givne konfigurationer, c_{start} og c_{goal} . Bemærk at dette netop er definitionen på en lokal planlægger, som er beskrevet i 3.5, og begge algoritmer kan bruges som en sådan. Da vi blot er interesseret i at finde én vej og ikke i at finde en vej med specielle kvaliteter, er problemet løst så snart en vej er fundet.

I kapitel 5 beskrives yderligere to algoritmer som også løser planlægningsproblemer, men gør det ved først at kortlægge C_f for derefter at kunne løse vilkårligt mange problemer hvor c_{start} og c_{goal} varierer. Disse kaldes for multi-forespørgsels.

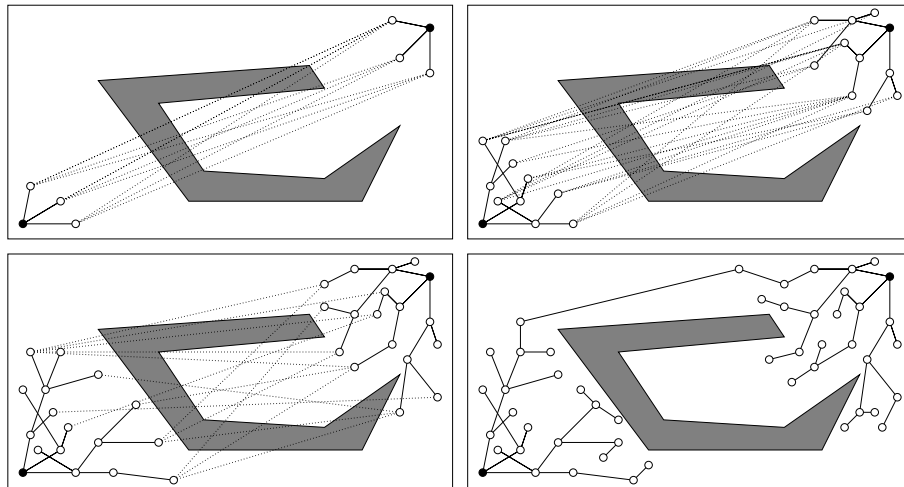
Begge forespørgsels-algoritmers grundlæggende strategi, går ud på at bygge to træer med rod i hhv. c_{start} og c_{goal} og løbende forsøge at forbinde disse to. I pseudo-kode kunne det se ud på følgende vis:

Algoritme 4.0.1: SINGLEQUERY(c_{start}, c_{goal}, M)

```
 $i \leftarrow 0$ 
while  $i < M$ 
    EXPANSION()
    if CONNECTION()
        return ( true )
     $i \leftarrow i + 1$ 
return ( false )
```

Her er dog tilføjet en variable M som bestemmer hvor mange gange algoritmen maksimalt forsøger at udvide træer. I figur 4.1 er vist et eksempel på hvordan en algoritme kunne forsøge at finde en sti. Eksemplet er kun lavet for at give et indtryk af hvad kapitlet handler om og er afbilleder ikke udførelsen af algoritmerne vi vil præsentere.

I mange enkelt-forespørgsels-algoritmer bygger man to konfigurations-træer, men der er generelle tilfælde hvor det kan være en fordel kun at bygge et træ. Hvis f.eks. c_{start} ligger i et meget snævert område og c_{goal} befinder sig i et meget åbent område, kan det være en fordel kun at bygge ét træ der har rod i c_{start} og løbende forsøge at forbinde dette til c_{goal} . Et scenarie som det beskrevne, møder man f.eks. indenfor *disassembly problems* hvor man er interesseret i at finde ud af om et legeme kan fjernes fra en sammenhæng. Ønsker man derimod at placere et legeme i en sammenhæng vil det modsatte scenarie være aktuelt. Et eksempel på dette kunne være at fjerne en bolt fra den motor



Figur 4.1: I hver af de fire iterationer, udvides hvert træ med en mængde knuder. De nye knuder i det ene træ forsøges derefter forbundet til de nye knuder i det andet træ. Så snart det lykkedes at forbinde to knuder, terminerer algoritmen fordi der er fundet en løsning på planlægningsproblemet.

den er skruet fast i. I vores gennemgang af algoritmerne bygges altid to træer, som vi kalder hhv. T_{start} og T_{goal} .

I gennemgangen af de to algoritmer, vil vi koncentrere os om to ting: Hvordan træerne forsøges udvidet og hvordan de forsøges forbundet.

4.1 Expansive-Spaces Trees (EST)

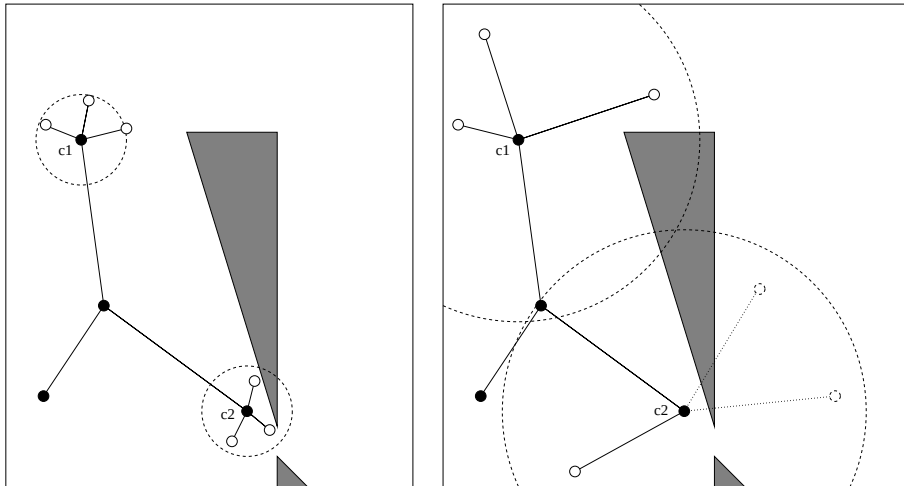
Den første af de to algoritmer er EST som præsenteret af Hsu et al. i [8]. Strukturen af EST er helt lig den vi gennemgik i 4. Hver gang $EXPANSION()$ bliver kaldt, udvides begge træer på følgende måde:

1. Vælg en konfiguration c med sandsynlighed $\frac{1}{w(c)}$ blandt konfiguration i træet. Funktionen w er forklaret efterfølgende.
2. Vælg K konfigurationer fra $N_d(c)$, altså nabolaget af c . Afstandsfunktionen er $dist_C$. Lad os kalde mængden af disse K konfigurationer for N .
3. For alle n i N , forsøg med sandsynlighed $\frac{1}{w(n)}$ at forbinde c og n . Hvis c og n kan forbindes, tilføjes n og en kant mellem c og n til træet.

Funktionen $w(x)$ beregner vægten af knuden x , som er antallet af elementer fra træet, der ligger i nabolaget af x . [8] bruger $dist_{max}$ til at afgrænse nabolaget x med. Vi har valgt at bruge $dist_C$ til at gøre dette, da vi mener, at det giver et bedre sammenligningsgrundlag i forhold til de andre algoritmer.

Valg af d Radiusen af nabolag kan have stor indflydelse på, hvor godt algoritmen klarer sig. Hvis d er for lille, vil træet blive udvidet med en masse konfigurationer, som kun giver meget lidt ekstra viden om C_f . Hvis d er for

stor, vil mange punkter fra N ikke kunne forbindes til c og der vil derfor blive tilføjet få konfigurationer til træet. I figur 4.2 er illustreret hvordan forskellige værdier for d kan påvirke udvidelsen af ét træ. Bemærk at træet ikke er et EST-træ men blot er tegnet for at illustrere hvordan de nye konfigurationer har en tendens til at ligge væk fra træet.



Figur 4.2: Der er fordele og ulemper ved at vælge et lille eller et stort d . Til højre er d fire gange så stor som til venstre, hvilket giver vidt forskellige resultater. I begge tilfælde forsøges $c1$ og $c2$ forbundet til tre nye konfigurationer. De hvide cirkler er nye i træet. De hvide stiplede cirkler er nye konfigurationer der ikke kan forbindes til træet. De sorte cirkler er eksisterende konfigurationer.

Vi har forsøgt at finde et godt d ved hjælp af forsøg, resultaterne kan ses i 8.2. Da d begrænser hvor hurtigt C_f kan udforsket, kan det være en god ide at arbejde med forskellige værdier for d i løbet af algoritmens udførsel.

Efter at have udvidet træerne med mellem 0 og K elementer, forsøger algoritmen at forbinde de to træer. I [8] beskrives forbindelsesskridtet på følgende måde:

Algoritme 4.1.1: CONNECTION()

```

for each  $c \in T_{start}$ 
  for each  $c' \in T_{goal}$ 
    if  $dist_C(c, c') < l$ 
      if LP( $c, c'$ )
        return ( true )
  return ( false )

```

Der er dog ingen grund til at forsøge at forbinde andet end de nye knuder i T_{start} til alle knuderne i T_{goal} og tilsvarende de nye knuder i T_{goal} til T_{start} . Denne justering reducerer denne del af algoritmens tidsforbrug med en lineær faktor.

Hvis CONNECTION() forbinder de to træer, er der kun ét træ tilbage. Ved at bruge bredde-først søgning startende i c_{start} , kan en kollisionsfri sti til c_{goal} findes.

4.2 Rapidly-exploring Random Trees (RRT)

I gennemgangen af EST, nævnte vi de overvejelser man kan gøre sig, når radiusen, d , på et nabolag skal vælges. Hvis d vælges for lille, sker udforskningen af C_f unødvendigt langsom. Hvis det derimod vælges for stort, er det svært at opdage smalle passager. Rapidly-exploring Random Trees algoritmen blev blandt andet præsenteret for at omgå nogen af disse svagheder. Algoritmens største bidrag i forhold til de foregående er dog at den er rettet mod effektiv løsning af *kinodynamic problems*, med både mange frihedsgrader og komplekse dynamiske begrænsninger. Vi vil dog ikke kigge på disse aspekter. RRT blev introduceret af LaValle og Kuffner i [11]

Overordnet ligner algoritmen altså EST, men måden træerne udvides på og måden de forsøges forbundet på er anderledes. Algoritmen som beskrevet herunder, er baseret på [12], hvor algoritmen er modificeret en smule i forhold til den oprindelige fremstilling i [11]. Da "RRT" nogen gange bruges om et enkelt træ der er bygget med RRT algoritmen, kalder vi den version af RRT der bygger to træer for BiRRT, når vi ønsker at tydeliggøre at det er denne version der skrives om. Vi har implementeret både RRT og BiRRT, men vil kun gennemgå BiRRT i dette afsnit.

Følgende pseudokode, viser hvordan to RRT-træer forsøges udvidet mod en uniformt samplet konfiguration, c_{rand} og i samme omgang forsøges forbundet. *Extend* vil blive gennemgået efterfølgende. Figur 4.3 kan give et indblik i hvordan udvidelsen sker.

Algoritme 4.2.1: BiRRT(c_{start}, c_{goal})

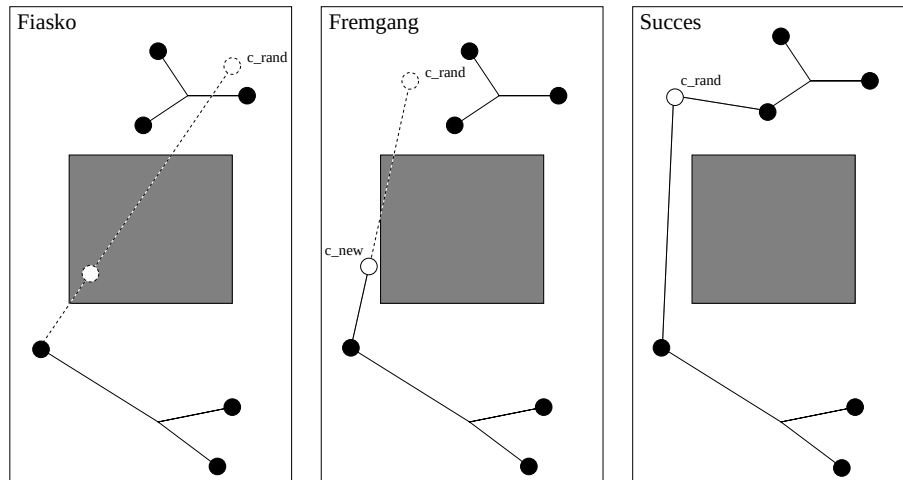
```

 $T_{start} \leftarrow \text{NEW EMPTY TREE}(c_{start})$ 
 $T_{goal} \leftarrow \text{NEW EMPTY TREE}(c_{goal})$ 
for  $i \leftarrow 0$  to  $K$ 
     $c_{rand} \in_R C_f$ 
     $c_{new} \leftarrow \text{EXTEND}(T_{start}, c_{rand})$ 
    if  $c_{new} \neq \text{NULL}$ 
        if  $\text{EXTEND}(T_{goal}, c_{new}) = c_{new}$ 
            return  $(\text{PATH}(c_{start}, c_{goal}))$ 
         $\text{SWAP}(T_{start}, T_{goal})$ 
return  $(\text{NULL})$ 

```

Hvor EST sampler i nabolaget omkring en tilfældigt valgt konfiguration i træet, starter RRT altså i stedet med at vælge konfigurationen, c_{rand} , der skal udvides imod - Og denne samples uniformt i hele C_f . Herefter forsøges træet udvidet mod samplen ved at bruge EXTEND som kan resultere i ét af følgende tre udfald:

1. Fiasko - Det lykkedes ikke at udvide træet. c_{new} bliver sat til NULL .
2. Fremgang - Det lykkedes at udvide træet i retning af samplen, men udvidelsen nåede ikke helt frem. c_{new} peger på en ny konfiguration der ligger på linjen mellem c_{rand} og den konfiguration i T_{start} der ligger nærmest c_{rand} .
3. Succes - Udvidelsen nåede frem til samplen. c_{new} peger på c_{rand} .



Figur 4.3: Tre eksempler på hvordan en iteration af BiRRT kan ændre grafen alt efter hvordan c_{rand} bliver valgt.

De tre udfald er hver især illustreret i figur 4.3.

I BiRRT afgør udfaldet, om de to træer skal forsøges forbundet. Når udvidelsen ender i fremgang eller succes, kaldes *extend* på det andet træ, således at det udvides i retning af den nye konfiguration. Når denne ekstra udvidelse ender i succes er træerne forbundet, og stien fra start til mål, som løser planlægningsproblemet, findes og returneres. Når det ikke lykkes, bytter træerne rolle, og der udføres en iteration mere. Sådan fortsætter algoritmen til problemet er løst eller en iterationsgrænse er nået.

Den nævnte forskel på denne implementation og den første implementation i [11] ligger i, hvilken konfiguration træerne udvides mod. I den første implementation udvides begge træer i hver iteration mod den samme tilfældige sample. Efter hvert træes udvidelse undersøges det, såfremt udvidelsen var en succes eller en fremgang, om den nye konfiguration allerede findes i det andet træ - Forstået således at den ligger indenfor en passende lille afstand af en konfiguration i det andet træ. I såfald er træerne forbundet og problemet er løst.

Extend funktionen er her vist i pseudokode:

Algoritme 4.2.2: EXTEND(T, c_{target})

```

 $c_{near} \leftarrow \text{NEAREST}(T, c_{target})$ 
 $c_{new} \leftarrow \text{NEWSTATE}(c_{near}, c_{target})$ 
if  $c_{new}$  not NULL
    T.ADDNODE( $c_{new}$ )
    T.ADDEDGE( $c_{near}, c_{new}$ )
return ( $c_{new}$ )

```

Funktionen finder først den konfiguration i træet, som ligger tættest på c_{target} . Den fundne konfiguration, c_{near} , sendes videre til NEWSTATE funktionen, som forsøger at tage et skridt imod c_{target} langs den lige linje, der forbinder de to konfigurationer i C . Som en "grådig" variant af algoritmen kan

man vælge at lade NEWSTATE kalde sig selv rekursivt, indtil den støder ind i en forhindring eller når frem til c_{target} . Man kan i såfald overveje, om man vil gemme alle konfigurationerne, der genereres, nogle af dem eller evt. kun den sidste. I både grådig og ikke-grådig version er skridtstørrelsen en parameter, der kan have indflydelse på algoritmens opførsel.

RRT har, som nævnt, fra starten været rettet mod *kinodynamic problems*. I forhold til *generalized movers problems* indeholder disse også dynamikbegrænsninger, som f.eks. accelerationsevne, der også skal overholdes af planlæggeren. Når algoritmen anvendes til at løse disse, udvider NEWSTATE ved at styre robotten via dens kontrolinput, f.eks. "fuld kraft frem på motor". Det kan man gøre ved at vælge et tilfældigt input og simulere udførelsen af det i et stykke tid. Eller man kan alternativt vælge at simulere alle mulige input, der kan gives til robotten, og herefter vælge det input, som har bragt robotten tættest på den konfiguration, der var målet for udvidelsen. Er inputmængden ikke endelig, kan man udvælge en endelig mængde ved sampling og sammenligne disse. Vi har udelukkende implementeret og brugt algoritmen til *generalized movers problems*.

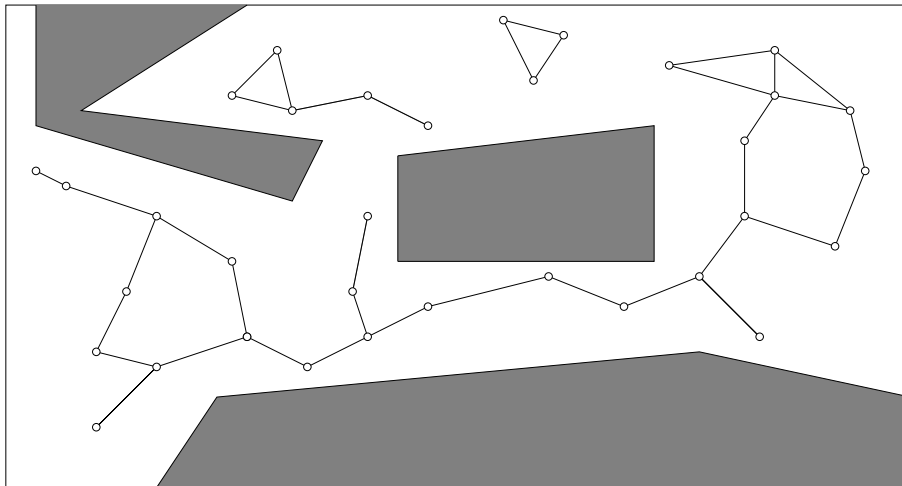
5 Multi-forespørgsels-algoritmer

Hvis man ønsker at planlægge mange stier for den samme robot i den samme scene, er det ikke særligt smart at starte på bar bund hver gang sådan som en single-forespørgsels-algoritme gør det. Det er derimod en god idé, at gemme oplysninger om vejene rundt i C_f . I dette kapitel beskriver vi to multi-forespørgsels-algoritmer som først konstruerer et kort over C_f for derefter hurtigt at kunne løse givne planlægningsopgaver.

I figur 5.1 er illustreret et eksempel på hvordan et kort kan se ud hvis det konstrueres på følgende måde:

- Sampel N konfigurationer uniformt tilfældigt i C_f .
- Forbind dem hver især med de k nærmeste konfiguration hvortil der findes en kollisionsfri sti.

Denne algoritme bliver i [4] og [5] kaldt PRM, men er blot en simplifikation af den algoritme vi gennemgår 5.1, der stammer fra [9] som er den originale artikel. I eksemplet er N 34 og k er to.



Figur 5.1: 34 konfigurationer der hver især er forsøgt forbundet til de to nærmeste konfigurationer.

Eksemplet viser udfordringen som multi-forespørgsels-algoritmerne prøver at overvinde, nemlig at bygge et kort der bedst muligt efterligner rummets topologi. Som det kan ses, er det ikke lykkedes ret godt i eksemplet og det

er ikke muligt at afgøre om robotten f.eks. kan flytte fra toppen af scenen til nedre venstre hjørne.

5.1 Probabilistic Roadmaps (PRM)

Probabilistic Roadmaps, PRM, blev introduceret i 1996 af Kavraki et. al [9] som en ny metode til at løse planlægningsproblemer, vi tidligere har beskrevet. Hovedfokus for PRM var at løse problemer med et højt antal frihedsgrader og en række problemer, som RPP havde meget lille sandsynlighed for at løse.

PRM bygger et roadmap, R , over C_f ved at sample konfigurationer i C_f og forsøger at forbinde dem med andre konfigurationer. Dette resulterer i en eller flere træer i C_f . Dette trin kaldes konstruktionstrinnet. Efterfølgende, forsøges roadmappet udvidet, sådan at der bliver færre, men større, træer. Dette trin kaldes udvidelsestrinnet. De to trin vil blive beskrevet i dette kapitel. Bemærk at i [5]s gennemgang af PRM, nævnes intet om udvidelsestrinnet.

Konstruktionstrinnet

I konstruktionstrinnet samples N konfigurationer, som løbenede forsøges forbundet til konfigurationerne i nabolaget af den nye konfiguration. Da formålet med kortet er at finde ud af om der er en sti mellem to konfigurationer og ikke at finde den korteste sti, er der ingen grund til at tilføje en kant der giver en cykel i grafen. Ved at undgå at tilføje cykler, forbliver R en skov. Figur 5.2 viser en enkelt iteration af algoritmen.

