

Korteste veje

i Europa Universalis 4

Daniel Bakhtiari
vjm788

Bruce Isiah Thomas Esplago
zwq140

Jesper Bull Holm Larsen
bsx137

Bachelorprojekt
Datalogi



KØBENHAVNS
UNIVERSITET

8. april 2024

Abstract

Europa Universalis IV (EU4) is one of many video games that make use of shortest path algorithms. In the case of EU4, the shortest path algorithm is used on a dynamic graph. In this project, we compare the performance of several shortest path algorithms and their ability to find the shortest path between two nodes on EU4's dynamic graph. In our experiments we have based our evaluations on the following parameters for each algorithm: days it took to complete a path, number of vertices searched, space and time measurements, nodes in the final path. When doing the experiments, we made a discovery regarding the parameter "days it took to complete a path", suggesting potential for finding better suited algorithms, that seek to minimize this parameter. The results show that the algorithms Bidirectional Dijkstra and Bidirectional A*, are the preferable choices of algorithms in EU4, when evaluating on the aforementioned parameters. Furthermore, we suggest using a "hybrid" algorithm, based on using specific algorithms for specific scenarios. These results however are based on randomness, as the movement of enemies can be unpredictable and therefore give misleading results.

Indhold

1	Introduktion	3
2	Undersøgelse	3
2.1	Dijkstra	3
2.2	Bellman-Ford	5
2.3	A* søge algoritme	6
2.4	Tovejs søgealgoritme	8
2.5	Floyd-Warshall	9
2.6	Johnsons algoritme	10
2.7	Mahadeokar-Saxena erstatningseveje algoritme	10
2.7.1	Tilfældet hvor $x \in M_i$	13
2.7.2	Tilfældet hvor $x \in U_i$	13
3	Korteste veje i EU4	14
3.1	Definition på den mest egnede algoritme	14
3.2	Grafens karaktertræk	15
4	Valg af algoritmer til implementation	17
5	Implementation	19
5.1	Simulering af spillet	19
5.2	Forarbejdelse af spildata	20
5.3	Visualisering af algoritmernes kørsel	21
5.4	Spiller logik	22
5.5	Fjende logik	22
5.6	Bemærkninger til implementerede algoritmer	23
6	Eksperiment	24
7	Resultater og analyse	25
7.1	Forskel mellem spring i fjende frekvens	26
7.2	Teoretiske grænser	29
8	Konklusion	32
8.1	Eventuelle forbedringer	33
A	Bilag	37
A.1	Resultater	37

1 Introduktion

I dette projekt undersøger vi korteste veje i spillet Europa Universalis IV (EU4) og sammenligner udvalgte algoritmers tilegnelse for EU4. For at kunne opdage om en algoritme egner sig til EU4, ser vi på køretid, pladsforbrug, prisen for vejen og antal knuder besøgt. Samtidig har vi valgt at undersøge korteste veje i EU4 grundet brugen af et dynamisk vejnetværk. Vejnetværket består af en række provinser, der kan forestille sig at være knuder i en graf. Grafen er dynamisk i den forstand, at knuder kan blive utilgængelige, hvis den pågældende provins er optaget af fjenden. Imellem provinserne findes kanterne, som har en vægt associeret med dem. Vægten er bestemt ud fra spillets tærren. Korteste veje problemet omhandler at finde korteste vej mellem to knuder. Det udspiller sig som reelt i en graf hvor der er flere knuder med kanter imellem dem. Målet er så at finde en vej fra A til B hvor summen af vejen er minimeret, dette kaldes for korteste vej. Vi vil løse dette problem ved at implementere diverse algoritmer, som vi vil bruge til at simulere en rute på i EU4 og derefter finde den korteste vej fra A til B.

2 Undersøgelse

Første del af projektet indeholder en undersøgelse og redegørelse af, hvilke algoritmer der findes (blandt andre) med formålet at finde korteste veje i en graf. Emner som tidskompleksitet, pladskompleksitet, grafens opbygning og eventuelle værktøjer som algoritmerne gør brug af, vil blive berørt.

2.1 Dijkstra

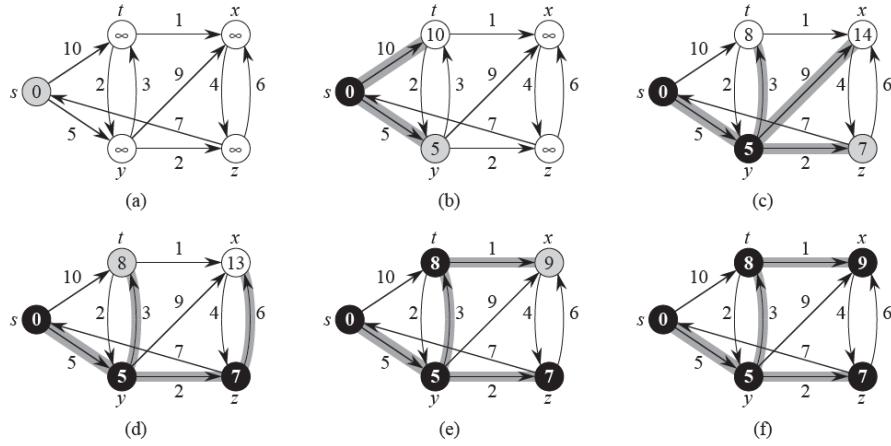
Den oprindelige Dijkstra algoritme finder alle korteste veje fra en given kilde- eller startknude til alle andre knuder i en graf [3]. Algoritmen er af typen enkelt-kilde (eng: single-source) og benytter sig af kanternes vægt, til at beregne de korteste veje. Det er et krav for algoritmen, at alle disse vægte er positive reelle tal. Med andre ord kan algoritmen ikke anvendes på grafinstanser med negative vægte. Et andet krav er, at grafen skal være orienteret. Lad os antage at vi har en graf G som består af knuderne V (eng: vertices) og kanterne E (eng: edges).

$$G = (V, E)$$

Algoritmen udregner iterativt den sammenlagte vægt u fra startknuden til alle andre knuder, og gemmer denne vægt i en liste S på det indeks, som den pårørende knude har i listen af knuderne V . I eksemplet fra bogen [2] betnytter algoritmen sig af en minimum-prioritets-kø Q . Under algoritmens initialisering fyldes alle V knuder ind i Q , hver især med en attribut der fortæller noget om vægten. Startknudens attribut sættes til 0, sådan at algoritmen starter ved den. Alle de resterende knuders attribut sættes til uendelig. Efter algoritmen har gemt vægten i S udføres *relaxation*-teknikken på de knuder, som er naboyer til den knude, hvis vægt lige er blevet gemt. Relaxation-teknikken går kort fortalt

ud på, at tjekke om vi har fundet et bedre estimat (vægt) end hvad vi hidtil kendte. Dog er algoritmen stadig det vi kalder *grådig*, idet at den lægger u over i S for hver iteration. Der ses kun på den billigste vægt algoritmen kender til, i den enkelte iteration.

Ved algoritmens terminering kan de sammenlagte vægte fra alle knuder til startknuden findes i S . Et opslag kunne være se på indeks 53 i S , som eksempelvis ville have vægten 36. Det ville betyde at den korteste, men også ”billigste” vej, som algoritmen fandt fra startknuden til knuden med indeks 53, ville være 36.



Figur 1: Eksempel på gennemkørsel af Dijkstra's algoritme [2]

I Figur 1 illustreres en gennemkørsel af algoritmen. Herunder ses en forklaring af de 6 skridt:

- I (a) skridtet bliver knuden s (startknuden) fjernet fra prioritets-køen Q , da den har den mindste værdi af alle knuderne. Vi farver s grå af samme årsag. Bemærk alle andre knuder end s har værdien uendelig.
- I (b) skridtet markerer vi s som værende besøgt, og tilføjer den derfor til S listen. Besøgte knuder farver vi sorte i dette eksempel. Ud fra s kan vi enten gå til t eller y , hvorfor vi opdaterer deres værdier fra uendelig til henholdsvis 10 for t og 5 for y . Den fede grå streg markerer den hidtil korteste vej til t og y . Vi farver knuden y grå, og den fjernes fra prioritets-køen Q , da den har lavest værdi.
- I (c) skridtet markerer vi t som værende besøgt, og tilføjer den derfor til S listen. Algoritmen opdager at der findes en kant fra y til t og at denne har lavere vægt end den hidtil kendte vej. Derfor bliver den fede grå streg sat på denne kant, sådan at den hidtil korteste vej fra s til t er $s \rightarrow y \rightarrow t$. Af samme årsag opdateres værdien på knuden t til 8 i stedet

for 10. Algoritmen opdager også en vej til knuden x og opdaterer derfor dens værdi til 14 (korteste kendte på nuværende tidspunkt). Det samme sker for knuden z , hvor værdien sættes til 7. Der tegnes henholdsvis grå fede streger til knuderne. Knuden z fjernes fra Q , da den har den mindste værdi af de ubesøgte knuder.

- I (d) skridtet besøger vi z , markerer den med sort og tilføjer den til S . Herefter opdages der en kortere vej til x : nemlig $s \rightarrow y \rightarrow z \rightarrow x$ hvor prisen er 13. Vi opdaterer derfor x værdien til 13. Algoritmen ser på ubesøgte knuder, og vælger at markere t grå, da den har den laveste værdi.
- I (e) skridtet markerer vi t med sort og opdager nu en endnu kortere vej til x gennem $s \rightarrow y \rightarrow t \rightarrow x$, hvor prisen er 9. Vi opdaterer x værdien til 9. De grå fede streger ændres af samme årsag. Vi markerer knuden x med grå, da den er ubesøgt og har lavest værdi.
- I (f) skridtet markerer vi x med sort og tilføjer den til S . Algoritmen opdager at prioritets-køen Q er tom, og terminerer derfor. Algoritmen har kørt færdig i dette skridt, og nu kendes alle korteste veje fra s til de andre knuder i grafen G .

Tidskompleksitet og pladskompleksitet

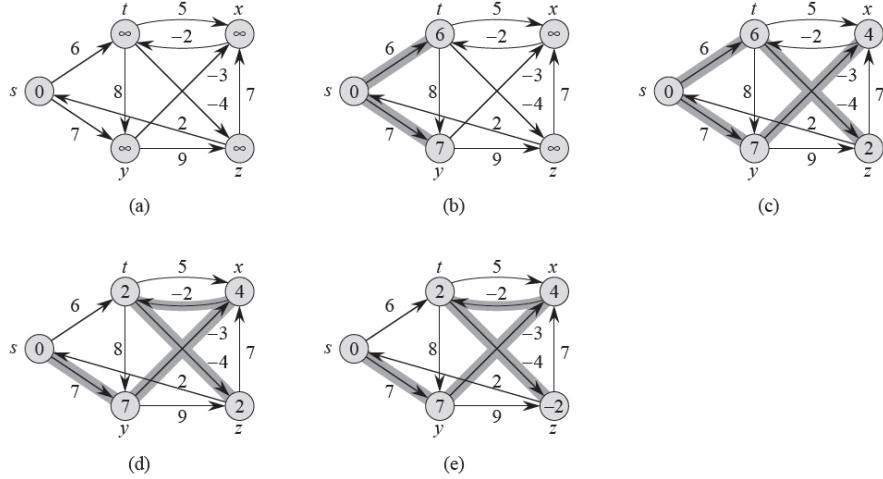
Ved brug af en minimum-prioritets-kø, som eksemplet i bogen [2] er Dijkstra's algoritme i væreste tilfælde øvre grænset af $O((|V| + |E|) \lg |V|)$. Hver gang vi udtrækker en knude med lavest værdi i Q er tidskompleksiteten øvre grænset af $O(\lg |V|)$. Antallet af gange vi skal gøre dette er $|V|$, siden vi skal udtrække alle knuder fra Q før algoritmen kan terminere. Derudover skal vi opdatere knudernes værdi, hvis vi finder en kortere rute end den hidtil kendte. Denne operation gør vi maksimalt $|E|$ gange. Antager vi at grafen er forbundet, gælder $|E| \geq |V| - 1$ og derfor ville derfor være $O(|E| \lg |V|)$.

Dijkstra's algoritme har en pladskompleksitet på $O(|V|)$, da antallet af knuder i Q til at starte med er $|V|$. Samtidig bruger vi også listen S til at holde styr på besøgte knuder. Endeligt er listen S fyldt med $|V|$ knuder. Derfor er pladskompleksiteten $O(|V| + |V|) = O(2|V|) = O(|V|)$ da vi ser bort fra konstanter i øvre grænser.

2.2 Bellman-Ford

Bellman-Ford er en *single-source* algoritme [1] [5], som også beregner den korteste vej. Denne alogritme har samme fremgangsmåde som Dijkstra, dog med den forskel at det ikke er et krav at alle vægte er positive reelle tal. Derfor kan algoritmen anvendes på grafinstanser med negative vægte. Dette er en fordel da det betyder at Bellman-Ford kan håndtere flere typer af grafer, end eksempelvis Dijkstras algoritme. Kravene for Bellman-Ford er at grafen skal være orienteret og at der ikke må findes negative cykler. Bellman-Ford går igennem alle kanter $V - 1$ gange, hvor V er antallet af knuder i grafen. For hver iteration minimeres

kanterne hvis det er muligt, således at vejen fra start knuden til alle knuder er de korteste veje.



