CHALMERS

Autonom marin ruttplanering med sökalgoritmer

Studie av sökalgoritmen i en skapad miljö

Examensarbete inom högskoleingenjörsprogrammet Mekatronik

Joel Fritsch

INSTITUTIONEN FÖR DATA OCH INFORMATIONSTEKNIK

CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2022
www.chalmers.se

Förord

Detta är ett examensarbete som utförs från mekatronikprogrammet på Chalmers Tekniska Högskola.

Arbetet examineras under institutionen för data och informationsteknik då det helt handlar om mjukvara och kod

Sammanfattning

Dagens verktyg för navigering på land så som google maps är väldigt avancerade och många använder dem dagligen. När det kommer till liknande verktyg för marint bruk ligger teknologin lång bakom. Navigering på sjön görs till stor del fortfarande manuellt; enstaka produkter som Navionic's "dock-to-dock" [1] ger liknande resultat som Google Maps men utbudet är fortfarande litet. I detta arbete kommer det undersökas vilka olika sökalgoritmer som skulle lämpa sig till användning inom autonom marin ruttplanering. Syftet blir att göra en grundstudie inom området för att underlätta vidare forskning och utveckling av denna typ av tjänster och produkter. Arbetet kommer att utföras fristående från sponsorer eller företag och är endast kopplat till Chalmers Tekniska Högskola.

Studien kommer att fokusera på åtta sökalgoritmer och undersöka ifall någon av algoritmerna skulle vara mer lämpad än de andra för denna typ av användningsområde. Ett JavaScript biblioteket för sökalgoritmer [2] kommer att utnyttjas och algoritmerna kommer att skrivas om för att passa studiens syfte. Testerna kommer att utföras på en egenskapad grafisk testmiljö som skall efterlikna svensk skärgård. Resultaten kommer att bestå av elva olika parametrar som kommer att användas för att jämföra algoritmernas svagheter och styrkor. Studien kommer att vara strikt teoretisk och ingen hårdvara kommer vara inblandad.

Resultaten av studien visar på att det bland de utvalda algoritmerna inte finns någon definitiv vinnare utan att alla algoritmer har sina styrkor och svagheter. Med detta sagt så visar sig algoritmen A* som den mest lovande väl passande algoritmen utan någon stor nackdel. Det visar sig också att prestandan av algoritmerna är till stor del kopplad till hur detaljerad sökmiljön är, det vill säga kvalitén hos sjökorten. Detta tyder på att det är det sjökorten som behöver bli mer detaljerade om teknologin för autonom marin ruttplanering skall gå framåt.

Abstract

Today's tools for navigation on land such as google maps are very advanced and many use them daily. When it comes to similar tools for marine use, the technology is a bit behind. Navigation at sea is still mostly manual, some products like Navionic's "dock-to-dock" [1] give similar result to google maps but the supply is still sparse. This project will examine which search algorithms would lend themselves well to autonomous marine route planning. The aim is to make a study that paves the way for more research and development in the type of services and products. The study will be conducted independent of any company or sponsor and will only be connected to Chalmers University of Technology.

The study will focus on eight different search algorithms to examine if any one of them would be better suited for this task. A JavaScript library for search algorithms will be used and the algorithms will be changed to fit the purpose of this study. The tests will be performed on a custom graphical environment that is trying to replicate Swedish archipelago. The result will consist of eleven parameters that will be used to compare the algorithms strengths and weaknesses. The study will be strictly theoretical, and no hardware will be involved.

The results of the study show that there is no clear winner of the chosen algorithms and that they all have strengths and weaknesses. With this said, the algorithm A* shows the most promise as being the most well rounded with a less glaring downside. The results show that no algorithm will always be the obvious choice and that the effectiveness of the algorithms largely corresponds with how detailed its search environment is. This indicated that for autonomous marine route planning to take a step forward, the sea charts need to become better.

Innehållsförteckning

1. Inledning

1.1 Bakgrund

Att planera sin färdväg i förväg har alltid varit av stor vikt när det kommer till att på bästa sätt ta sig från punkt A till punkt B. Historiskt sett har dessa färder och handelsrutter skett på havet då det var väldigt svårt att färdas långa sträckor på land utan ett etablerat vägnät. Denna ruttplanering gjordes då traditionellt av en expert inom navigering som bl.a. använde sig av kartor, lodning, egen kartläggning och himlen för att hitta en rutt till önskat mål.

Ett av de stora framstegen inom ruttplanering var det militära positioneringssystemet Global Positioning System (GPS) som gjorde entré för allmänheten på 80-talet. GPS klarade av att ytterst noggrant ge sin användare en väldigt noggrann position på jorden med hjälp av satelliter, och denna nya teknik revolutionerade ruttplanering. Idag har automatiska ruttplanerare och GPS blivit så vanligt att de finns i nästan allas händer genom mobiltelefoner och datorer. Vi har vant oss så mycket vid olika former av ruttplanerare att få av oss hade varit bekväma utan en tillhands och kravet på vad de skall klara av ökar parallellt med att dessa ruttplanerare konstant förbättras.

Företaget Google har idag med sin applikation Google Maps satt standarden på vad som förväntas av en automatisk ruttplanerare på land, men samtidigt lever de flesta alternativen på sjön inte upp till denna standard. Dagens privata ruttplanerare anpassade för sjöbruk, eller ”plotters” som de tidigare också kallats, saknar som oftast en utmärkande egenskap och det är möjligheten att bara kunna ange en slutdestination och få förslag på en eller flera alternativa rutter som tar dig hela vägen från punkt A till punkt B och tar hänsyn till dina ev. övriga önskningar. Idag måste användaren som oftast fortfarande ”plotta” ut sina egna rutter med hjälp av brytpunkter så därav namnet.

Det finns många anledningar till att det skulle vara bra att marin ruttplanering tog ett steg framåt och blev mer autonom. Stora fraktfartyg som kör långa sträckor som över Atlanten eller liknande använder sig redan av modeller för att skära ner på kostnader i form av bränsleförbrukning. Men det finns alltid förbättringar att göra och med en algoritm som gjorde uträkningen skulle den kunna anpassa rutten i realtid för att ta hänsyn till oväder eller andra förändringar i miljön för att på så sätt förbättra både ekonomi, miljöpåverkan och säkerhet. När det kommer till fritidsbåtar skapas oftast rutter helt för hand och ofta så finns det ingen rutt alls. Detta gör att en autonom ruttplanerare skulle drastiskt kunna dra ner på bränsleförbrukningen för båtar i fritidsbruk och därför miljöpåverkan. Och eftersom GPS är någon som redan finns på i stort sett alla båtar så skulle det inte vara svårt att inkludera en funktion för att undvika områden med redan mycket trafik för att på så sätt förhindra krockar.

1.2 Syfte

Projektet är tänkt att analysera olika ruttplanerings-algoritmer för att skapa ett statistiskt underlag för vilka algoritmer som kan tänkas vara lämpliga vid utveckling av en automatisk ruttplanerare för fritidsbåtar. Detta innefattar en mängd olika algoritmer samt en virtuell testmiljö som skall kunna utföra dessa analyser.

1.3 Mål

Målet för projektet är att utvärdera ifall det finns en klar vinnare eller ifall olika algoritmer kommer att prestera bäst i olika scenarion. Utifrån detta kommer projektet försöka komma fram till vilken algoritm som skulle vara mest lämpad till en marin ruttplanerare för fritidsbåtar.

1.4 Avgränsningar

Algoritmerna kommer att utvecklas i JavaScript och kommer inte simuleras på existerande digitala sjökort utan på egengenererade marina miljöer där sjömärken, farleder, väder, vind och andra dynamiska element kommer att exkluderas. I övrigt kommer ingen hårdvara vara inblandad och analyserna begränsas till digital miljö. Och därmed kommer endast ett antal förvalda algoritmer att analyseras.

2. Teoretisk referensram

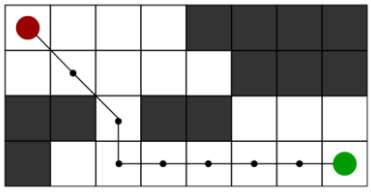
2.1 Sökalgortitmer

Sökalgoritmer har funnits länge och är en mycket bred term som innefattar alla typer av algoritmer som söker efter ett givet objekt bland en större mängd objekt [3]. En For-loop [4] skulle kunna klassas som en typ av linjär sökalgoritm för att hitta ett givet objekt i en array [5]. I dagens samhälle har sökalgoritmer en uppsjö av användningsområden såsom att hitta ett specifikt objekt på en server eller den kortaste vägen från punkt A till B för en bil, så kallad ruttplanering.

2.2 Ruttplanering

Användningsområdet för sökalgoritmen i detta projekt kommer att vara ruttplanering. Alla sökalgoritmerna som fokuserar på ruttplanering går att placera i en av kategorierna uninformed search, informed search och local search. Detta projekt kommer att arbeta med algoritmer som faller i kategorin informed search och uninformed search. Alla tre typerna av algoritmer letar efter en optimal lösning till ett problem, men för att sänka söktiden gör informed search detta med vad som kan kallas kvalificerade gissningar.

När man ska simulera en sökning för en sträcka, som i en marin rutt, används ofta ett rutnät där alla individuella rutor kallas noder och har upp till 8 grannar (se figur 2.1).



Figur 2.1, Beskriver miljön som algoritmerna söker i.