Konstruktionstrinnet ser, i pseudokode, således ud:

Algoritme 5.1.1: CONSTRUCT()

comment: Bygger grafen (V,E)

$V \leftarrow \emptyset$

$E \leftarrow \emptyset$

for $i \leftarrow 1$ **to** N

$c \in_R C_f$

$N_c \leftarrow$ a set of candidate neighbors of c chosen from V

$V \leftarrow V \cup c$

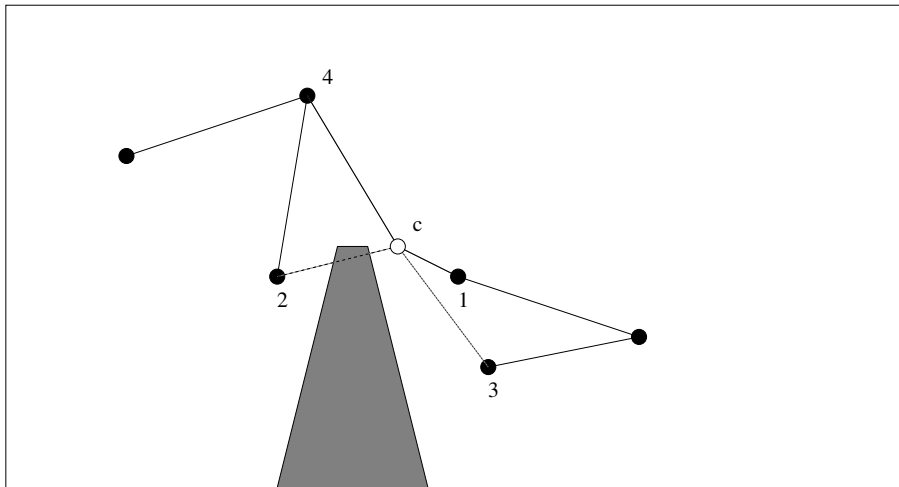
for each $n \in \text{SORT}(N_c)$

if not SAMECONNECTEDCOMPONENT(c, n) **and** FREE(c, n)

$E \leftarrow E \cup (c, n)$

De tilfældige konfigurationer findes ved uniform sampling, som beskrevet i afsnit 3.4. Der samples indtil en fri konfiguration er fundet, hvilket giver en uniform sampling af C_f .

Eftersom at sandsynligheden for at en linje i C skærer en forhindring, er større jo længere den er, sorteres konfigurationerne i nabolaget efter stigende afstand sådan at den første knuder der forsøges forbundet med c er den nærmeste konfigurationerne i R der ligger i nabolaget. Nabolag er defineret i 3.4. [9] foreslår at bruge *swept-volume* som afstandsfunktion, men forkaster det igen da det er for dyrt at beregne og ender med at bruge $dist_{max}$. Vi har dog valgt at bruge $dist_C$ ligesom i de andre algoritmer.



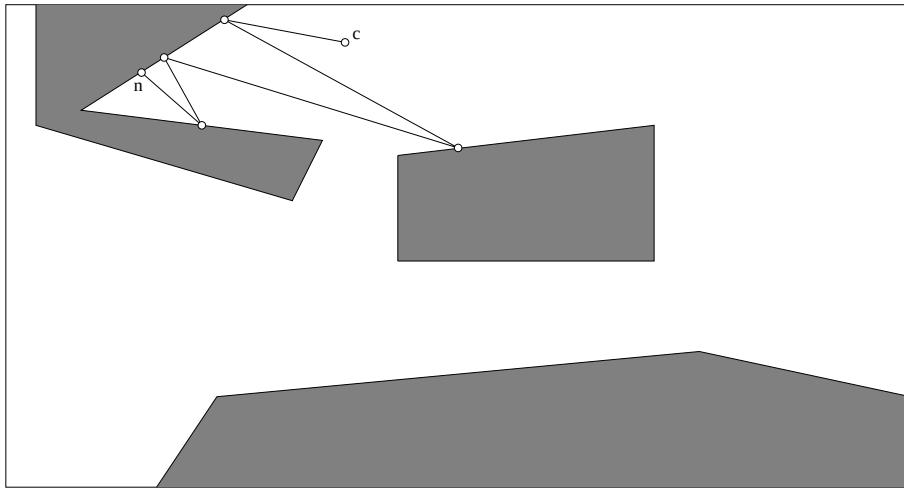
Figur 5.2: c forsøges forbundet til de fire nærmeste konfigurationer i stigende afstand. Først forbindes til 1 da der findes en kollisionsfri sti mellem c og 1. En kollision er netop grunden til at c og 2 ikke forbindes. c og 3 er allerede i samme sammenhængskomponent og forbindes derfor ikke. Til sidst forbindes c og 4 og vi har kun ét træ tilbage.

Artiklen nævner ikke hvordan sammenhængskomponenterne repræsenteres og holdes opdaterede. Vi har implementeret en *union-find* struktur, der effektivt løser opgaven. Strukturen er nærmere beskrevet i afsnit 6.1. Bemærk at hver gang der tilføjes en ny kant til E , slås to sammenhængskomponenter sammen, hvilket kræver en union operation. $\text{FREE}(c,n)$ er et kald til lige-linje-LPen som er beskrevet i 3.5.

Udvidelsestrinnet

Mængden af knuder som konstruktionstrinnet genererer, dækker C_f uniformt, men træerne forbinder ikke nødvendigvis alle de sammenhængende dele af C_f . Især scener, der er forbundet med smalle koridorer, giver problemer. Områder med med smalle koridorer kaldes svære områder, hvorimod åbne områder uden forhindringer er lette.

Udvidelsestrinnet, er et forsøg på at udvide R , på en sådan måde at der bliver færre sammenhængskomponenter. At to sammenhængskomponenter ikke er blevet til ét under konstruktionstrinnet, betyder at de enten ligger for langt fra hinanden eller at de ikke kan se hinanden direkte på grund af en forhindring der blokerer. I Udvidelsestrinnet vælges gentagende gange en konfiguration hvorfra der bygges en zig-zag-sti, en såkaldt *random bounce walk*, der forsøger at forbinde sammenhængskomponenter der ikke tidligere er blevet forbundet. En *random-bounce walk* starter med udgangspunkt i c og bevæger sig i en tilfældig retning indtil en forhindring mødes, derefter vælges en ny tilfældig retning, som følges indtil en forhindring mødes. På denne vis forsættes et fastlagt stykke tid. I vores implementation bruger vi 0,05 sek.. Et eksempel på en *random-bounce walk* kan ses i figur 5.3. I modsætning til [9], som indsætter en kant mellem c og n i R og lader denne kant indeholde informationer om



Figur 5.3: Eksempel på en random-bounce walk der starter i c . Hver gang en forhindring mødes, vælges en ny tilfældig retning hvori stien skal gå.

stien der førte til n , har vi valgt at indsætte alle de nye knuder, der bliver genereret. Dette giver en større graf, men gør også at alle kanter i R er konsistente med at vi fortolker kanter om kanterne i en visibilitygraf. Når n er indsat i R , forsøges konfigurationen forbundet til alle andre sammenhængskomponenter i R på samme vis som det sker i konstruktionstrinnet.

Efter konstruktionsfasen er store dele af C_f allerede dækket, men muligvis ikke forbundet igennem snævre områder. Derfor er det vigtigt at vælge konfigurationerne hvorfra der forsøges udvidet i svære områder. I [9] bruges et mål for sværhed, $w(c)$ for området omkring en konfiguration c . Jo større sværhed en konfiguration, c , har, jo større sandsynlighed er der for udvide fra c .

Der er tre forslag til hvordan $w(c)$ kan defineres.

1. Jo færre konfigurationer der er i nærheden af c , jo højere er $w(c)$.
2. Definer d_c som afstanden fra c til et nærmeste sammenhængskomponent der ikke indeholder c . $w(c)$ kan beregnes som $\frac{1}{d_c}$. $w(c)$ skal dog stadig normaliseres.)
3. Hvis der under konstruktionstrinnet holdes styr på hvor stor en andel kaldene til LP, der har fejlet, kan $\frac{failure}{tries}$ bruges som $w(c)$. Dette forhold vil netop være stort, hvis der er et stort antal konfigurationer omkring c , som ikke kan forbindes - Dette er indikation af at c ligger i et svært område. Bemærk at hvis LP mislykkedes i at forbinde c med en anden konfiguration, n , bliver både c og n talt op.

PRM bruger den tredje metode. Som den sidste del af konstruktionstrinnet, beregnes $r_f(c) = \frac{failure}{tries+1}$. Som første del af udvidelsestrinnet, beregnes $w(c) = \frac{r_f(c)}{\sum_{a \in V} r_f(a)}$, således at $\sum_{c \in V} w(c) = 1$.

Den sidste del af udvidelsestrinnet er at fjerne alle små sammenhængskomponenter. [9] påstår at det giver gode resultater at fjerne komponenter der in-

deholder mindre end 0,01% af det samlede antal konfigurationer. Der fremføres dog ingen begrundelser for dette.

Forespørgselsfasen

Efter konstruktionen af roadmappet, R , er formålet at finde en sti mellem to konfigurationer, s og g . R består af en eller flere sammenhængskomponenter. Der kan kun findes en sti fra s til g hvis begge konfigurationer kan forbindes, med en kollisionsfri sti, til samme sammenhængskomponent i R . Hvis disse stier findes, er det muligt at finde vejen mellem s og g f.eks. ved hjælp af en breddeførstsøgning.

At forbinde c_{start} og c_{goal} til et sammenhængskomponent, S , er identiske problemer. Vi nøjes derfor med at kigge på c_{start} . At forbinde s til S bliver gjort med samme teknik som når c skal forbindes til R . Først kigges på nærmeste naboer sorteret efter stigende afstand. Hvis der fra en af disse findes en kollisionsfri sti fra s til S , forbindes s til S - og vi er færdige. Hvis ikke det lykkedes at forbinde s til S på denne måde, foretages en random-bounce-walk fra s . Disse nye tilstande tjekkes nu for en kollisionsfri sti til deres naboer i S , hvis en sådan findes, forbindes s til S . Der prøves maksimalt $N_{RB-QUERY}$ at lave random-bounce-walk.

5.2 Sampling-based Roadmap of Trees (SRT)

SRT blev introduceret i [4] i 2003. I modsætning til en stor del af resultaterne, der blev præsenteret efter den originale PRM artikel, [9], som har beskæftiget sig med *single query* algoritmer, er SRT en *multi query* algoritme, ligesom PRM er det.

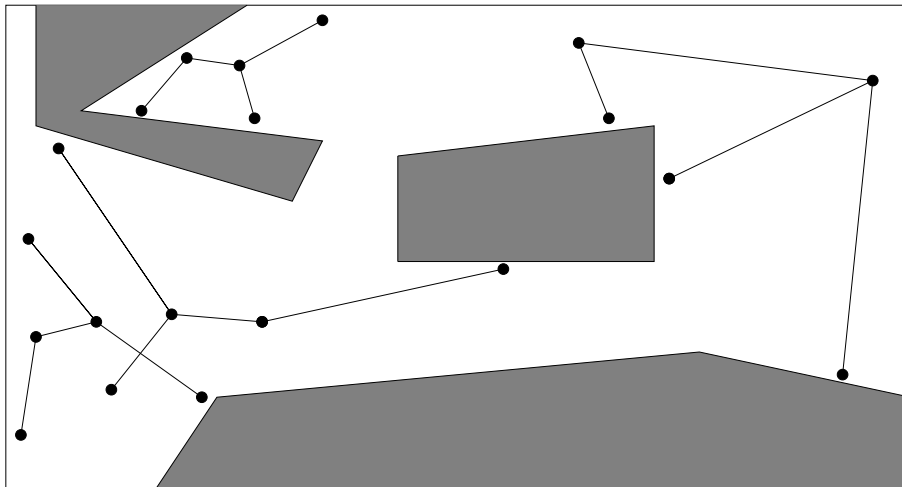
Grundlæggende, har SRT en hel del til fælles med PRMs måde at gøre tingene på, og det skinner da også igennem at Lydia Kavraki har været medforfatter på begge artikler.

PRM bliver præsenteret på følgende vis i [4] og [5]. Der vælges n konfigurationer i C_f , som hver især forsøges forbundet med de k nærmeste knuder med en lige-linje-LP. Denne simple algoritme bliver brugt som udgangspunkt for SRT, men SRT bygger en graf med milesten som knuder i stedet for konfigurationer. En milesten sten er et RRT-træ eller EST-træ med et fast antal konfigurationer. Vi har valgt kun at bruge RRT-træer. Det vil sige at roadmappet indeholder en mængde træer der alle er bygget med RRT-algoritmen. Se eksemplet i figur 5.4, hvor n er fire og det endnu ikke er forsøgt at forbinde milestenene. Hver af de fire milesten består af fem konfigurationer.

Den helt store forskel på PRM og SRT, er at SRT bruger RRT som LP. RRT er en meget langsommere algoritme end lige-linje-LP, men finder til gengæld oftere en vej mellem to konfigurationer.¹

En problemstilling som de tidligere præsenterede algoritmer ikke har givet anledning til, er at afgøre afstanden mellem konfigurationstræer. SRT bruger afstanden mellem træer til at finde de k nærmeste naboer med. I [4] bruges en afstandsfunktion, der er defineret på C på de to træers centroide. Træets centroider er en konfiguration hvis koordinater svarer til gennemsnittet af alle

¹Se afsnit 3.5 for en diskussion af valget af LP.



Figur 5.4: Fire milesten der hver består af fem konfigurationer. Træerne er ikke 100% korrekte RRT-træer, men tanken er at de er bygget med RRT-algoritmen.

træets elementer. Vi har valgt at gøre det på samme måde, og bruger $dist_C$ til at afgøre afstande mellem centroider.

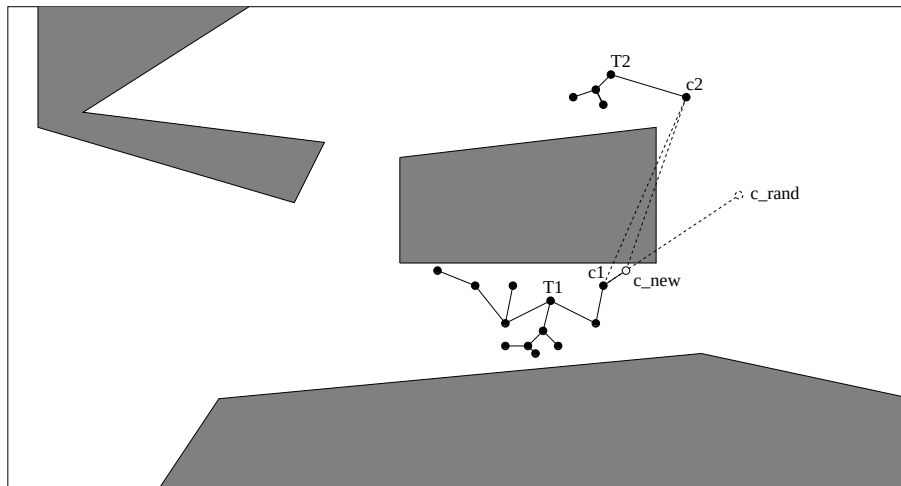
Forbindelse af milesten Efter at have samlet milesten og fundet de k nærmeste milesten til hver af dem, skal de forbindes. I modsætning til PRM, bruger SRT ikke kun en lige-linje-LP, når to træer, T_1 og T_2 , skal forbindes, men derimod følgende lokal planlægger: (Se figur 5.5 for eksempel):

- Vælg en tilfældig konfiguration c_{rand} og find de to konfigurationer, c_1 og c_2 fra hhv. T_1 og T_2 som er tættest på c_{rand} .
- Hvis en lige-linje-LP kan forbinde c_1 og c_2 , returneres kanten (c_1, c_2) .
- Ellers findes en ny konfiguration c_{new} ved at gå op til $step_size$ ud af den lige linje mellem c_1 og c_{new} . c_{new} og kanten (q_1, q_{new}) indsættes i T_1 . Hvis c_{new} og c_2 kan forbindes med en lige-linje-LP, returneres stien fra c_1 til c_2 over c_{new} .
- Hvis c_{new} og c_2 ikke kan forbindes, køres proceduren forfra men med de to træers rolle byttet om.

Denne procedure udføres indtil der er blevet fundet en sti, dog maksimalt et fastlagt antal gange. I figur 5.5 bliver de to træer ikke forbundet, men kanten (q_1, q_{new}) indsættes i T_1 .

Forespørgsler

Start- og slutkonfigurationen prøves hver især forbundet til de k nærmeste milestones. Dette gøres ved at bruge den lokale planlægger, der er beskrevet ovenfor, på hver af de k milesten og hhv. start- og slutkonfigurationen.



Figur 5.5: Da c_1 og c_2 ikke kan forbindes direkte, samples en konfiguration c_{rand} som giver anledning til en ny konfiguration c_{new} . c_{new} ligger maksimalt $step_size$ fra c_1 . Da c_2 ikke kan forbindes til c_{new} , forbindes T_1 og T_2 ikke i denne omgang.

Distribueret SRT

I [14] beskrives en distribueret udgave af SRT, med nær lineær skalering over processorer. Udgangspunktet er at det meste af SRTs udførselstid ligger i beregningen af RRT-træerne og arbejdet med at forbinde disse. Bygningen af et enkelt RRT-træ er helt uafhængigt af de andre, så det lige til at distribuere. At distribuere arbejdet med at forbinde træerne er mere omstændeligt, men kan stadig gøres effektivt over flere maskiner. Vi har dog ikke arbejdet med at distribuere beregningerne.

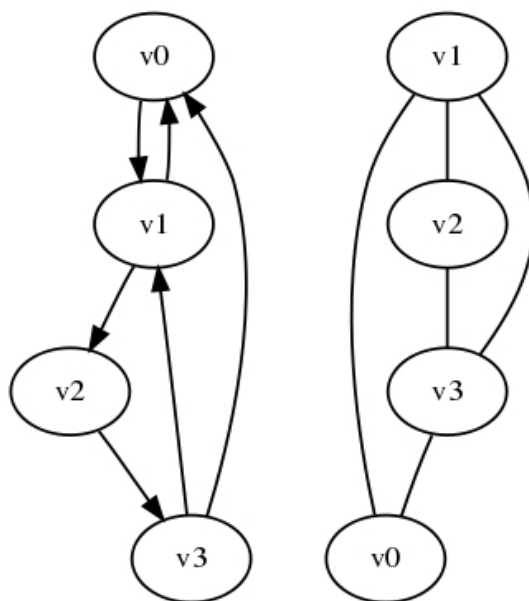
6 Hjælpeværktøjer

6.1 Graf

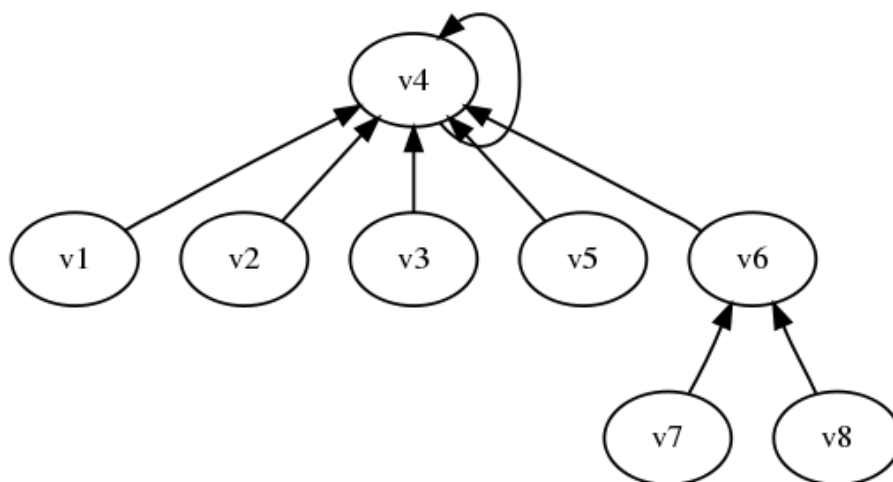
En graf består af en række knuder, V , og en række kanter, E , der forbinder par af knuder fra V , altså

$$E = \{(u, v) | u \in V, v \in V\}$$

En graf kan enten være orienteret eller ikke-orienteret. I en orienteret graf er kanterne ikke nødvendigvis symmetriske. Dvs. kanten (v, u) behøves ikke findes bare fordi (u, v) gør. I en ikke-orienteret graf er alle kanter symmetriske, så det gælder at $(u, v) = (v, u)$. To knuder, u og v , kaldes naboer, hvis der findes en kant $(u, v) \in E$.



Figur 6.1: Hhv. en orienteret og en ikke-orienteret graf.



Figur 6.2: Alle otte knuder er i samme sammenhængskomponent og er identificeret ved $v4$.

Repræsentation

En graf kan repræsenteres på en række forskellige måder. De mest almindelige er "nabomatrice" eller "naboliste". Da knuderne i graferne, vi arbejder med, har ganske få naboer, ofte et konstant antal, bruger vi en nabolisterepræsentation. Pga. det konstante antal naboer, bruger grafen $O(n)$ plads, hvorimod en nabomatricerepræsentation bruger $O(n^2)$.

Sammenhængskomponenter

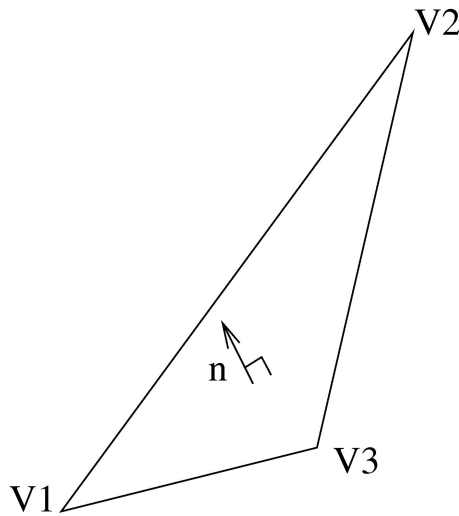
For at afgøre om en række knuder er et sammenhængskomponent, bruger vi union-find datastrukturen, som blandt andet beskrives i [6][4.2.3]. Som navnet antyder, understøtter datastrukturen to operationer nemlig $\text{UNION}(c_1, c_2)$, som forener to sammenhængskomponenter, og $\text{FIND}(v)$, som fortæller i hvilket sammenhængskomponent v ligger.