Figur 2: Eksempel på gennemkørsel af Bellman-Ford algoritmen. Startknuden er s [2]

I Figur 2 illustreres en gennemkørsel af Bellman-Ford. Herunder ses en forklaring af de 5 skridt:

- I (a) skridtet illustres grafen før der relax-teknikken udføres.
- Fra skridt (b)-(d) relaxeres der således at vejene fra start knuden ud til alle knuder er minimeret.
- I (e) ses den endelige graf.

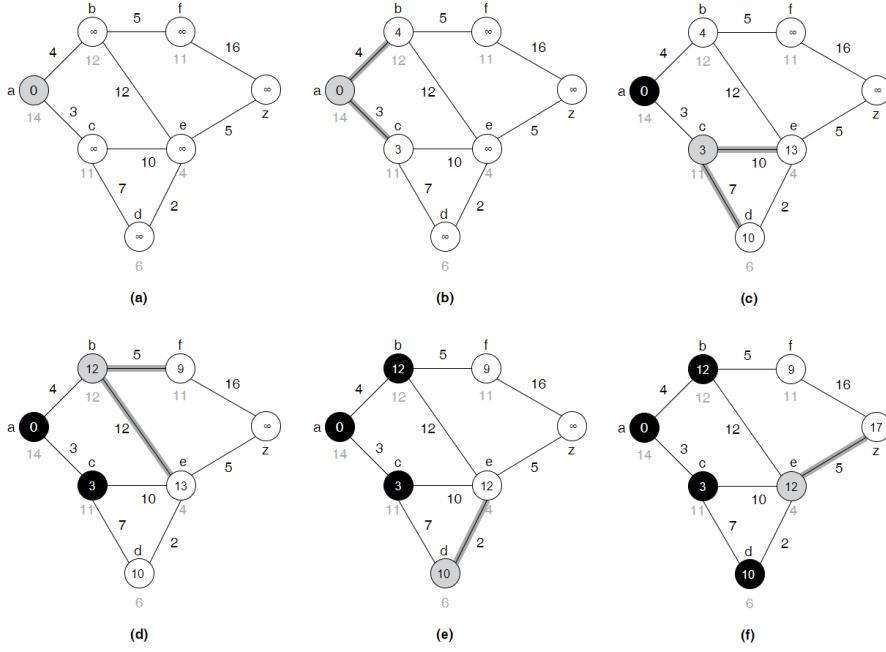
Tidskompleksitet og pladskompleksitet

I værste tilfælde er tidskompleksiteten for Bellman-Ford $O(VE)$ [2], da algoritmen besøger alle knuder og kanter i grafen. Pladskompleksiteten er $O(V)$ da den besøger alle knuder.

2.3 A* søge algoritme

A* (udtale: A-stjerne) er endnu en *single-source* algoritme, som finder den korteste vej fra A til B [6]. Algoritmen ligner Dijkstras algoritme og har derfor den samme fremgangsmåde og egenskaber. Der er dog en tilføjelse til algoritmen, som er dens brug af heuristik. Heuristikken er ofte repræsenteret som den euclidiske afstand mellem start- og slutknuden, og bliver som regel lagt sammen

med kantens vægt, hvilket udgør den en total afstand. Dette er en fordel for A*, da den bruger heuristikken til at estimere retningen af den korteste vej og den har derfor også en fornemmelse af hvor tæt den er på sin slutknude. A* er typisk mere effektiv end Dijkstra, da den ved brug af heuristikken besøger færre knuder.



Figur 3: Eksempel på gennemkørsel af A* algoritmen. De grå tal repræsenterer heuristikken fra den pågældende knude til Z og de sorte tal repræsenterer kantvægte.

I Figur 3 illustreres en gennemkørsel af A*. Herunder ses en forklaring af de 6 skridt:

- I (a) skridtet ses grafen før der relaxeres. De grå tal er afstanden fra knuden til Z , som er slut knuden.
- I (b) skridtet tjekkes knude A 's naboer, og vi marker A som værende besøgt.
- I (c) skridtet tages den knude med den laveste totale afstand fra forrige knude som er C og kanterne tjekkes.
- I (d) skridtet tages den næste knude med laveste totale afstand B og kanterne tjekkes.

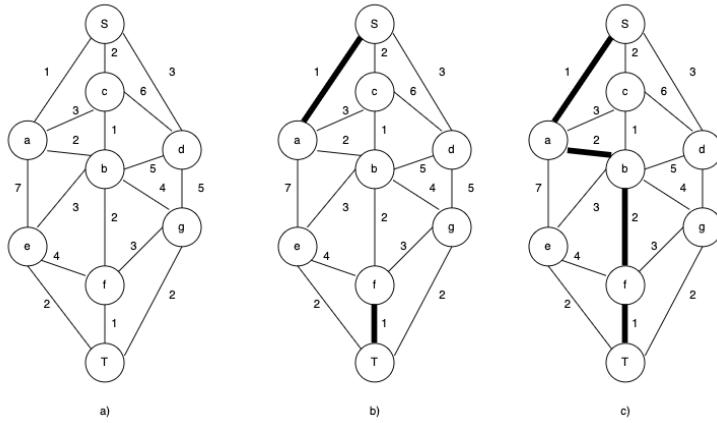
- I (e) skridtet går vi til knude E fra knude D , da den har den laveste totale afstand.
- I (f) skridtet har vi nået vores mål og en rute er fundet: A-C-D-E-Z.

Tidskompleksitet og pladskompleksitet

I værste tilfælde er tidskompleksiteten for A^* den samme som Dijkstras, da A^* er magen til Dijkstra, hvis vi så bort fra heuristikken. Heuristik-opslag bliver udført i $O(1)$ tid, hvorfor tidskompleksiteten for A^* er $O(|V| + |E|) \lg |V|$). I værste tilfælde er pladskompleksiteten $O(|V|)$, hvor V er antallet af knuder.

2.4 Tovejs søgealgoritme

En tovejs søgealgoritme har til formål at køre to af den samme algoritme på samme tid, hvor den ene bevæger sig fra startknuden mod slutknuden, og den anden fra slutknuden mod startknuden [9]. Den korteste vej bliver fundet når de to algoritmer mødes på midten. Herefter når algoritmen et stopkriterie, og de to veje lægges sammen til én, således at vi har en vej fra A til B . Der findes forskellige varianter af denne tovejs søgealgoritme, hvor de mest bemærkværdige varianter er baseret på A^* eller Dijkstra. En tovejs søgealgoritme er markant mere effektiv til at finde korteste veje på kortere tid, end de fleste andre algoritmer, som f.eks. Dijkstra.



Figur 4: Eksempel på gennemkørsel af tovejs søgealgoritme baseret på Dijkstra.

I Figur 4 illustreres en gennemkørsel af en tovejs søgealgoritme baseret på Dijkstra. Herunder ses en forklaring af de tre skridt:

- I (a) skridtet starter Dijkstra algoritmen fra både S og T .
- I (b) skridtet tager Dijkstra de første grådige valg.

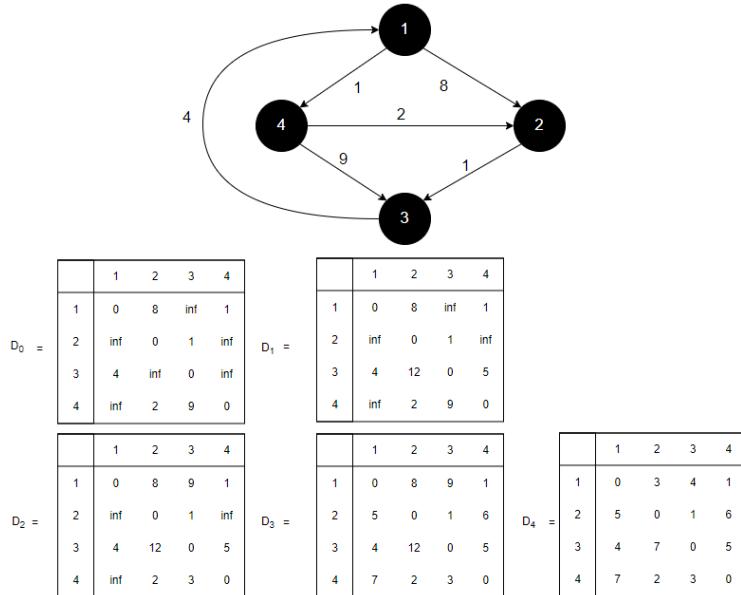
- I (c) skridtet tages der igen grådige valg. Ved knude B ses en sammenfletning af de to Dijkstra algoritmer. Vi har nu den korteste vej fra S til T og algoritmen terminerer.

Tidskompleksitet og pladskompleksitet

Tidskompleksiteten og pladskompleksiteten i en tovejs søgealgoritme afhænger af hvilken type tovejs søgealgoritme der er på tale. Generelt ville en tovejs søgealgoritme i værste tilfælde have dens tids- og pladskompleksitet halveret fra den algoritme den er baseret på.

2.5 Floyd-Warshall

Floyd-Warshall er en algoritme der beregner korteste veje i en orienteret vægtet graf. Algoritmen kan både håndtere positive og negative vægte, men kan ikke håndtere negative cykler. Floyd-Washall er en alle-par (eng: all-pairs) korteste vej algoritme, hvilket betyder at den på en enkel gennemkørsel af algoritmen finder alle de korteste veje mellem alle knuder i grafen [4]. I Floyd-Warshall algoritmen opstilles en $N \times N$ matrice med knuderne og herefter køres algoritmen.



Figur 5: Eksempel på gennemkørsel af Floyd-Warshall.

I Figur 5 illustreres en gennemkørsel af Floyd-Warshall. På figuren kan man se selve grafen, og de matricer der dannes udfra grafen. Herunder ses en forklaring af skidtene:

I kroppen for Floyd-Warshall findes tre for-løkker: k , i og j . Alle for-løkker kører indtil $|V|$, som er antallet af knuder. Den inderste for-løkke j er omkranset af i , som er omkranset af den yderste for-løkke k . Måden hvorpå relax-teknikken udføres på, er ved at tjekke om $dist[i][j] > dist[i][k] + dist[k][j]$, hvor $dist$ betyder afstanden (eng: distance) i $N \times N$ matricen. Den mindste værdi indsættes i matricen på det pågældende indeks. Denne opdatering af matricen kan ses i form af et eksempel, som vist på Figur 5, hvor fem matricer ses. Disse matricer er i teorien én enkelt matrice, men eksemplet viser hvordan matricen ser ud på fem forskellige tidspunkter. Når opdateringen af matrice D_i er færdig, ser vi den næste opdatering af matrice D_{i+1} . Et eksempel på en opdatering ses på skridtet mellem D_3 og D_4 , hvor $dist[1][2] > dist[1][4] + dist[4][2]$ ækvivalerer til $8 > 1+2$. Indekset bliver derfor opdateret til 3, da summen er mindre.

Tidskompleksitet og pladskompleksitet

I værste tilfælde er tidskompleksiteten for Floyd-Warshall $O(|V|^3)$ [2], da der i algoritmens krop forekommer tre for-løkker, hvor der i hver løkke gennemgås $|V|$ knuder. I værste tilfælde er pladskompleksiteten $O(|V|^2)$, da vi for hver knude skal finde en vej til alle andre knuder.

2.6 Johnsons algoritme

Johnsons algoritme minder meget om Floyd-Warshall algoritmen, da den også er en *all-pairs* korteste vej algoritme. Den anvender både Bellman-Ford og Dijkstra algoritmerne [7]. Den starter med at bruge Bellman-Ford til at eliminere negative kanter og finde negative cykler. Derefter bruger den Dijkstra til at finde den korteste vej fra alle knuder til alle knuder.

I Figur 6 illustreres en gennemkørsel af Johnson. Herunder ses en forklaring:

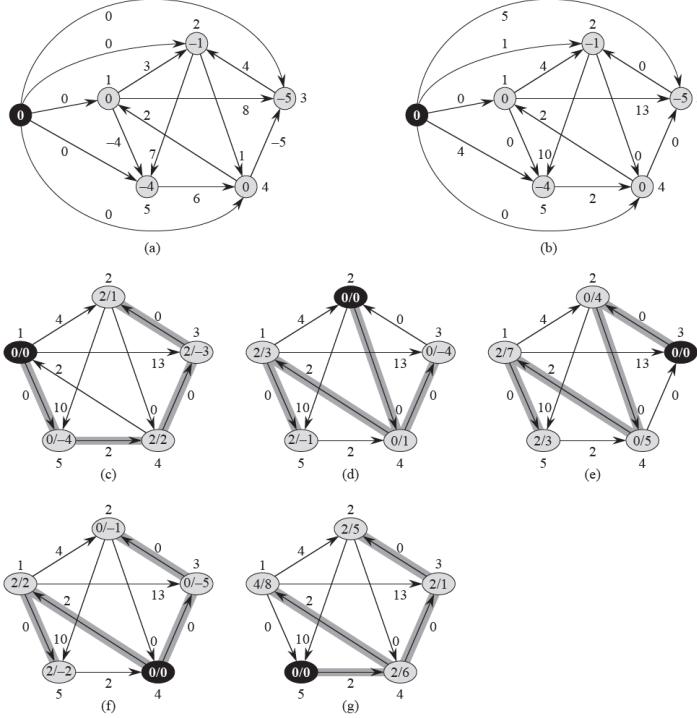
- I (a) skridtet tilføjes en ny knude, som har en kant med vægten 0 til alle andre knuder i grafen.
- I (b) skridtet besøges alle kanter og negative kanter bliver elimineret.
- I (c-g) skridtet fjernes den knude der blev tilføjet fra første skridt, hvorefter Dijkstras algoritme køres.