Dessa noder har olika egenskaper. Egenskaper som en nod ofta har är, koordinater så att algoritmen vet var denna noden är placerad i rutnätet, en flagga för ifall noden kan korsas, en kostnad som bestämmer t.ex sträckan, faran eller svårigheten denna noden representerar samt en parent för att bland annat kunna återskapa rutten efter en färdig sökning. Det finns inget tak på hur många egenskaper dessa noder kan tilldelas, fler egenskaper resulterar i en komplexare karta med fler parametrar som påverkar sökalgoritmens resultat.

```
X = 3
Y = 2

isLand = False

Weight = 1

parentX = 2
parnetY = 2
```

Figur 2.2, Exempel på parametrar för en algoritm.

Algoritmen kommer att söka sig från en bestämd startpunkt från ruta till ruta tills dess att den har hittat den kortaste vägen till målet eller insett att det är omöjligt att nå målet. Den kortaste vägen innebär den kombination av sammankopplade noder som får den lägsta adderade kostnaden, alltså inte nödvändigtvis vägen med minst antal noder i sig.

2.3 Uninformed search och informed search

När en algoritm endast tar hänsyn till kostnaden av nästa nod när den skall utvärdera sitt steg är den en så kallad uninformed algorithm. Detta innebär att algoritmen inte bryr sig om ifall en ny nod är ett steg bort från målet, så länge det är noden med den lägsta kostnaden av möjliga noder så kommer den att blir algoritmens nästa steg, detta kallas att en nod expanderas av algoritmen. Det finns även algoritmer som faller i uninformed kategorin men bortser helt från kostnader, såsom depth first search och breadth first search. Dessa algoritmer söker helt regelmässigt och har ingen intelligens att utesluta uppenbart dåliga alternativ med hjälp av kostnader. Denna typ av algoritmer kommer inte att testas i detta projekt.

Påbyggnaden till uninformed algoritmer är vad som kallas informed algoritmer: man säger att dessa algoritmer gör kvalificerade gissningar. Vad som gör gissningarna till kvalificerade kallas heuristik. Detta innebär att algoritmen utnyttjar ytterligare en parameter när den evaluerar vilken nod som ska bli nästa steg. Denna parameter är kortaste möjliga sträckan från den evaluerade noden till mål. Detta leder till att algoritmen inte behöver söka alla möjliga alternativ utan snabbt kan bestämma att en nod som går ett steg bakåt inte är attraktiv eftersom heuristiken kommer ge en högre kostnad jämfört med att gå framåt. Genom att introducera detta moment ökas sökhastigheten signifikant.

2.4 A*

A* är kanske den mest välkända algoritmerna när det kommer till ruttplanering och publicerades 1968 [6]. Den faller i kategorin informed search eftersom den utnyttjar heuristik. Formeln som algoritmen använder för att bestämma sitt nästa steg när den söker från A till B ser ut som följande $f(n) = g(A,n) + h(n,B)$. Där f är kostnaden som skall minimeras för varje steg, g är den sammanlagda kostnaden för alla steg från A till och med n och h är den heuristiska kostnaden från n till B.

När algoritmen skall bestämma nästa steg från den nuvarande noden n används denna formel för att tilldela alla angränsande noder ett f värde. Detta kallas att evaluera en nod. När en nod har evaluerats och tilldelats sitt f värde placeras den i listan som består av alla noder som algoritmen tidigare har evaluerat men inte ännu expanderat, ofta känd som openlist. Utöver detta läggs även den nuvarande noden n till som parent för den evaluerade noden $n + 1$. Detta görs så att algoritmen vet vilken väg som resulterade i kostnaden för noden så att den vet varifrån den ska expandera noden i framtiden om den inte väljer att göra det nu. När dom nya grannarna har lagts i openlist väljer algoritmen noden med det lägsta f-värdet i openlist och expanderar till den från dess parent.

När algoritmen har expanderat till noden med lägst f värde flyttas denna från openlist till en lista bestående av tidigare expanderade noder, ofta känd som closedlist. Efter detta upprepas processen och grannarna till den expanderade noden evalueras. Ifall algoritmen kommer fram till en nod som redan ligger i closedlist eller openlist, så evalueras den igen och det nya f värdet jämförs med det gamla för att se ifall en kortare väg till noden har upptäckts. Ifall det nya f värdet är mindre än det gamla uppdateras både detta och parent, om detta inte uppfylls

behålls den gamla evalueringen. Detta upprepas tills dess att n är B. För att återskapa rutten använder algoritmen parametern parent. Den börjar med att kolla vilken parent noden B har, vilket blir B-1, och sparar undan denna nod i en lista. Den kollar sedan vilken parent noden B-1 har och sparar undan den i samma lista. Denna process upprepas till dess att en nod har parent A vilket betyder hela rutten är sparad.

2.5 Djikstras algoritm

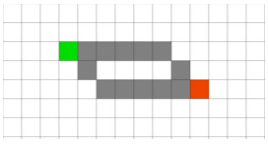
Dijkstras algoritm är fader algoritmen till A* och publicerades 1959 [7], den fungerar på ett liknande sätt, skillnaden är att den är en uniformed algoritm. Detta betyder att den inte utnyttjar heuristik utan endast evaluerar sina steg baserat på kostnaden av en specifik nod. Skulle man sätta $h(n, B) = 0$ för A* så får man Dijkstras algoritm, skillnaden mellan de två algoritmerna blir att A* riktar sin sökning mot målet medan Dijkstras kommer att söka i en större mängd noder. Detta leder till att A* generellt sett har snabbare sökningen men med en minimal men existerande chans att inte hitta den optimala vägen medan Dijkstras garanterar ett optimalt resultat till uppoffringen av en långsammare sökning.

2.6 Best First search

Best first search är en informed algoritm med en aggressiv heuristik. Vad detta betyder är att det har lagts till en stor multiplikator på $h(n, B)$ vilket gör att denna väger mycket tyngre i $f(n) = g(A, n) + h(n, B)$. Detta leder till att algoritmen nästan aldrig hittar den optimala vägen då den bryr sig mer om att röra sig mot målet än att undvika kostsamma noder eller omvägar. Vinsten man får med detta är en extremt effektiv algoritm som snabbt kan hitta en väg i komplicerade miljöer och sällan behöver söka många noder. Fundamentalt fungerar algoritm på samma sätt som A* med open set och closed set, utöver den aggressiva heuristiken ligger den stora skillnaden i att best first aldrig kan evaluera om en redan evaluerad nod.

2.7 Jump Point Search

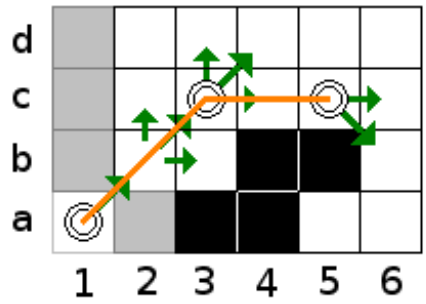
Jump point search är en utvidgning av A* som riktas mot att förbättra sökningar på kartor med konstant kostnad och kan inte användas på kartor där alla noder inte har samma kostnad. Fundamentalt fungerar algoritmen på samma sätt med ett open set och ett closed set samt $f(n) = g(A, n) + h(n, B)$. Skillnaden ligger i hur algoritmen lägger till nya noder i open set och hur den expanderar noder. Jump point search försöker att minimera söktiden genom att endast lägga till ”intressanta” noder i open set samt ignorera symmetrier såsom att två vägar kan ha samma kostnad (se figur 2.3).



Figur 2.3, Exempel på symmetri i sökning.

Detta sker genom en rad sökningar, utifrån en nod startas vinkelräta, vertikala och diagonala sökningar. När en intressant nod har upptäckts läggs den till i open set, den aktuella sökningen avslutas och en ny sökning utifrån f påbörjas. Vad som gör en nod till intressant följer en rad med regler och kan vara svårt att sammanfatta men skulle kunna definieras som att sökningen

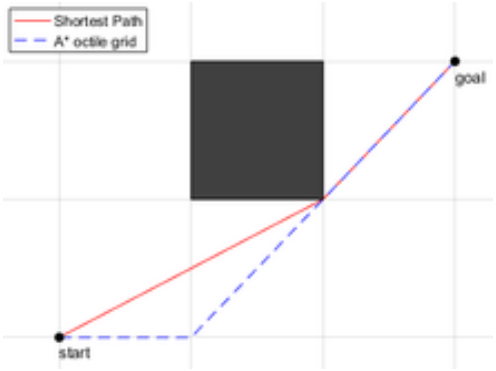
gör en ny upptäckt. I bilden nedan är upptäckten att det nu är möjligt att gå till höger, vilket gör att en ny nod expanderas



Figur 2.4, Exempel på ny upptäckt.

2.8 Theta*

Som många andra algoritmer är Theta*, även kallad any angle, också en påbyggnad av A*. Målet med algoritmen är att en specifik nods parent inte behöver vara dess granne vilket den måste i A*. Algoritmen fungerar fundamentalt på samma sätt som A* men utnyttjar en siktlinjefunktion, vilket gör att den inte behöver följa kartans rutnät på samma sätt. Theta* evaluerar en ny nod Q från noden N, ifall parent(N) har siktlinje till Q så ignoreras N och parent(Q) sätts parent(N). Detta upprepas till dess att siktlinjen bryts, först då expanderas noden som senast hade siktlinje. Detta är vad som skiljer A* från Theta*, vinsten man får av detta kommer oftast inte i form av snabbhet eller effektivitet utan att slippa köra den slutliga vägen genom efterbehandling för att uppnå visuellt acceptabel rutt i form av vägpunkter istället för hela linjer.



Figur 2.5, Exempel på siktlinje.