Idéen er at bygge en træstruktur for hvert sammenhængskomponent, hvor roden definerer sammenhængskomponenten. Se figur 6.2

$\text{UNION}(c_1, c_2)$ Når to sammenhængskomponenter skal slås sammen til én, sætter man blot roden af den ene til at pege på roden af den anden. Da stier fra en knude i træet til roden bliver komprimeret på et senere tidspunkt, dvs. at knuder kommer til at pege direkte på roden, vælger vi at indsætte det mindste træ som et undertræ i det største træ.

$\text{FIND}(v)$ Ved at følge stien af knuder indtil roden nås, kan det afgøres hvilke sammenhængskomponent en knude ligger i. Ved at sætte alle knuder i stien fra v til roden til at pege på roden, tager en efterfølgende forespørgelse på en af disse knuder konstant tid.

I [6] argumenteres for at en serie af n union og find operationer tager $O(n \log^*(n))$. Vi vil ikke gennemgå disse argumenter.



Figur 6.3: En trekant med hjørnepunkter og normal

6.2 Trekanter

En trekant, t , er repræsenteret ved dens tre hjørner, v_1 , v_2 og v_3 .

Vi siger, at forsiden af en trekant er den side hvor vektoren n , som beregnes som $(v_2 - v_1) \times (v_3 - v_1)$, peger væk fra. n kaldes for t s normal. I matematikken er n blot én af to normaler, men vi bruger den ovenstående definition, som er udbredt indenfor computer grafik. Se figur 6.3.

Trekantsnet

For at kunne repræsentere mere komplekse forhindringer og robotter, sammensættes en mængde trekanter til at danne overfladen af et objekt. Denne måde at repræsentere objekter på, er standardmetoden indenfor computer-grafik og bruges også af hovedparten af tekniske tegneprogrammer. Traditionelt bruges udtrykket "triangle mesh". Vi har valgt at oversætte dette til "trekantsnet".

Af grunde, der vil blive forklaret i kapitlet om kollisionstest, har vi et ønske om, at alle trekanter har deres forside til at pege væk fra trekantsnettes indre. Dette er tilfældet, hvis alle trekanter har deres hjørner angives i rækkefølge med uret, når man kigger på objektet ude fra.

6.3 Biased Bingo Bag

I flere af planlægningsalgoritmerne, har vi brug for en datastruktur, der kan udtrække et tilfældigt element blandt en række vægtede kandidater.

Vi har lavet en datastruktur, som vi kalder Biased Bingo Bag, der kan bruges til formålet på følgende måde: Hvert element har en vægt, w , mellem 1 og D . Sandsynligheden for at et element, i , udtrækkes er $\frac{w_i}{\Sigma}$ hvor Σ er summen

af alle vægtene. Til det formål datastrukturen er udviklet til, er der brug for følgende operationer:

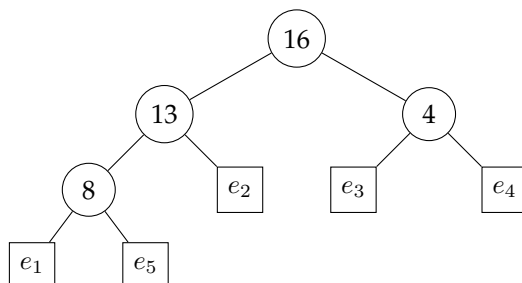
- INSERT(e) - Indsætter e og tildeler det standardværdien D .
- SELECT() - Returnerer et tilfældigt element baseret på vægtene.
- DECREASE(e) - sænker vægten på e med 1. Er e s vægt allerede 1, gør dette kald ingenting.

Ved at opfatte vægtene som intervaller med forskellige størrelse, som på figur 6.4, vil et tilfældigt valgt heltal fra mængden $[0, \Sigma)$, kunne bruges til at udvælge et element med en sandsynlighed, der stemmer overens med vægtene.



Figur 6.4: Fem elementer med vægte: $e_1: 8, e_2: 3, e_3: 4, e_4: 2, e_5: 5$

Ved at opretholde et balanceret binært søgetræ, som indeholder elementerne i bladene og i de interne knuder indeholder summen af det venstre undertræ, kan alle operationerne udføres i $O(\log n)$ tid. Se figur 6.5.



Figur 6.5:

Da knuderne udelukkende består af et heltal og en pointer til et element, har vi valgt at gemme disse data i to arrays. Eksemplet giver anledning til de to arrays, der kan ses i figur 6.6

value[i]	16	13	4	8	3	4	2	8	5	...
elem[i]					e_2	e_3	e_4	e_1	e_5	...

Figur 6.6:

Følgende pseudokode viser hvorledes de tre operationer er implementeret. UPDATE(i, δ) er en hjælpefunktion, der ændrer summen op igennem træet, når størrelsen af et interval er blevet ændret eller et nyt interval er blevet tilføjet.

Algoritme 6.3.1: INSERT(e)

```

if ISEEMPTY()
   $n[1].value \leftarrow D$ 
   $n[1].elem \leftarrow e$ 
   $sum \leftarrow sum + D$ 
   $e.nodeIndex \leftarrow 1$ 
else
   $parent \leftarrow \frac{last}{2}$ 
   $left \leftarrow last$ 
   $right \leftarrow left + 1$ 
   $n[left].value \leftarrow n[parent].value$ 
   $n[left].elem \leftarrow n[parent].elem$ 
   $n[right].value \leftarrow D$ 
   $n[right].elem \leftarrow e$ 
   $n[right].elem.nodeIndex \leftarrow right$ 
   $n[left].elem.nodeIndex \leftarrow left$ 
   $last \leftarrow last + 2$ 
   $\Sigma \leftarrow \Sigma + D$ 
  UPDATE( $right, D$ )

```

Algoritme 6.3.2: UPDATE(i, δ)

```

 $parent \leftarrow \text{if } parent \bmod 2 = 0$ 
   $n[parent].value \leftarrow n[parent].value + \delta$ 
if  $parent \neq 1$ 
  UPDATE( $parent, \delta$ )

```

Algoritme 6.3.3: DECREASE(e)

```

 $i \leftarrow e.nodeIndex$ 
if  $n[i].value > 1$ 
   $n[i].value \leftarrow n[i].value - 1$ 
  UPDATE( $i, -1$ )
   $\Sigma \leftarrow \Sigma - 1$ 

```

SELECT(p) returnerer et element givet et heltal mellem 0 og $\Sigma - 1$,

Algoritme 6.3.4: SELECT(p)

```

input ( $p$ ) :  $p \in [0, \Sigma)$ 
 $i \leftarrow 1$ 
while  $last > 2i$ 
  if  $p < n[i].value$ 
     $i \leftarrow 2i$ 
  else
     $i \leftarrow 2i + 1$ 
     $p \leftarrow p - n[i].value$ 
return ( $n[i].elem$ )

```

6.4 Naboer i C

Alle de planlægningsalgoritmer, vi har implementeret, har et behov for effektivt at kunne finde en eller flere nærmeste naboer til en given konfiguration. Ved de n nærmeste naboer til en konfiguration x , forstås de n konfigurationer, der givet en afstandsfunktion, har de n mindste afstande til x .

Den naive løsning er at beregne afstanden til samtlige konfigurationer og vælge konfigurationerne med de n mindste afstande. For at løse problemet effektivt har vi brug for en søgestruktur, der kan finde naboerne hurtigere - og særligt at den kan gøre det i de højdimensionelle rum, planlægningsproblemerne giver anledning til.

Et kD -træ er en datastruktur, som effektivt finder nærmeste naboer. Elementerne i disse træer er punkter i $\mathbb{R}^k$ og afstandsfunktionen er euklidisk afstand i rummet. Strukturen løser nabo-problemet effektivt, selv i høje dimensioner.

I forhold til vores algoritmer er der dog to problemer med kD -træer:

1. kD -træet er en statisk struktur, men planlægningsalgoritmerne har alle brug for en dynamisk struktur for at kunne indsætte elementer løbende.
2. Konfigurationer eksisterer i et ikke-euklidisk rum, i det rotationerne er beskrevet ved quaternioner. Quaternionerne kan repræsenteres af rummet $\mathbb{R}P^3$ - En 3D-sfære i $\mathbb{R}^4$, hvor modsatte poler er identiske quaternioner. For at kunne bruges i vores algoritmer, skal træet altså kunne benytte en afstandsfunktion, der arbejder i både euklidiske rum og i $\mathbb{R}P^3$.

Yershova et al. [16] beskriver hvordan kD -træet kan tilpasses til at håndtere $\mathbb{R}P^3$ og yderligere til at være dynamisk, i den forstand at det effektivt understøtter indsættelse af nye elementer løbende.

I de følgende afsnit vil vi beskrive dette tilpassede kD -træ, som vi har implementeret til vores algoritmer. Først definerer vi afstandsfunktionen for vores konfigurationer. Derefter beskriver vi konstruktionen af og søgningen i statiske kD -træer, som arbejder med denne afstandsfunktion, der er defineret i både euklidiske rum og $\mathbb{R}P^3$. Til sidst beskriver vi den videre tilpasning, der gør strukturen dynamisk.

Afstandsfunktion for C

Rummet for vores konfigurationer er en kombination af robottens forskydning i $\mathbb{R}^3$ og dens rotationer i $\mathbb{R}P^3$. Afstandsfunktionen er således en kombination af afstande i disse to rum. Translationens afstand er euklidisk, mens en rotationsafstand er længden af buen, på quaternionernes 3D-sfære, mellem to quaternioner. For hele C er den samlede afstand kvadratroden af summen af del-rummenes kvadrerede afstande.

Inden vi definerer afstandsfunktionen vil vi lige minde om definitionen af en konfiguration: $(t, Q) = ([t_x, t_y, t_z], \{q_1, \dots, q_n\})$.

Delafstandene for forskydningen og de enkelt rotationer kan vægtes med vægtene $w_0, \dots, w_n$, således at f.eks. rotation af ét led er dyrere end rotation af et andet. Dette er dog ikke noget vi har eksperimenteret med. Afstandsfunktionen for konfigurationer i C er defineret i figur 6.7.

Rum	Afstandsfunktion
$\mathbb{R}^3$	$dist_{\mathbb{R}^3}(t_1, t_2) = \sqrt{(t_{1x} - t_{2x})^2 + (t_{1y} - t_{2y})^2 + (t_{1z} - t_{2z})^2}$
$\mathbb{R}P^3$	$dist_{\mathbb{R}P^3}(q_\phi, q_\chi) = \cos^{-1} q_\phi \cdot q_\chi $
C	$dist_C((t_1, Q_1), (t_2, Q_2)) = \sqrt{w_0 dist_{\mathbb{R}^3}^2(t_1, t_2) + \sum_{i=1}^n w_i dist_{\mathbb{R}P^3}^2(q_{1i}, q_{2i})}$

Figur 6.7: Afstandsfunktion mellem to konfigurationer

Udover afstande mellem konfigurationer, skal træet også bruge afstande mellem konfigurationer og de kasser, som et kD -træ opdeler rummet i. For $\mathbb{R}$ benyttes Hausdorff-afstanden, der for et punkt p og en kasse R , blot er afstanden mellem p og det nærmeste punkt i R . For $\mathbb{R}P^3$ forstås kassen som et rektangel i $\mathbb{R}^4$, der indeholder en del af quaternionernes enhedssfære. Afstanden til den er Hausdorff-afstanden i $\mathbb{R}^4$ mellem kassen og quaternionens vektor $[x, y, z, w]$, eller mellem kassen og den alternative vektor $[-x, -y, -z, -w]$, svarende til den modsatte, men ækvivalente quaternion, hvis denne afstand er mindre. Den kombinerede afstand for C er igen kvadratroden af summen af del-rummenes kvadrerede og vægtede afstande.

I figur 6.8 ses den formelle definition. I definitionerne angiver R en kasse, og når denne sendes videre til delrumsfunktioner, antages det at funktionerne selv finder de informationer i kassen, som er relevante for deres beregninger.

Som et eksempel ville en enkelt robot, med et enkelt led, med forskydning og rotation i 3D give afstandsfunktionen:

$$dist_C((t, Q), R) = \sqrt{w_0 dist_{\mathbb{R}^3}^2(t, R) + w_1 dist_{\mathbb{R}P^3}^2(q_1, R) + w_2 dist_{\mathbb{R}P^3}^2(q_2, R)}$$

Statisk kD -træ med afstandsfunktion for $\mathbb{R}^3$ og $\mathbb{R}P^3$

Konstruktionen af kD -træet foregår som et almindeligt kD -træ. Dvs. konfigurationer opfattes som vektorer i $\mathbb{R}^d$ og der tages ikke højde for den specielle afstandsfunktion og de alternative, men ækvivalente, quaternion-vektorer.

Rum	Afstandsfunktion
$\mathbb{R}$	$dist_{\mathbb{R}}(p, [a, b]) = \min_{r \in [a, b]}(dist_{\mathbb{R}}(p, r))$
$\mathbb{R}^m$	$dist_{\mathbb{R}^m}([p_1, \dots, p_m], R) = \sqrt{\sum_{i=1}^m dist_{\mathbb{R}}^2(p_i, R)}$
$\mathbb{R}P^3$	$dist_{\mathbb{R}P^3}(q_{\phi}, R) = \min(dist_{\mathbb{R}^4}(q_{\phi}, R), dist_{\mathbb{R}^4}(-q_{\phi}, R))$
C	$dist_C((t, Q), R) = \sqrt{w_0 dist_{\mathbb{R}^3}^2(t, R) + \sum_{i=1}^n w_i dist_{\mathbb{R}P^3}^2(q_i, R)}$

Figur 6.8: Afstandsfunktion mellem en konfiguration og en kasse

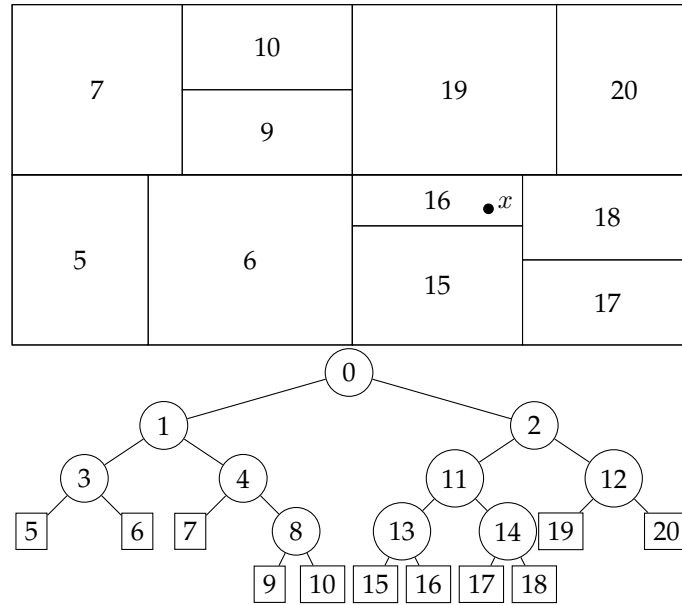
Konfigurationens forskydning giver således de tre første dimensioner og quaternionerne tilføjer blot fire ekstra dimensioner hver, i form af deres vektorer. En konfiguration for en robot med ét led, som i det tidligere eksempel, vil således fortolkes som en vektor i $\mathbb{R}^{11}$.

Den "almindelige" konstruktionen forløber ved at rummet rekursivt opdeles i mindre kasser. For en given kasse, vælges den længste side som den dimension kassen skal opdeles i. Hvis der er konfigurationer på begge sider af midten af denne side, opdeles kassen der. Ellers placeres linjen lige inden den nærmeste konfiguration på den side hvor konfigurationerne ligger. Således gøres den tomme side, så stor som muligt. For hver kasse gemmes information om hvilken dimension, der blev splittet efter. Opdelingen fortsætter indtil hver kasse kun indeholder op til et vist antal elementer. Disse små kasser bliver således blade i træet og kassernes elementer gemmes i heri.

Søgning Træet skal bruges til at svare på spørgsmålet "Hvilke n konfigurationer ligger tættest på konfigurationen x ?". Søgningen udføres ved hjælp af rekursive kald ned gennem træet og dets kasser. Kaldene finder først frem til det blad/den kasse, som x ligger indeni. Her findes de n nærmeste konfigurationer blandt bladets konfigurationer. Efterfølgende følges stien til bladet baglæns og der kaldes rekursivt på nabo-kasserne i hvert niveau, såfremt afstanden mellem en given kasse og x , ikke er større end det dårligste af de n hidtil bedste naboer - Altså hvis der er en mulighed for at kassen indeholder en af de naboer, der skal med i det endelige svar. Figur 6.9 viser et eksempel på et kD -træ i 2D.

Imodsætning til konstruktionen, benytter søgningen sig af de "rigtige" afstandsfunktioner, som vi definerede i tabellerne ovenfor. Da den kombinerede afstandsfunktion for et givent planlægningsproblems rum kan have relativt mange faktorer, benytter [16] sig så vidt muligt af en inkrementel beregning af afstandene mellem kasser og søgekonfigurationen x , som gør træet mere effektivt. Nærmere forklaret betyder det, at vi givet afstanden til en kasse B beregner afstanden til de to kasser, som B er opdelt i, ved hjælp af afstandsfunktionen svarende til den dimension/det delrum, hvori B blev delt i to, fremfor hele den kombinerede afstandsfunktion.

Lad os illustrere den inkrementelle beregning med et eksempel. Antag at vi har en konfiguration (t, Q) og en kasse B , som er opdelt i kasserne, B_l og B_r . Opdelingen af B , antager vi, er sket i en dimension hørende til den tredje



Figur 6.9: Et eksempel på et kD -træ med to dimensioner og et søgepunkt x heri. Punktet ligger i kasse nummer 16, så der søges ned til denne først. Dernæst bliver kasse 15 undersøgt, og efterfølgende kasse 14. Er 14 ikke for langt væk, undersøges kasserne 17 og 18 ligeledes. Næstefters undersøges kasse nummer 12, samt dennes delkasser og til sidst er det kasse nummer 1, der checkes og hele dennes undertræ, så længe kasserne ikke ligger for langt væk i forhold til de naboer, der allerede er fundet.

quaternion q_3 . Vi er givet afstanden mellem (t, Q) og B , lad os kalde afstanden d_{box} . Vi beregner nu afstanden fra q_3 til B_l og B_r , lad os kalde afstandene $d(q_3, B_l)$ og $d(q_3, B_r)$:

$$d(q_3, B_l) = \text{dist}_{\mathbb{R}P^3}^2(q_3, B_l)$$

$$d(q_3, B_r) = \text{dist}_{\mathbb{R}P^3}^2(q_3, B_r)$$

Nu vil vi gerne finde afstandene $d_{B_l} = \text{dist}_C((t, Q), B_l)$ og $d_{B_r} = \text{dist}_C((t, Q), B_l)$, men uden brug af dist_C .

Lad os, uden tab af generalitet, antage at $d(q_3, B_l) \leq d(q_3, B_r)$. Det medfører at afstanden mellem q_3 og B_l er den samme, som afstanden mellem q_3 og B . Enten fordi q_3 ligger i B_l og dermed også i B , eller fordi q_3 ligger udenfor B , og B_l , men på B_l 's side af den opdelende linje, der adskiller B_l og B_r . At afstanden fra q_3 til B_l og B er den samme, betyder videre at $d_{B_l} = d_{box}$, altså at afstanden mellem (t, Q) og B_l er lig afstanden mellem konfigurationen og B .

Den anden afstand, d_{B_r} , kan nu beregnes ved at fjerne $d(q_3, B_l)$ faktoren fra d_{box} , og i stedet tilføje $d(q_3, B_r)$. Herved opnås resultatet, da kasserne i alle andre dimensioner er lige langt fra konfigurationen, og faktorerne dermed er de samme.

Alt i alt har vi altså:

$$d_{B_l} = d_{box}$$

$$d_{B_r} = d_{box} - d(q_3, B_l) + d(q_3, B_r)$$

Vi vil nu vise og gennemgå pseudokoden for søgning i træet.

Algoritme 6.4.1: KDTREE.SEARCH(x, n)

```

 $d_{box} \leftarrow dist_C^2(x, root.GETBOX())$ 
 $d_{neighbors} \leftarrow new\ double[]\{\infty, \dots, \infty\}$ 
 $neighbors \leftarrow new\ Configuration[n]$ 
 $root.SEARCH(x, d_{box}, d_{neighbors}, neighbors)$ 
return ( $neighbors$ )

```

Søgningen starter med at afstanden mellem x og træets yderste kasse beregnes med $dist_C$. To tomme lister laves. $neighbors$ skal løbende opdateres så den indeholder de n nærmeste naboer, som søgningen har fundet på et givent tidspunkt. $d_{neighbors}$ indeholder afstandene til x for de konfigurationer, der ligger i $neighbors$.

Søgningen sendes nu videre til roden i træet.