Tidskompleksitet og pladskompleksitet

I værste tilfælde er tidskompleksiteten for Johnson $O(|V|^2 \log |V| + |V||E|)$, hvis der gøres brug af en fibonacci minimum-prioritets-kø, da den først kører Bellman-Ford og herefter Dijkstra [2]. I værste tilfælde er pladskompleksiteten $O(|V|^2)$, da vi for hver knude skal finde en vej til alle andre knuder.

2.7 Mahadeokar-Saxena erstatningseveje algoritme

Grundet naturen af grafen som EU4's vejfinder algoritmer kører på, har vores valg af den dynamiske algoritme som vi vil kigge på, været baseret på det. Den



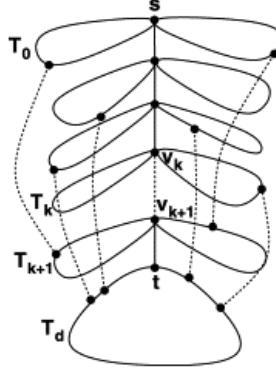
Figur 6: Eksempel på en gennemkørsel af Johnsons algoritme [2].

dynamiske algoritme vi har valgt at kigge på, er i dette tilfælde en erstatningsveje algoritme (eng: replacement paths algoritihm).

Erstatningsveje algoritmer går overordnet ud på at løse følgende problem: Givet en graf G , kildeknuden s og målknuden t , for hver knude v i den korteste vej mellem s til t i G , skal der findes en korteste vej s til t i grafen $G - v$. Der findes variationer til dette problem, hvor man f.eks. i stedet kigger på kantesvigt, men vi vil fokusere på knudesvigt, da det er hvad vi i EU4 har med at gøre.

Mahadeokar og Saxena giver en løsning på erstatningsveje problemet i deres videnskabelige artikel ”Faster replacement paths algorithms in case of edge or node failure for undirected, positive integer weighted graphs” [8], for 3 forskellige sager som de klassificerer som 1. kantsvigt problemer, 2. knudsvigt problemer og 3. omvej kritiske problemer. Vi vil fokusere på 2. knudsvigt problemer.

Lad $G = (V, E)$ være en ikke-orienteret graf (eng: undirected graph) med positive kante vægte. For $s, t \in V$, lad P være en kortest vej mellem kildeknude s og målknude t . Vi kan betegne P som $P = \{v_0, v_1, v_2, \dots, v_d\}$ hvor $s = v_0$ og



Figur 7: Korteste vej træet med roden s er opdelt i sæt T_0, \dots, T_d [8]

$t = v_d$. For hver knude v_i i P , skal vi finde en kortest vej s til t i grafen $G - v_i$.

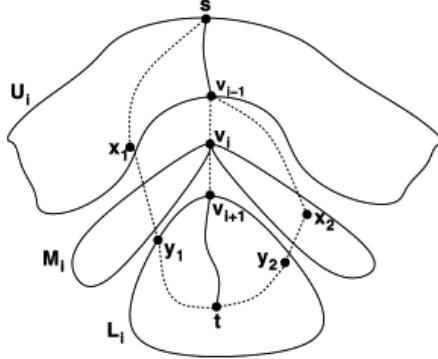
Ideen med algoritmen er at opsætte korteste veje træer (eng: shortest path trees) forankret (eng: rooted) i s og t , der vil hjælpe med at finde erstatningsveje, skulle der opstå et knudsevigt. Vi finder dette træ X ved at køre en korteste veje algoritme med s som roden. Derudover finder vi træet Y ved at køre en korteste veje algoritme med t som roden. På X , udfører vi forudseende traversering (eng: preorder traversal), hvor vi vil gemme $\text{pre}(v)$, $\text{desc}(v)$ og $\alpha(v)$ som betegner preorder nummeret, antallet af efterkommere, v inkl. og $\alpha(v) = \text{pre}(v) + \text{desc}(v)$. T_i vil betegne de efterkommere (eng: descendants) af v_i i X som ikke er efterkommere af v_{i+1} (se figur 7). Hvis en knude v_i bliver utilgængelig, bliver træet X delt op i 3 dele. Lad $\tau(v_i)$ betegne sættet af knuder som er efterkommere i X af knuden v_i , v_i inkluderet. Knuderne af $\tau(v_{i+1})$ er også en del af $\tau(v_i)$ sættet.

- Undertræet U_i for "upper", består af sættet $\tau(s) \setminus \tau(v_i)$
- Undertræet L_i for "lower" består af sættet $\tau(v_i) \setminus (\{v_i\} \cup \tau(v_{i+1}))$
- De resterende dele undertræet M_i for "middle" består af $\tau(v_{i+1})$

figur 8. illustrerer ovennævnte. Erstatningsvejen for knuden v_i kommer til at bestå af en sammenkædning af følgende:

1. $P_{G-v_i-L_i}(s, x)$: korteste vej fra s til en vilkårlig knude x i grafen $G - v_i - L_i$.
2. (x, y) : en kant (x, y) hvor $x \in G - v_i - L_i$ og $y \in L_i$.
3. $P_{L_i}(y, t)$: korteste vej fra y til t .

Da grafen er ikke-orienteret, ved vi at prisen på vejen $P_{L_i}(y, t)$ er afstanden (vægt) mellem t og y , hvilket blev beregnet da vi fandt Y . Vi kender også prisen på (x, y) , derfor mangler vi kun at finde prisen på $P_{G-v_i-L_i}(s, x)$. Da x kan befinde sig i U_i eller M_i , finder man erstatningvejen på to måder, baseret på hvor x hører til.



Figur 8: Korteste vej træet med roden s . Hvis knuden v_i bliver utilgængelig, bliver træet delt op i sæt U_i , M_i og L_i . Erstatningsvejen $\{s, \dots, x_1, y_1, \dots, t\}$ svarer til når $x_1 \in U_i$. Erstatningsvejen $\{s, \dots, x_2, y_3, \dots, t\}$ svarer til når $x_2 \in M_i$ [8]

2.7.1 Tilfældet hvor $x \in M_i$

Hvis $x \in M_i$, må vejen fra s til x indeholde knuder fra M_i (se figur 8 med x_2 og y_2). Vi opstiller en modifieret graf G'_i som er opbygget af følgende:

- Skrump alle U_i 's knuder til en superknude (eng: supernode) s'_i . Tilføj s'_i til G'_i .
- For hver knude $x \in M_i$, tilføj en tilsvarende knude x' til G'_i
- For hver kant (x, y) hvor $x \in M_i$ og $y \in M_i$, tilføj en tilsvarende kant (x', y') til G'_i med samme pris, dvs. $c(x', y') = c(x, y)$, hvor $c(x, y)$ betegner prisen for kanten (x, y)
- For hver kant (x, y) hvor $x \in U_i$ og $y \in M_i$, tilføj en tilsvarende kant (x', y') til G'_i med prisen $c(x', y')$, som er $c(x', y') = d_G(s, x) + c(x, y)$ (bevis i [8]), hvor $d_G(s, x)$ betegner afstanden fra s til x i grafen G

Herefter finder vi korteste vej træet forankret i s'_i . Nu vil vi for hver kant (x, y) hvor $x \in M_i$ og $y \in L_i$ finde

$$\psi_i(x, y) = d_{G'_i}(s', x') + c(x, y) + d_G(y, t)$$

Af dem vil vi finde det $\psi_i(x, y)$ med laveste værdi, og det vil være den kant vi bruger til vores erstatningsvej.

2.7.2 Tilfældet hvor $x \in U_i$

Hvis $x \in U_i$, er det blot kanten (x, y) vi vil finde. Vi er interesseret i at kigge på de kanter (x, y) hvor $\text{pre}(x)$ ligger i

$$1 \leq \text{pre}(x) \leq \text{pre}(v_{i-1}) \text{ eller } \alpha(v_i) \leq \text{pre}(x) < \alpha(s)$$

og $\text{pre}(y)$

$$\text{pre}(v_{i+1}) \leq \text{pre}(y) < \alpha(v_{i+1})$$

figur 8 illustrerer ovennævnte med x_1 og y_1 . Som før, vil vi for hver af disse kanter (x, y) finde

$$\psi_i(x, y) = d_{G'_i}(s', x') + c(x, y) + d_G(y, t)$$

Af dem vil vi finde det $\psi_i(x, y)$ med laveste værdi, og det vil være den kant vi bruger til vores erstatningsvej.

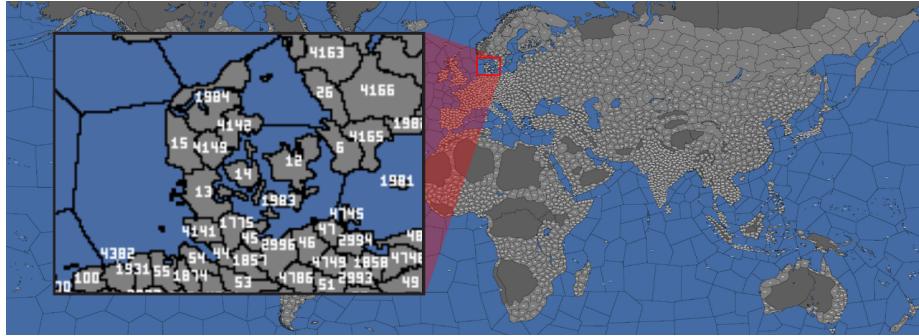
Ved brug af denne algoritme og forbehandling af de oprettede træer, hævder forfatterne af artiklen [8], at algoritmen har, ved et knudsesvigt på den fundne rute, en worst-case køretid på $O(|E| + d^2)$, hvor d betegner afstanden fra s til t .

3 Korteste veje i EU4

I dette projekt undersøger vi korteste veje i spillet Europa Universalis IV (EU4). Spillet består af et kort over verdens provinser som set i Figur 9. Man kan bevæge sine tropper fra provins til provins. Tropper betegner soldater i spillet, som spilleren kan bevæge mellem provinserne. Både spillere og fjender har tropper. Spillet kan derfor repræsenteres som en graf hvor hver provins er en knude og hvor kanterne er forbindelserne mellem provinserne. I spillet spiller man imod andre spillere og/eller computeren, og der kan derfor opstå fjendtlige tropper i visse provinser. Medmindre man eksplisit giver en ordre om at angribe, vil man gerne have at den rute som sine tropper tager, undgår fjenderne. Dette kan blive repræsenteret som utilgængelige knuder på grafen, hvilket gør grafen dynamisk. Derudover kan de fjendtlige tropper også bevæge sig fra provins til provins. Når man vil bevæge sine tropper, får man givet et estimat på hvor mange dage det tager at rejse fra provins til provins. Dette estimat er påvirket af forskellige variabler, såsom terrænet af provinsen, trop-type, vejret osv. Disse variabler kan man modellere som vægten på kanterne.

3.1 Definition på den mest egnede algoritme

Formålet med projektet er at finde den mest egede algoritme til at finde den korteste vej på en dynamisk graf af typen som set i EU4. ”Den mest egnede algoritme” har vi valgt at kvantificere som dagene det tager at bevæge sig fra provins a til provins b , da dette ultimativt er det vigtigste mål indefor spillets rammer. Med dette fastlagt, kan vi begynde at kigge på de parametre, som bliver påvirket af at vælge algoritmen, der minimerer antallet dage en tur tager og vælge en passende algoritme baseret på dem. Disse parametre inkluderer køretid, pladsforbrug, besøgte knuder, knuder i den endelige vej og opdateringer på vejen. Det er med dette i mente vi har haft intuitionen om, at det ville være interessant at sammenligne en dynamisk korteste vej algoritme og de klassiske statiske algoritmer, som anvendes i en dynamisk sammenhæng.



Figur 9: EU4 kortet og dets provinser [10]. Lysegrå provinser med et provins-ID betegner landets provinser. Blå provinser betegner havets provinser og de mørkegrå provinser betegner provinser hvor tropper ikke kan færdes (havet og provinser hvor tropper ikke kan færdes, har også provins-ID'er, dog er de udeladt for denne illustration).