2.9 Bi-directional searching

Bi-directional searching är en typ av sökning som går att tillämpa på alla algoritmer, det innebär att två sökningar sker samtidigt, en med utgångspunkt i startnoden och en med utgångspunkt i slutnoden. Sökningen är klar när dessa två möts och skapar en sluten väg från start till slut, vinsten med detta är att sökningarna oftast går snabbare men förlusten blir är att det inte längre går att garantera en optimal rutt eftersom ruten ses som klar vid första tillfället som halvorna möts.

3. Metod

Projektet kommer att inledas en fördjupning i ämnet grafbaserade sökalgoritmer, för att utvärdera vilka algoritmer som kan vara lämpade för denna uppgift.

När detta teoretiska stadie är klart kommer en utvecklingsfas att börja där bl.a. testmiljön skall skapas. I denna testmiljö kommer utvalda algoritmer att testas hur de klarar av de ställda kraven.

1. Planering
2. Teknisk fördjupning i sökalgoritmer
3. Utveckling av en testmiljö för utvalda algoritmer
4. Anpassning av utvalda algoritmer för att passa testmiljön
5. Analys i testmiljön av dessa algoritmer.
6. Utvärdering av resultat.

3.1 Planering

I början av projektet togs en ungefärlig tidsplanering fram med uppskattningar om de olika momentens tidskonsumtion.

Informationsinhämtning av dagens kommersiella ruttplanerare.	2v
Studier i algoritmer	3v
Studier i programspråk	1v
Samanställning av teoretisk info	1v
Utveckling av testmiljö	6v
Utveckling av algoritmer	4v
Sammanställning av resultaten samt rapportskrivning	7v

Figur 2.6, planeringsrapport.

3.2 Teknisk fördjupning i sökalgoritmer

Redan innan projektet startades fanns det en god uppfattning om vilka algoritmer som skulle vara aktuella för en undersökning som denna men för att säkerhetsställa den tidigare kunskapen gjordes en djup analys. Andra liknande studier [8] låg till grund för att hitta vilka algoritmer som skulle vara relevanta men även mer allmänna data om sökalgoritmer samlades in för att skapa en djup bild av vad som finns [9]. Fundamentalt skiljande algoritmer och optimationssätt som ant colony undersöktes också [10] samt en allmän förståelse om hur nuvarande marin ruttplanering utförs [11] [12] [13].

Eftersom projektet handlar om att jämföra algoritmerna som skall undersökas så valdes JavaScript som programspråk då det finns ett stort bibliotek [2] med redan klara algoritmer och utvecklingen av dessa inte skulle vara relevant. Dessa algoritmer kommer endast behöva modifieras marginellt för att nå det stadie som är passande för att göra en undersökning av detta slag.

Som utvecklingsmiljö valdes Visual Studio Code då det är mycket lämpat för att skriva både html och JavaScript och därmed kunna skapa en enkel hemsida för att visualisera testmiljön och algoritmerna.

3.3 Utveckling av en testmiljö för utvalda algoritmer

Eftersom algoritmerna redan existerade fick testmiljö funktionaliteten utformas efter hur dessa var uppbyggda för att nå korrekt funktionalitet.

Rent visuellt utvecklades testmiljön för att efterlikna en svensk skärgård med fokus på att undersöka relativt korta sträckor. Stor tyngd lades i att få en slumpmässig generering av varje karta för att kunna testa många scenarion och ge en rättvis bild av när det olika algoritmerna presterar bäst. För en bättre visualisering och felsökning av algoritmerna skapades enkel grafik och ett UI för testmiljön.

3.4 Anpassning av utvalda algoritmer

För att kunna göra en rättvis undersökning med givande resultat krävdes vissa ändringar till algoritmerna för att ge jämlika förutsättningar att prestera bra. Dessa ändringar gjordes utifrån parametrar som ansågs vara viktiga att ta hänsyn till men även för att anpassningar algoritmerna till att kunna agera funktionellt på den marina testmiljön.

3.5 Analys

Algoritmerna testades och jämfördes utifrån en mängd olika parametrar på ett flertal slumpmässigt genererade kartor, för att inte ge någon algoritm ett övertag inkluderades olika typer av hinder och svårigheter. Vissa av dessa parametrar är relevanta ur en kommersiell synvinkel medan andra är strikt knutna till hur algoritmen presterar.

3.6 Utvärdering

Parametrarna som hade analyserats rangordnades baserat på hur viktiga de var till en säker, fungerande och effektiv ruttplanerare. Denna rangordning kom till viss det från tidigare förståelse om vad en säker ruttplanerare innebär, som att det är viktigare att inte åka allt för nära land än att algoritmen gör sin uträkning lite snabbare. Men även från kunskapen som hade samlats in understudiens gång. Anledningen till att sunt förnuft används är för att det inte finns något facit på en optimal ruttplanerare, detta gör det svårt att helt objektivt rangordna vilka karakteristiker som är viktiga hos en ruttplanerare. Resultaten av hur algoritmerna presterade på dessa parametrar jämfördes helt objektivt och det slutgiltiga valet av den bästa algoritmen baserades på dessa resultat.

4. Genomförande

4.1 Översikt

Första steget var att skapa ett skelett för testmiljön, och för att kunna göra detta krävdes en förståelse om hur algoritmerna läser av sin omgivning. Eftersom biblioteket med algoritmer var skrivet i javaScript bestämdes det att grafik för miljön skulle skapas på en enkel hemsida med hjälp av html för att underlätta sammanfogningen av algoritmerna med denna. I biblioteket fanns redan en funktion för att skapa kartor som skulle representera miljön som algoritmerna rörde sig i. Denna modifierades för att bättre kunna uppnå det som projektet söker. Funktionen skapade endast en blank matris som sedan behövde vidareutvecklas till sitt avsedda syfte, såsom terräng som likna skärgård och viktning för noder runt om terräng. Efter att ett acceptabelt stadie av testmiljön har uppnåtts påbörjas implementeringen av en algoritm för att senare expandera till att inkludera alla algoritmer. Små ändringar gjordes i algoritmerna för att ge ett så bra resultat som möjligt. Vid sidan av utveckling av miljön utvecklas också ett UI för att visualisera alla nya funktioner. Efter att en naturlig generering av miljö var nådd blev nästa fokus att automatisera denna för att kunna skapa nya kartor utan att starta om programmet samt att kunna köra om flera algoritmer på samma karta. När detta var uppnått blev nästa fokus att uppgift att kunna ta ut data från algoritmerna, och det skapades en rad funktioner för att läsa och spara hur algoritmen utförde sin sökning. Dessutom utnyttjades data som redan sparas av algoritmen själv.

4.2 Vad ändrades i algoritmerna

För att algoritmerna skulle fungera som tänkt krävdes att de anpassades så att de kunde ta hänsyn till icke uniform kostnad på kartan. Detta krävdes eftersom vissa noder kommer att ha en högre kostnad då de ligger nära land vilket var något basalgoritmerna inte var kapabla att ta hänsyn till. Detta gjordes genom att gå igenom koden och lägga till en kostnads-parameter för funktion som skapade matrisen som låg till grund för kartan och sedan ge algoritmerna förmågan att läsa av denna parameter. Parametern var som standard tom eftersom den ökade kostnaden skulle ligga på noder runt land vilket gjorde att steget att ändra vikten måste vara ett av dom sista för skapandet av varje individuell karta.

4.3 UI och grafik

Grafik utvecklas i samband med att testmiljön blir mer avancerad och UI skapas när en acceptabel grafik har uppnåtts. Grafiken började som ett enkelt rutnät där varje ruta representerar den korresponderande noden. Varje ruta programmerades sedan till att kunna anta några olika färger som representerade nodernas tillstånd. Blå står för att detta är en nod som algoritmen får expandera och grå står för att detta är en nod som algoritmen inte får expandera.

Efter en sökning representerar rött att algoritmen har provat att expandera denna nod men har kommit fram till att den inte ingår i en optimal väg. Vägen som algoritmen kommer fram till som optimal presenteras av gröna noder.

Sedan lades möjligheten att interagera med rutnätet till i from av att trycka på en ruta och sedan hålla nere och dra till en annan ruta och på så bestämma start och stopp för en sökning. När detta var uppnått så påbörjades UI:n vilket bestod av knappar för alla funktioner som

programmet hade. Detta inkluderade att välja vilken algoritm som skulle köras samt att rensa kartor för att kunna köra en ny algoritm på samma karta.

4.4 Auto-generering av miljö, kartor och viktning

Kartan skapades med hjälp av en redan existerande funktion som tog matriser av ettor och nollor som input och sedan skapade en array med noder som algoritmen kan läsa. Nollor betydde att noden skulle vara expanderbar och ettor betydde att noden skulle ses som hinder. Funktion matades med en matris med endast nollor vilket skapade en karta där alla noder var expanderbara. Det skapades sedan funktioner för att modifiera kartorna till den typen av miljö och viktning som söktes. Detta innefattade generering av hinder och en speciell viktning för vissa noder. Alternativt hade det gått att göra en matris med både ettor och nollor för att sedan mata funktionen med denna. Det valdes att inte göra på detta sätter då funktionen inte kunde ta hänsyn till viktning och kartan hade därmed ändå behövt post-processing.