Algoritme 6.4.2: NODE.SEARCH($x, d_{box}, d_{neighbors}[], neighbors[]$)

```

 $d_{worst} \leftarrow d_{neighbors}[d_{neighbors}.length - 1]$ 
if  $d_{box} < d_{worst}$ 
     $T \leftarrow$  delrum hvori den givne kasse blev splittet.
     $d_1 \leftarrow dist_T^2(x, v_1.GETBOX())$ 
     $d_2 \leftarrow dist_T^2(x, v_2.GETBOX())$ 
    if  $d_1 < d_2$ 
         $v_1.SEARCH(x, d_{box}, d_{neighbors}, neighbors)$ 
         $v_2.SEARCH(x, d_{box} - d_1 + d_2, d_{neighbors}, neighbors)$ 
    else
         $v_2.SEARCH(x, d_{box}, d_{neighbors}, neighbors)$ 
         $v_1.SEARCH(x, d_{box} - d_2 + d_1, d_{neighbors}, neighbors)$ 

```

Når søgningen når en given knude, v , findes afstanden for den konfiguration, x_{worst} i $neighbors$, som ligger længst fra x . Hvis afstanden mellem x og v 's kasse, som er givet ved d_{box} , er mindre end x_{worst} 's afstand til x , så er det muligt at der ligger konfigurationer i kassen, som er tættere på x end x_{worst} . Derfor skal søgningen fortsætte ned til knudens børn. Er det ikke tilfældet skal der ikke søges videre her.

Når søgningen skal sendes videre til knudens børn, udføres den inkrementelle beregning af afstandene mellem x og børnenes kasser. Søgningen fortsætter først ned til det barn, hvis kasse er nærmest x .

Algoritme 6.4.3: LEAF.SEARCH($x, d_{box}, d_{neighbors}[], neighbors[]$)

```

for each  $x_{leaf} \in leaf.GETCONFIGURATIONS()$ 
  if  $dist_C^2(x, x_{leaf}) < \text{en værdi } x_i \text{ i } d_{neighbors}$ 
    Skub værdier mod højre fra position  $i$ , og lad den sidste værdi forlade listen,
    Indsæt  $dist_C^2(x, x_{leaf})$  i listen på position  $i$ .
    Opdater  $neighbors$  tilsvarende.

```

Søgning i træets blade foregår ved at løbe bladets liste af konfigurationer igennem. Hvis en konfiguration x_{leaf} fra bladet har en kortere afstand til x end én eller flere af konfigurationerne i $neighbors$, indsættes x_{leaf} foran alle disse i listen, og det sidste element i listen fjernes helt fra listen. Således er listen altid sorteret efter afstand med den nærmeste nabo forrest i listen.

Der benyttes kvadrerede afstande for at undgå unødvendige og dyre kvadratsberegninger.

Dynamisk kD -træ

Planlægningsalgoritmerne har som tidligere nævnt brug for at kunne indsætte nye knuder i kD -træet løbende. For at gøre strukturen dynamisk på denne måde, består det ikke af et enkelt træ, men i stedet af en liste af op til $\log n$ træer, givet n elementer i strukturen. Mere præcist indeholder strukturen et statisk kD -træ for hvert 1-tal i den binære repræsentation af n . Træet for det i 'te ciffer, som er 1, indeholder 2^i elementer. Med $n = 5$ er der altså to statiske kD -træer i listen: Et med ét element i og et med fire elementer i.

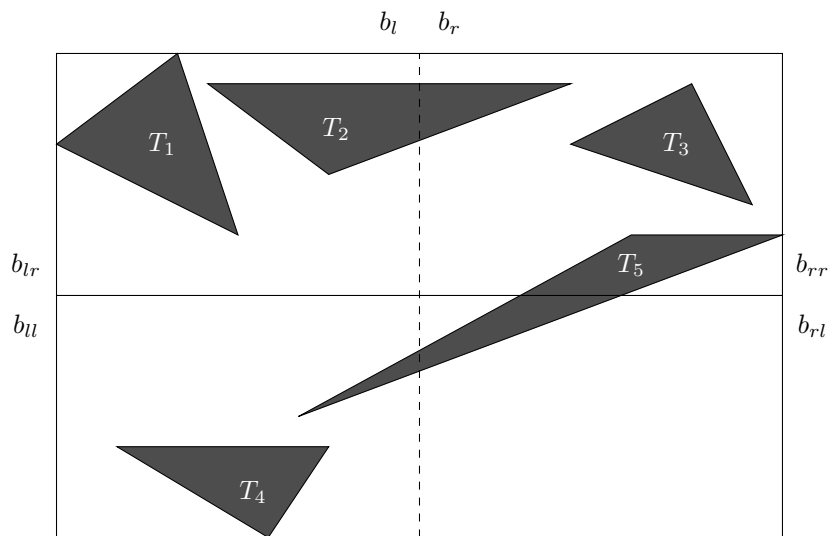
Når et element tilføjes, slettes træerne svarende til de cifre, der er ændret til 0'er. Elementerne fra de slettede træer, samt det nye element, danner et nyt statisk træ, svarende til det ciffer, der er blevet ændret til 1'er. Tilføjes der et element til træet med $n = 5$, slettes det statiske træ, som indeholder ét element, altså fra listen, og elementer derfra samt det nye element lægges i et nyt træ. Herefter er $n = 6$, og træet indeholder igen to statiske træer, men nu med hhv. fire og to elementer.

Søgning En søgning i denne dynamiske struktur løses ved at udføre søgningen på hvert af de "indre" statiske træer et ad gangen. Det andet træ i rækken tager udgangspunkt i det forrige træs svar, og søger således ikke i kasser, der er længere væk end de n nærmeste naboer, der er fundet hidtil - I samme træ eller foregående træer.

6.5 Kollisionstest

I vores planlægningsalgoritmer, består både robot og forhindringer af trekanter i rummet. Da der kan være tale om titusindevis af trekanter i både robot og forhindringer, har vi brugt tid på at implementere en kollisionstest der kan afgøre om en robot skærer en forhindring langt mere effektivt end den naive metode. Eftersom specialet ikke omhandler kollisionstest, er det ikke nødvendigt at læse dette afsnit for at forstå planlægningsalgoritmerne. Det er dog vigtigt at vide at et meget stor del af udførselstiden for alle planlægningsalgoritmerne bliver brugt på kollisionstest og at en forbedring i kollisionstesteren derfor vil betyde en forbedring for algoritmerne.

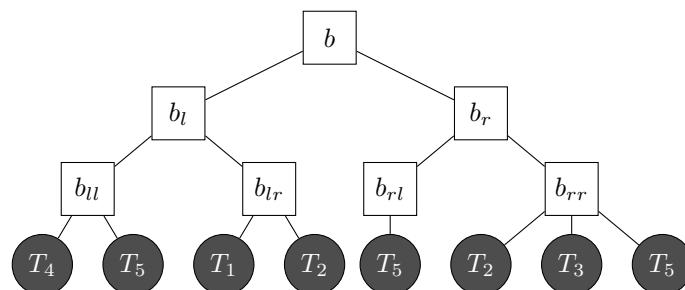
Alle robotterne vi har brugt i test består af kun otte trekanter per lem, og vi har på baggrund af dette valgt at teste alle trekanter for kollision med scenen. Over trekanterne i scenen, bygger vi et såkaldt *Bounding Volume Hierarchy*, BVH, som er et binært søgetræ der rekursivt opdeler rummet i mindre og mindre kasser. Repræsentationen af kasserne er beskrevet i afsnit 6.5. I figur 6.10 er illustreret hvordan en scene med fem trekanter rekursivt deles op. Det resulterende BVH-træ kan ses i figur 6.11. Bemærk at der ikke er nogen garantier for dybden af træer, men at det i praksis virker fornuftigt. Hver gang kassen skal deles, sker det tværs af kassens længste led.



Figur 6.10: Opdelingen af en kasse sker altid på den længste led af kassen. I dette eksempel, bliver

Som nævnt i afsnit 6.2, er trekantsnet hule skaller hvor selve skallen består af trekanter. En robot kan være i kollision med en scene på én ud af to måder:

- De kan skære hinanden, hvilket betyder at mindst én trekant fra robotten skærer mindst én trekant fra scenen skærer hinanden.



Figur 6.11: Konstruktion af et BVH-træ med tre niveauer.

- Robotten er helt indeholdt i en forhindring. Selv om der ikke er nogen trekanter der skærer hinanden, ønsker vi at denne situation opfattes som en kollision.

Gennemgangen af de to tilfælde vil ske i de næste fire delafsnit. Det første vil beskrive søgning i et BVH-træ efter en skæring mellem en trekant og et trekantsnet. De næste vil beskrive hvorledes man kan afgøre om en trekantsnet er komplet indeholdt i et andet. I de sidste to opsummerer to artikler af Tomas Akenine-Möller hvori han præsenterer hurtige metoder til at afgøre kasse-trekant-kollision og trekant-trekant-kollision.

Søgning efter en skæring i et BVH-træ

Givet en trekant, t , er vi interesseret i hurtigst muligt at udelukke så mange af scenens trekanter som muligt. Idéen bag at bruge et BVH-træ er, at man ved at teste om t skærer en indre knude i træet, lynhurtigt kan afgøre om t kan skære nogle af de trekanter knuden indeholder. Hvis ikke trekanten skærer kassen, der svarer til den indre knude, betyder det at alle trekanter i kassen ligger så langt væk fra t at der ikke er mulighed for en kollision. Der er derfor ingen grund til at undersøge kassen og dens elementer yderligere. Skærer t kassen, er det nødvendigt at undersøge begge kassens børn. Hvis kassen er et blad i træet, testes t for kollision med alle kassens trekanter. Algoritmen til at afgøre kollisioner mellem to trekanter er beskrevet senere i dette afsnit. Pseudokoden for søgning efter en skæring kan ses i algoritme 6.5.1.

Algoritme 6.5.1: FINDINTERSECTION($bvhNode, t$)

```

if not BOXTRIANGLEINTERSECTION( $bvhNode, t$ )
  return ( false )
if ISLEAF( $bvhNode$ )
  for each  $t_2 \in bvhNode.triangles$ 
    if TRIANGLETRIANGLEINTERSECT( $t, t_2$ )
      return ( true )
  return ( false )
if FINDINTERSECTION( $bvhNode.left, t$ )
  return ( true )
if FINDINTERSECTION( $bvhNode.right, t$ )
  return ( true )
return ( false )

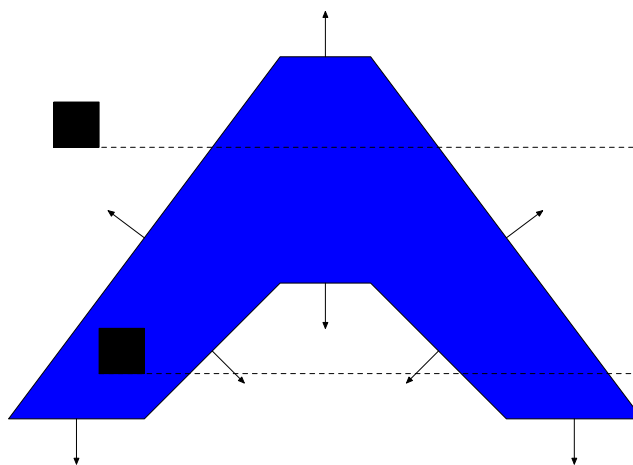
```

BOXTRIANGLEINTERSECTION beskrives i afsnit 6.5 og TRIANGLETRIANGLEINTERSECT i afsnit 6.5

Er robotten indeni scenen?

Da trekantsnet er hule, er vi interesseret i at trekantsnet der ligger indeni hinanden giver anledning til en skæring selvom deres "overflader" ikke skærer hinanden. Denne situation kan opstå når et objekt er placeret fuldstændigt inden i et andet.

I figur 6.5 er vores løsning illustreret. Den udnytter at alle trekanters normaler peger udad.



Figur 6.12: Den samme robot placeret hhv. indeni og udenfor en forhindring. Hvis robotten er udenfor forhindringen, vil den første trekant, en stråle fra roboten møder, have retning mod roboten. Hvis robotten er indeni forhindringen, er det modsatte tilfældet.

For at kunne løse opgaven effektivt, har vi brugt en teknik fra *ray tracing* hvor man er interesseret i at finde den første trekant der bliver ramt af en stråle der bliver skudt ind i scenen for at kunne give hver pixel den rigtige farve. I [15] er givet en algoritme til at finde den nærmeste skæring mellem en stråle og et BVH-træ. Strålen vi tester op i mod BVH-træet starter i en af robotens trekants hjørnepunkt og har samme retningen $(1, 0, 0)$. Ved at beregne prikproduktet mellem strålen og normalen på den nærmeste trekant, kan det afgøres om robotten ligger inden i forhindringen. Hvis prikproduktet er positivt er det sandt, ellers ikke.

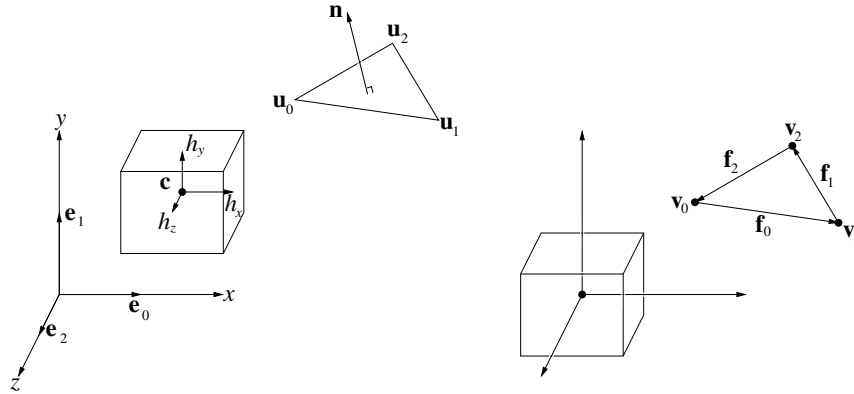
En hurtigt kasse-trekant kollisionstest

Vi har implementeret kasse-trekant-kollisionstesten fra [1]. Hver kasse er repræsenteret ved et centerpunkt, c og en vektor h der definerer størrelsen af kassen, som det kan ses i figur 6.13. Først flyttes origin for det lokale koordinatsystem til c . Herefter laves i alt 13 test som alle forsøger at finde en akse hvorpå trekanten og kassen ikke overlapper. Hvis en sådan akse findes hvis og kun hvis kassen og trekanten ikke overlapper. Vi vil ikke gennemgå de 13 test, men de kan læses i [1][s. 2].

En hurtigt trekant-trekant kollisionstest

Vi har implementeret trekant-trekant-kollisionstesten som Thomas Möller introducerer i [13]. I det dette afsnit vil vi beskrive den grundlæggende idé i algoritmen.

Lad os kalde de to trekanten for T_1 og T_2 og planerne hvori de ligger for π_1 og π_2 .



Figur 6.13: Figuren stammer fra [1] og viser repræsentationen af BVH-kasserne.

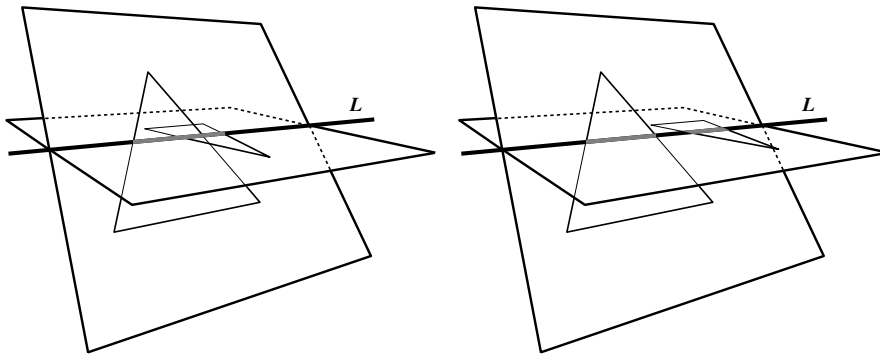
Først udføres to symmetriske test der beregner om T_1 har hjørner på begge sider af π_2 . Hvis dette ikke er tilfældet, kan de to trekanter ikke skære hinanden. Tilsvarende udføres testen for T_2 og π_1 .

Lad L være linjen der opstår i skæringen mellem de to planer. Hvis de to første test er positive, gælder det at:

$$L \cap T_1 \neq \emptyset \wedge L \cap T_2 \neq \emptyset$$

Hvis de to trekanter deler punkter på L , skærer de hinanden. Der er med andre ord en skæring hvis følgende gælder:

$$L \cap T_1 \cap T_2 \neq \emptyset$$



Figur 6.14: Figuren stammer fra [13] og illustrerer hvordan at en test mellem de to intervaller som trekanterne og linjen L danner, kan afgøre om to trekanter skærer hinanden.

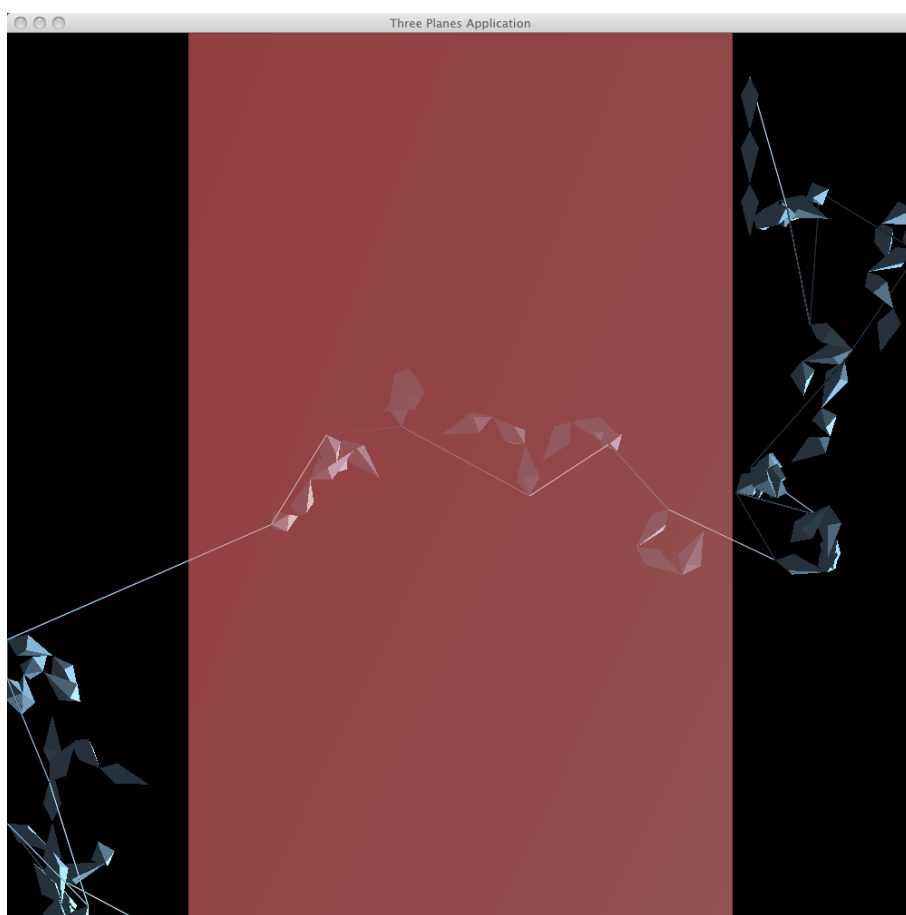
7 Status for implementationen

Implementationen kan udføre algoritmerne EST, RRT, PRM og SRT på scener der er opbygget af trekantsnet og robottor med vilkårligt mange led. Den største udfordring for at løse meget store problemer er at gøre kollisionstesteren mere effektiv.

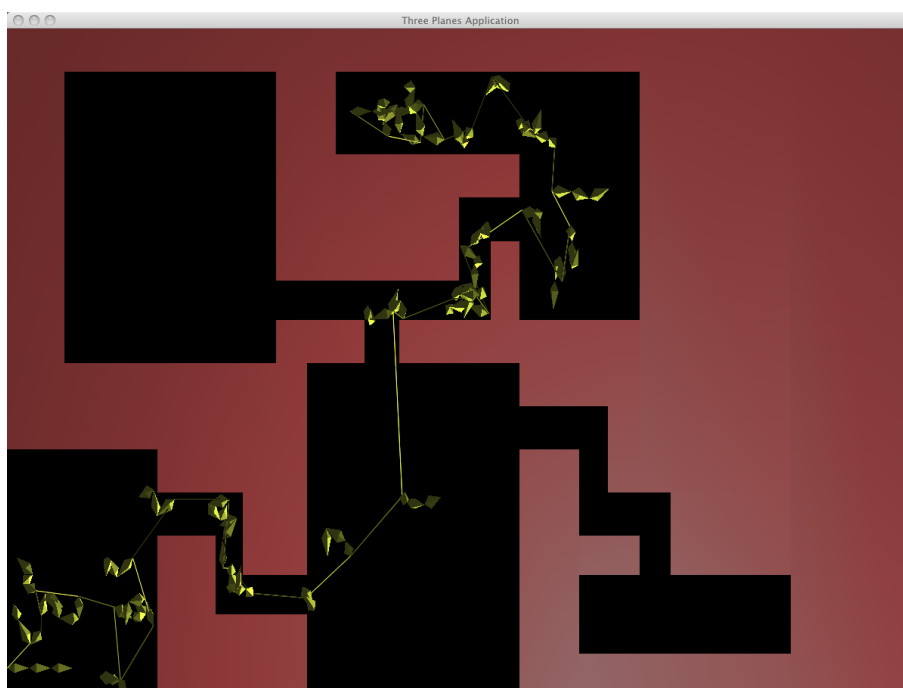
Programmeringssprog Implementationen er lavet i Java SE 1.5 og udviklet ved hjælp af Eclipse 3.3. For at kunne vise OpenGL-visualiseringer af planlægningsalgoritmernes resultater, har vi brugt JOGL biblioteket.

System Alle vores test er kørt på computere med to 3 GHz Intel Xeon processorer og 2 GB RAM. Operativsystemet har været Fedora 8.