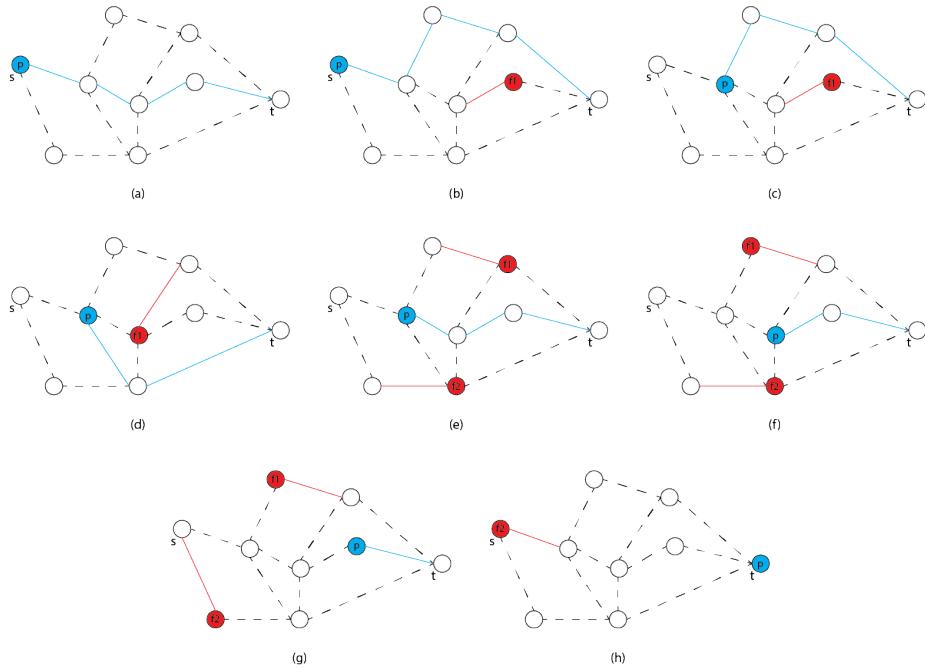
3.2 Grafens karaktertræk

Grafen består som helhed af provinser, hvor provinserne der grænser op til hinanden, er naboer forbundet af en kant. Grafen er dynamisk i den forstand, at når man f.eks. vil finde den korteste vej P , bestående af provinser v_i , fra provins s til t , kan en fjende optage en af disse provinser. Medmindre eksplisit givet en ordre om at engagere fjendtlige tropper, vil man gerne have at det undgås. Udover fjendens nuværende position, ved man også hvor de fjendtlige tropper er på vej hen, altså kender man til provinsen de er på vej til. Der er ingen begrænsninger for om spilleren kan bevæge sig frem og tilbage mellem provinserne. Derfor kan vi modellere grafen som at være ikke-orienteget. Tropper kan derfor bevæge sig fra provins a til provins b , men også fra provins b til provins a . Bevægelsen af alle troppers (spiller og fjendtlig) bevægelse er styret af et diskret globalt ur. Hver trop har deres eget interne ur, der bruges som alarm til hvornår det er tid til at bevæge sig. Alarmen følger det globale ur. Det er på denne måde, at grafen håndterer hvor lang tid en trop skal stå på en provins, før den kan bevæge sig til den næste. Det globale ur uddybes i Kapitel 5. Alle kanter i grafen har en vægt, som også skal tages stilling til, når algoritmen vælger den korteste vej.

Vægten skyldes forskellige former for terræn og andre variabler. Havet består, ligesom landet, også af provinser som i sagens natur ikke er forskellig fra landets provinser, når det kommer til hvordan de opfører sig. Forskellen ligger i kantvægtene, der er givet mellem land-hav forbindelser (kyster osv.). På Figur 9 ses 4942 provinser, hvor enkelte provinser er markeret med en mørkegrå farve, som symboliserer alle de provinser hvorpå en spiller ikke kan have tropper stående. Nogle af disse provinser er ufarbare, dvs. de er områder med bjerge som ikke kan krydses. Trækker vi antallet af ufarbare provinser fra, er vi nede på at grafen har 3925 provinser, hvor spilleren godt kan have tropper stående.

Figur 10 viser et eksempel på hvordan kørslen af en korteste vej algoritme på denne slags graf kunne se ud. Følgende forklarer kørslen af Figur 10.

- (a) Spillerens tropper p 's korteste vej P fra knude s til t er fundet
- (b) Fjendens tropper f_1 bliver rejst på en knude i P , ny korteste vej P for p fundet
- (c) p bevæger sig
- (d) f_1 bevæger sig, ny korteste vej for p fundet
- (e) Fjendtlige tropper f_2 bliver rejst på en knude i P , f_1 bevæger sig, ny korteste vej P for p fundet
- (f - g) f_1 bevæger sig, f_2 bevæger sig og p bevæger sig
- (h) f_1 tropperne ned sættes, f_2 bevæger sig og p når i mål



Figur 10: Eksempel på kørsel af en vilkårlig dynamisk korteste vej algoritme, på den type dynamiske graf spillet har med at gøre. s betegner startknuden, t betegner målknuden, blå knude og kanter repræsenterer spillerens tropper og røde knuder og kanter repræsenterer fjendtlige tropper. Algoritmen er ikke baseret på nogen faktisk algoritme, valgene algoritmen tager er skræddersyet for eksemplets skyld.

4 Valg af algoritmer til implementation

Ud fra vores valgte case (EU4) har vi en dynamisk graf, da knuder kan blive utilgængelige undervejs, som nævnt tidligere. Af denne årsag har vi et skærpet udvalgt af algoritmer, der kan anvendes på vores problem. Vi har derfor valgt at implementere følgende algoritmer:

- Dijkstra
- A*
- Tovejs Dijkstra
- Tovejs A*
- Bellman-Ford

Umiddelbart er ingen af disse algoritmer beregnet til at finde korteste veje i dynamiske grafinstanser, da det er statiske algoritmer. Af forskellige årsager (som vi vil udpense herunder) har vi alligevel udvalgt dem. Vores graf er af lille størrelse, selvom det er et kort over provinser i hele verden. Dette betyder at alle de udvalgte algoritmer (bortset fra Bellman-Ford, som besøger alle knuder) bør være i stand til, at finde den korteste vej hurtigt, selvom vi laver opdateringer på grafen undervejs. Disse opdateringer forekommer kun på den pågældende rute, som algoritmen allerede har udvalgt til at være den hidtil korteste vej.

Der kan sættes spørgsmåltegn til, hvorfor vi har valgt at implementere en langsom statisk algoritme som **Bellman-Ford**. Årsagen er blandt andet, at vi inden eksperimentet forventede at kunne køre algoritmen én enkelt gang og herefter kende alle korteste ruter fra kilde-knuden. Dette virker også i en statisk sammenhæng, men i øjeblikket hvor knuder begynder at blive utilgængelige opstår problemet, hvor nogle af de tidligere beregnede korteste ruter er afhængige af de knuder som bliver utilgængelige. Dette var ikke et problem vi havde tænkt over, før vi havde implementeret algoritmen. Ydermere ville vi også have en algoritme med i eksperimentet, som vi på forhånd havde en ide om, ville have mindre effektivitet end de andre algoritmer. Årsagen til dette er ikke for at belyse de andre algoritmer, men for at sætte fokus på hvorfor nogle algoritmer ikke egner sig til opdateringer i grafinstanser.

Da der ser ud til at være forskellige definitioner for en dynamisk algoritme i offentlig litteratur, er vi nødt til at præcisere den type dynamisk algoritme, der passer til den type graf, som spillet EU4 gør sig brug af. Den første type af algoritmer der defineres som dynamisk, er af slagsen der håndterer opstæn af utilgængelige knuder under kørslen af algoritmen. For eksempel kan det ske, at der under et skridt i kørslen af Dijkstras algoritme opstår et knudesvigt. En dynamisk version af Dijkstras algoritme ville i denne forstand kunne håndtere dette, og fortsætte med at finde den korteste vej. Den anden type dynamisk algoritme er den af slagsen, som håndterer knudesvigt på en korteste vej, efter den

korteste vej mellem to knuder er fundet. Et eksempel på dette, er givet i den dynamiske Mahadeokar-Saxena erstatningseveje algoritme fra undersøgelsen. Det er de dynamiske algoritmer af den sidstnævnte type som vi er interesseret i at kigge på.

Selvom Mahadeokar-Saxena erstatningseveje algoritmen opfylder kravet om den type dynamisk algoritme der passer til EU4's graf-type, dækker den stadigvæk ikke alle krav som grafen i EU4 kræver. Algoritmen håndterer nemlig kun tilfældet af knudsesvigt på den første givet korteste vej. Altså kan der efter opståen af et knudsesvigt på den første korteste vej ikke håndteres opstandelsen af endnu et knudsesvigt på den nye korteste vej. En dynamisk graf i EU4 skal kunne håndtere knudsesvigt uanset antallet nødvendige opdateringer på den korteste vej.

Vi har ikke valgt at implementere nogle ægte dynamiske algoritmer, da vi i undersøgelsen af korteste vej algoritmer fandt ud af, at dynamiske algoritmer er noget mere komplekse og avanceret end statiske algoritmer. Grundet en tidslig horisont og for projektets bedste, har vi derfor valgt at fokusere på anvendelsen af statiske algoritmer i dynamisk forstand. Samtidig har vi en opfattelse om, at vores graf er for lille til, at en dynamisk algoritme ville være effektiv på den. Dette har vi ikke et bevis for, men tanken bunder i, at en dynamisk algoritme skal udføre tidskrævende forarbejde (eng: pre-processing), før den gør nytte til at finde korteste veje.

Vi har udvalgt **Dijkstras** algoritme, blandet andet fordi den tager det grådige valg. Dette gør det muligt for os at stoppe algoritmen, når vi er nået stopknuden (eller destinationen) i grafen. På denne måde kan en masse beregninger ses bort fra, da disse ikke betyder noget for den pågældende rute som algoritmen arbejder på. En anden grund til at vi har udvalgt netop Dijkstra, er fordi det er en klassisk korteste vej algoritme og umiddelbart den første der bliver tænkt på, når problemet om at finde korteste veje nævnes. Derfor synes vi at det kunne være spændende, at se hvordan den klarer sig med at finde korteste veje i en dynamisk sammenhæng. Dernæst har vi udvalgt **A***, da den er baseret på Dijkstras algoritme, men har selve heuristik delen med sig. Vi tror på at euklidiske afstande i netop dette eksempel, med et verdenskort, ville gavne resultatet gevældigt. Det er derfor også vores forventning at A* kan finde nogle kortere veje end Dijkstra. Desuden har vi udvalgt to algoritmer, hvis grundlag stammer fra henholdsvis A* og Dijkstra, men i en mere avanceret version. Her taler vi om **tovejs A*** og **tovejs Dijkstra**. Vi forventer at disse algoritmer vil være særdeles effektive i vores case (EU4), da vi hele tiden kender en start -og stopknude. Her kan den ene del af tovejs algoritmen tage udgangspunkt i startknuden, mens den anden i stopknuden. Vi er interesseret i at se hvor effektiv tovejs-versionen af A* og Dijkstra er i denne sammenhæng, når sammenlignet med den originale version.

5 Implementation

I dette afsnit forklares dele af implementationen og de grundlæggende valg vi har truffet undervejs, samt vores tanker og hensigt bag valgene.

5.1 Simulering af spillet

En kort forklaring af spillets systemer vil hjælpe med at afklare valgene og begrænsningerne bag vores simulering af spillet. Europa Universalis IV, er af ”Grand Strategy” genren. Grand Strategy spil består ofte af håndteringen af en eller flere nationers/partiers ressourcer, hvilket indebærer hæren, økonomien, politikken og mere. Ofte er grænsefladen som spilleren interagerer med, et eller flere kort, og menuer hvorpå man kan tilgå de forrigenævnte ressourcer (se Figur 11). Hæren man styrer i EU4 er opdelt i mindre regimenter. Disse tropper visualiseres på spillekortet som set på den grønne forstørrelse i Figur 11. Tropperne kan som tidligere nævnt, bevæge sig mellem provinser givet en destination. Det tager dog tid for tropperne at bevæge sig mellem disse provinser. Tiden det tager er baseret på spillets tids-system.

EU4 er et ”real-time strategy” spil, altså kører der et globalt ur hvilket al-



Figur 11: Europa Universalis IV’s brugergrænseflade. Det grønne forstørrelse viser en spillerenhed. Den røde forstørrelse viser datoen og spilhastigheden.

le spilleres handlinger er bundet af, i modsætning til ”turn-based strategy” spil, hvor man ligesom i skak, skiftes om at tage en handling hver. Typisk kører det globale ur diskret, hvilket i EU4’s tilfælde er repræsenteret af dage. Hastigheden af spillet kan ændres ved at justere på hvor mange sekunder der går per dag. Tiden det tager at bevæge sine tropper mellem provinser, er altså repræsenteret

af dage og er afhængig af adskillige variabler. Vores simulering af spillet er derfor også baseret på dage som i spillet. Hver trop i vores simulering har en intern dag-nedtæller. Nedtælleren sættes til at være kantvægten mellem de provinser, som tropperne er i gang med at bevæge sig mellem. På denne måde sørger vi for at alle tropper, ligesom i spillet, er bundet af det globale ur: dagene.

5.2 Forarbejdelse af spildata

For at kunne implementere algoritmerne har en stor del af dette projekt gået ud på, at klargøre de forskellige data, der tilsammen udgør grafen. I de første uger hvor vi arbejdede på projektet, kontaktede vi virksomheden bag EU4 for at høre, om det var muligt at få en kopi af deres graf. Dette var desværre ikke en mulighed, så vi gik i gang med selv at udforme grafen. I mappen hvori spillet bliver installeret, kunne vi finde en masse spildata. Der var blandt andet en fil, som indeholdt alle provinser og deres koordinater på verdenskortet. Den vigtigste data for vores projekt, nemlig kanterne mellem knuderne, var ikke til at finde i nogle af spilfilerne. Dog fandt vi et .bmp billede af hele verden, hvor alle provinserne var afgrænset af hver deres unikke RGB farvekode. Vi har derfor været i stand til, at opdage alle kanter i grafen ved brug af dette billede og filen med provins data.