Auto-genereringen av kartor började skapas efter grafiken då det var lättare att bedöma ifall den fungerade som det önskades när det gick att visualisera den. Det började med att generera cirklar av land på bestämda platser på kartan. Detta var också ett sätt att se att grafiken fungerade som den skulle. Detta fortsatte med att generera cirklarna på slumpmässiga platser på kartan samt ge dem en irreguljär generering i storlek och symmetri. Det upptäcktes att resultatet blev mycket bättre om kartan byttes till att genereras som endast land och sedan skapa vatten med hjälp av dessa cirklar. Det lades till sist till knappar på UI för att enkelt kunna få mer eller mindre land i nästa karta utan att behöva starta om programmet och ändra i koden. Att generera nya kartor gjordes enkelt med hjälp av att göra en knapp som körde om matris funktionen.

Viktningen av kartan gjordes genom att skapa en funktion som söker alla noder i den färdiga kartan för att ta reda på vilka av dem som ligger angränsande land, nära land och långt ifrån land. Noder som ligger angränsande till land tilldelas en hög kostnad, noder nära eller långt ifrån land tilldelas en medelhög respektive låg kostnad.

4.5 Integration av algoritmer

Integrationen av den första algoritmen började med att söka på en karta med endast noder som gick att expandera. Enkla, manuellt placerade hinder lades sedan till för att se ifall algoritmen tog hänsyn till dessa och för att bekräfta att den planerade metoden med att lägga till hinder efter att kartan skapats skulle fungera. Samma metod användes för att bekräfta att algoritmen tog hänsyn till viktningen. Manuella kluster av noder med höga kostnader placerades för att se om algoritmen undvek dessa. Efter att dessa steg fungerat implementerades de resterande algoritmerna på liknande sätt.

4.6 Framtagning av resultat

För att skapa ett resultat krävdes visst förarbete. Vissa parametrar som algoritmen använder under körning anses som relevant data för den slutliga evalueringen. Men stora delar av den eftersökta data är mer knuten till hur algoritmen utför sin sökning utifrån ett tredjepersons perspektiv, alltså hur ser den slutliga rutten ser ut, inte hur den togs fram. För att ta fram denna data behövdes det skapas funktioner som externt analyserade hur algoritmerna gjorde sina sökningar och spara undan denna data under körning.

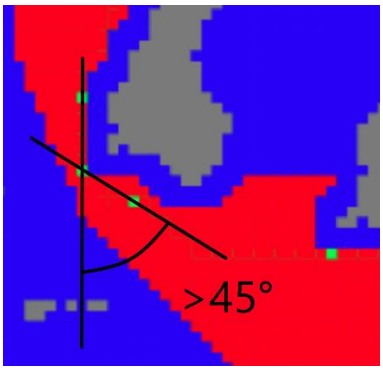
Funktionen “Smoothen Path” som redan fanns i biblioteket skrevs om så att den fungerade med de ändringar som hade gjorts till algoritmerna. Funktionen jämnar ut långa sträckor som ligger i samma siktlinje för att en bättre representation av den faktiska rutten. Rutten består sedan endast av de nödvändiga noderna, liknande vägpunkter. Detta görs för att bättre representera den verkliga rutten som en båt skulle ta. Detta utnyttjades på två sätt: Det skapades separata parametrar som analyserade den utjämnade vägen, och det gav en bättre bild av hur algoritmen presterade i en marin miljö.

Parametrarna som skapades är följande 11:

Basepath near land instances: Hur många gånger den icke utjämnade rutten expanderar en nod som gränsar till land.

Smoothpath near land instances: Hur många gånger den utjämnade rutten korsar en nod som gränsar till land, tas fram genom att dra ett streck mellan de olika vägpunkterna och se vilka noder som korsas.

Smoothpath not ok turns: hur många gånger den utjämnade rutten gör en sväng som är skarpare än 45 grader.



Figur 2.7, Illustration till ”Smoothpath not ok turns”.

Smoothpath length längden på den utjämnade rutten.

Basepath not ok turns hur många gånger den icke utjämnade rutten gör en sväng som är skarpare än 45 grader på två steg eller kortare.

Basepath length: Längden på en icke utjämnade rutten.

Smoothpath ok turns: hur många svängar som är mindre skarpa än 45 grader som den utjämnande rutten gör.

Basepath ok turns: hur många svängar som är mindre skarpa än 45 grader som den icke utjämnande rutten gör.

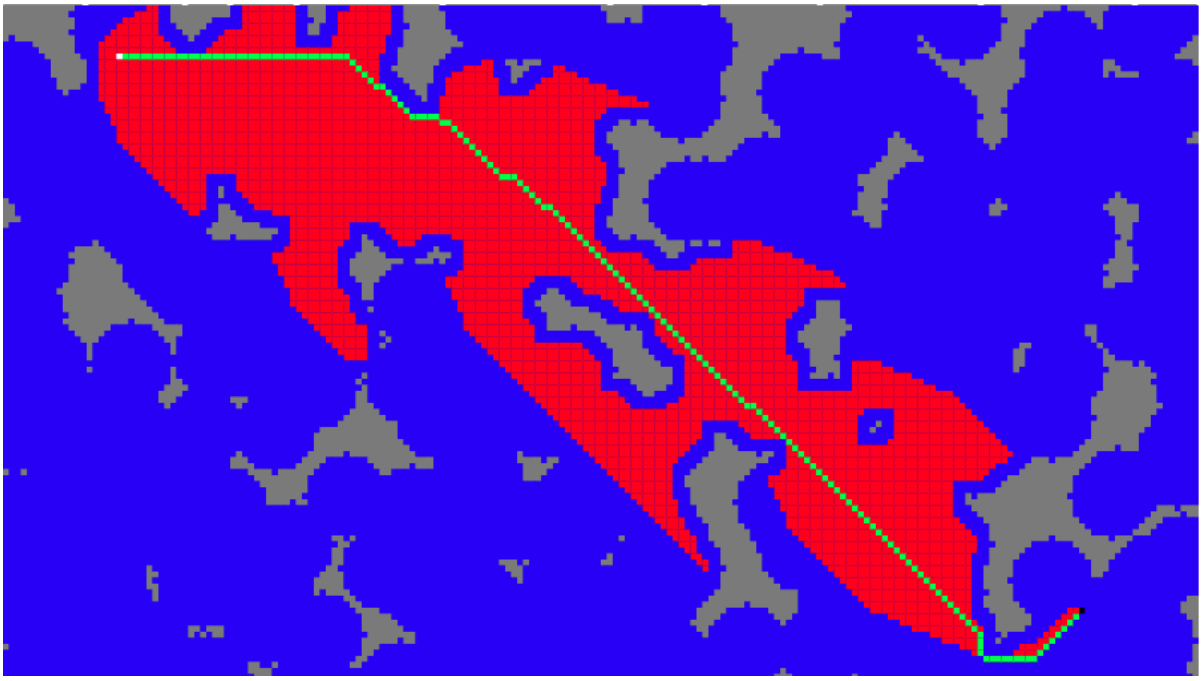
Searched nodes: Totala antalet noder som har sökts.

Runtime: totala tiden för sökningen.

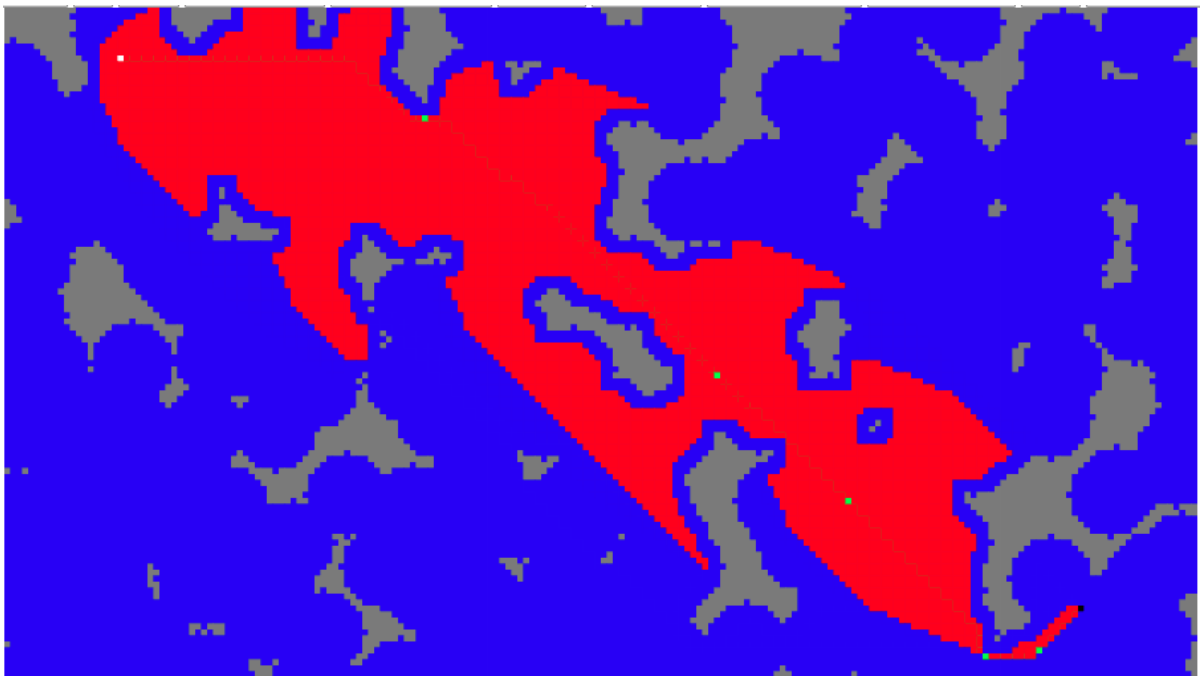
Nodes in basepath: Totala antalet noder i rutten.