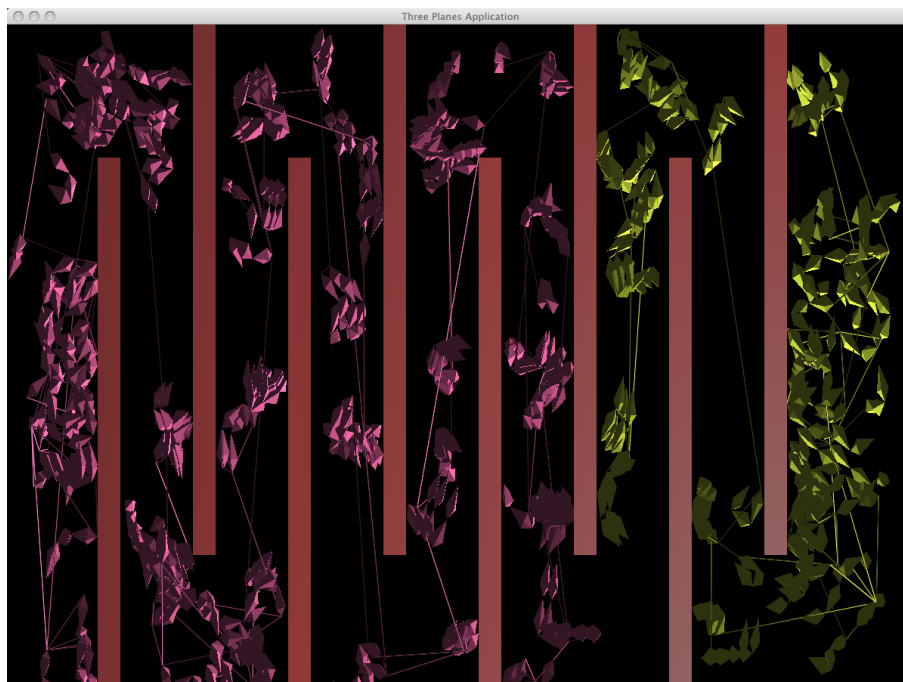
Visualisering Igennem udviklingen af planlægningsalgoritmerne, har vi brugt både 2D og 3D renderinger, til at visualisere konfigurationer, roadmaps, scener og fundne stier for at verificere at implementationerne havde den forventede opførsel. Eksempler på hvordan vi har brugt visualiseringen kan ses i figur 7.1-7.4.



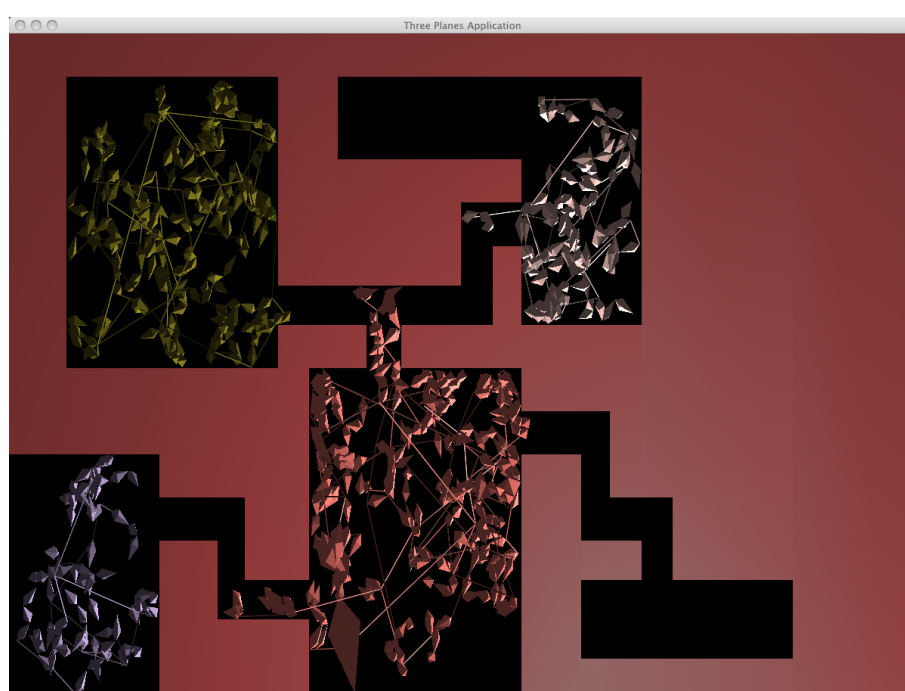
Figur 7.1: EST har i dette tilfælde fundet en sti igennem tunnel-2. Da væggene er halvgennemsigtige, kan man se stien der går igennem tunnelen.



Figur 7.2: EST har i dette tilfælde fundet en sti igennem bunkeren.



Figur 7.3: Labyrinten set fra oven efter en kørsel af PRM. Som det kan ses var det lige ved at lykkedes at forbinde de komponenter.



Figur 7.4: Bunkeren set fra oven efter en kørsel af SRT.

Del II

Test og sammenligning

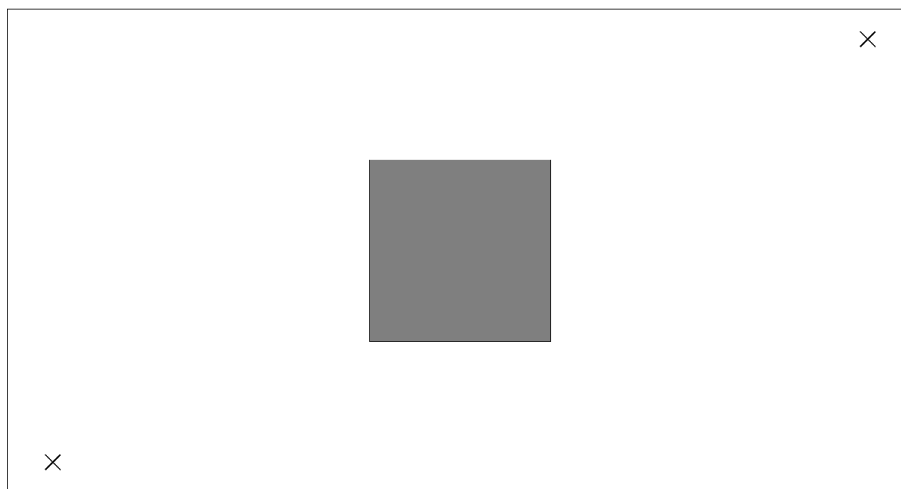
8 Test

I dette kapitel vil vi beskrive vores empiriske studier af vores implementationer af planlægningsalgoritmerne, som vi har præsenteret i del I. I disse studier har vi kigget på hver enkelt algoritmes resultater i forbindelse med en række scener og ligeledes sammenlignet algoritmerne med hinanden. I første afsnit beskriver vi de scener vi har konstrueret til vores forsøg og forklarer intentionen bag dem. I andet afsnit kigger vi nærmere på de parametre, der kan skrues på, i forbindelse med hver enkelt algoritme, og hvorledes ændringer af disse påvirker resultaterne. Sidst sammenligner vi, i afsnit tre, algoritmerne med hinanden på en række punkter.

8.1 Testscener

For at kunne finde gode parametre til de enkelte algoritmer og for at kunne sammenligne algoritmerne, har vi lavet en række testscener. Hver scenes formål vil blive beskrevet i de følgende afsnit. Alle scenerne er tredimensionelle, men er afbilledet som todimensionelle projektioner på xy-planet.

Enkelt kasse



Figur 8.1: Simple kasse

Den første scene, indeholder en enkelt forhindring - En kube placeret midt i scenen. Start- og slutkonfigurationerne ligger på hver sin side af kuben, placeret på kubens ene diagonal. Formålet med denne scene er at teste algoritmerne på et meget simpelt problem, som dog ikke kan løses direkte af en lige-linje-planlægger. I forsøg med denne scene kan kubens størrelse desuden varieres for at undersøge hvordan forholdet mellem størrelsen af C_f og C påvirker algoritmerne.

Tunnel

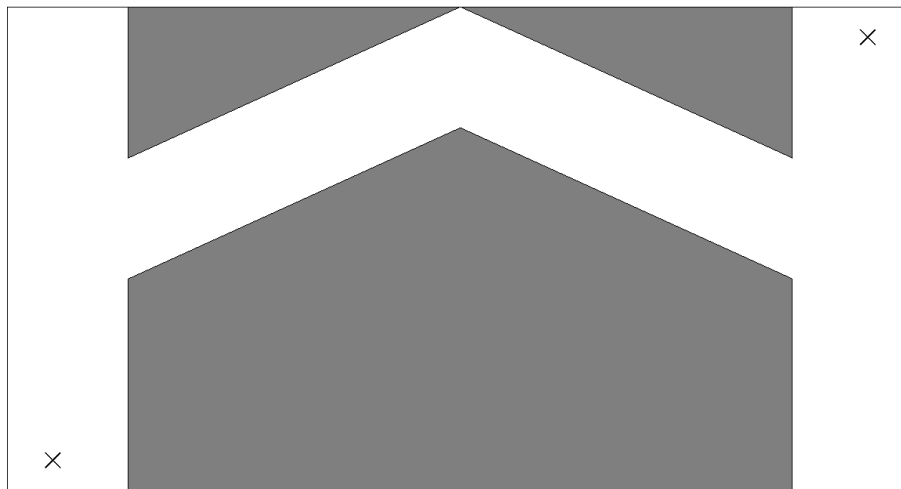


Figur 8.2: Tunnel

Den typiske akilleshæl for planlægningsalgoritmer er at finde vej gennem smalle passager i C_f . Denne scene er konstrueret for at teste hvordan algoritmerne klarer netop sådanne smalle passager. Midt i scenen er der placeret en tyk mur, og igennem denne går en smal tunnel, der forbinder de to frie rum på hver side af muren. I forsøgene varieres bredden af tunnelen. Tunnellenbredden gælder for både y- og z-dimensionen, og tunnellen er ligeledes centreret i de to dimensioner. Start- og slutkonfiguration ligger i modsatte hjørner af scenen.

Tunnel med hjørne

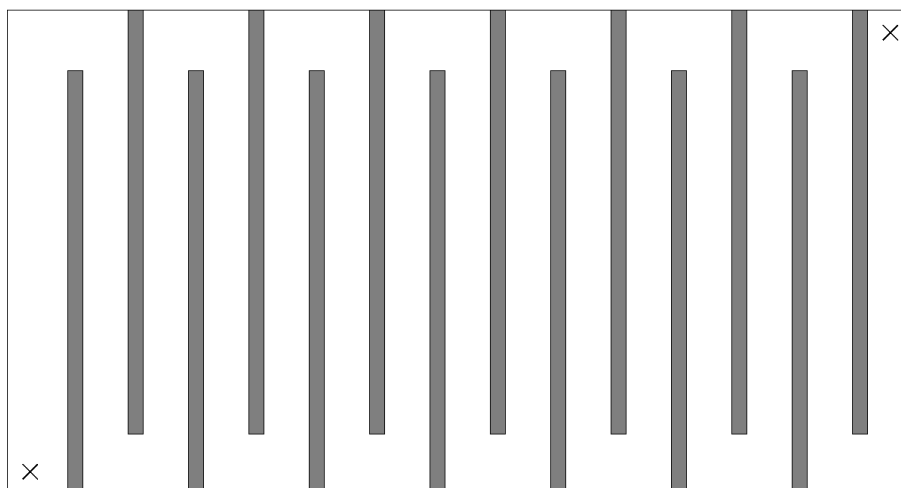
Denne scene er en variation af den forrige. Der er igen en smal tunnel i en mur, der forbinder to frie rum. Tunnellen går ikke lige igennem muren. Den går skråt ind, og laver et knæk inde i muren, inden den når den anden side af muren. Således kan en lige-linje-planlægger ikke "se" gennem tunnelen. Vi har taget scenen med, for at undersøge hvordan det påvirker algoritmernes evne til at løse opgaven, at der ikke kan ses direkte igennem tunnelen. Bredden af tunnelen kan igen varieres, og start- og slutkonfigurationerne ligger som i tunnelscenen.



Figur 8.3: Tunnel med knæk

Labyrint

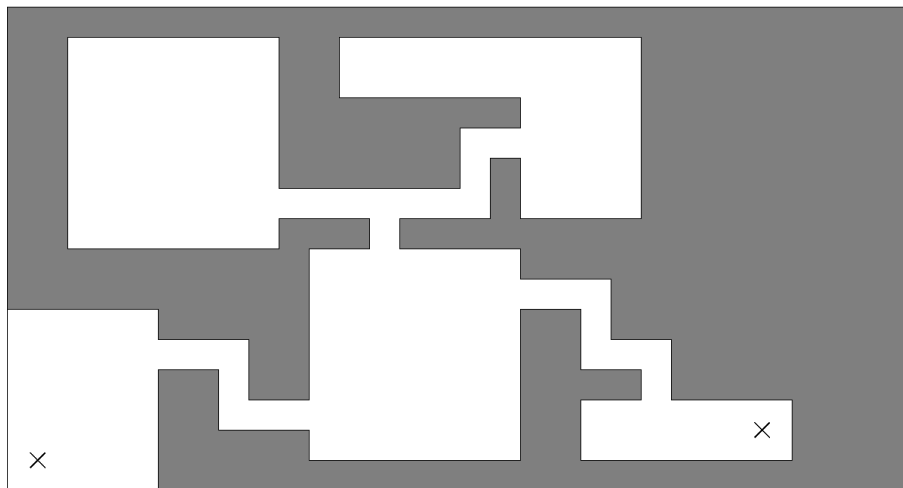
Denne scene er konstrueret som en flad og simpel labyrint. Korridorerne i labyrinten er adskilt af tynde mure, hvilket giver anledning til mange konfigurationer, der ligger meget tæt på hinanden, men som ikke kan se hinanden. Det kan både skyldes at naboerne er adskilt af muren, men det kan ligeledes ske i det tilfælde at to konfigurationer ligger i samme korridor, men så at si-ge vender omvendt i forhold til hinanden og ikke har mulighed for at vende rundt. Se figur 8.4. Vi har konstrueret scenen, for at undersøge hvordan algoritmerne håndterer disse mange naboer, som ikke kan ses.



Figur 8.4: Labyrint

Bunker

Denne scene en række rum i C_f , der er forbundet med tunneler. Med scenen ønsker vi at undersøge hvordan algoritmerne påvirkes af de ekstra rum, som ikke er relevante i forhold til at løse forespørgslen. Samtidig giver scenen mulighed for at se algoritmernes effektivitet på en relativt kompleks probleminstans.



Figur 8.5: Bunker

8.2 Parametre

Planlægningsalgoritmerne har alle en række parametre, som har betydning for hvordan dele af algoritmerne opfører sig. Hvilke værdier disse parametre har kan således påvirke algoritmernes evne til at løse et givent problem i god eller dårlig retning. I dette afsnit vil vi undersøge hver algoritme og dens parametre. Vi vil via tests undersøge hvilke værdier der er optimale for hver af de scener vi beskrev i forrige afsnit. Vi vil ligeledes undersøge om vi, for en given algoritme, kan finde generelt "gode" værdier, eller om algoritmen kræver tilpasning til givne problemer, for at fungere godt. Herunder vil vi også prøve at vurdere om algoritmer, der ikke har generelt gode parameterværdier, er relativt svære eller nemme at justere til en god indstilling.

Enkelt-forespørgselsalgoritmer

Vi starter med at kigge på de to enkelt-forespørgselsalgoritmer, EST og RRT. Vi kigger hver enkelt parameter i algoritmerne uafhængigt af andre parametre. Dvs. med udgangspunkt i et standard parametersæt, tager vi fat i parametrene én ad gangen, og tester et interval af værdier, mens resten af sættes parametre holdes fast i deres standardværdier.

For hver værdi i intervallet afvikles algoritmen fem gange på de fem scener, vi har beskrevet. Hver udførsel begrænses til en fast mængde tid. Forespørgselskonfigurationerne er i alle tests, de konfigurationer vi har angivet

ved scenernes beskrivelser. Vi gemmer den samlede succesrate for en given scene og et givent parametersæt, og gemmer ligeledes udførselstiderne forbundet med de succesfulde udførsler. Det giver alt i alt en tabel for hver parameter, hvor vi kan aflæse udførselstider/succesrater for en given værdi af parameteren for hver enkelt scene.

Hvorvidt et parametersæt for en af disse algoritmer er bedre end et andet afgør vi ved at kigge på succesraterne og udførselstiderne.

EST

EST-algoritmen har disse parametre vi kan justere:

1. Antallet af iterationer, der udføres - eller mængden af tid, der afsættes til at løse problemet.
2. Antallet af samples, der findes i nabolaget ved udvidelse af et træ, K . (omkring 10)
3. Radius på et nabolag, d .
4. Afstandsgrænsen for forbindelsesforsøg.

I [8] nævnes ingen specifik værdi for d , men blot at jo mere snævert et rum er, jo mindre bør d være.

Vi har testet parametrene ved at afprøve værdier i et interval omkring de værdier man har anvendt i artiklen. Hver udførsel har haft et tidsloft på fem minutter. Har algoritmen ikke løst problemet inden loftet er nået, noteres udførslen som en ikke-succesfuld.

Antal samples i nabolag

I figur A.2 ses en tabel med udførselstiderne for forskellige værdier for parameteren K , og i figur A.1 ses de tilhørende succesrater. Artiklens værdi var 10. Vi har derfor testet jævnt i intervallet [5, 25]. Generelt synes tendensen at være at mindre K giver hurtigere udførselstider. For alle scener, på nær Bunker, er den hurtigste snittid opnået med enten $K = 5$ eller $K = 10$. For Tunnel-2 var 10 kun marginalt hurtigere end 5. For Bunker, ser vi ikke nogen klar tendens i den ene eller anden retning. Alt i alt må 5 altså være en generelt god værdi for K .

Radius på et nabolag

I forhold til nabolagsradiusen nævnte artiklen, som sagt ingen specifik værdi, men blot en tommelfingerregel om at man bør anvende små d i snævre konfigurationsrum. Det giver da også god mening at man bør tage mindre skridt i forbindelse med træernes udvidelser, når der ikke store rum at bevæge sig i. Vi forventer således at store d vil give gode udførselstider på de simpleste af vores baner, hvor der er relativt god plads til at foretage store skridt, som kan bringe robotten i mål i løbet af relativt få skridt, mens små skridt vil kræve flere iterationer i algoritmen. Tilsvarende forventer vi at de store d vil gøre det svært for algoritmen at løse problemerne i de mere komplekse og snævre scener.

I forbindelse med vores indledende visuelle tests af algoritmen har vi fundet en værdi på 100, der syntes at gøre algoritmen effektiv. Vi har taget udgangspunkt i denne, og samplet jævnt i intervallet [50, 500]. Resultaterne ses i tabellerne i figur A.3 og A.4.

For de relativt lette scener, Box, Tunnel og Tunnel-2, ser vi den forventede tendens: Store d giver anledning til hurtig løsning af problemet. Dog ser vi at succeraten falder et hak på Tunnel-2 scenen ved værdier større end 350. Yderligere ser vi at 500 tilsyneladende er for stort i Box-scenen hvor udførselstiden er steget en anelse i forhold til $d = 450$. Man kunne forestille sig at det skyldes at 450 giver lige tilpas store skridt, mens 500 blot giver anledning til unødigt lange skridt, som LP'en bruger lidt ekstra tid på at checke.

I forhold til banerne, Bunker og Labyrinth, ser vi ligeledes den forventede effekt: Algoritmen kan ikke løse problemet når d bliver for stort, men indtil en grænse er store d stadig hurtigere end små. Mere uventet ser vi at den mindste værdi, 50, giver problemer på de to scener, hvor succeraten er 0%.

Alt i alt er det altså en parameter, der kan optimeres markant i forhold til givne scener. Men i forbindelse med vores scener kunne man alligevel med en værdi på 150 eller 200, få en overordnet god algoritme, der trods alt løser de simple scener indenfor kort tid, hvis man forestiller sig en bruger, der sidder og venter på svar.

Afstandsgrænsen for forbindelsesforsøg

Afstandsgrænsen for forbindelsesforsøg afgør hvornår algoritmen forsøger at forbinde nye knuder i det ene træ, med knuder i det andet træ. Vi forventer store værdier for denne parameter vil give en hurtig udførselstid på lette baner, som f.eks. banerne Box og Tunnel. I Box er der store åbne rum hvor robotten kan bevæge sig i lange lige linjer, så træerne vil hurtigt kunne forbindes her, når afstandsgrænsen tillader det. I Tunnel kan algoritmen, med de rette samples, forbinde træerne med en enkelt lige linje hele vejen gennem tunnellen. På den anden side forventer vi at store afstandsgrænser kan give problemer i scener med snævre rum, da algoritmen vil spille tid på at afprøve forbindelse inden træerne er tæt nok på hinanden. Man kunne f.eks. forestille sig en afstandsgrænse, der er så stor at træerne hele tiden kan se hinanden, selvom de er på hver sin side af murene i Tunnel og Tunnel-2 banerne.

I artiklen bruges som nævnt 0,5 i forhold til scener reduceret til enhedskuber. Hvordan denne afstand relaterer sig til vores afstandsfunktioner er svært at sige. Vi har indledningsvis forsøgt os med en afstand på 200, hvilket gav fornuftige resultater. Vi har testet omkringliggende værdier i intervallet [100, 1000]. Resultaterne ses i figur A.5 og A.6.

Som forventet ser vi at store værdier gør algoritmen mindre effektiv på Labyrinth scenen, dog ser vi ikke et tilsvarende resultat på Bunker-scenen. En af de mellemste værdier ser ud til at kunne bruges generelt.

Opsummering

I figur 8.6 har vi samlet de bedste parameterværdier for hver scene og et parametersæt, der måske vil fungere godt generelt.