Vi har i en dobbelt for-løkke anvendt en peger som gennemløber alle pixels i .bmp billedet af hele verden. Det ene loop itererer igennem højden og det andet igennem bredden. Pegeren er i stand til, at fortælle hvilken RGB-farve den står på i billedet. Ud fra denne farve kan vi slå op i en spilfil, der indeholder provins-ID'er og deres tilhørende unikke RGB-farve. Hver gang pegeren flytter sig til en anden farve, end hvad den stod på tidligere, ved vi at vi har bevæget os fra en provins til en anden. Vi kan derfor gemme en kant ved at nedskrive de to provins-ID'er. Sammen med kanten gemmer vi også dens vægt, hvilken vi udregner ved at summere de to provinser/knuders vægte.

Med henblik på heuristik, som bruges af henholdsvis A* og tovejs A*, har vi valgt at basere heuristikken på den euklidiske afstand mellem provinserne. Vi har ud fra det tilgængelige spildata anvendt alle positioner af provinserne, hvor vi i en dobbelt for-løkke gennemløber alle provinsernes positioner. Vi beregner derefter den euklidiske afstand mellem provinserne ved at anvende biblioteket `numpy` og dets funktion `linalg.norm`. Herefter skalerer vi vores heuristik således:

$$h(x) \leq d(x, y) + h(y) \quad (1)$$

Her er $h(y)$ heuristikken for den nye knude og $h(x)$ er heuristikken for den forrige knude. Vægten af kanten, sammenlagt med den nye knude $h(y)$'s heuristik, skal være større eller lig den forrige knude $h(x)$'s heuristik. Dette gør at vores heuristik er konsekvent/tilladelig (eng: admissible), og dermed er de algoritmer der bruger heuristik også garanteret i at finde en optimal rute. Den præcise skaleringsfaktor har vi opdaget ved at finde den største værdi i heuristikken.

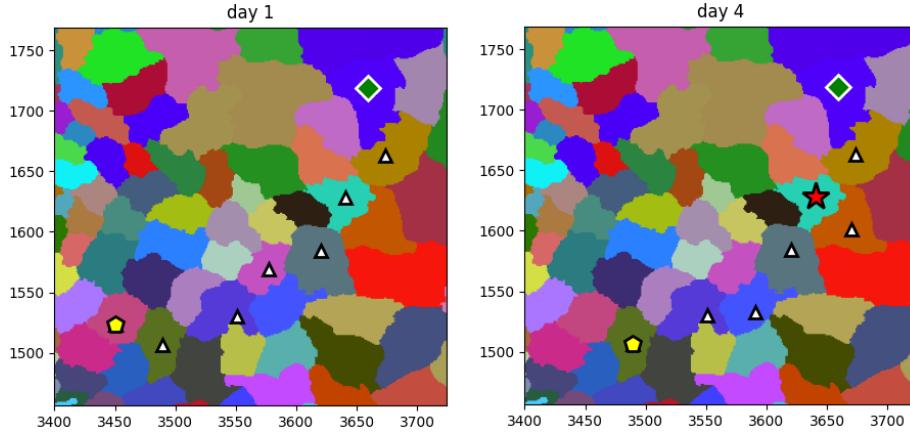
Denne værdi viste sig at være 5946, 29633, altså den største euklidiske afstand mellem to provinser i vores graf. Derudover har vi også fundet frem til, at den mindste vægt der kan forekomme på kanter i grafen, er 2. Vi har derfor skaleret alle værdier i heuristikken ved at dividere den enkelte værdi med skaleringsfaktoren og herefter lagt heltallet 1 til. I denne process opnår vi reglen (1). Efter denne skalering bliver vores data skrevet ind i en matrix der er 4942×4942 stor. Denne matrix indeholder alle knuder der findes på grafen, og ved at lave et opslag som eg. række 8 kolonne 17, kan vi læse den skalerede euklidiske afstand mellem provinserne med ID 8 og 17.

Havde vi i stedet undladt en skalering af heuristikken, ville heuristikken være utiladelig (eng: non-admissible), hvilket betyder at reglen (1) ikke holder og dermed ville A* og tovejs A* besøge langt færre knuder, på bekostning af en mindre optimal vej. Dette ville i vores eksempel medføre et højere antal dage på de fundne veje, men en hurtigere terminering af algoritmen, da den besøger færre knuder. Havde antal dage, som det tager at gennemføre ruten, ikke været den vigtigste faktor, kunne der argumenteres for, at man godt ville tillade den indbyggede korteste vej algoritme at finde ruter som er ”tæt” på det optimale, hvis algoritmen skal kunne eksekvere hurtigt. Det giver derfor mening at justere på heuristikken, ved at nedskalere de originale euklidiske afstande, men med en faktor så heuristikken forbliver *utiladelig*. Bemærk, algoritmer som bruger denne heuristik vil finde korteste veje, som er tæt på det optimale, men på kortere tid og med mindre ressourcer. Denne justering af skaleringsfaktoren kan bestemmes manuelt, indtil der opnås et punkt, hvor antal dage som det tager at flytte sine tropper fra start til slut, er så tæt på det optimale som muligt, samtidig med at antal knuder besøgt er inden for en vis grænse (subjektivt bestemt).

5.3 Visualisering af algoritmernes kørsel

For vores egen skyld og for nemmere at kunne udvikle på simuleringen, har vi visualiseret algoritmernes kørsel i realtid. Fra spilfilerne har vi anvendt det .bmp billede over verden, som tidligere nævnt, til at kunne visualisere spilleren og fjendernes skiftende positioner i grafen. Vi har gjort brug af Python biblioteket `matplotlib`, hvor vi har sat dette .bmp billede som baggrund og herefter plottet spilleren og fjendernes positioner. For hver opdatering der sker på grafen tegnes et nyt plot, og det tidligere plot bliver overskrevet af det nye. På denne måde bliver plottet dynamisk i den forstand, at vi visuelt kan se når spilleren og fjenderne bevæger sig rundt i verdenskortet. Ud over spilleren og fjenderne har vi også plottet den hidtil korteste rute, som algoritmen har fundet og spilleren bevæger sig igennem. Dette illustreres på Figur 12.

I den venstre visualisering på Figur 12 ses første rejsegang fra start provinsen til stop provinsen. Her har tovejs A* algoritmen fundet frem til, hvad den mener at være den korteste rute. Ingen fjendtlige tropper har vist sig på vejen endnu. Dette ændrer sig i den højre visualisering, hvor en fjende på fjerde-dagen optager en af knuderne på vejen, og algoritmen er derfor nødt til at beregne en ny korteste vej udenom fjenden.



Figur 12: Grafisk visualisering af en algoritmes kørsel på EU4 kortet. Gul femkant repræsenterer spilleren, grøn diamant repræsenterer slutprovinsen, hvide trekantede repræsenterer provinser på den fundne vej mellem start og stop provinsen, røde stjerner repræsenterer fjender. Algoritme anvendt: tovejs A*, fjende frekvens: 30%

5.4 Spiller logik

Spilleren er repræsenteret som den første knude i den rute der gives via den valgte korteste vej algoritme. Spillerens rolle i vores simulation af spillet er at bevæge sig langs dens given korteste vej fra start- til målprovins. Fjendens nuværende position og næste position er kendt af spilleren. Hvor mange dage fjenden bliver på sin position, er dog ikke kendt. Dette betyder at spilleren også behandler fjendens næste bevægelse som en utilgængelig knude. Der kan opstå situationer, hvor spillerens korteste vej algoritme ikke kan finde en alternativ vej, på samme tidspunkt hvor spillerens næste provins er optaget af en fjende. Tiden det tager for spilleren at bevæge sig mellem provinser, er afhængig af kantens vægt. Udover dette er der ikke mere spilleren skal håndtere, da det er den korteste vej givet til spilleren, som sørger for det meste af spillerens reelle ”valg” på grafen. Spilleren er blot en visualisering af den kilde knude, hvorfra den valgte algoritme baserer sine beslutninger fra.

5.5 Fjende logik

Vi har valgt at simulere fjendtlige tropper på en måde, der adskiller sig fra spillet. Vi mener at det er unødvendigt, at bruge ressourcer på, at lave en 1:1 simulation af fjende logikken. Derfor har vi valgt at simulere fjenderne på en måde, som fundamentalt ikke er forskellig, fra den måde de ville interagere med spilleren og deres fundne korteste vej.

Spillertropperne p får givet en korteste vej P mellem knuderne s og t . Vi har

valgt, at fjendtlige tropper kun kan blive rejst på P , udover knuderne s og t . Da vi kun er interesseret i situationen, hvor en fjende interagerer med spillerens korteste vej, ligeledes hvor fjenden kom fra, er det tilstrækkeligt blot at simulere fjendtlige tropper på disse provinser. De fjendtlige troppers logik er styret af variablerne frekvens og vejlængde. Frekvensen givet til de fjendtlige tropper styrer chancen for rejsningen af en fjendtlig trop på en provins på $P - \{s, t\}$. Der kan maksimalt blive rejst 1 fjendtlig trop per dag. Vejlængde-variablen styrer hvor mange provinser, en fjendtlig trop bevæger sig, før den bliver nedsat. Provinserne på den fjendtlige vej bliver valgt en ad gangen, for hver provins den står på, hvorfor den ikke finder hele ruten. En fjendtlig trop vælger en tilfældig nabo provins, som den bevæger sig hen til.

Vi har valgt at opsætte visse begrænsninger for de fjendtlige troppers logik, hvilket nødvendigvis ikke er den samme logik, som de besidder i spillet. Uden begrænsningerne ville der opstå problemer, som vi alligevel ikke er interesseret i, og kan derfor forkaste sager hvori sådanne problemer opstår. Det første mulige fjende problem vi adresserer, er sammenstødet af en fjende og spilleren. Det kan ske at en fjendtlig trop beslutter sig for at bevæge sig hen til provinsen som spillerens trop står på, hvis kanten som fjenden rejser igennem har en lavere vægt, end den spilleren er i gang med at rejse igennem. I spillet vil dette betyde at tropperne nu ville begynde at kæmpe. Da vi ikke er interesseret i dette scenarie, har vi valgt at sætte en begrænsning op for fjendtlige tropper, der gør at de ikke kan vælge at bevæge sig på den provins som spilleren på det tidspunkt står på.

Det næste mulige fjende problem vi adresserer, er når spilleren nærmer sig målknuden. Når spilleren nærmer sig målknuden, bliver antallet af knuder i P lavere. Dette betyder at vi med en høj nok fjende frekvens kan ende i en situation, hvor der i de omringende provinser til målknuden forekommer en besættelse af fjendtlige tropper. Derfor kan det tage længere tid for spilleren, at nå frem til dets destination. Vi har derfor valgt at sætte en begrænsning for, hvor lang spillerens fundne korteste vej skal være, før at det er tilladt at rejse nye fjender på vejen. Vi har i vores implementation valgt at sætte vejlængden til 4, hvilket resulterer i, at der ikke kan rejses nye fjendtlige tropper på spillerens korteste vej, hvis vejen indeholder 4 eller færre provinser.

5.6 Bemærkninger til implementerede algoritmer

For alle de implementerede algoritmer (bortset fra Bellman-Ford) har vi valgt at stoppe søgningen, når vi er nået stopknuden. Dette giver os en hurtigere ekskvering af algoritmen, og vi slipper for unødvendige vejsøgninger, som er ud over start- og stopknuderne. Endvidere betyder dette også, at tidskompleksiteterne som beskrevet i undersøgelsen aldrig vil blive nået.

For algoritmer som anvender en minimum-prioritets-kø, har vi i implementationen anvendt Python biblioteket `heapdict` til at oprette denne kø. Biblioteket giver os mulighed for at oprette, indsætte og udtrække elementer fra køen.

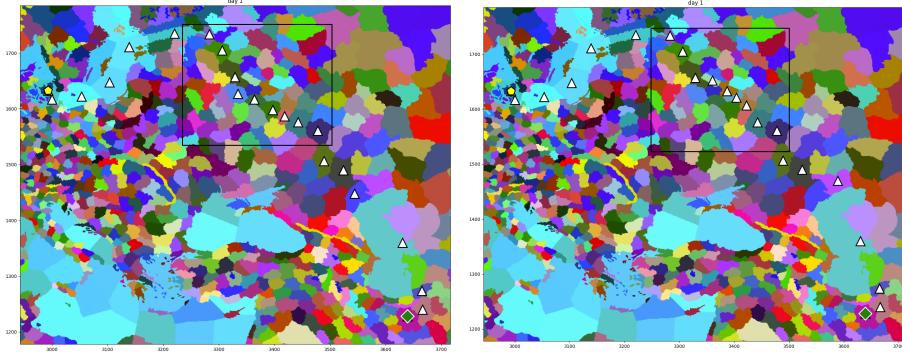
Med henblik på algoritmer som gør brug af en heuristik (A^* og tovejs A^*), har vi i implementationen anvendt den skalerede, tilladelige heuristik som omtalt i Kapitel 5.2.