Alla parametrarna skall vara så låga som möjligt.

Normal rutt jämfört med smoothen.



Figur 2.8, Exempel på rutt utan efterbehandling.



Figur 2.9, Exempel på rutt med efterbehandling.

För att lättare kunna skapa ett givande resultat så har parametrarna olika stor betydelse, där högst i listan väger tyngst och sist väger minst. Inga numeriska värden är kopplade till detta då det är svårt att veta precis hur betydelsefulla olika parametrar är. Den existerande rankingen har grundats på en sammansättning av kunskap som har erhållits under projektet.

5. Resultat

5.1 Framtagning

För att ge alla algoritmer en chans att visa sin starka sida så gjordes tester på 15 slumpmässigt genererade kartor med alla algoritmer. På vissa av dessa 15 kartor så gjordes kortare sökningar, andra hade mer öppet vatten och vissa dominerades av land. Detta gjordes för att skapa en så sann bild som möjligt över hur algoritmerna presterar i olika scenarion.

5.2 Resultaten

Basepath near land instances: Många av algoritmerna har fått samma resultat och endast gått nära land på karta 9. Det var oundvikligt att inte går nära land på denna karta och 4 steg var det bästa som gick att få. Resultaten är som förväntat då Best first finder samt Jump point är mycket aggressiva algoritmer som nästan alltid kommer att gå rakt mot målet och helt ignorera andra vikter.

Basepath near land instances	A*	A* bi	best first	best first bi	dijkstras	dijkstras bi	jump point	Theta*
karta 1	0	0	12	24	0	0	47	0
karta 2	0	0	16	27	0	0	50	0
karta 3	0	0	12	26	0	0	68	0
karta 4	0	0	17	22	0	0	60	0
karta 5	0	0	11	14	0	0	40	0
karta 6	0	0	14	27	0	0	45	0
karta 7	0	0	11	19	0	0	67	0
karta 8	0	0	12	26	0	0	38	0
karta 9	4	4	17	26	4	4	91	1
karta 10	0	0	14	20	0	0	36	0
karta 11	0	0	13	26	0	0	64	0
karta 12	0	0	12	12	0	0	75	0
karta 13	0	0	12	25	0	0	70	0
karta 14	0	0	13	26	0	0	52	0
karta 15	0	0	12	26	0	0	37	0
AVG	0,266667	0,266667	13,2	23,06666667	0,266667	0,26666667	56	0,066667

Smoothpath near land instances: Lite mer sprida resultat men det är fortfarande ett flertal algoritmer som har lyckats undvika hinder helt på alla kartor förutom karta 9. Vägen som A* bi och Dijikstra bi har valt att ta ser ut att lika många noder nära land som vanliga A* och Dijikstra men faller lite här när rutten jämnas ut. Alla resultat där smoothpath har varit med måste tänkas över lite extra då detta är en efterbehandling och inte hur algoritmen faktiskt skapade rutten. Det är därför i dessa fall extra viktigt att väga in hur rutten såg ut på kartan för att få mer kontext till denna data.

Smoothpath near land instances	A*	A* bi	best first	best first bi	dijkstras	dijkstras bi	jump point	Theta*
karta 1	0	0	13	29	0	0	47	23
karta 2	0	0	19	29	0	0	52	54
karta 3	0	0	5	19	0	0	73	48
karta 4	0	0	22	24	0	0	49	45
karta 5	0	0	7	18	0	0	17	22
karta 6	0	0	14	24	0	0	39	42
karta 7	0	0	6	23	0	0	52	42
karta 8	0	0	12	30	0	0	36	40
karta 9	5	10	22	26	5	10	102	50
karta 10	0	0	14	22	0	0	28	17
karta 11	0	0	10	36	0	0	54	29
karta 12	0	0	33	33	0	0	93	29
karta 13	0	0	18	18	0	0	61	57
karta 14	0	0	23	23	0	0	53	32
karta 15	0	0	11	27	0	0	50	59
AVG	0,333333	0,666667	15,26667	25,4	0,333333	0,66666667	53,7333333	39,26667

Smoothpath not ok turns: Ett allt som allt mycket jämnare resultat men Jump point och Theta* ligger snäppet före. Även här måste resultaten tas kritiskt och bilden av rutten i åtanke då detta är en egenskriven räknare som inte tar hänsyn till hur de olika algoritmerna expanderar.

Smoothpath not ok turns	A*	A* bi	best first	best first bi	dijkstras	dijkstras bi	jump point	Theta*
karta 1	1	1	4	7	1	1	0	1
karta 2	1	1	2	2	1	2	0	2
karta 3	1	0	6	2	1	0	1	2
karta 4	3	3	10	4	4	4	1	1
karta 5	0	0	2	2	0	0	0	0
karta 6	2	1	3	1	2	1	2	1
karta 7	1	1	8	7	1	1	0	1
karta 8	2	2	3	3	2	1	0	0
karta 9	4	3	5	4	3	5	4	2
karta 10	1	1	0	0	0	0	0	0
karta 11	1	1	5	2	2	2	1	0
karta 12	2	3	6	5	3	3	5	4
karta 13	5	3	3	3	5	2	3	2
karta 14	2	2	0	0	2	2	1	2
karta 15	4	4	7	7	3	3	4	1
AVG	2	1,733333	4,266667	3,266666667	2	1,8	1,46666667	1,266667

Smoothpath length: Även här är algoritmerna jämlika och det är inget som står ut allt för mycket, vilket är förväntat då alla algoritmer jobbar mot att hitta den snabbaste vägen. Jump point står ut lite med några enstaka steg kortare rutt men har som det sågs tidigare det absolut sämsta resultatet när det gäller att styra undan land. Återigen så är det viktigt att studera kartorna för att se ifall smoothpath funktionen på något sätt gjorde dessa resultat mindre pålitliga.

Smoothpath length	A*	A* bi	best first	best first bi	dijkstras	dijkstras bi	jump point	Theta*
karta 1	201,24	200,71	214,26	217,73	201,23	200,71	202,93	195,74
karta 2	253,14	249,93	218,36	216,07	253,68	251,18	206,21	235,01
karta 3	220,24	219,21	262,95	239,39	220,99	220,05	215,03	217,03
karta 4	231,68	232,45	243,51	202,82	232,44	232	201,44	214,19
karta 5	173,36	171,08	175,64	184,38	176,51	175	175,27	213,61
karta 6	217,08	216,37	191,89	185,28	217,84	216,3	185,73	213,61
karta 7	279,91	280,81	292,52	291,57	280,83	280,15	276,34	275,34
karta 8	213,42	212,79	214,76	213,4	212,67	213,25	210,9	215,44
karta 9	292,03	291,06	304,93	290,28	292,58	291,87	279,79	282,7
karta 10	234,9	235,82	232,3	232,45	233,4	233,14	233,26	231,43
karta 11	264,66	263,85	271,29	261,32	264,2	265,16	258,27	257,53
karta 12	191,2	203,88	202,77	202,6	192,24	192,03	190,11	286,33
karta 13	200,94	218,37	226,88	215,83	200,28	196,77	168,56	190,97
karta 14	156,87	157,22	148,5	148,5	157,14	156,36	147,81	148,84
karta 15	265,41	264,79	330,71	272,41	266,15	266,92	231	255,52
AVG	226,4053	227,8893	235,418	224,9353333	226,812	226,059333	212,176667	228,886

Basepath not ok turns: Snarlika resultat men jump point är ännu en gång best med de två versionerna av Dijikstra tätt bakom.

Basepath not ok turns	A*	A* bi	best first	best first bi	dijkstras	dijkstras bi	jump point	Theta*
karta 1	1	1	5	9	0	1	1	1
karta 2	1	1	6	9	0	0	1	6
karta 3	1	0	7	8	0	0	0	4
karta 4	2	2	8	9	2	1	0	5
karta 5	0	0	5	3	0	0	0	1
karta 6	0	1	7	7	1	1	0	1
karta 7	2	1	5	6	1	1	0	4
karta 8	1	3	0	4	0	0	0	3
karta 9	0	1	4	9	0	0	0	4
karta 10	1	2	1	3	0	0	0	1
karta 11	3	1	4	5	0	0	0	1
karta 12	3	2	5	7	1	2	1	4
karta 13	1	3	8	10	1	1	2	4
karta 14	0	0	6	8	0	0	0	3
karta 15	3	2	8	7	1	0	0	3
AVG	1,266667	1,333333	5,266667	6,933333333	0,466667	0,46666667	0,33333333	2,857143

Basepath length: Som förväntat så speglas dessa resultat med Length smooth path.

Basepath length	A*	A* bi	best first	best first bi	dijkstras	dijkstras bi	jump point	Theta*
karta 1	207,49	209,73	230,7	241,09	207,49	207,49	208,17	195,74
karta 2	262,58	262,58	235,59	233,49	262,58	262,58	210,72	235,01
karta 3	229,88	229,88	294,77	275,66	229,88	229,88	223,15	217,03
karta 4	242,99	243,58	272,04	224,2	242,99	242,99	209,05	214,19
karta 5	182,85	184,5	190,89	191,82	182,85	182,85	182,85	169,74
karta 6	226,39	226,98	209,69	220,94	226,39	226,39	193,16	213,61
karta 7	289,3	289,3	312,55	312,45	289,3	289,3	289,3	275,34
karta 8	222,83	224,49	231,84	237,64	222,83	222,83	219,32	215,44
karta 9	302,3	304,06	330,14	320,14	302,3	302,88	288,16	282,7
karta 10	243,5	244,33	240,47	246,47	243,5	243,5	238,23	231,43
karta 11	273,09	273,67	290,14	286	273,09	273,09	266,64	257,53
karta 12	203,92	212,46	219,82	218,26	203,92	203,92	197,98	186,33
karta 13	206,43	229,65	249,04	243	206,43	206,43	180,78	190,97
karta 14	160,85	161,43	178,65	190,79	160,85	160,85	155,98	148,84
karta 15	272,01	273,67	350,3	288	272,01	272,01	234,4	216,7071
AVG	235,094	238,0207	255,7753	248,6633333	235,094	235,132667	219,859333	216,7071

Smoothpath ok turns: Här gör Theta* bra ifrån sig följt av best first bi. Men längre upp syns det att detta är till stor del pga att många av best firsts svängar klassas som “not ok” medan tex A* har sitt höga värde här. Theta* presterar dock bra på båda hållen och tar allt som allt inte många svängar.