	Bunker	Labyrinth	Tunnel-2	Tunnel	Generelt
K	25	5	10	5	5
d	150	200	300	350	150
Afstandsgrænse	500	300	400	800	600

Figur 8.6: Parametersæt for EST

RRT

RRT-algorithmens parametre:

1. Antallet af iterationer, der udføres - eller mængden af tid, der afsættes til at løse problemet.
2. Hvor store skridt tages der ved udvidelse.
3. Laver algoritmen "grådige" udvidelser eller ej.

De oprindelige artikler nævner ingen værdier for parametrene, så vi har taget udgangspunkt i de værdier, der syntes at virke fornuftigt i forbindelse med implementationen. Således har vi taget udgangspunkt i en skridtstørrelse på 100, og testet jævnt i intervallet [25, 200]. Alle testværdierne er afviklet med algoritmen i både grådig og ikke-grådig tilstand. Hvert parametersæt er blevet testet fem gange på hver scene, og til hver kørsel har vi afsat fem minutter. Finder algoritmen ikke en løsning indenfor det tidsrum, noterer vi et mislykket forsøg. Når det lykkes, noterer vi et succesfuldt forsøg, og gemmer yderligere udførelstiden. Parametersættene har vi sammenlignet på både succesrate og udførelstid.

I figurene B.1 og B.2 vises tabeller med hhv. succesrater og udførelstider for algoritmen i ikke-grådig tilstand. Tomme felter betyder at alle forsøg med det givne parametersæt har været mislykkede. Generelt ser man i tabellen en tendens til at større skridt giver lavere udførelstid. Hvilket man måtte forvente, da de store skridt giver anledning til at kunne udforsket scenerne med færre skridt, og dermed løse problemet med færre iterationer. Yderligere ser vi at algoritmen ikke har formået at løse problemet på Labyrinth med de to mindste skridtstørrelser. Man kan forestille sig at algoritmen med de små skridt ganske enkelt "går" for langsomt til at nå igennem den relativ lange bane.

Når algoritmen afvikles i grådig tilstand forventer vi at hastigheden for små skridt forbedres i en sådan grad at fordelen ved store skridt bliver langt mindre eller helt forsvinder. Grådigheden fører netop til at algoritmen kan fortsætte en udvidelse langt, ligesom et stort skridt. Der er naturligvis et overhead ved at tage flere små skridt langs stien i forhold til blot at tage ét stort skridt helt til stiens ende. Tilgængelig undgår de små skridt problemerne ved de store skridt: At der ikke er plads til dem. Vi forventer altså, at den grådige tilstand kombineret med små skridt vil give den gode hastighed ved store skridt og samtidig de små skridts evne til at manøvrere i snævre rum.

I figurene B.3 og B.4 ses testenets resultateter for algoritmen i grådig tilstand.

Som vi forventede, viser resultaterne markant bedre udførselstider ved små skridt. Og samtidig er succesraten for de snævre scener, Labyrinth og Bunker, blevet markant bedre. Det er altså en klar fordel at benytte algoritmen i grådig tilstand. Ser vi på udførselstiderne, ligger de bedste for alle scener på eller lige opad parameterværdien 125. Denne må derfor være en generelt god værdi.

Opsummering

I figur 8.7 har vi samlet de bedste parameterværdier for hver scene og et generelt parametersæt.

	Bunker	Labyrinth	Tunnel-2	Tunnel	Generelt
Grådig	ja	ja	ja	ja	ja
Skridt	100	125	150	75	125

Figur 8.7: Parametersæt for RRT

Multi-forespørgselsalgoritmer

De to multi-forespørgselsalgoritmer, PRM og SRT, vil vi ligesom enkelt-forespørgselsalgoritmerne teste ved at kigge på de enkelte parametre uafhængigt af andre parametre. Vi tester igen hvert parametersæt fem gange for hver scene, og der testes med et loft på konstruktionen af algoritmernes roadmaps. En enkelt test består dog ikke kun af en forespørgsel, men af 100 forespørgsler. Den første forespørgsel er forespørgslen beskrevet ved scenerne. Resten er består af tilfældige start- og slutkonfigurationer i C_f genereret af den uniforme sampler. For hver scene er de 100 forespørgsler blevet genereret og gemt, således at alle test i dette afsnit er udført med samme forespørgsler.

For at sammenligne parametersættene kigger vi på succesrater og udførselstider for forespørgslerne.

PRM

Ved PRM-algoritmen har vi følgende parametre at skrue på (Værdierne fra [9] er angivet i parentes):

1. Antallet af samples, der laves i konstruktionsskridet - Eller mængden af tid, der afsættes til at sample i, T_c .
2. Mængden af tid afsat til udvidelsesskridtet, T_e . ($2 * T_c$)
3. Antallet af nærmeste naboer en sample forsøges forbundet til. (30)
4. Afstandsgrænsen for forbindelsesforsøg. (0,5 med scener reduceret til enhedskuber)
5. Mængden af tid afsat til en enkelt *random bounce walk* i udvidelsesfasen, $T_{RB-expand}$. (0,01 sekund)

6. Mængden af tid afsat til en enkelt *random bounce walk* ved forespørgsler, $T_{RB-query}$. (0,05 sekund)
7. Antallet af *random bounce walks*, der må udføres ved en forespørgsel. (45)

Vi lader vores test tage udgangspunkt i værdierne fra [9]. Værdierne for forespørgsler har vi ikke ændret på.

Tid afsat til sampling/udvidelse

Algoritmens tid er, som nævnt, holdt fast i vores tests. Den første parameter vi kigger på er derfor hvordan den afsatte tid fordeles mellem konstruktionsfasen og udvidelsesfasen. Vi forestiller os at en overvægt af tid i udvidelsesfasen vil gavne algoritmen i scener som Tunnel og Tunnel2. I disse scener har vi to store områder i C_f , som algoritmen bør have nemt ved at lave samples i og det bør også være relativt let at forbinde samples, som ligger i det samme af de to områder. Derimod forventer vi at konstruktionsfasen vil have svært ved at forbinde de to områder, hvorfor udvidelsesfasen bør gives tid nok til at "ramme" tunnellen med sine random walks.

En overvægt på konstruktionstiden forventer vi kan være positivt i en scene som Labyrinth, hvor man kan forestille sig at samplingen kan ramme uniformt hele vejen igennem gangen fra start til slut. Giver der nok tid til sampling må man så kunne forvente at de vil ligge tæt nok til at de også kan forbindes i konstruktionsfasen. Sagt på en anden måde, kan man sige at hele banen er lige svær, så der er ikke særlige områder, som udvidelsesfasen bør gives tid til at løse.

I artiklen gives konstruktionsfasen 1/3 af tiden og udvidelsesfasen 2/3. Vi har testet algoritmen med en tidsandel for konstruktionsfasen fra 10% til 90% med et interval på 10%-point. Resultaterne af vores test ses i figurene C.1 og C.2.

Resultaterne viser en klar præference for at lægge mest tid i konstruktionsfasen i forhold til scenerne Tunnel, Tunnel-2, Labyrinth og Bunker. Især Labyrinth får markant bedre succesrate med høj vægt på konstruktionsfasen. I forhold til Box-scenen ser vi en næsten konstant succesrate, dog med et lille dyk ved 10%.

En tidsfordeling med 80% til konstruktionsfasen og 20% til udvidelse, ligner ud fra resultaterne som en god generel indstilling.

Antal nærmeste naboer

Den næste parameter vi undersøger, er antallet af nærmeste naboer, som konstruktionsfasen forsøger at forbinde nye samples til. I forhold til Labyrinth-scenen har vi en forventning om at høje værdier, kan være bedre end lave, evt. kun til en hvis grænse. Scenens smalle mure giver netop anledning til at de nærmeste naboer ikke kan nås, hvorfor en større mængde naboer kan øge sandsynligheden for at få nogle af de naboer med, som ligger i samme korridor, som den givne konfiguration. Til de andre baner er det mindre oplagt hvilken betydning parameterværdien kan have.

Vi har testet værdier i intervallet [10, 100] med skridt på 10. Resultaterne ses i figurene C.3 og C.4.

På succesraterne ser vi den forventede præference for høje værdier på Labyrinth-scenen, dog med en størst succes med værdien 90. På de andre scener svinger raterne også, men der synes ikke at være en sammenhæng med parameterværdien. Man kunne måske have forestillet sig at de høje værdier, ville gøre algoritmen mindre effektiv på disse baner, men parametren for maksimal afstand for forbindelsesforsøg, sørger for at ingen af naboerne ligger håbløst langt væk. Det kan i praksis betyde at algoritmen reelt prøver at forbinde til langt færre naboer end den valgte værdi for parameteren "Antal nærmeste naboer".

Alt i alt har parameteren altså ikke betydning på de fleste af vores scener, men der findes scener, som f.eks. Labyrinth, der foretrækker relativt høje værdier. Spørgsmålet er så om der findes scener, hvor en relativ lille værdi foretrækkes. Vi kan ikke umiddelbart forestille os en sådan bane. Vi vælger derfor 90 som en generelt god værdi for parameteren.

Afstandsgrænsen for forbindelsesforsøg

Denne parameter afgør i forbindelse med konstruktionsfasen, om en given nabo er tæt nok på en ny sample til at en forbindelse skal forsøges lavet. Vi forventer at snævre scener vil foretrække relativt lave værdier i forhold til simple og åbne scener. I de snævre scener vil fjerne naboer oftere ligge på den anden side af en forhindring, og hvor et forbindelsesforsøg vil være forgæves. På de åbne scener vil der være bedre muligheder for sådanne lange forbindelsesforsøg. Om det vil gavne at kunne lave lange forbindelser i de åbne scener er dog ikke sikkert. Nærmere naboer vil alligevel blive forsøgt først, så man kan forestille sig at roadmappet bliver det samme uanset om lange forbindelser tillades eller ej.

Vi har testet værdier i intervallet [100, 1000] med skridt på 100. I figurene C.5 og C.6 ses resultaterne af vore tests.

Resultaterne viser at de mindste værdier, 100 og 200, generelt set giver relativt dårlige succesrater, og specielt at de giver meget lave rater på de sværreste baner, Bunker og Labyrinth. Vi ser heller ikke en præference for de laveste værdier efter 200, så vores forventning om at lave værdier ville være gode på disse scener viser sig at være forkert. For alle scenerne er raterne og de succesfulde forespørgsels udførselstider mere eller mindre stabile for værdier fra 300 til 1000. En værdi i det interval ser altså ud til at være en generelt god værdi.

Tidstærskel for et enkelt random walk i udvidelsesfasen

Hvor lang tid udvidelsesfasen i alt har til at udføre *random walks*, blev afgjort af den første parameter vi kiggede på. Den næste parameter vi undersøger afgør hvor længe et enkelt *random walk* fortsætter. Jo længere tid der gives, jo længere kan det enkelte *random walk* naturligvis nå, men samtidig kan der selvfølgelig udføres et mindre antal *walks* i hele udvidelsesfasen. Parametren justerer altså på et kompromis mellem længde og antal af *walks*. Hvordan vægtingen påvirker algoritmen er umiddelbart svært at sige. På den ene side vil lange *walks* have større sandsynlighed, end korte, for at kunne nå frem til et andet sammenhængskomponent. På den anden side vil et større antal *walks*

have større sandsynlighed for at “ramme” smalle korridorer, som typisk er udfordringen for udvidelsesfasen.

Vi har testet i intervallet $[0,002; 0,020]$ med skridt af 0,002. Resultaterne er vist i figurene C.7 og C.8.

Der synes ikke at fremgå en klar præference af resultaterne. Man kan måske ane en lille overvægt af høje rater ved værdier større end eller lig 0,01.

Opsummering

I figur 8.8 har vi samlet de bedste parameterverdier for hver scene og et generelt parametersæt.

	Bunker	Labyrinth	Tunnel-2	Tunnel	Generelt
T_c	80%	80%	80%	90%	80%
Nærmeste naboer	100	90	60	30	90
Afstandsgrænse	700	400	700	400	700
$T_{RB-expand}$	16ms	16ms	18ms	2ms	16ms

Figur 8.8: Parametersæt for PRM

SRT

SRT-algoritmens parametre:

1. Antallet af milepæle, der laves - eller mængden af tid, der afsættes til at bygge et road-map.
2. Fordelingen af tid mellem sampling- og forbindelsesfase, hvis algoritmen afvikles med tidsgrænse.
3. Antallet af nærmeste naboer en sample forsøges forbundet til.
4. Antallet af tilfældige naboer en sample forsøges forbundet til.
5. Antallet af konfigurationer, der afprøves når træer forsøges forbundet.

I [4] nævnes der ikke værdier for parametrene. Vi har prøvet os frem med visuelle tests for at finde et udgangspunkt for vores tests.

Tidsfordeling mellem samplings- og forbindelsesfase

Da vi tester med en tidsgrænse, er den første parameter vi kigger på, fordelingen af tiden mellem sampling- og forbindelsesfasen i algoritmen. Forbindelsesfasen itererer over de samlede milesten. Mængden af arbejde, der kan laves i den fase, er altså begrænset af hvor mange milesten, samplingsfasen har nået at generere. Derfor forventer vi at meget lave vægtninger af samplingsfasen, vil generere for lidt arbejde til forbindelsesfasen og dermed give

en ikke-optimal algoritme. Samtidig er det selvfølgelig nødvendigt at forbindelsesfasen tildes nok tid til at nå omkring de fleste af milestenene. Hvordan fordelingen bliver optimal afhænger således af hvor mange tidsenheder det tager at sample en tilsvarende mængde arbejde til forbindelsesfasen. Dette forhold afhænger naturligvis af andre parametre, så som hvor mange samples algoritmen konfigureres til at generere i hver milestens RRT-træ.

Vi har testet tildelingen af tid til sampling fra 10% til 90% med skridt på 10%-point. Resultaterne ses i figur D.1 og D.2.

Resultaterne viser en svag præference for lave værdier på banerne Box, Tunnel og Labyrinth. På Tunnel-2 er det ligeledes de lave der giver de bedste rater, og forskellen er en anelse mere udtalt på denne bane. For Bunker scenen er det i stedet en 60% tildeling til sampling, der giver det bedste resultat. Det kan givetvis skyldes at denne scene har mere end blot ét eller to store åbne rum i C_f , i forhold til de andre scener. I Tunnel scenerne vil man givetvis ikke vinde noget ved at sample mere, når der er lavet en sample på hver side af muren - med mindre man er heldig at ramme tunnelen med en sampling. I Bunker scenen er der flere tunneller, eller gange, og man kan forestille sig at det vil være en fordel for algoritmen at få samplet tilstrækkeligt til at der er milesten tæt på alle tunnelindgangene, som forbindelsesfasen kan bruge til at forbinde de åbne rum.

Antal nærmeste naboer

Som i PRM er der en parameter, der angiver hvor mange naboer en given sample skal forsøges forbundet til. Imodsætning til er PRM er der dog ikke specificeret en maksimal afstand for disse forbindelsesforsøg. Vi forventer at værdien derfor skal være lavere end værdien var for PRM. I vores visuelle tests nåede vi tilfredsstillende resultater med en værdi på fem. For PRM var det vigtigt i forbindelse med Labyrinth scenen at kigge på en relativt stor mængde naboer, da de nærmeste ofte vil ligge på den anden side af en af murene i scenen. Med SRT's RRT-træer fremfor enkelt konfigurationer og langt kraftigere lokalplanlægger, er det svært at sige om en høj værdi igen er en fordel.

Vi har testet værdier fra de fem, som vi fandt ved visuelle tests og op til 55, for at checke om de relativt høje værdier fra PRM i mod forventning skulle være bedre. Resultaterne er vist i figurene D.3 og D.4.

For de lave værdier er der ikke de store udsving i succesraterne. Tager vi de større værdier fra PRM med, ser vi at succesraten faktisk bliver en anelse bedre på Tunnel og Tunnel-2 scenerne.

Antal tilfældige knuder

Udover et antal nærmeste naboer, forsøger SRT også at forbinde en given sample til et antal tilfældige naboer. Ifølge [4] skyldes det problemer med afstandsfunktionen i forbindelse med snævre områder. Det uddybes ikke nærmere i artiklen. Vi forventer at værdien skal være lav, da disse tilfældige milesten kan lægge meget langt fra sample-milestenen.

Vi har testet værdier i intervallet [1, 9] med skridt på 2. Resultaterne ses i figur D.5 og D.6.

Som ved de nærmeste naboer, ser parameteren igen ikke ud til at påvirke resultaterne markant.

SRT-RRT skridtstørrelse

Da SRT benytter sig af RRT, har vi parametrene fra RRT at skrue på igen. Vi har valgt ikke at teste grådigheden, da den var klart bedst i forbindelse med RRT-testene. Vi kigger derfor først på skridtstørrelsen for RRT. Vi forventer at små skridt vil være en fordel, da de har større sandsynlighed for succes - Og kombineret med grådighed, kan udvidelsen alligevel nå langt, med minimalt overhead.

Vi har testet værdier i intervallet [25, 200]. Resultaterne er vist i D.7 og D.8.

Resultaterne viser ikke markant afhængighed af parameteren. En værdi på 125, som vi valgte i RRT-testen også, virker som et fornuftigt bud på en generelt god værdi.

SRT-RRT samples

Denne parameter angiver hvor mange samples RRT skal lave hver gang SRT sampler en ny milesten. Vi forventer at værdien ikke skal være "alt for høj". Træet skal lave samples nok til at udforske det rum den ligger i, men hvis der genereres milesten flere gange i samme rum, kan det være spildt arbejde hvis milestene hver især bruger meget tid på at udforske det samme rum. I scener med meget snævre rum eller som udelukkende består af korridorer kan man på den anden side forestille sig at der knap så ofte vil blive lavet milesten i samme område, og træernes udforskning derfor ikke er spildt arbejde.

Vi har testet parameteren med værdier i intervallet [20, 100] med skridt på 20. I figurene D.9 og D.10 ses resultaterne.

For Bunker-scenen ser vi at de mindste værdier klarer sig bedst, omend ikke meget bedre end de høje. For de resterende scener er de høje værdier klart bedre, og 100 ligner en generelt god værdi.

SRT-BiRRT (LP) samples

Den sidste parameter for SRT er antallet af samples RRT maksimalt må bruge til at forbinde to træer, når den agerer lokal planlægger for SRT. Det er oplagt at LP'ens evne til at finde vej øges hvis sample-grænsen for den øges. På den anden side vil den så også spille mere tid på de opgaver den alligevel ikke løser. Der er altså et kompromis i forhold hvor meget tid man vil lade LP'en bruge på en opgave.

Vi har testet med værdier i intervallet [10, 100] med skridt på 10. Resultaterne er vist i figurene D.11 og D.12.

For Tunnel, Tunnel-2 og Labyrinth ser den mindste værdi ud til at være bedst. For Bunker giver den mindste værdi en anelse dårligere succesrate, mens parameteren ikke viser nogen effekt ved Box-scenen. Den mindste værdi ligner et godt bud på en generelt god værdi.

Opsummering

I figur 8.9 ses parametersættene vi nåede frem til.

	Bunker	Labyrinth	Tunnel-2	Tunnel	Generelt
$T_{sampling}$	50%	30%	10%	30%	30%
Nærmeste naboer	55	15	55	55	55
Tilfældige naboer	9	1	5	3	3
Skridt	125	200	175	100	125
RRT-samples	80	60	100	100	100
BiRRT-samples	80	20	10	10	10

Figur 8.9: Parametersæt for SRT

8.3 Sammenligning af algoritmerne

Vi har testet algoritmerne med de parametersæt vi nåede frem til ved vores parametertests. For hver scene har vi testet parametersættet med de individuelt bedste værdier, og vi har ligeledes på hver scene testet det parametersæt vi på baggrund af parametertestene forventer kan være generelt godt. Testene er udført på samme måde som parametertestene. Algoritmerne er udført fem gange med hvert parametersæt, på hver scene. For PRM og EST er der udført 100 forespørgsler for hver kørsel. Alle tests er udført med følgende grænser for udførselstid:

Tunnel 15 sekunder.

Tunnel-2 20 sekunder.

Labyrinth 45 sekunder.

Bunker 60 sekunder.

I figur 8.10 ser vi succesraterne for alle fire algoritmer, og i figur 8.11 ses udførselstiderne for EST og RRT.

	Bunker	Labyrinth	Tunnel	Tunnel-2
EST (opt)	60.0%	100.0%	40.0%	80.0%
EST (std)	60.0%	100.0%	80.0%	60.0%
RRT (opt)	60.0%	0.0%	100.0%	100.0%
RRT (std)	0.0%	0.0%	100.0%	100.0%
PRM (opt)	44.0%	100.0%	100.0%	100.0%
PRM (std)	33.8%	100.0%	100.0%	99.8%
SRT (opt)	23.2%	38.2%	89.6%	65.4%
SRT (std)	22.8%	35.2%	97.4%	82.2%

Figur 8.10: Succesrater for EST, RRT, PRM og SRT

		Bunker	Labyrinth	Tunnel	Tunnel-2
	min	31057.0	14907.0	3064.0	4795.0
EST (opt)	avg	40643	17501	5753	7973
	max	56074.0	20043.0	8441.0	15361.0
	min	43394.0	26996.0	5031.0	13320.0
EST (std)	avg	47599	29268	8046	13989
	max	51900.0	31468.0	11286.0	14824.0
	min	44446.0		2310.0	2230.0
RRT (opt)	avg	47803		4482	3268
	max	53217.0		7965.0	4514.0
	min			1640.0	2062.0
RRT (std)	avg			3103	5376
	max			6578.0	15306.0

Figur 8.11: Udførselstider for EST og RRT. (ms)

For enkeltforespørgelsesalgoritmerne ser vi at EST løser opgaven hver gang på Labyrinth, hvor RRT slet ikke formår at løse opgaven indenfor tidsgrænsen. Vi forestiller os at det skyldes RRTs måde at udvide på. Uniformt tilfældige samples over hele C , vil naturligvis, på den aflange bane, meget ofte styre RRT direkte ind i en af murene. ESTs lokale sampling synes at være langt mere meningsfuldt i scenen. Man kan forestille sig at det er samme årsag der ligger bag RRTs dårlige succesrate på Bunker-scenen med sit generelle parametersæt, om end den klarer sig ligeså godt som EST med sit "optimale" sæt.