6 Eksperiment

For at sammenligne algoritmerne og deres effektivitet til at finde korteste veje i EU4, har vi opstillet et eksperiment. Dette eksperiment tager udgangspunkt i de implementerede algoritmer, samt grafen som vi har udarbejdet ved brug af spildata fra installationsmappen. I eksperimentet har vi nogle forskellige tests, som hver algoritme skal igennem. Vi udvælger, ved brug af Python biblioteket `random`, 100 tilfældige start/stop provinser (samme 100 for alle algoritmer), som herefter kategoriseres som enten korte, mellem eller lange ruter. Vi mäter rutens distance ved at kigge på den første vej, som algoritmen finder (inden opdateringer i grafen). Ruterne er kategoriseret således:

- 0 - 10 provinser = kort rute
- 11 - 20 provinser = mellem rute
- over 20 provinser = lang rute

Alle algoritmer (bortset fra Bellman-Ford) skal igennem disse 100 start/stop provinser 4 gange hver, hvor der i første test er ingen chance for, at fjender kan optage knuder på den fundne vej. Dette er tilfældet, hvor der ikke sker nogle opdateringer på grafen (statisk graf). De resterende 3 gange er chancen henholdsvis 10%, 30% og 50%. I disse tilfælde bliver grafen gjort dynamisk, da fjender kan opstå på ruten. Grunden til at vi har valgt 10% bunder i, at denne chance dækker godt over forholdene, i spillet EU4, hvor spilleren befinner sig i fredelige omgivelser. Der er i løbet af spillet forskellige områder, som kan være lidt, moderat eller meget besat af fjenden. De 30% passer godt på den moderate fjendebesættelse af områderne, som f.eks. kunne være på grænsen mellem fjenden og sit eget territorium under krig. Imens svarer de 50% til områder med høj fjendebesættelse, og det skal forestille sig at være situationer, hvor man befinner sig på fjendtligt territorium under en krig. Vi har valgt at opdele rutelængder, siden det giver et bedre indblik i, om de implementerede algoritmer udviser forskellig effektivitet afhængigt af rutens længde. Årsagen til at vi har valgt at kategorisere ruterne, som vi har gjort, er baseret på spillets egenskaber. De korte ruter skulle forstille sig at være afstanden fra sit territorium til naboen, og mellem ruter skal forstille sig at være afstanden fra sit territorium til sin nabos nabo. I vores eksperimenter tester vi for parametrene: antal dage det tager at udføre en rute, antal besøgte knuder, tid det tager algoritmen at blive færdig, pladsforbrug, knuder i den endelige vej, samt opdateringer på grafen. Det er her værd at pointere, at knuder i den endelige vej repræsenterer knuder der eksisterer i ruten fra start til slut. Det betyder ikke nødvendigvis at det tager flere dage at udføre ruten, da den sammenlagte vægt af kanterne i en rute med



Figur 13: Kørsel af Dijkstra (venstre) og Tovejs Dijkstra (højre), samt de forskellige fundne veje, ud fra de samme start- og stopprovinser. Bemærk forskellen i de illustrede sorte kasser.

flere knuder, kan være mindre end den sammenlagte vægt af kanterne i en rute med færre knuder.

Eksperimentet er udført på et system med processoren Intel Core i7-8700K ved 3.70 GHz og 16 GB RAM.

7 Resultater og analyse

Ud fra sammenligningen af de gennemsnitlige resultater for de dynamiske ruter, som set på Tabel 6, 8, 10, 12 og 14 i Bilag A.1, kan det ses at det endelige antal dage, som det havde taget at gennemføre en vej, varierer. Tovejs Dijkstra er i gennemsnittet færre dage om, at gennemføre en givet vej på dynamiske grafer. Disse resultater kommer bag på os, da vi havde regnet med at alle algoritmer ville ende med at have de samme resultater, ift. antal dage det tager at gennemføre ruterne. Vi har dog en forklaring på, hvorfor der er varians i de endelige antal dage det har taget, at gennemføre de samme vej. Det skyldes tilfældighed med hensyn til fjendernes bevægelse, som godt kunne gå hen og påvirke resultaterne. Variationen af dage per algoritme skyldes at der findes mere end en optimal (antal dage) korteste vej, som set på Figur 13. Det kan være at knuderne på den korteste vej, som Dijkstra finder, ikke nødvendigvis er de samme knuder som A* finder. Dette påvirker algoritmen på punkter, som antal dage, besøgte knuder, tid og pladsforbrug, da algoritmen bliver nødt til at omdirigere, hvilket tager længere tid og får algoritmen til at besøge flere knuder.

På Tabel 12 ses det at tovejs A* algoritmen i gennemsnittet besøger færrest antal knuder, hvilket vi havde regnet med. Yderligere ses det fra resultaterne, at tovejs A* bruger mere plads og at det tager længere tid at køre algoritmen. Dette resultat kommer bag på os, da vi havde forestillet os at tovejs algoritmer-

ne ville være hurtigere end de oprindelige algoritmer.

Med fokus på de gennemsnitlige resultater for A* i Tabel 8 og for Dijkstra i Tabel 6, ses det at A* bruger færre antal dage til at gennemføre ruterne, besøger færre antal knuder og at der er færre knuder i den endelige vej. Dijkstra havde af alle algoritmer den laveste køretid og det mindste pladsforbrug.

Da vores implementation af Bellman-Ford besøger alle knuder i grafen, og ikke stopper ved slutknuden, har vi valgt ikke at køre de 100 tilfældige start/stop tests igennem for denne algoritme. Vi har af denne årsag kørt 1 kort, 1 mellem og 1 lang rute for algoritmen. Hvis man kigger på Bellman-Fords gennemsnitlige resultater i Tabel 14, kan det ses at algoritmen havde det højeste antal dage, besøgte knuder, knuder i den endelige vej og tog længst tid at køre, men brugte mindst plads. Grunden til at den besøger flest knuder og tager længst tid, skyldes at Bellman-Ford besøger alle knuder i en graf, som en del af dens algoritme. Det kan derfor have en indvirkning på algoritmens effektivitet. Med henblik på Bellman-Fords pladsforbrug, som var det laveste blandt algoritmerne, kan en forklaring være, at algoritmen ikke gør brug af nogen kø eller hob.

Hvis vi kigger på sammenligningen af tovejs algoritmerne og deres respektive algoritmer, ses det at tovejs algoritmerne bruger færre dage på at udføre en rute, samt at de besøger færre antal knuder og at knuder i den endelige vej er færre. Det ses også at tovejs algoritmerne er mere ressourcekrævende, hvilket bl.a. ses på pladsforbruget og tiden det tager, at køre algoritmerne. Disse resultater passer fint med, at en tovejs algoritme teoretisk set besøger færre knuder. Grunden til at tovejs algoritmerne tager længere tid, skyldes formentlig den konstante tid det tager at udføre operationerne i tovejs algoritmerne, da tiden sparet på det færre antal besøgte knuder ikke er nok til at ”dække” for dem. Som set på Figur 14, er det grundet tiden som det tager at oprette strukturen til tovejs Dijkstra, der gør den langsommere på den korte vej, hvilket ses på ”Overall Time - Setup Time” tallene. Til den længere vej, er det grundet tiden det tager at håndtere en enkelt knude, der er skyld i at tovejs søgealgoritmer tager længere tid, hvilket ses på ”Average V Operation Time” tallene.

7.1 Forskel mellem spring i fjende frekvens

Med henblik på Tabel 7, 13, 5, 11 og 9 i Bilag A.1 kan det udledes, at fjende frekvensen (chancen for at en fjende optager en provins på ruten) påvirker pladsforbruget og antal besøgte knuder markant. En formodentlig grund kan være at algoritmen skal køre forfra, hver gang en fjende optager en provins på den hidtil fundne vej. Dette er også et forventet resultat, idet algoritmernes besøgte knuder samt pladsforbrug bliver summereret for hver opdatering på grafen. Jo højere fjende frekvens, jo flere opdateringer på grafen, som logisk set resulterer i brug for mere hukommelse og kiggen på knuder.

Det er værd at lægge mærke til fordelingen af korte, mellem og lange ruter.

```

Short route:
Dijkstra n = 451
Overall Time: 0.0364079475402832
Setup Time: 0.02743220329284668
Overall - Setup Time: 0.008975744247436523
Average V Operation Time: 1.9901871945535527e-05

Bidirectional Dijkstra n = 163
Overall Time: 0.04487943649291992
Setup Time: 0.041887760162353516
Overall - Setup Time: 0.0029916763305664062
Average V Operation Time: 3.693427568600501e-05
-----
Long route:
Dijkstra n = 3728
Overall Time: 0.08377599716186523
Setup Time: 0.019946575164794922
Overall - Setup Time: 0.06382942199707031
Average V Operation Time: 1.685404470550144e-05

Bidirectional Dijkstra n = 1995
Overall Time: 0.12865614891052246
Setup Time: 0.036901235580444336
Overall - Setup Time: 0.09175491333007812
Average V Operation Time: 9.20310063491255e-05

```

Figur 14: Tal på køretiden af vores implementation for Dijkstra og tovejs Dijkstra på en kort og en lang vej, hvor n betegner besøgte knuder, overall betegner køretiden for hele algoritmen, setup betegner køretiden for opsætningen af datastrukturen og average V operation betegner den gennemsnitlige tid det tager at håndtere en knude

I alle resultaterne for eksperimentet, har algoritmerne kun 11 korte ruter i kvalificeringen. Her kan vi dog udelukke Bellman-Ford fra iagttagelsen, da vi kun har kørt 1 kort, 1 mellem og 1 lang rute i eksperimentet. Selve årsagen til at der er få korte ruter i eksperimentet skyldes klart, at vi i koden udvælger tilfældige start- og stop provinser, som algoritmen herefter skal arbejde på. Her er chancen for at få udvalgt to provinser, som ligger tæt på hinanden ikke lige så stor, som chancen for at de lå langt fra hinanden. Iagttagelsen om at der forekommer lav kvantitet af korte ruter, er værd at resonere om, da eksperimentets kvalitet øges ved forekomster af flere korte ruter. Dog fortæller resultaterne for de korte ruter noget om den generelle tendens.

Tages der tages udgangspunkt i de lange ruter og med henblik på antal besøgte knuder, er der en væsentlig forskel på hvordan tallene stiger mellem algoritmerne. Dette ser vi nærmere på i tabellen herunder.

Algoritme	Procentvis ændring (0-10%)	Procentvis ændring (10-30%)	Procentvis ændring (30-50%)
Dijkstra	864,8902%	232,8150%	214,3505%
A*	689,7145%	290,4750%	195,9143%
Tovejs Dijkstra	650,4561%	293,4837%	165,9479%
Tovejs A*	651,1026%	289,0087%	161,3250%
Bellman-Ford	800,0000%	550,0000%	222,7273%

Tabel 1: Procentvis ændring i antal besøgte knuder på *lange* ruter. Viser ændringen hvor fjende frekvensen går fra henholdsvis 0-10%, 10-30% og 30-50%.

Som det kan ses i Tabel 1 er der store udsving i den procentvise ændring af antal knuder, såfremt vi går fra en fjende frekvens på 0% til en fjende frekvens på 10%, når vi sammenligner algoritmerne. Bellman-Ford er oppe på en stigning på 800%, når vi går fra 0% fjendefrekvens til 10%. Det er forventet at Bellman-Ford har den største procentvise ændring her, da algoritmen skal besøge alle knuder i grafen for hver kørsel. Årsagen til heltal er fordi Bellman-Ford har kørt 8 gange, ergo er der sket opdateringer på grafen 8 gange. Dette tal er ikke et gennemsnit, da vi kun har kørt én rute for Bellman-Ford.

Ses der i stedet på algoritmen med den mindste procentvise ændring, ved fjende frekvens 0-10%, er det umiddelbart tovejs Dijkstra der er tale om. Her kan det udledes, at denne algoritme påvirkes mindst i antal besøgte knuder, når vi går fra statisk til dynamisk ved 10% fjende frekvens. Vi kan dog se at tovejs A* har en procentvis ændring i denne kolonne, som er ca. ét procentpoint højere end for tovejs Dijkstra. Det er værd at bide mærke i at forskellen på den procentvise ændring for tovejs-versionerne, er betydeligt mindre end forskellen ved mellem A* og Dijkstra (ikke-tovejs-versionerne).

Desuden kan vi se på søjlen (*Procentvis ændring (10-30%)*) i Tabel 1, at der ikke er ligeså store udsving blandt algoritmerne, når vi ser på springet mellem 10 og 30% fjende frekvens i den procentvise ændring (der ses bort fra Bellman-Ford). Af dette kan vi udlede at den største forskel mellem algoritmerne, på den procentvise ændring i antal besøgte knuder, forekommer ved øjeblikket, hvor grafen går fra statisk til dynamisk. Dette betyder også, at der er markant forskel på, hvordan de statiske algoritmer forholder sig, når de bliver anvendt i en dynamisk sammenhæng. Her skal det dog nævnes at tallene også er påvirket af tilfældigheden i fjendernes bevægelse.

Med fokus på den sidste søjle i Tabel 1 ses den procentvise ændring i springet fra 30-50% fjende frekvens. Tendensen i denne søjle fortæller at den procentvise ændring generelt er lavere end springet fra 10-30% fjende frekvens. I denne sammenhæng ses en større ændring i antal knuder besøgt ved justeringer i lille chance, kontra justeringer i mellem/høj chance for at fjender opstår på ruten. Det kan også udledes at tovejs Dijkstra og tovejs A* har den laveste procentvise ændring i denne søjle, hvorfor de må være mindst påvirket af fjende frekvensen ud af de fem algoritmer.