Smoothpath ok turns	A*	A* bi	best first	best first bi	dijkstras	dijkstras bi	jump point	Theta*
karta 1	4	5	7	5	4	5	5	1
karta 2	10	9	9	8	10	8	9	5
karta 3	10	9	12	5	10	10	6	3
karta 4	12	12	8	6	12	10	5	5
karta 5	4	6	2	4	4	4	2	3
karta 6	8	8	8	6	10	9	3	4
karta 7	9	9	7	5	9	8	10	5
karta 8	7	7	4	2	5	6	4	5
karta 9	13	12	8	10	13	11	9	6
karta 10	3	3	6	3	4	5	3	2
karta 11	9	10	6	4	7	7	7	4
karta 12	10	10	8	8	8	9	4	2
karta 13	10	8	10	8	9	10	4	5
karta 14	8	6	6	6	8	7	5	2
karta 15	13	12	14	7	13	13	3	9
AVG	8,666667	8,4	7,666667	5,8	8,4	8,13333333	5,26666667	4,066667

Basepath ok turns: Det är egentligen bara jump point och möjligtvis Dijikstra som står ut lite med att inte använda så mycket svängar. Theta* resultat kan ignoreras här, det kommer att förklaras i diskussion varför.

Basepath ok turns	A*	A* bi	best first	best first bi	dijkstras	dijkstras bi	jump point	Theta*
karta 1	15	20	23	20	9	10	17	1
karta 2	28	30	23	20	18	20	11	-3
karta 3	35	38	39	24	19	26	17	-1
karta 4	39	38	29	22	29	34	15	-3
karta 5	30	40	11	18	10	10	11	1
karta 6	33	33	15	26	25	26	11	4
karta 7	28	21	33	28	21	20	17	-1
karta 8	19	31	23	24	9	15	9	2
karta 9	32	36	31	28	27	34	23	2
karta 10	24	23	11	20	17	20	7	0
karta 11	30	37	17	26	23	20	13	2
karta 12	25	27	39	38	27	26	23	2
karta 13	23	26	24	33	19	24	14	1
karta 14	21	16	23	26	15	18	15	0
karta 15	21	24	47	30	27	26	13	-1
AVG	26,86667	29,33333	25,86667	25,53333333	19,66667	21,9333333	14,4	0,4

Searched nodes: Även här presterar Jump point bäst med otroligt mycket färre noder sökta än resterande algoritmer. Det syns även att dijikstras är den sämsta av dom alla vilket var väntat pga dijikstras sätt att söka på.

Searched nodes	A*	A* bi	best first	best first bi	dijkstras	dijkstras bi	jump point	Theta*
karta 1	5125	1568	247	397	23581	17062	83	5869
karta 2	13168	2503	1488	532	27268	15405	324	13269
karta 3	9078	3200	910	1452	27206	14160	531	8914
karta 4	13386	5864	2008	676	26703	13491	312	12415
karta 5	504	740	166	320	20078	15775	40	855
karta 6	3190	2602	268	340	23000	16821	38	2388
karta 7	6199	6514	316	569	29695	21749	75	6472
karta 8	6307	1382	241	385	25935	18012	184	5943
karta 9	35457	6021	1114	511	35504	20769	82	32933
karta 10	4222	2368	223	262	27008	19547	62	3086
karta 11	6290	1348	1140	558	27911	18201	69	6517
karta 12	9547	8077	2700	4877	22244	12469	849	8995
karta 13	9921	5623	2345	2360	21536	14837	430	9332
karta 14	2471	2015	400	753	20906	12274	81	2421
karta 15	11602	4887	518	632	29189	18620	57	11946
AVG	9097,8	3647,467	938,9333	974,9333333	25850,93	16612,8	214,466667	8757

Runtime: Här är det Best first bi, som tidigare inte har presterat så bra, som gör bäst ifrån sig. Detta skulle kunna vara exempel på varför en allt för simpel sökprocess kan leda till att andra områden lider. Det syns också att bi-directional versionerna presterar lite bättre över lag.

runtime----------	A*	A* bi	best first	best first bi	dijkstras	dijkstras bi	jump point	Theta*
karta 1	28,53	9,73	1,94	3,33	31,57	35,05	7,95	71,82
karta 2	30,72	28,57	19	3,08	36,34	35,06	11,3	81,94
karta 3	30,07	25	3,15	3,63	37,98	34,52	12,6	47,37
karta 4	36,23	28,5	17,69	2,27	37,57	36,8	9,79	55,86
karta 5	1,34	2,21	0,63	1,71	33,67	38,79	4,58	5,16
karta 6	20,7	23,58	1,37	1,59	35,88	39,66	1,23	23,16
karta 7	25,48	26,9	2,5	2,56	43,98	55,62	6,49	39,23
karta 8	26,03	2,8	1,35	4,36	38,46	17,6	3,19	37,12
karta 9	72,15	8,94	16,52	1,71	125,94	58,75	6,33	145,91
karta 10	24,2	3,76	2,65	0,63	41,19	19,82	1,34	30,24
karta 11	27,9	2,73	2,85	1,21	55,06	19,16	5,91	46,24
karta 12	33,15	11,56	35,24	6,55	34,42	15,33	10,9	46,03
karta 13	20,99	9,78	1,6	1,79	26,83	16,54	10,76	51,23
karta 14	2,24	1,85	0,34	0,69	22,71	15,14	6,51	22,69
karta 15	26,29	8	1,38	3,13	34,18	20,91	1,29	60,97
AVG	27,068	12,92733	7,214	2,549333333	42,38533	30,5833333	6,678	50,998

Nodes in basepath: Även här står Theta* ut väldigt mycket vilket kommer förklaras i diskussionen. För resterande algoritmer så är resultaten som förväntat som speglar endast hur lång rutten är.

Nodes in basepath	A*	A* bi	best first	best first bi	dijkstras	dijkstras bi	jump point	Theta*
karta 1	165	166	182	187	165	165	169	4
karta 2	235	235	23	191	235	235	179	9
karta 3	211	211	244	234	211	211	208	8
karta 4	215	216	225	190	215	215	191	8
karta 5	159	159	166	163	159	159	159	5
karta 6	210	211	173	171	210	210	166	8
karta 7	229	229	239	236	229	229	229	8
karta 8	182	182	179	179	182	182	176	5
karta 9	242	245	257	247	242	243	232	10
karta 10	189	189	181	187	189	189	180	4
karta 11	219	220	217	217	219	219	208	6
karta 12	178	194	191	194	178	178	182	8
karta 13	183	200	202	203	183	183	159	10
karta 14	137	138	149	157	137	137	140	6
karta 15	220	220	290	219	220	220	177	12
AVG	198,2667	201	194,5333	198,3333333	198,2667	198,333333	183,666667	7,4

5.3 Sammanfattning

Sammanfattning av alla genomsnittsresultat över dom 15 kartorna, avrundade till två decimaler.

	A*	A* bi	best first	best first bi	dijkstras	dijkstras bi	jump point	Theta*
Basepath near land instances	0,27	0,27	13,20	23,07	0,27	0,27	56,00	0,07
Smoothpath near land instances	0,33	0,67	15,27	25,40	0,33	0,67	53,73	39,27
Smoothpath not ok turns	2,00	1,73	4,27	3,27	2,00	1,80	1,47	1,27
Smoothpath length	226,41	227,89	235,42	224,94	226,81	226,06	212,18	228,89
Basepath not ok turns	1,27	1,33	5,27	6,93	0,47	0,47	0,33	2,86
Basepath length	235,09	238,02	255,78	248,66	235,09	235,13	219,86	216,71
Smoothpath ok turns	8,67	8,40	7,67	5,80	8,40	8,13	5,27	4,07
Basepath ok turns	26,87	29,33	25,87	25,53	19,67	21,93	14,40	0,40
Searched nodes	9097,80	3647,47	938,93	974,93	25850,93	16612,80	214,47	8757,00
Runtime	27,07	12,93	7,21	2,55	42,39	30,58	6,68	51,00
Nodes in basepath	198,27	201,00	194,53	198,33	198,27	198,33	183,67	7,40

6. Diskussion

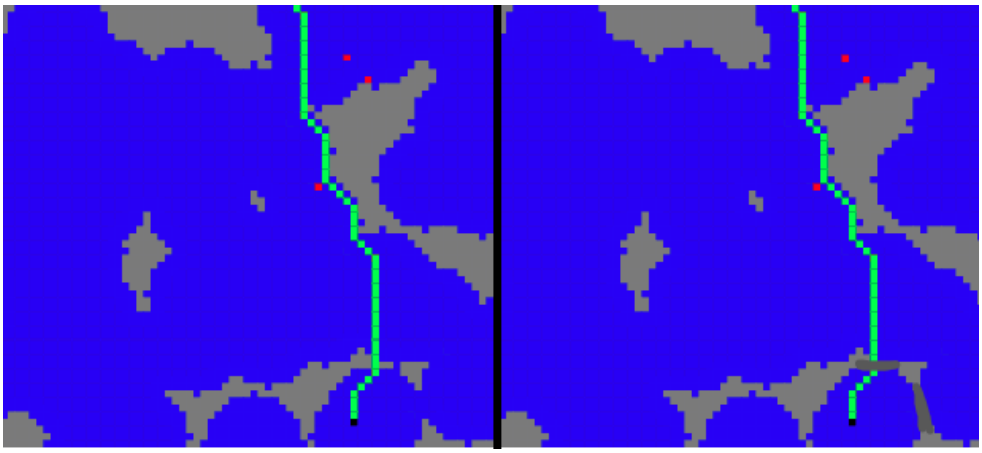
Utfallen av analysen är inte oväntade då rent teoretiskt sett är vissa av dessa algoritmer mer lämpade för denna typ av uppgift. Med detta sagt så finns det fortfarande ingen klar vinnare som den bästa algoritmen och det finns mycket diskutera då många parametrar spelar in i resultaten.