På Tunnel-scenerne er rollerne byttet om, og RRT har løst problemerne ved hver udførsel.

Kigger vi på PRM og SRT, ser vi en klar vinder i PRM. PRM har i alle scener har klaret sig bedst, og i nogle tilfælde markant bedre end SRT.

Alt i alt må vi konkludere at i forbindelse med enkeltforespørgelsesalgoritmer ser algoritmerne, EST og RRT, ud til at have hver sine styrker. Dog klarer EST sig mere hæderligt på de scener, som RRT er bedst til at løse, end det er tilfældet den anden vej rundt. Med en variabel d parameter implementeret i EST, forestiller vi os, at EST kunne blive endnu bedre.

Til multiforespørgelsesopgaver ser PRM ud til at være et bedre valg end SRT i alle tilfælde. Vi har dog en mistanke om at vores version af SRT ikke yder optimalt. Det kan muligvis skyldes en fejl i vores implementation, som gøres algoritmen mindre effektiv uden dog at give anledning til så store fejl at vi har kunne se dem på tests. Alternativt kan det skyldes at vi ikke bruger de optimale parametre

9 Konklusion

I dette speciale har vi præsenteret en række forskellige planlægningsalgoritmer som kan løse generalized movers problem. Formålet har været været at sammenligne de forskellige algoritmers evner til at løse forskellige problemer. For at kunne lave en sammenligning har vi beskrevet og implementeret følgende algoritmer:

- Expansive-Spaces Trees
- Rapidly-exploring Random Trees
- Probabilistic Roadmaps
- Sampling-based Roadmap of Trees

For at kunne lave effektive implementationer af algoritmerne, har vi skulle finde løsninger på en række ikke trivielle problemer. Disse løsninger er alle beskrevet i specialet, men de mest komplekse og dermed også dem vi har brugt mest tid på at er følgende:

Repræsentation af rotationer, hvortil vi på baggrund af [10] har valgt at bruge quaternioner.

Vægtning af tilfældigt udvalgte elementer, løste vi med en datastrukturen *biased bingo bag* som vi selv har konstrueret.

At finde naboer i konfigurationsrummet, har vi løst effektivt ved at implementere et *kd*-træet fra [16], som er specielt rettet mod denne slags robotplanlægningsproblemer.

At finde kollisioner mellem trekantsnet, løser vi med et BVH-træ, som vi har implementeret således at det har den nødvendige effektivitet.

Alle algoritmerne bruger en række forskellige parametre og en stor del af vores test er udført for at finde gode parametre. Vores test har vist at der stor forskel på hvordan de enkelte algoritmer klarer sig på forskellige scener og at deres parametre også kan afhænge af scenen. For eksempel klarer de to enkelforespørgsels-algoritmer sig bedst på forskellige scener og at vi derfor ikke kan udnævne en som vinder. Med multi-forespørgsels-algoritmerne har vi set at PRM klarer sig bedre end SRT på trods af dennes langt mere avancerede lokale planlægger.

Litteratur

- [1] Tomas Akenine-Möller. Fast 3d triangle-box overlap testing, 2001.
- [2] Jérôme Barraquand, Lydia Kavraki, Jean-Claude Latombe, Rajeev Motwani, Tsai-Yen Li, and Prabhakar Raghavan. A random sampling scheme for path planning. *Int. J. Rob. Res.*, 16(6):759–774, 1997.
- [3] Jérôme Barraquand and Jean-Claude Latombe. Robot motion planning: a distributed representation approach. *Int. J. Rob. Res.*, 10(6):628–649, 1991.
- [4] K. E. Bekris, B. Y. Chen, A. M. Ladd, E. Plaku, and L. E. Kavraki. Multiple query probabilistic roadmap planning using single query planning primitives. pages 656–661, October 2003.
- [5] Howie Choset, Kevin M. Lynch, Seth Hutchinson, George A Kantor, Wolfram Burgard, Lydia E. Kavraki, and Sebastian Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, June 2005. ISBN 0-262-03327-5. Kapitel 7.
- [6] Michael T. Goodrich, Roberto Tamassia, and Michael Goodrich. *Algorithm Design: Foundations, Analysis, and Internet Examples*. Wiley, September 2001.
- [7] K.K. Gupta and Zhenping Guo. Motion planning for many degrees of freedom: sequential search with backtracking. *Robotics and Automation, IEEE Transactions on*, 11(6):897–906, Dec 1995.
- [8] David Hsu, Jean-Claude Latombe, and Rajeev Motwani. Path planning in expansive configuration spaces. *International Journal of Computational Geometry and Applications*, 9(4/5):495–, 1999.
- [9] Lydia Kavraki, Petr Svestka, Jean-Claude Latombe, and Mark Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. Technical Report CS-TR-94-1519, 1994.
- [10] James Kuffner. Effective sampling and distance metrics for 3d rigid body path planning. In *Proc. 2004 IEEE Int'l Conf. on Robotics and Automation (ICRA 2004)*. IEEE, May 2004.
- [11] Steven M. LaValle and James J. Kuffner Jr. Randomized kinodynamic planning, 1999.
- [12] Steven M. LaValle and James J. Kuffner Jr. Randomized kinodynamic planning. *I. J. Robotic Res.*, 20(5):378–400, 2001.

- [13] Tomas Möller. A fast triangle-triangle intersection test. *J. Graph. Tools*, 2(2):25–30, 1997.
- [14] E. Plaku and L. E. Kavraki. Distributed sampling-based roadmap of trees for large-scale motion planning. pages 3879–3884, Barcelona, Spain, April 2005.
- [15] Peter Shirley and R. Keith Morley. Realistic ray tracing. pages 134–138, 2003.
- [16] A. Yershova and S. M. LaValle. Improving motion-planning algorithms by efficient nearest-neighbor searching. *Robotics, IEEE Transactions on [see also Robotics and Automation, IEEE Transactions on]*, 23(1):151–157, Feb. 2007.

A Resultater for EST-parameter tests

	Box	Bunker	Labyrinth	Tunnel-2	Tunnel
5.0	100.0%	100.0%	100.0%	100.0%	100.0%
10.0	100.0%	100.0%	100.0%	100.0%	100.0%
15.0	100.0%	100.0%	100.0%	100.0%	100.0%
20.0	100.0%	100.0%	100.0%	100.0%	100.0%
25.0	100.0%	100.0%	100.0%	100.0%	100.0%

Figur A.1: Test af K-parameter for EST, succesrater.

		Box	Bunker	Labyrinth	Tunnel-2	Tunnel
	min	1675.0	113051.0	109682.0	5795.0	6586.0
5.0	avg	3661.8	126728.2	112063.4	11480.0	8102.8
	max	4570.0	140013.0	118289.0	17228.0	10741.0
	min	4116.0	103410.0	117735.0	8065.0	7604.0
10.0	avg	5165.8	127639.8	122579.2	11109.4	8477.2
	max	6007.0	172863.0	124503.0	14133.0	9375.0
	min	3565.0	96562.0	120362.0	10177.0	7636.0
15.0	avg	5034.4	125527.4	132579.0	12341.8	9153.8
	max	6657.0	155668.0	140828.0	14104.0	10696.0
	min	4571.0	104179.0	128743.0	10246.0	7520.0
20.0	avg	5576.6	129720.6	140509.0	12211.0	11071.0
	max	6912.0	165069.0	151851.0	13954.0	14273.0
	min	3108.0	115588.0	138716.0	10633.0	8203.0
25.0	avg	5101.6	122369.2	142251.2	13032.6	10419.6
	max	6571.0	135240.0	144578.0	18641.0	12789.0

Figur A.2: Test af K-parameter for EST, udførselstider (ms).

	Box	Bunker	Labyrinth	Tunnel-2	Tunnel
50.0	100.0%	0.0%	0.0%	100.0%	100.0%
100.0	100.0%	100.0%	100.0%	100.0%	100.0%
150.0	100.0%	100.0%	100.0%	100.0%	100.0%
200.0	100.0%	60.0%	100.0%	100.0%	100.0%
250.0	100.0%	40.0%	100.0%	100.0%	100.0%
300.0	100.0%	0.0%	80.0%	100.0%	100.0%
350.0	100.0%	0.0%	100.0%	100.0%	100.0%
400.0	100.0%	0.0%	20.0%	80.0%	100.0%
450.0	100.0%	0.0%	40.0%	80.0%	100.0%
500.0	100.0%	0.0%	0.0%	80.0%	100.0%

Figur A.3: Test af d-parameter for EST, succesrater.

		Box	Bunker	Labyrinth	Tunnel-2	Tunnel
	min	14222.0			18909.0	12095.0
50.0	avg	16444.6			30994.2	16510.2
	max	20182.0			60957.0	20717.0
	min	1164.0	65499.0	109210.0	4458.0	3131.0
100.0	avg	2814.8	99455.0	118081.2	8312.8	3639.2
	max	4131.0	121147.0	127640.0	12160.0	4189.0
	min	869.0	40872.0	51198.0	4538.0	2512.0
150.0	avg	1386.2	84702.8	62379.0	6590.0	3019.6
	max	1651.0	124716.0	82496.0	7501.0	3623.0
	min	828.0	55212.0	28212.0	3805.0	1916.0
200.0	avg	1145.8	91408.7	32451.8	4425.6	2638.8
	max	1462.0	116262.0	41515.0	5459.0	3756.0
	min	860.0	112516.0	29221.0	2114.0	1611.0
250.0	avg	1099.8	198219.0	38417.2	3481.0	2753.8
	max	1396.0	283922.0	53854.0	4177.0	4200.0
	min	719.0		25491.0	2551.0	1360.0
300.0	avg	906.4		74473.5	3276.2	2188.2
	max	1223.0		196061.0	5001.0	2747.0
	min	591.0		27804.0	2602.0	1432.0
350.0	avg	748.8		84929.2	4938.6	1894.6
	max	928.0		216580.0	8085.0	2273.0
	min	520.0		18839.0	2562.0	1389.0
400.0	avg	751.0		35125.5	3273.5	2296.4
	max	1050.0		51412.0	4957.0	3469.0
	min	488.0		238416.0	2306.0	1412.0
450.0	avg	646.4		238416.0	2981.25	17852.2
	max	747.0		238416.0	3345.0	61620.0
	min	751.0			2243.0	1238.0
500.0	avg	803.6			4127.25	2004.8
	max	900.0			6375.0	2237.0

Figur A.4: Test af d-parameter for EST, udførselstider (ms).

	Box	Bunker	Labyrinth	Tunnel-2	Tunnel
100.0	100.0%	100.0%	100.0%	100.0%	100.0%
200.0	100.0%	100.0%	100.0%	100.0%	100.0%
300.0	100.0%	100.0%	100.0%	100.0%	100.0%
400.0	100.0%	100.0%	100.0%	100.0%	100.0%
500.0	100.0%	100.0%	100.0%	100.0%	100.0%
600.0	100.0%	100.0%	100.0%	100.0%	100.0%
700.0	100.0%	100.0%	100.0%	100.0%	100.0%
800.0	100.0%	100.0%	100.0%	100.0%	100.0%
900.0	100.0%	100.0%	100.0%	100.0%	100.0%
1000.0	100.0%	100.0%	100.0%	100.0%	100.0%

Figur A.5: Test af afstandsgrensen for forbindelsesforsøg for EST, succesrater.

		Box	Bunker	Labyrinth	Tunnel-2	Tunnel
	min	4225.0	113176.0	114423.0	8840.0	5917.0
100.0	avg	5524.4	139549.8	122345.4	12499.8	8993.4
	max	7649.0	172262.0	129551.0	13791.0	14793.0
	min	3782.0	98753.0	111030.0	8464.0	6759.0
200.0	avg	4503.8	105731.2	117164.6	13004.6	9655.8
	max	5730.0	116815.0	122630.0	15525.0	11638.0
	min	2774.0	94181.0	110873.0	8206.0	4573.0
300.0	avg	3664.8	125802.2	117012.8	9939.6	6873.2
	max	4332.0	139486.0	123175.0	12258.0	9125.0
	min	2680.0	85395.0	113362.0	5783.0	5120.0
400.0	avg	3192.2	113525.8	120525.6	8124.8	6540.8
	max	3837.0	152795.0	128819.0	10216.0	8830.0
	min	1946.0	71692.0	114275.0	6948.0	3935.0
500.0	avg	2793.2	95923.0	120766.6	9957.6	4530.6
	max	3609.0	147260.0	133999.0	11477.0	4985.0
	min	1396.0	86043.0	108164.0	6567.0	2853.0
600.0	avg	2054.6	112681.0	117431.8	8269.6	3963.2
	max	2542.0	157141.0	130974.0	10647.0	5167.0
	min	1552.0	96955.0	100547.0	6805.0	2951.0
700.0	avg	2243.4	116654.2	118606.8	9085.8	3460.6
	max	2643.0	135903.0	132315.0	12042.0	3844.0
	min	1271.0	80844.0	116524.0	9199.0	1860.0
800.0	avg	1862.8	116045.4	129863.4	12152.2	2283.6
	max	2451.0	149501.0	138565.0	16938.0	2569.0
	min	1671.0	75517.0	149036.0	7514.0	2136.0
900.0	avg	2298.4	121686.4	163117.4	12107.0	2584.0
	max	3038.0	187099.0	175676.0	15759.0	3442.0
	min	1802.0	69980.0	176597.0	9235.0	1656.0
1000.0	avg	2235.8	119077.2	187052.6	17202.4	2329.6
	max	3101.0	145836.0	199791.0	25255.0	3041.0

Figur A.6: Test af afstandsgrensen for forbindelsesforsøg for EST, udførselstider (ms).

B Resultater for RRT-parameter tests

	Box	Bunker	Labyrinth	Tunnel-2	Tunnel
25.0	100.0%	20.0%	0.0%	100.0%	100.0%
50.0	100.0%	20.0%	0.0%	100.0%	100.0%
75.0	100.0%	20.0%	100.0%	100.0%	100.0%
100.0	100.0%	20.0%	100.0%	100.0%	100.0%
125.0	100.0%	20.0%	100.0%	100.0%	100.0%
150.0	100.0%	20.0%	100.0%	100.0%	100.0%
175.0	100.0%	20.0%	100.0%	100.0%	100.0%
200.0	100.0%	20.0%	100.0%	100.0%	100.0%

Figur B.1: Succesrater ved test af skridtstørrelse for ikke-grådig RRT.

		Box	Bunker	Labyrinth	Tunnel-2	Tunnel
	min	17575.0	264312.0		9026.0	8770.0
25.0	avg	19582.8	264312.0		11001.6	12688.4
	max	23712.0	264312.0		15979.0	18546.0
	min	4487.0	153183.0		3947.0	2523.0
50.0	avg	5129.6	153183.0		5035.2	4663.0
	max	5910.0	153183.0		6559.0	9369.0
	min	2114.0	179606.0	181306.0	3584.0	2038.0
75.0	avg	2373.0	179606.0	187350.4	4280.0	2855.8
	max	2590.0	179606.0	198817.0	5777.0	3798.0
	min	1789.0	139459.0	125756.0	3277.0	2969.0
100.0	avg	2192.2	139459.0	151182.4	3692.0	3411.0
	max	2557.0	139459.0	181685.0	4525.0	4003.0
	min	1313.0	135156.0	96785.0	2770.0	2179.0
125.0	avg	1832.0	135156.0	119667.8	3253.6	2947.8
	max	2560.0	135156.0	143272.0	3596.0	3360.0
	min	1044.0	288054.0	101387.0	2086.0	1999.0
150.0	avg	1693.2	288054.0	116123.2	2904.2	2668.8
	max	2432.0	288054.0	133208.0	3449.0	3900.0
	min	1262.0	136168.0	102479.0	2496.0	2072.0
175.0	avg	1727.6	136168.0	109831.4	3141.8	2411.4
	max	2243.0	136168.0	117128.0	4246.0	2927.0
	min	983.0	119561.0	93345.0	2934.0	1995.0
200.0	avg	1373.2	119561.0	110503.0	3289.6	2337.8
	max	2129.0	119561.0	124866.0	3776.0	2599.0

Figur B.2: Udførselstider ved test af skridtstørrelse for ikke-grådig RRT (ms).

	Box	Bunker	Labyrinth	Tunnel-2	Tunnel
25.0	100.0%	100.0%	100.0%	100.0%	100.0%
50.0	100.0%	100.0%	100.0%	100.0%	100.0%
75.0	100.0%	100.0%	100.0%	100.0%	100.0%
100.0	100.0%	100.0%	100.0%	100.0%	100.0%
125.0	100.0%	100.0%	100.0%	100.0%	100.0%
150.0	100.0%	80.0%	100.0%	100.0%	100.0%
175.0	100.0%	100.0%	100.0%	100.0%	100.0%
200.0	100.0%	100.0%	100.0%	100.0%	100.0%

Figur B.3: Test af skridtstørrelse for grådig RRT, succesrater.

		Box	Bunker	Labyrinth	Tunnel-2	Tunnel
	min	207.0	94668.0	116247.0	2636.0	1006.0
25.0	avg	399.6	119238.6	125864.4	4022.2	2011.8
	max	572.0	139457.0	135025.0	5534.0	2836.0
	min	228.0	46426.0	94609.0	2534.0	1822.0
50.0	avg	361.6	117912.2	112866.8	4300.0	2831.0
	max	500.0	152935.0	122412.0	6166.0	4635.0
	min	297.0	128842.0	94638.0	1836.0	1295.0
75.0	avg	407.4	159316.8	108136.6	3088.2	1716.2
	max	575.0	213060.0	116223.0	4514.0	2416.0
	min	178.0	71860.0	87113.0	1981.0	1093.0
100.0	avg	381.0	113439.2	111083.6	3846.2	2462.0
	max	613.0	152119.0	130324.0	5798.0	3187.0
	min	179.0	66443.0	83546.0	2382.0	1097.0
125.0	avg	234.4	136103.8	95285.4	2761.6	1796.4
	max	376.0	195643.0	104230.0	3116.0	2421.0
	min	212.0	60092.0	90450.0	1935.0	1411.0
150.0	avg	359.8	132690.0	101474.6	2535.6	2254.4
	max	672.0	266473.0	115370.0	3498.0	3226.0
	min	199.0	110184.0	91766.0	1859.0	1942.0
175.0	avg	238.6	180517.6	103538.8	3251.6	2981.2
	max	316.0	265927.0	117374.0	5380.0	4940.0
	min	185.0	161716.0	82005.0	2023.0	1099.0
200.0	avg	282.4	240858.6	96786.8	2847.0	1970.4
	max	435.0	286660.0	107434.0	3811.0	2417.0

Figur B.4: Test af skridtstørrelse for grådig RRT, udførselstider (ms).

C Resultater for PRM-parameter tests

	Box	Bunker	Labyrinth	Tunnel-2	Tunnel
0.1	97.2%	17.8%	15.6%	43.2%	52.8%
0.2	99.8%	17.2%	23.0%	43.8%	53.0%
0.3	100.0%	18.0%	49.2%	44.2%	71.8%
0.4	100.0%	19.4%	66.6%	56.8%	71.8%
0.5	100.0%	25.2%	78.6%	77.6%	90.6%
0.6	100.0%	22.2%	93.4%	93.8%	90.6%
0.7	100.0%	27.2%	100.0%	99.8%	100.0%
0.8	100.0%	41.4%	100.0%	100.0%	90.6%
0.9	100.0%	28.0%	100.0%	100.0%	100.0%
1.0	100.0%	35.2%	100.0%	100.0%	100.0%

Figur C.1: Test af tidsfordeling i PRM, succesrater.

		Box	Bunker	Labyrinth	Tunnel-2	Tunnel
0.1	min	16.0	71.0	15.0	10.0	10.0
	avg	31	165	98	28	25
	max	311.0	489.0	392.0	80.0	170.0
0.2	min	15.0	36.0	17.0	10.0	10.0
	avg	27	120	88	28	21
	max	95.0	390.0	402.0	135.0	134.0
0.3	min	15.0	45.0	14.0	11.0	12.0
	avg	25	109	84	28	25
	max	88.0	439.0	412.0	187.0	116.0
0.4	min	15.0	26.0	19.0	11.0	9.0
	avg	25	99	80	27	24
	max	100.0	225.0	275.0	105.0	64.0
0.5	min	15.0	30.0	16.0	16.0	14.0
	avg	24	99	81	34	28
	max	86.0	347.0	331.0	127.0	159.0
0.6	min	16.0	33.0	20.0	15.0	12.0
	avg	24	104	79	38	28
	max	93.0	792.0	167.0	318.0	105.0
0.7	min	15.0	20.0	20.0	24.0	17.0
	avg	25	79	66	39	27
	max	96.0	219.0	193.0	133.0	108.0
0.8	min	14.0	23.0	20.0	25.0	14.0
	avg	24	93	65	39	25
	max	84.0	264.0	194.0	218.0	110.0
0.9	min	17.0	32.0	15.0	29.0	18.0
	avg	26	79	60	43	28
	max	144.0	197.0	192.0	152.0	99.0
1.0	min	17.0	19.0	12.0	25.0	23.0
	avg	24	76	58	40	33
	max	93.0	245.0	183.0	200.0	158.0

Figur C.2: Test af tidsfordeling i PRM (forespørgselstider), forespørgselstider (ms).