7.2 Teoretiske grænser

Vi har valgt at sammenligne resultaterne fra vores eksperimenter med teoretiske grænser. I forbindelse med den dynamiske graf i EU4, giver det mening at tilføje en ekstra variabel til alle algoritmers tidskompleksitet. Denne variabel skal være et heltal, hvis værdi er antallet af opdateringer på grafen. Den statiske teoretiske grænse (tidskompleksitet) skal derfor ganges med denne variabel. Vi kalder denne variabel for U (eng: updates). Tidskompleksiteten for eksempelvis A* ville derfor se ud på følgende måde: $O(((|V| + |E|) \lg |V|)U)$. Ud fra vores undersøgelse har vi i Tabel 2 rangeret algoritmernes tidskompleksitet på følgende måde (hvor 1 er bedst): Her ville det være interessant at sætte fokus på

Rang	Algoritme	Tidskompleksitet
1	Tovejs Dijkstra	$O(\frac{1}{2}(V + E) \lg V))$
2	Tovejs A*	$O(\frac{1}{2}(V + E) \lg V))$
3	Dijkstra	$O((V + E) \lg V)$
4	A*	$O((V + E) \lg V)$
5	Bellman-Ford	$O(V \cdot E)$

Tabel 2: Rangeringer af algoritmer ift. tidskompleksitet.

om algoritmerne efterlever denne rangering, når anvendt i EU4 sammenhæng. Af flere årsager har vi ikke mulighed for at få et svar på dette i vores projekt. For det første er tidskompleksiterne i Tabel 2 baseret på algoritmernes køretid i

tilfældet, hvor vi vil finde alle korteste veje fra startknuden til de andre knuder i grafen. Vores eksperiment bunder i vores implementation, hvor vi (som tidligere nævnt) stopper algoritmen, når vi er nået stopknuden. De ovenstående øvre grænser bliver derfor aldrig nået i vores case med EU4. Dette gør sig dog ikke gældende for Bellman-Ford, da den i vores implementation gennemsøger hele grafen. Et andet aspekt, som gør at vi får utilregnelige resultater, har sit fokus i tilfældigheden for fjendernes bevægelse. Som tidligere nævnt kan en algoritme være ”heldigere” på en specifik rute mellem start og stop, end en anden algoritme.

Eftersom vi netop stopper søgningen for de andre algoritmer, giver det mere mening at se på en ”lavere” øvre grænse. Denne grænse kender vi ikke, men vi har et kvalificeret gæt. Dette gæt bunder i *antal knuder i den endelige vej*, som kan ses i Tabel 7, 13, 5, 11 og 9 i Bilag A.1 for alle algoritmerne. Her kunne en øvre grænse for antal besøgte knuder for eksempelvis Dijkstras algoritme være $O(\pi K^2) = O(K^2)$, hvor $K =$ mindste antal knuder i den endelige vej. Vi er nået frem til gættet, da Dijkstras algoritme i teorien maksimalt besøger de knuder, som ligger i et cirkelareal rundt om kildeknuden, hvor radius på cirklen er K . I forbindelse med dette gæt kan vi ydermere gætte på, at køretiden ville være øvre grænset af $O((K^2 \lg K^2)U)$, da vi nu kun besøger K^2 antal knuder. Her er $U =$ antal opdateringer på grafen, som beskrevet i Kapitel 4. For at dette gæt holder i teorien, bliver vi nødt til at antage, at knuderne i grafen er fordelt uniformt. Dvs. at hvis vi ser på et udsnit u_1 af grafen, hvor størrelsen er s , skal der i et andet udsnit u_2 af størrelsen s forekomme lige mange knuder i udsnittet. Denne abstraktion gør sig ikke gældende i vores graf, da kantvægten mellem knuderne ikke nødvendigvis er fordelt på grafen som den euklidiske afstand mellem provinserne fordelt i verdenskortet. Dette er dog det tætteste vi kommer på et kvalificeret gæt.

Et gæt på en øvre grænse for antal knuder besøgt ved tovejs Dijkstra ville være $O(\frac{1}{2}(\pi K^2)) = O(\frac{1}{2}(K^2))$, da den ene del af algoritmen starter med at søge fra startknuden og den anden fra stopknuden. Et gæt på køretiden ville derfor være $O(\frac{1}{2}(K^2 \lg K^2)U)$. Det samme forhold af grænserne gør sig gældende for henholdsvis A* og tovejs A*. Grunden til at vi ikke har et bud på en lavere grænse for algoritmer som anvender heuristikken, er fordi vi anvender en tillidelig nedskaleret heuristik. Ud fra resultaterne af eksperimentet kan det ses at A* i gennemsnit besøger færre knuder end Dijkstra, men forskellen er så minimal at vi ikke har et bud på en lavere grænse end dette for A*.

Vi kan i Bilag A.1 se at vores gæt på de øvre grænser ikke stemmer overens med resultaterne, og dette skyldes klart antagelsen om at knuderne er fordelt uniformt - hvilket de ikke er i spillet. I stedet for kan vi se på en nedre grænse for antal knuder i den endelige vej blandt algoritmerne. Vi kan igen se på $K =$ mindste antal knuder i den endelige vej. Denne værdi findes ved fjende frekvens 0%, og den nedre grænse ville derfor være $\Omega(K)$. Her er det interessant at se hvor meget algoritmerne afviger fra den nedre grænse, når fjende frekvensen stiger.

Med udgangspunkt i Tabel 3 ses den procentvise afvigelse fra den nedre grænse

Algoritme	Procentvis afvigelse (0 vs. 10%)	Procentvis afvigelse (0 vs. 30%)	Procentvis afvigelse (0 vs. 50%)
Dijkstra	20,5063%	35,2553%	75,1550%
A*	18,0770%	44,4513%	62,6569%
Tovejs Dijkstra	11,9871%	25,2293%	37,9836%
Tovejs A*	12,4955%	27,1947%	39,7502%
Samlet gennemsnit	15,7665%	33,0326%	53,8864%

Tabel 3: Procentvis afvigelse i antal knuder i endelige vej. Baseret på gennemsnit ud af de 100 ruter. Bellman-Ford udeladt pga. for lille teststørrelse.

$\Omega(K)$. Vi har valgt at udelade Bellman-Ford fra denne sammenligning, da der kun er kørt 3 ruter for algoritmen og dermed fortæller tallene ikke meget om den generelle tendens. Afvigelserne er forskellige blandt algoritmerne udelukkende pga. tilfældigheden i fjendernes bevægelse, hvorfor det er uhensigtsmæssigt at sammenligne algoritmerne på denne basis. Det giver derfor mere mening at se på det samlede gennemsnit af de forskellige afvigelser. Dette kan også ses i Tabel 3, hvor en interessant tendens er, at afvigelsen stiger mest ved springet fra fjende frekvens 10% til 30%. Her tales der om ca. en fordobling (15,7665% vs. 33,0326%) når vi regulerer fjende frekvensen op med 20%, fra 10% til 30%. Reguleres der yderligere 20% op i fjende frekvens (fra 30% til 50%) har vi ikke længere at gøre med en fordobling, men en stigning af lavere karakter. Af dette kan det udledes at forskellen blandt afvigelser, er størst ved spring i lave fjende frekvenser.

Ydermere kan vi se på en nedre grænse $\Omega(D)$ for mindste antal dage det tager at gennemføre den korteste vej. Denne værdi kan også findes ved fjende frekvens 0%. Den procentvise afvigelse fra den nedre grænse, baseret på antal dage, kan ses i Tabel 4. Igen er afvigelserne forskellige blandt algoritmerne udelukkende pga. tilfældigheden i fjendernes bevægelse, hvorfor det her også er uhensigtsmæssigt at sammenligne algoritmerne på denne basis. Af denne årsag ser vi igen på det samlede gennemsnit, hvor afvigelserne fra $\Omega(D)$ ser ud til at være større end for den tidlige omtalte grænse $\Omega(K)$. Det kan dog udledes at tendensen for forskellen mellem afvigelserne i Tabel 4 ligner tendensen for Tabel 3. Vi ser ca. en procentvis fordobling fra fjende frekvens 10% til 30%, men en mindre stigning ved 30% til 50%. Af netop denne årsag ser det ud til at tendensen for antal knuder i den endelige vej følges/korrelerer med tendensen for antal dage det tager at gennemføre den korteste vej.

Algoritme	Procentvis afvigelse (0 vs. 10%)	Procentvis afvigelse (0 vs. 30%)	Procentvis afvigelse (0 vs. 50%)
Dijkstra	25,5309%	45,7307%	97,6401%
A*	22,8601%	56,5046%	83,0879%
Tovejs Dijkstra	16,6095%	35,3980%	56,0626%
Tovejs A*	17,0975%	39,4101%	61,6306%
Samlet gennemsnit	20,5245%	44,2608%	74,6053%

Tabel 4: Procentvis afvigelse i antal dage det tager at gennemføre den korteste vej. Baseret på gennemsnit ud af de 100 ruter. Bellman-Ford udeladt pga. for lille teststørrelse.

8 Konklusion

Efter udførelsen af vores eksperiment og vores analyse af resultaterne, er vi kommet tættere på hvilke algoritmer der virker mest effektivt på en graf som den i Europa Universalis 4. Det er umiddelbart tovejs-versionerne af Dijkstra og A*, der efterlader sig de bedste resultater når den vigtigste faktor er prisen for ruten. Denne pris er, som tidligere nævnt, målt i enheden *antal dage*, da denne oversætter bedre til egenskaberne for EU4. Forskellen i antal dage per algoritme var en overraskelse, dog fandt vi, som tidligere nævnt, en forklaring på dette. Fra dette bemærker vi dog at det er tovejs algoritmerne der konsekvent finder endelige veje, der tager kortere dage at gennemføre. Dette får os til at tro at der muligvis er uforsættige elementer ved tovejs algoritmerne, der leder til dette. Et gæt på dette, er antallet af knuder i den valgte optimale vej. Færre knuder på den valgte optimale vej, leder til færre mulige utilgængelige knuder. Hvad det dog egentligt kunne være, har vi ikke et svar på.

Derudover er en stor fordel ved anvendelsen af tovejs-versionerne kontra de oprindelige Dijkstra og A* versioner, at der bliver besøgt langt færre knuder. Dette burde samtidig gøre køretiden for tovejs algoritmerne hurtigere, men dette er ikke tilfældet i vores eksperiment, da tovejs algoritmerne bl.a. ikke er implementeret sideløbende/parallelt (eng: concurrent) og at grafen ikke er stor nok. Selvom tovejs algoritmerne i vores eksperiment bruger flere ressourcer end de oprindelige, er vi stadig tilbøjelige til at sige, at de egner sig bedst til brug i EU4. Denne tilbøjelighed bunder i at tidsforskellen er så minimal, at der ikke er en observerbar forskel i selve spillet. EU4 kræver kvik hardware for at kunne spilles, og med nutidens moderne maskiner, er den minimale tidsforskell ligegyldig for spiloplevelsen. Havde forskellen været betydeligt større, ville det give mere mening at fravælge tovejs algoritmerne til fordel for de oprindelige.

Jævnfør analysen af resultaterne fra eksperimentet er vi på nuværende tidspunkt i stand til at foretrakke nogle algoritmer frem for andre, når specifikke

situitioner opstår. Vælger spilleren eksempelvis at rykke sine tropper til en tætliggende provins, kan vi ud fra resultaterne i Tabel 7, 13, 5, 11 og 9 i Bilag A.1 fortrække at anvende tovejs A*, hvis fjende frekvensen er 10%. Vælger spilleren derimod at flytte sine tropper til en fjern provins, når fjende frekvensen er 50%, er det også tovejs A* der er det foretrukne valg af algoritme. Her besøges der både færre knuder end for de andre algoritmer, og der vejen tager færre dage. Ved en fjende frekvens på 30% hvor spilleren flytter sine tropper til en mellemliggende provins, er det umiddelbart også tovejs A*, der er det foretrukne valg. Det skal dog nævnes at forskellen på antal dage er så minimal her, at tovejs Dijkstra også kunne vælges, da den besøger færre knuder. I de resterende tilfælde vil tovejs Dijkstra altid være den foretrukne algoritme at anvende til, at finde den korteste vej i EU4. Det skal dog nævnes, at disse observationer og anbefalinger bunder i vores resultater, hvor algoritmerne ikke står på lige fod, grundet tilfældigheden i fjendernes bevægelse.

8.1 Eventuelle forbedringer

Med henblik på hvad der kunne være gjort anderledes i vores projekt, giver det mening at tale om de valg vi har truffet undervejs. Da vi ikke kunne finde noget data i spilfilerne omkring kanter, har vi (som tidligere nævnt) selv opdaget disse. I forhold til vægten af kanterne har vi benyttet os af attributten `movement_cost`, hvilken kan slås op for alle knuder i grafen. Her har vi bestemt at vægten for en kant K mellem knuderne A og B er bestemt ud fra:

$$vægt(K) = \text{movement_cost}(A) + \text{movement_cost}(B)$$

I stedet for denne løsning, kunne vi have ladet kanterne være uvægtede og holde vægten hos knuderne. Her ville algoritmerne i stedet for skulle se på de mulige provinser, der kunne bevæges til, ud fra den nuværende knude (den som spilleren står på). Valget ville derfor blive truffet ud fra knudernes vægt, nemlig bevægelsesomkostningen (eng: movement cost) i stedet for en kants vægt. Dette ville dog give det samme resultat, men implementationsmæssigt ville det være tætttere på hvordan vægtenes egenskaber ser ud i EU4.