6.1 Förklaring av algoritmernas beteende

I tidigare kapitel gick rapporten igenom hur de olika algoritmerna var strukturellt uppbyggda men det har inte nämnts mycket om hur detta reflekteras i hur algoritmerna beter sig när de söker. Detta är väldigt viktigt för att förstå hur resultatet skall tydas och varför de ser ut som de gör.

A* och Dijikstras är båda traditionella algoritmer och baserat på hur denna undersökning görs samt vad som undersöks så har de lite av ett försprång för att prestera bra. Därför är det viktigt att ha kontexten på hur de resterande algoritmerna beter sig.

Jump point search är som sagt en vidareutveckling av A* med en potential för att prestera bättre än den, och borde på papper göra det också. Så varför ligger den så dåligt till på många av resultaten? Det som gör Jump point till en vidareutveckling från A* är att den bättre kan se mönster i kartor och därmed hoppa mer än ett steg i taget för att göra långa sträckor ett stort hopp istället för många små. Detta låter självklart mycket bra, men avvägningen här blir att den gör ett antagande att alla noder har samma kostnad. Land är fortfarande land men noderna runt om land som har en högre kostnad och ses som “Basepath near land instances” ser algoritmen som vilken annan vanlig nod som helst vilket gör att den inte kommer undvika det “grunda” vattnet runt om land och därför aldrig kunna prestera bra på dessa punkter. Men detta gör istället att den presterar bättre på antalet svängar den tvingas ta, antalet sökta noder samt även längden på rutten, som blir den absolut kortaste som går att få. Hade algoritmerna skrivits från grunden istället för att utnyttja ett existerande bibliotek hade det kanske vart möjligt att skriva om Jump point search så att den istället såg alla noder med en ”vikt > 1” som land. Detta hade i sin tur skapat nya problem som i fallet nedanför där det inte längre hade funnits en möjlig rutt. Detta hade blivit en övervägning i hur man vill att algoritmen fungerar.



Figur 3.0, Förklaring av problem med ”vikt>1 == Land”.

Best First Search har också problem med att hålla sig undan land men detta är beroende på en mycket aggressiv heuristik i form av en multiplikator på $h(n)$. Den presterar dock inte lika dåligt som Jump point när det kommer till att gå nära land. Detta är för att den fortfarande tar hänsyn till individuell viktning på noderna, men inte lika mycket som tex A*. Det syns dock att förutom just att undvika land så presterar Best first genomsnittligen sämre än Jump Point på allt. Detta är pga att den, till skillnad från Jump Point, inte är smartare än sin föregångare A* utan bara aggressivare. Den kommer i många sammanhang ta väldigt långa omvägar till målet eftersom balansen i $F(g,h)$ nu är bruten då h väger mycket tyngre än g. Med andra ord så bryr sig Best First inte längre om att hitta den optimala rutten från A till B utan endast att den snabbt hittar någon rutt från A till B och kommer därmed sällan avvika från sitt första försök.

Theta* är en mycket effektiv algoritm som har hamnat i ett lite orättvist test och hade förmodligen presterat mycket bättre under andra omständigheter. Anledningen till de oväntade resultaten som Theta* ibland producerade beror på att den till skillnad från resterande algoritmer ignorerar rutnätet som kartan bygger på. Som nämnts tidigare placerar inte Theta* en ny nod för varje steg den tar utan endast när det inte längre kommer finnas siktlinje från den senast expanderande noden till den nya. Eftersom Theta* tar hänsyn till nodernas viktning kommer den aldrig att placera en ny expansion nära land. Men siktlinje funktionen bryr sig inte om vilken vikt noderna som korsas har utan endast att de inte är land. Detta leder till noderna som Theta* placerar inte ligger nära land men rutten som skapas kommer ofta att gå nära land ändå. Detta gör att “Smooth path” funktionen inte kommer att göra någon visuell skillnad på Theta* då den redan har optimerats på detta sätt. Men kommer att göra skillnad på resultaten då den efter optimeringen drar ett streck mellan alla noder för att se hur många av dessa noder som ligger nära land. Detta förklarar den stora skillnaden mellan “Basepath near land instances” och “Smoothpath near land instances”. Eftersom Theta* inte hoppar från en nod till en av dess grannar utan kan ta stora kliv kommer ok turns inte fungera då den kräver att noderna ligger bredvid varandra. Det undersöktes ifall det var möjligt att skiva om Theta* så att dess siktlinjefunktion såg ”vikt > 1” som land. Men det var inte vart möjligt att testa detta utan att skriva om hela algoritmen från grunden och göra en djupdykning i hur Theta* fungerar och detta fanns de inte tid för.

Bi directional searching påverkar inte resultaten allt för mycket och behöver inte någon större förklaring. Det kan vara bra att inse att detta inte är samma algoritm som söker från två håll utan att det är två separata algoritmer. Den ena söker från end till start och den andra från start till end, sökningen är klar när de möts. Eftersom detta blir två separata scenarion kan det inte längre garanteras en optimal rutt. För att kunna garantera det krävs att sökvägen eller miljön ser likadan ut för de båda hållen. Allt detta resulterar i marginellt sämre resultat än samma algoritm utan Bi förutom i “Runtime” och “Searched nodes” vilket är helt förväntat.

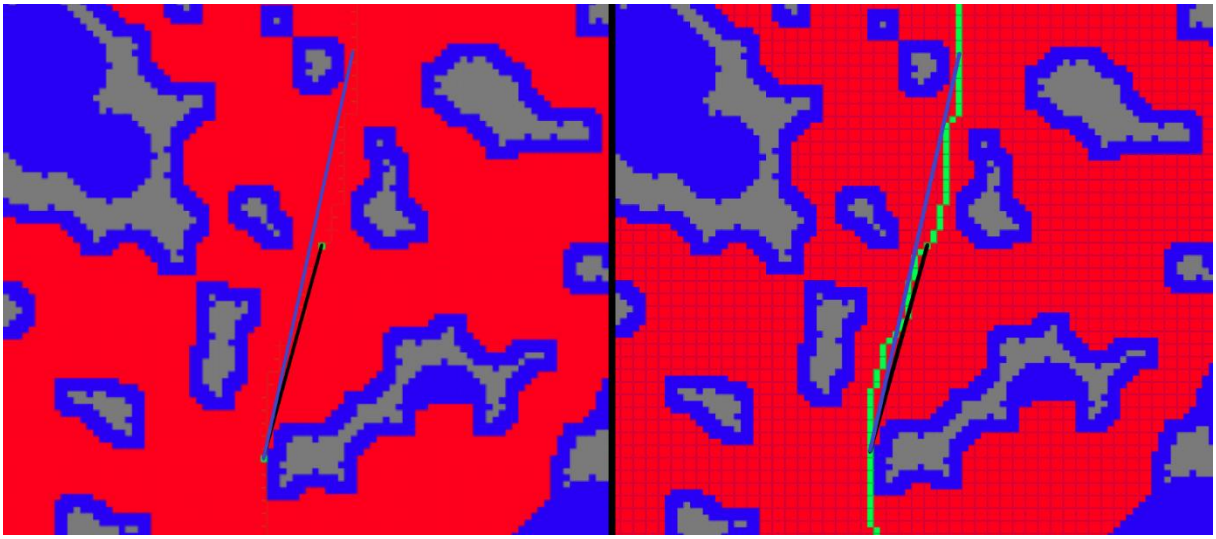
6.2 Introduktion till problem med studien

6.2.1 Smooth path

Det finns ett flertal förbättringsområden i detta projekt, funktionen “smooth path” ligger till grund för en stor del av dessa.

Något som till en början var en osäkerhet ifall det skulle utnyttjas var funktionen smooth path då den inte var helt rättvis mot alla algoritmer. Hur funktionen fungerar är mycket likt Theta* fast på en existerande rutt. Den börjar med att se hur många noder fram vi kan hoppa på rutten utan att förlora siktlinje från nuvarande position. När siktlinjen bryts så hoppar den till noden

innan och tar bort allt mellan dessa två punkter. Den börjar sedan om processen från den nya noden, detta upprepas tills dess att funktionen når mål. Detta resulterar i en mycket renare och realistisk rutt där endast de nödvändiga noderna visas. Men redan här uppstår ett problem, eftersom den inte letar igenom hela rutten innan den placerar sin första vägpunkt utan bara vid första brytningen så vet den inte ifall det finns noder längre fram som hade haft siktlinje.