	Box	Bunker	Labyrinth	Tunnel-2	Tunnel
10.0	99.8%	18.2%	20.8%	55.0%	52.8%
20.0	100.0%	18.6%	30.2%	44.2%	70.0%
30.0	100.0%	18.4%	31.4%	66.6%	81.2%
40.0	99.8%	18.6%	37.8%	49.8%	53.0%
50.0	100.0%	25.0%	33.6%	66.2%	62.4%
60.0	100.0%	18.2%	34.6%	66.4%	53.0%
70.0	100.0%	20.0%	45.8%	60.0%	53.0%
80.0	100.0%	19.2%	46.8%	52.8%	71.8%
90.0	100.0%	19.2%	58.6%	61.4%	81.2%
100.0	100.0%	26.6%	46.0%	44.0%	53.0%

Figur C.3: Test af antal nærmeste naboer i PRM, succesrater.

		Box	Bunker	Labyrinth	Tunnel-2	Tunnel
	min	18.0	55.0	19.0	15.0	12.0
10.0	avg	28	140	87	35	22
	max	94.0	390.0	374.0	164.0	95.0
	min	21.0	46.0	21.0	17.0	13.0
20.0	avg	31	136	100	29	28
	max	152.0	463.0	421.0	194.0	129.0
	min	19.0	46.0	20.0	19.0	15.0
30.0	avg	31	139	93	39	30
	max	144.0	259.0	339.0	199.0	102.0
	min	18.0	47.0	20.0	16.0	11.0
40.0	avg	30	139	93	32	23
	max	93.0	384.0	393.0	164.0	91.0
	min	19.0	54.0	24.0	19.0	12.0
50.0	avg	31	153	103	39	26
	max	80.0	528.0	439.0	186.0	101.0
	min	21.0	44.0	22.0	18.0	14.0
60.0	avg	30	140	109	40	23
	max	89.0	396.0	748.0	196.0	70.0
	min	22.0	47.0	21.0	20.0	14.0
70.0	avg	32	139	103	37	26
	max	165.0	419.0	407.0	181.0	114.0
	min	21.0	56.0	25.0	17.0	16.0
80.0	avg	30	139	107	34	30
	max	89.0	435.0	361.0	133.0	107.0
	min	22.0	50.0	24.0	17.0	15.0
90.0	avg	32	140	105	38	33
	max	102.0	401.0	375.0	436.0	169.0
	min	19.0	39.0	20.0	17.0	14.0
100.0	avg	32	123	93	33	27
	max	96.0	3387.0	547.0	242.0	107.0

Figur C.4: Test af antal nærmeste naboer i PRM (forespørgselstider), forespørgselstider (ms).

	Box	Bunker	Labyrinth	Tunnel-2	Tunnel
100.0	89.4%	11.0%	12.4%	43.8%	51.8%
200.0	100.0%	19.0%	41.8%	54.0%	52.8%
300.0	100.0%	21.4%	63.8%	77.0%	90.6%
400.0	100.0%	22.8%	96.2%	93.8%	100.0%
500.0	99.8%	19.2%	83.0%	65.8%	100.0%
600.0	100.0%	19.8%	96.0%	88.4%	71.8%
700.0	100.0%	26.4%	76.2%	99.2%	90.6%
800.0	100.0%	19.2%	92.6%	77.0%	90.6%
900.0	99.8%	19.0%	84.2%	65.8%	90.6%
1000.0	100.0%	22.2%	92.0%	66.4%	81.2%

Figur C.5: Test af afstandsgrense for forbindelseforsøg i PRM, succesrater.

		Box	Bunker	Labyrinth	Tunnel-2	Tunnel
100.0	min	13.0	42.0	22.0	19.0	15.0
	avg	40	141	85	34	25
	max	228.0	431.0	322.0	310.0	104.0
200.0	min	19.0	34.0	16.0	17.0	11.0
	avg	29	136	105	35	22
	max	93.0	322.0	376.0	393.0	78.0
300.0	min	19.0	51.0	23.0	19.0	16.0
	avg	29	155	111	41	30
	max	85.0	414.0	281.0	215.0	101.0
400.0	min	20.0	39.0	36.0	14.0	19.0
	avg	29	148	111	44	31
	max	146.0	328.0	305.0	152.0	111.0
500.0	min	18.0	59.0	28.0	18.0	21.0
	avg	28	157	110	38	31
	max	97.0	462.0	292.0	184.0	108.0
600.0	min	19.0	68.0	34.0	20.0	13.0
	avg	28	146	110	40	25
	max	94.0	368.0	358.0	179.0	96.0
700.0	min	17.0	48.0	23.0	29.0	16.0
	avg	30	162	108	43	30
	max	140.0	443.0	346.0	195.0	89.0
800.0	min	16.0	56.0	30.0	17.0	15.0
	avg	29	143	114	36	30
	max	99.0	333.0	325.0	167.0	103.0
900.0	min	19.0	52.0	27.0	16.0	13.0
	avg	28	140	112	34	30
	max	140.0	291.0	326.0	173.0	105.0
1000.0	min	17.0	40.0	33.0	16.0	14.0
	avg	29	128	114	36	29
	max	153.0	277.0	344.0	203.0	95.0

Figur C.6: Test af afstandsgrænse for forbindelseforsøg i PRM, forespørgselstider (ms).

	Box	Bunker	Labyrinth	Tunnel-2	Tunnel
0.0020	99.0%	18.6%	48.0%	70.2%	100.0%
0.0040	99.6%	19.6%	71.6%	59.8%	62.2%
0.0060	100.0%	18.4%	34.8%	55.2%	62.2%
0.0080	99.8%	18.4%	32.2%	43.4%	62.4%
0.01	100.0%	18.2%	46.2%	55.4%	81.2%
0.012	100.0%	19.4%	35.0%	71.8%	71.8%
0.014	100.0%	19.4%	64.0%	70.4%	71.8%
0.016	100.0%	23.0%	75.0%	77.6%	71.8%
0.018	100.0%	19.2%	70.2%	88.8%	71.6%
0.02	100.0%	18.4%	51.2%	66.2%	53.0%

Figur C.7: Test af tidsgrænse for et enkelt random walk i PRMs udvidelsesfase, succesrater.

		Box	Bunker	Labyrinth	Tunnel-2	Tunnel
	min	28.0	41.0	20.0	22.0	29.0
0.0020	avg	36	93	93	46	38
	max	194.0	185.0	220.0	141.0	116.0
	min	23.0	37.0	20.0	18.0	15.0
0.0040	avg	31	92	91	37	27
	max	120.0	255.0	187.0	222.0	73.0
	min	23.0	44.0	23.0	23.0	11.0
0.0060	avg	34	142	117	40	28
	max	153.0	609.0	1578.0	175.0	124.0
	min	20.0	28.0	22.0	19.0	14.0
0.0080	avg	31	146	112	31	29
	max	100.0	547.0	408.0	177.0	180.0
	min	19.0	43.0	21.0	18.0	11.0
0.01	avg	28	136	99	36	29
	max	78.0	337.0	385.0	245.0	158.0
	min	18.0	44.0	21.0	17.0	11.0
0.012	avg	28	151	93	36	25
	max	97.0	1854.0	401.0	121.0	93.0
	min	15.0	29.0	13.0	13.0	10.0
0.014	avg	24	116	87	33	25
	max	91.0	329.0	272.0	211.0	98.0
	min	16.0	34.0	16.0	14.0	11.0
0.016	avg	24	128	91	33	22
	max	148.0	662.0	467.0	174.0	68.0
	min	14.0	28.0	14.0	12.0	11.0
0.018	avg	23	122	86	31	23
	max	92.0	307.0	342.0	116.0	64.0
	min	12.0	28.0	13.0	11.0	10.0
0.02	avg	22	115	77	28	19
	max	94.0	423.0	291.0	104.0	81.0

Figur C.8: Test af tidsgrænse for et enkelt random walk i PRMs udvidelsesfase, forespørgselstider (ms).

D Resultater for SRT-parametertests

	Box	Bunker	Labyrinth	Tunnel-2	Tunnel
0.1	100.0%	14.0%	26.2%	81.2%	77.8%
0.2	100.0%	19.0%	34.2%	66.6%	85.6%
0.3	100.0%	20.0%	35.0%	72.4%	86.6%
0.4	100.0%	20.0%	31.4%	60.2%	71.8%
0.5	100.0%	22.4%	27.6%	53.2%	82.0%
0.6	100.0%	21.4%	25.6%	57.0%	63.2%
0.7	100.0%	21.0%	28.2%	60.6%	63.8%
0.8	100.0%	21.0%	29.2%	62.2%	69.0%
0.9	100.0%	21.0%	25.6%	50.0%	63.4%

Figur D.1: Test af tidsfordeling mellem SRT konstruktionsfaser, succesrater.

		Box	Bunker	Labyrinth	Tunnel-2	Tunnel
	min	74.0	386.0	151.0	102.0	92.0
0.1	avg	166	847	823	404	269
	max	322.0	2778.0	1902.0	1102.0	1147.0
	min	88.0	440.0	166.0	102.0	119.0
0.2	avg	182	886	751	435	398
	max	374.0	2406.0	1750.0	1288.0	1397.0
	min	83.0	276.0	142.0	101.0	91.0
0.3	avg	184	856	789	440	393
	max	442.0	1872.0	1951.0	1465.0	1548.0
	min	65.0	270.0	196.0	102.0	117.0
0.4	avg	185	842	760	429	376
	max	406.0	2894.0	1814.0	1555.0	1548.0
	min	104.0	303.0	168.0	127.0	115.0
0.5	avg	194	860	769	404	400
	max	506.0	2868.0	2035.0	1718.0	1636.0
	min	75.0	317.0	143.0	111.0	106.0
0.6	avg	188	808	720	421	360
	max	417.0	1756.0	2213.0	1757.0	1863.0
	min	101.0	289.0	178.0	134.0	141.0
0.7	avg	191	784	738	426	365
	max	417.0	1528.0	1712.0	1839.0	1584.0
	min	86.0	287.0	180.0	123.0	116.0
0.8	avg	190	817	817	456	368
	max	404.0	1927.0	1781.0	1782.0	1790.0
	min	100.0	250.0	167.0	111.0	113.0
0.9	avg	197	563	743	365	364
	max	425.0	1646.0	1863.0	1589.0	1818.0

Figur D.2: Test af tidsfordeling mellem SRT konstruktionsfaser, forespørgselstider (ms).

	Box	Bunker	Labyrinth	Tunnel-2	Tunnel
5.0	100.0%	21.2%	30.0%	58.4%	72.8%
15.0	100.0%	22.0%	31.0%	53.0%	72.8%
25.0	100.0%	21.0%	29.0%	59.4%	72.8%
35.0	100.0%	21.2%	27.6%	57.8%	73.2%
45.0	100.0%	20.0%	29.6%	55.2%	72.8%
55.0	100.0%	22.8%	30.6%	63.8%	77.6%

Figur D.3: Test af antal nærmeste naboer, succesrater.

		Box	Bunker	Labyrinth	Tunnel-2	Tunnel
	min	78.0	339.0	126.0	94.0	96.0
5.0	avg	164	774	653	407	363
	max	435.0	2731.0	1653.0	1735.0	1416.0
	min	82.0	329.0	157.0	106.0	113.0
15.0	avg	172	829	747	374	379
	max	383.0	2102.0	1677.0	1505.0	1554.0
	min	73.0	254.0	176.0	133.0	103.0
25.0	avg	172	759	701	423	363
	max	577.0	1827.0	1726.0	1595.0	1585.0
	min	83.0	307.0	156.0	121.0	105.0
35.0	avg	170	757	698	440	384
	max	457.0	2385.0	2137.0	1370.0	1646.0
	min	73.0	231.0	165.0	101.0	106.0
45.0	avg	165	665	722	376	364
	max	359.0	1795.0	1658.0	1391.0	1760.0
	min	59.0	249.0	119.0	82.0	75.0
55.0	avg	135	592	608	400	353
	max	318.0	1744.0	1571.0	1451.0	1373.0

Figur D.4: Test af antal nærmeste naboer, forespørgselstider (ms).

	Box	Bunker	Labyrinth	Tunnel-2	Tunnel
1.0	100.0%	21.6%	27.4%	52.0%	71.4%
3.0	100.0%	22.2%	27.2%	56.2%	76.4%
5.0	100.0%	21.0%	27.0%	56.6%	67.4%
7.0	100.0%	20.6%	27.2%	54.6%	71.4%
9.0	100.0%	22.8%	27.0%	53.0%	71.8%

Figur D.5: Test af antal tilfældige naboer, succesrater.

		Box	Bunker	Labyrinth	Tunnel-2	Tunnel
	min	75.0	331.0	184.0	124.0	124.0
1.0	avg	182	852	728	348	383
	max	416.0	2126.0	1677.0	1780.0	1665.0
	min	91.0	323.0	121.0	136.0	113.0
3.0	avg	179	846	727	396	424
	max	388.0	2123.0	1975.0	1767.0	2015.0
	min	93.0	402.0	187.0	117.0	101.0
5.0	avg	186	822	765	410	378
	max	468.0	1457.0	1733.0	1661.0	1784.0
	min	88.0	350.0	169.0	146.0	105.0
7.0	avg	191	805	748	406	364
	max	421.0	1841.0	1902.0	1714.0	1537.0
	min	95.0	381.0	142.0	121.0	129.0
9.0	avg	185	896	715	381	417
	max	373.0	2424.0	1739.0	1383.0	1814.0

Figur D.6: Test af antal tilfældige naboer, forespørgselstider (ms).

	Box	Bunker	Labyrinth	Tunnel-2	Tunnel
25.0	100.0%	20.2%	28.4%	54.2%	68.8%
50.0	100.0%	20.8%	28.0%	50.8%	73.6%
75.0	100.0%	22.8%	32.4%	58.4%	66.4%
100.0	100.0%	20.8%	28.6%	53.0%	79.8%
125.0	100.0%	23.4%	33.0%	59.2%	75.6%
150.0	100.0%	21.2%	31.8%	62.6%	72.0%
175.0	100.0%	21.8%	31.6%	64.2%	72.6%
200.0	100.0%	22.8%	33.8%	56.6%	65.6%

Figur D.7: Test af skridtstørrelse for SRT-RRT, succesrater.

		Box	Bunker	Labyrinth	Tunnel-2	Tunnel
	min	89.0	265.0	171.0	102.0	99.0
25.0	avg	184	811	779	382	370
	max	355.0	1624.0	1868.0	1758.0	1395.0
	min	92.0	252.0	198.0	124.0	123.0
50.0	avg	188	827	693	373	394
	max	357.0	1781.0	2057.0	1340.0	1595.0
	min	94.0	282.0	132.0	125.0	112.0
75.0	avg	184	885	747	418	385
	max	717.0	3651.0	1998.0	1394.0	1781.0
	min	95.0	295.0	155.0	124.0	132.0
100.0	avg	184	798	713	371	414
	max	414.0	1759.0	1747.0	1953.0	1997.0
	min	94.0	340.0	198.0	124.0	124.0
125.0	avg	184	890	738	446	400
	max	480.0	3368.0	1824.0	1822.0	1608.0
	min	98.0	295.0	122.0	121.0	104.0
150.0	avg	189	876	723	458	425
	max	414.0	2463.0	2338.0	1775.0	1765.0
	min	72.0	391.0	131.0	116.0	118.0
175.0	avg	183	951	700	433	406
	max	358.0	2111.0	1919.0	1669.0	1744.0
	min	93.0	320.0	146.0	125.0	115.0
200.0	avg	182	770	669	382	344
	max	379.0	1513.0	2030.0	1681.0	1525.0

Figur D.8: Test af skridtstørrelse for SRT-RRT, forespørgselstider (ms).

	Box	Bunker	Labyrinth	Tunnel-2	Tunnel
20.0	100.0%	20.0%	26.6%	51.8%	68.0%
40.0	100.0%	21.8%	27.0%	57.6%	74.0%
60.0	100.0%	21.6%	32.4%	59.2%	81.2%
80.0	100.0%	22.4%	28.6%	60.2%	76.0%
100.0	100.0%	20.4%	31.8%	66.0%	90.8%

Figur D.9: Test af antal samples for SRT-RRT, succesrater.

		Box	Bunker	Labyrinth	Tunnel-2	Tunnel
	min	62.0	260.0	146.0	97.0	107.0
20.0	avg	180	718	683	350	330
	max	450.0	1506.0	2071.0	1654.0	1552.0
	min	81.0	262.0	126.0	100.0	104.0
40.0	avg	167	823	674	390	388
	max	376.0	2748.0	2080.0	1565.0	1536.0
	min	79.0	300.0	113.0	127.0	85.0
60.0	avg	163	776	720	406	417
	max	453.0	1660.0	1965.0	1430.0	1555.0
	min	68.0	388.0	133.0	102.0	102.0
80.0	avg	161	819	665	414	374
	max	345.0	1879.0	1631.0	1790.0	1256.0
	min	81.0	331.0	134.0	103.0	97.0
100.0	avg	172	755	681	433	443
	max	381.0	1571.0	1811.0	1499.0	1519.0

Figur D.10: Test af antal samples for SRT-RRT, forespørgselstider (ms).

	Box	Bunker	Labyrinth	Tunnel-2	Tunnel
10.0	100.0%	20.0%	34.8%	65.0%	81.4%
20.0	100.0%	21.0%	35.2%	59.0%	73.8%
30.0	100.0%	21.4%	30.0%	53.6%	68.4%
40.0	100.0%	21.8%	29.0%	55.0%	68.6%
50.0	100.0%	21.8%	26.6%	55.4%	69.6%
60.0	100.0%	22.2%	30.0%	58.6%	80.6%
70.0	100.0%	20.8%	28.8%	54.4%	73.4%
80.0	100.0%	23.2%	29.4%	58.0%	74.6%
90.0	100.0%	21.8%	27.4%	54.6%	68.6%
100.0	100.0%	22.4%	28.8%	59.8%	68.2%

Figur D.11: Test af antal samples for SRT-BiRRT (LP), succesrater.

		Box	Bunker	Labyrinth	Tunnel-2	Tunnel
10.0	min	92.0	329.0	126.0	86.0	93.0
	avg	179	805	767	396	410
	max	403.0	1495.0	1786.0	1371.0	1273.0
20.0	min	94.0	288.0	151.0	103.0	117.0
	avg	195	802	812	413	377
	max	493.0	1566.0	1702.0	1620.0	1634.0
30.0	min	82.0	347.0	178.0	137.0	110.0
	avg	195	834	747	389	358
	max	498.0	1792.0	2097.0	1692.0	1909.0
40.0	min	100.0	330.0	129.0	126.0	98.0
	avg	194	927	745	400	392
	max	424.0	2629.0	1676.0	1411.0	1728.0
50.0	min	86.0	365.0	139.0	114.0	115.0
	avg	194	922	721	386	404
	max	437.0	2519.0	1782.0	1872.0	1776.0
60.0	min	100.0	216.0	102.0	82.0	77.0
	avg	199	626	606	375	361
	max	526.0	2208.0	1889.0	1717.0	1580.0
70.0	min	68.0	231.0	128.0	96.0	104.0
	avg	138	561	636	374	427
	max	516.0	1383.0	1820.0	1727.0	1758.0
80.0	min	64.0	262.0	118.0	87.0	71.0
	avg	140	615	587	411	397
	max	372.0	2285.0	2183.0	1855.0	1748.0
90.0	min	58.0	202.0	106.0	75.0	79.0
	avg	139	666	529	349	376
	max	582.0	3128.0	2204.0	1693.0	1755.0
100.0	min	60.0	268.0	110.0	106.0	99.0
	avg	141	654	537	406	373
	max	315.0	2384.0	1704.0	1781.0	1687.0

Figur D.12: Test af antal samples for SRT-BiRRT (LP), forespørgselstider (ms).

Indeks

- C , 14
- C_f , 15
- $N_d(x)$, 15
- $\mathbb{R}P^3$, 37
- $dist_C$, 38
- $dist_{\mathbb{R}^3}$, 38
- $dist_{max}$, 15
- kD -træ, 37
- Cell Decomposition, 7
- Generalized movers problems, 3
- Local planner, 16
- Union-find, 27

- Afstandsfunktion, 38
- Assembly planning, 4

- Biased Bingo Bag, 34
- Bounding Volume Hierarchy, 43
- Brushfire, 8
- BVH-træ, 43

- Centroide, 29

- Dynamisk kD -træ, 42

- enhedsquaternion, 11
- Euler vinkler, 12

- Flexible objects, 4
- Forhindringer, 14
- Frihedsgrad, 13

- Generalized movers problem, 3
- Graf, 32

- Hausdorff-afstand, 38
- Heuristic planners, 10

- Interpolation af konfigurationer, 15
- Interpolering af quaternioner, 13

- Kasse-trekant-kollisionstest, 45
- Kinodynamic problems, 3
- Kollisionstest, 42
- Konfigurationsrummet, 14

- Landkort, 17
- Led, 13
- Lige-linje-LP, 16
- Lokal planlægger, 16
- LP, 16

- Manipulation planning, 3
- Milesten, 29
- Multiple movers problem, 3

- Piano movers problem, 3
- Potential-field methods, 7
- potentialefunktioner, 7
- PRM, 26
- Probabilistic Roadmaps, 26

- quaternion, 11

- Randomized Path Planner, 8
- Rotation, 11
- Rotationsmatricer, 12
- RPP, 8

- Sampling af konfigurationer, 15
- Sampling-based Roadmap of Trees, 29
- SRT, 29

- Trekant-trekant-kollisionstest, 45
- Trekantsnet, 34
- Triangle mesh, 34

- Uniform sampling af quaternioner, 12
- Union-find, 33
- Visibility graphs', 17