Figur 3.1, Förklaring av problem med siktlinje.

Detta resulterar i att "Smooth path" ibland placerar onödiga vägpunkter, detta sågs inte som något stort problem då det är något som påverkar alla algoritmer lika mycket.

Ett större problem uppstår med att denna funktion inte var gjord för att hantera olika kostnad på noderna utan letade endast efter siktlinje mot land. Detta ledde till att samma problem som Theta* har uppstod när funktionen kördes på kartor med projektets viktning.

Eftersom det var viktigt att se hur algoritmerna klarade denna kostnad så skrevs funktionen om till att tappa siktlinje även på noder med vikt >1. Ett stort problem här var att mycket av koden för "Smooth path" var sammanvävd med annan kod och det var alldeles för tidskrävande att fundamentalt skriva om funktionen för vårt syfte vilket gjorde att funktionen inte är perfekt.

Lösningen fungerade bra men hade sina svagheter, som tex när en algoritm tar flera steg nära land så kommer funktionen att placera många onödiga vägpunkter eftersom den alltid ser vikt >1 i siktlinje. Detta var något som var speciellt synligt på best first och jump point som båda tar många steg nära land. Eftersom nästan alla algoritmerna beter sig olika kommer detta att påverka olika mycket på deras resultat vilket inte är rättvist och resultatet reflekterar snarare hur kompatibel "Smooth path" är med en viss algoritm.

Trots detta valdes det att använda funktionen då den i de flesta fall fungerar som den skall, men också för att så länge utvärderandet görs med dessa problem i åtanke så kommer det inte att påverkade resultatet för algoritmerna allt för negativt. Om denna studie skulle gjorts om så "Smooth path" definitivt ett av de områdena som skulle setts över, antingen för att skriva om algoritmen ännu mer för att bättre hantera alla scenarion eller för att helt uteslutas.

6.2.2 För simpel undersökning?

Ytterligare något som i våra ögon hindrar undersökning att nå de mest verklighetstrogna resultaten är komplexiteten av testmiljön. Något som undersökningen till en början hade mycket större förväntningar på men som det snabbt insågs skulle vara alldeles för komplicerat var en mer omfattande viktning. Till en början fanns förhoppningar om att skapa en testmiljö som var mycket mer verklighetstrogen med vattendjup, vind, grund, farleder mm men det insågs efter en del tids investering att detta skulle bli ett helt arbete i sig och att det inte riktigt vara syftet för denna undersökning. I en perfekt värld hade dessa tester gjorts på riktiga sjökort. Viktningen som för tillfället används är visserligen inte dålig men en väldigt förenklad version av vad en algoritm hade behövt manövrera om den hade använts som en ruttplanerare.

Vidare med testmiljön så skapas det också en mindre rättvis miljö att utvärdera dessa algoritmer med när funktionen som tar fram resultaten är något som har skapats i efterhand. Detta gör att utöver att testmiljön samt ”Smooth path” i sig förmodligen är bias mot vissa algoritmer så är det omöjligt att objektivt ta fram data eftersom det inte är någon som är inbakat i algoritmernas kod utan en extern funktion som i efterhand har analyserat vad algoritmen har gjort och därmed något där jag var tvungen att bestämma hur den skulle agera.

6.2.3 För litet urval?

Något som förmodligen skulle kunnat förbättras med denna undersökning är spektret av kartor. Visserligen har alla kartor en unik miljö men en större mängd test på kartor av olika storlekar är något som definitivt skulle gjort gett ett mer ärligt resultat. Som vi ser så ligger vissa algoritmers styrka i hur snabba dom är medan andra har sin styrka i pålitlighet eller effektivitet. Tester på större kartor hade gett en bättre representation av algoritmernas styrkor. Anledning till att detta inte gjordes var till viss del tiden det tog att spara undan data för varje test men framför allt att svårigheterna som kom med att miljögenereringen inte var modulär och krävde justering så fort kartan blev större eller mindre.

6.2.4 Allmänna förbättringar, vidare undersökningar?

Skulle detta arbete göras om hade det första på listan av saker att göra bättre varit att skriva all kod från grunden. Otroliga mängder tid gick åt att leta i kod som någon annan hade skrivit och skriva om den för att uppnå funktionaliteten som eftersöktes. Att skriva allt själv hade självklart varit ännu mer tidskrävande men hade resulterat i algoritmer och funktioner som var helt anpassade för detta arbete. Mer tid hade också lagts på att se över ifall det finns något sätt att arbeta runt Theta* siktlinjeproblem samt Jump point oförmåga att arbeta med icke uniform kostnad.

Vidare undersökningar på detta skulle vara hur man fysiskt skulle implementera en ruttplanerare med autopilot i en båt, är dagens algoritmer smarta nog?

6.2.5 Säkerheten hos ett sådant system

Något som är viktigt att tänka på om man försöker automatisera marin ruttplanering är att bara för att det fungerar på land så finns det inga garantier att det kommer fungera lika bra på vattnet. När google maps lägger ut en rutt från punkt A till punkt B så har den ett väldigt strikt regelverk som den kan förhålla sig till. Detta existerar inte till sjöss, ett sjökort innehåller

visserligen mycket information men inte tillräckligt mycket för att garantera att en planerad rutt inte skulle krocka med någon annan. Det finns inga vägar för algoritmen att förhålla sig till, farleder hjälper men de är inte lika strukturerade som vägtrafik. En båt har varken manöverenheten eller bromssträckan hos en bil och havsbotten är i ständig rörelse. Grund kommer och går med tidvattnet och med åren. Det finns i nuläget marina GPSer som stödjer denna typ av ruttplanering men även dessa behöver en översikt innan man kan lita på den planerade rutten. För att en marin ruttplanerare skall kunna bli helt självgående så behövs det mer information på sjökort och mer regler för fartyg och båtar.

7. Slutsats

Mycket av problemen som har uppstått under projektets gång har varit relaterade till att koden inte var anpassad för det projektet ville utnyttja den för. Hade projektet kunnat göras om med mer tid och kunskap så skulle “egen kod” vara högst på listan av saker att ta med.

Men på det stora hela ses denna undersökning som lyckad men något som ska studeras kritiskt. Man ser också att nästa alla algoritmerna har sina områden där de skulle vara det optimala valet. Men när det kommer till just ruttplanering så skulle A* eller Jump Point i våra ögon passa bäst.

A* presterade alltid bra på alla punkter och om man skulle skriva en mer komplicerad ruttplanerare med fler parametrar i sin viktning, så som väder och vind skulle A* vara perfekt.

Bryr man sig inte om viktning utan vill endast ha en optimal, snabb och effektiv ruttplanerare så skulle det bästa valen vara jump point som på just dessa punkter är en rak uppgradering från A*.

Det visar sig också att prestandan av algoritmerna till stor det är knuten till hur detaljerade testmiljön som dom söker i är. Detta betyder att för att få effektivare autonoma marina ruttplanerare så borde fokus ligga på mer detaljerade sjökort.

8. Referenser

[1] Navionics. (2021). Hemsida för Navionics Dock-to-Dock tjänst. https://www.navionics.com/fin/charts/features/dock-to-dock-autorouting [Acc 2021-03-15].

[2] Github, (2017). bibliotek för sökalgoritmer. https://github.com/qiao/PathFinding.js/ [Acc 2019-09-02]

[3] Technopedia Inc. (2021). Technopedia explains search algorithm. https://www.techopedia.com/definition/21975/search-algorithm [Acc 2019-09-04]

[4] W3schools. (2021). JavaScript for loop. https://www.w3schools.com/js/js_loop_for.asp [Acc 2019-09-04]

[5] MDN Web Docs. (2021). Array description. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array [Acc 2019-09-04]

[6] Peter E.Hart, Nils J. Nilsson, Bertram Raphael. (1968). A Formal Basis for the Heuristic determination of Minimum Cost Paths. https://www.cs.auckland.ac.nz/courses/compsci709s2c/resources/Mike.d/astarNilsson.pdf [Acc 2019-09-15]

[7] E.W. Dijikstra. (1959). A Note on Two Problems in Connexion with Graphs. http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf [Acc 2019-09-16]

[8] Witold Kazimierski, Natalia Wawrzyniak. (2016). https://www.researchgate.net/publication/281389664_Analysis_of_Graph_Searching_Algorithms_for_Route_Planning_in_Inland_Navigation [Acc 2019-09-16]

[9] Peter Sanders, Dominik Schultes. (). Engineering Fast Route Planning Algorithms* http://algo2.iti.kit.edu/schultes/hwy/weaOverview.pdf [Acc 2019-09-19]

[10] Helong Wang. (2018). Voyage optimization algorithms for ship safety and energy-efficiency https://research.chalmers.se/publication/503070/file/503070_Fulltext.pdf [Acc 2019-09-26]

[11] Angelica Andersson. (2015) Multi-objective optimisation of ship routes. http://publications.lib.chalmers.se/records/fulltext/218341/218341.pdf [Acc 2019-09-26]

[12] Myung-Il Roh. (2013). Determination of an economical shipping route considering the effects of sea state for lower fuel consumption. https://www.sciencedirect.com/science/article/pii/S2092678216303958 [Acc 2019-10-15]

[13] Laura Walther, Anisa Rizvanolli, Mareike Wendebourg, Carlos Jahn. (2016). Modeling and Optimization Algorithms in Ship Weather Routing. https://www.sciencedirect.com/science/article/pii/S2405535216300043 [Acc 2019-10-17]