Vedrørende eksperimentet har vi også truffet en række valg, som har været en bias igennem udførelsen af det. For det første har vi anvendt randomiseringsbiblioteket `random` til at udvælge de 100 tilfælde start- og stopprovinser, som algoritmerne skal arbejde ud fra. Vi har først kategoriseret de fundne ruters længde efter at den pågældende algoritme har termineret første gang. Her kunne der i stedet have været en kategorisering af start- og stopknuderne under udvælgelsen, hvor der i stedet ville blive set på den euklidiske afstand mellem start og stop. På denne måde kunne eksperimentet have ca. lige mange korte, mellem og lange ruter kategoriseret, inden algoritmen får tildelt start- og stop knuderne. Som tidligere nævnt ville eksperimentets resultater også kunne sige mere om den generelle tendens, hvis vi havde forøget antallet af start- og stopknuder. Det perfekte eksperiment ville være eksperimentet, hvor algoritmerne får tildelt

alle tænkelige start- og stopprovinser i grafen. Dette ville dog kræve en del tid at gennemkøre, men bias i form af udvælgelse ville være elimineret. Antallet af forskellige kombinationer af start- og stopprovinser kan vi udregne med følgende formel:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

hvor n = antal knuder i grafen og k = antal knuder behøvet for at danne en kombination. Vi får derfor følgende antal kombinationer af unikke start- og stopprovinser:

$$\binom{3925}{2} = \frac{3925!}{2!(3925-2)!} = 7.700.850$$

På baggrund af det høje antal kombinationer af unikke start- og stopprovinser, har vi ikke valgt at gennemkøre mere end de 100 tilfældigt udvalgte, grundet en tidsmæssig horisont. Havde vi undersøgt alle kombinationer af start og stop provinser, ville vi gætte på at den bedste algoritme (ud fra teorien) ville være tovejs A*. Dette er ikke tilfældet i vores eksperiment, da fjenderne (som tidligere nævnt) bevæger sig tilfældigt på kortet for alle algoritmerne. Denne egenskab for grafen gør, at en algoritme som tovejs Dijkstra kan opnå bedre resultater end tovejs A*, hvis den har været ”heldig” med fjenderne. Med en større teststørrelse ville tovejs A* derfor i teorien opnå enten samme resultat som tovejs Dijkstra, eller bedre.

Gennem implementationen og eksperimentet har vi omdiskuteret det vi kalder fjende frekvensen, hvor vi kan skrue op eller ned for chancen for, at en fjende kan optage en provins på ruten. Justeringen på denne frekvens bidrager til resultater, som fortæller noget om, hvilken betydning denne chance har for algoritmernes effektivitet. Som tidligere nævnt justeres fjende frekvensen med formålet at simulere områder med lav, mellem og høj fjendebættelse. Der bliver i vores implementation udvalgt tilfældige provinser på den aktuelle vej, hvor fjenderne bliver rejst, og de bevæger sig herefter til 4 tilfældige nabo provinser. Dette betyder at fjenderne, pga. tilfældighed, ikke har samme lokationer på den n ’te dag for alle algoritmerne. En måde at forbedre dette i et fremtidigt projekt, kunne være at tilfældigt udvælge et bestemt antal knuder fra en liste, der indeholder alle knuder i grafen, som værende ikke-tilgængelige. I simuleringen kunne man derfor nulstille denne liste for hver dag der går, og udvælge ikke-tilgængelige knuder på ny. På denne måde ville fjenders lokationer på de specifikke tidspunkter ikke være forskellige blandt algoritmerne, da man kan anvende et såkaldt ”seed”. Et seed gør at de tilfældige valg der bliver truffet, genskabes/er de samme for alle kørsler.

For at kunne måle pladsforbruget af algoritmerne, har vi gjort brug af hukommelses-diagnoseringsværktøjet `tracemalloc`. Under opsætningen af eksperimentet afprøvede vi forskellige værktøjer til at måle pladsforbruget, hvor det umiddelbart var `tracemalloc`, som gav de mest konsistente målinger. Måden at værktøjet fungerer på, er ved at starte en måling på linjen over funktionskaldet på den

funktion man vil undersøge pladsforbruget for, og på linjen efter funktionskallet stoppe målingen. Her kan man spørge værktøjet, hvor meget plads der blev allokeret mellem start og stop. Alternativt kunne man i selve kroppen for algoritmerne spørge, hvad hver enkelt allokeret variabel har af størrelse i hukommelsen, og summere disse størrelser.

Det er vores opfattelse at ægte dynamiske algoritmer skal bruge en større graf, end den vi har tilgængelig i EU4, til at få bedre resultater. Vi ville derfor gerne have haft en ægte dynamisk korteste vej algoritme med i eksperimentet, for at kunne be- eller afkræfte denne opfattelse.

Litteratur

- [1] Richard Bellman. "On a routing problem". I: *Quarterly of Applied Mathematics* 16 (1958), s. 87–90. URL: <https://www.cs.yale.edu/homes/lans/readings/routing/bellman-routing-1958.pdf>.
- [2] Thomas H. Cormen m.fl. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: <http://mitpress.mit.edu/books/introduction-algorithms>.
- [3] Edsger W Dijkstra m.fl. "A note on two problems in connexion with graphs". I: *Numerische mathematik* 1.1 (1959), s. 269–271.
- [4] Robert W. Floyd. "Algorithm 97: Shortest Path". I: *Commun. ACM* 5.6 (jun. 1962), s. 345. ISSN: 0001-0782. DOI: 10.1145/367766.368168. URL: <https://doi.org/10.1145/367766.368168>.
- [5] D. R. Ford og D. R. Fulkerson. "Flows in Networks". I: (1962). URL: <https://www.rand.org/content/dam/rand/pubs/reports/2007/R375.pdf>.
- [6] Peter E. Hart, Nils J. Nilsson og Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". I: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), s. 100–107. URL: <http://ai.stanford.edu/users/nilsson/OnlinePubs-Nils/PublishedPapers/astar.pdf>.
- [7] Donald B. Johnson. "Efficient Algorithms for Shortest Paths in Sparse Networks". I: *J. ACM* 24.1 (jan. 1977), s. 1–13. ISSN: 0004-5411. DOI: 10.1145/321992.321993. URL: <https://doi.org/10.1145/321992.321993>.
- [8] Jay Mahadeokar og Sanjeev Saxena. "Faster replacement paths algorithms in case of edge or node failure for undirected, positive integer weighted graphs". I: *Journal of Discrete Algorithms* 23 (2013). 23rd International Workshop on Combinatorial Algorithms (IWOCA 2012), s. 54–62. ISSN: 1570-8667. DOI: <https://doi.org/10.1016/j.jda.2013.08.001>. URL: <https://www.sciencedirect.com/science/article/pii/S1570866713000658>.
- [9] Ira Pohl. "Bi-directional Search". I: *Machine Intelligence* 6 (1971), s. 127–140. URL: <https://aitopics.org/download/classics:49B0B811>.
- [10] Paradox Wikis. *Province ID map*. [Online; accessed May 25, 2022]. 2021. URL: https://eu4.paradoxwikis.com/File:Province_ID_map.png.

A Bilag

A.1 Resultater

Tabel 5: Dijkstra resultater. FF = fjende frekvens i procent, P = antal veje. Alle data er gennemsnit.

Kort rute	FF = 0, P = 11	FF = 10, P = 11	FF = 30, P = 11	FF = 50, P = 11
Dage	21,0	23,1818	26,3636	57,0
Besøgte knuder	212,1818	400,6364	2349,0909	16532,3636
Tid (sek)	0,0654	0,142	0,6102	2,786
Pladsforbrug (MB)	0,9732	2,1175	8,7542	36,8543
Knuder i endelige vej	7,1818	7,6364	8,5455	16,2727
Opdateringer på vejen	1	1,1818	7,8182	34,3636
Mellem rute	FF = 0, P = 51	FF = 10, P = 51	FF = 30, P = 51	FF = 50, P = 51
Dage	45,1176	57,2353	70,7647	83,2941
Besøgte knuder	1268,9216	7773,0784	26367,902	48204,1961
Tid (sek)	0,1115	0,8815	2,8875	4,9421
Pladsforbrug (MB)	1,2203	10,3611	32,582	54,8854
Knuder i endelige vej	15,098	18,3529	21,6863	24,9608
Opdateringer på vejen	1	8,0784	27,7059	46,7843
Lang rute	FF = 0, P = 38	FF = 10, P = 38	FF = 30, P = 38	FF = 50, P = 38
Dage	75,2368	97,0263	108,8684	139,0789
Besøgte knuder	2912,7368	25191,9737	58650,6842	125718,0263
Tid (sek)	0,1785	2,0219	5,0973	10,4094
Pladsforbrug (MB)	1,2262	18,0302	48,4652	97,0586
Knuder i endelige vej	24,6842	30,6053	33,2895	41,0263
Opdateringer på vejen	1	14,3158	40,7895	81,8947
Alle ruter	FF = 0, P = 100	FF = 10, P = 100	FF = 30, P = 100	FF = 50, P = 100
Dage	47,1181	59,1478	68,6656	93,1243
Besøgte knuder	1464,6134	11121,8962	29122,5590	63484,8620
Tid (sek)	0,1185	1,0151	2,8650	6,0458
Pladsforbrug (MB)	1,1399	10,1696	29,9338	62,9328
Knuder i endelige vej	15,6547	18,8649	21,1738	27,4199
Opdateringer på vejen	1,0000	7,8587	25,4379	54,3475

Tabel 6: Dijkstra resultater. FF = fjende frekvens både 10, 30 og 50, P = antal veje. Alle data er gennemsnit af de ovenstående gennemsnit (kun de dynamiske).

Dynamiske ruter	FF = 10/30/50, P = 300
Dage	73,6459
Besøgte knuder	34576,4391
Tid (sek)	3,3087
Pladsforbrug (MB)	34,3454
Knuder i endelige vej	22,4862
Opdateringer på vejen	29,2147

Tabel 7: A* resultater. FF = fjende frekvens i procent, P = antal veje. Alle data er gennemsnit.

Kort rute	FF = 0, P = 11	FF = 10, P = 11	FF = 30, P = 11	FF = 50, P = 11
Dage	21,0	30,4545	43,0909	42,4545
Besøgte knuder	192,0	1095,0909	6347,0	9506,8182
Tid (sek)	0,0701	0,3839	1,3533	2,009
Pladsforbrug (MB)	1,1263	5,9591	18,6131	28,6081
Knuder i endelige vej	7,1818	9,5455	13,0909	12,1818
Opdateringer på vejen	1	4,1818	14,9091	23,3636

Mellem rute	FF = 0, P = 51	FF = 10, P = 51	FF = 30, P = 51	FF = 50, P = 51
Dage	45,1176	55,5294	72,451	82,0588
Besøgte knuder	1233,5294	5959,098	24036,6471	46609,8824
Tid (sek)	0,1162	0,7054	2,9403	5,4139
Pladsforbrug (MB)	1,368	8,6798	35,8651	64,8794
Knuder i endelige vej	15,098	17,9608	22,1176	24,3137
Opdateringer på vejen	1	5,7059	26,7451	49,3137

Lang rute	FF = 0, P = 38	FF = 10, P = 38	FF = 30, P = 38	FF = 50, P = 38
Dage	75,2368	87,6842	105,6842	134,2895
Besøgte knuder	2882,6842	19882,2895	57753,0789	113146,5263
Tid (sek)	0,185	1,75	5,2643	10,1988
Pladsforbrug (MB)	1,3787	17,21	52,28	101,5821
Knuder i endelige vej	24,6842	27,9474	32,6316	39,8947
Opdateringer på vejen	1	12,0263	38,7632	75,9737

Alle ruter	FF = 0, P = 100	FF = 10, P = 100	FF = 30, P = 100	FF = 50, P = 100
Dage	47,1181	57,8894	73,7420	86,2676
Besøgte knuder	1436,0712	8978,8261	29378,9087	56421,0756
Tid (sek)	0,1238	0,9464	3,1860	5,8739
Pladsforbrug (MB)	1,2910	10,6163	35,5861	65,0232
Knuder i endelige vej	15,6547	18,4846	22,6134	25,4634
Opdateringer på vejen	1,0000	7,3047	26,8058	49,5503

Tabel 8: A* resultater. FF = fjende frekvens både 10, 30 og 50, P = antal veje. Alle data er gennemsnit af de ovenstående gennemsnit (kun de dynamiske).

Dynamiske ruter	FF = 10/30/50, P = 300
Dage	72,6330
Besøgte knuder	31592,9368
Tid (sek)	3,3354
Pladsforbrug (MB)	37,0752
Knuder i endelige vej	22,1871
Opdateringer på vejen	27,8869

Tabel 9: Tovejs Dijkstra resultater. FF = fjende frekvens i procent, P = antal veje. Alle data er gennemsnit.

Kort rute	FF = 0, P = 11	FF = 10, P = 11	FF = 30, P = 11	FF = 50, P = 11
Dage	21,0	23,7273	27,4545	28,3636
Besøgte knuder	86,8182	199,8182	781,6364	1096,3636
Tid (sek)	0,1094	0,5171	1,3089	2,064
Pladsforbrug (MB)	1,9474	8,4811	22,2566	34,9727
Knuder i endelige vej	7,1818	7,7273	8,5455	8,2727
Opdateringer på vejen	1	3,3636	10,4545	17
Mellem rute	FF = 0, P = 51	FF = 10, P = 51	FF = 30, P = 51	FF = 50, P = 51
Dage	45,1176	54,0	63,6471	70,8431
Besøgte knuder	485,7843	2989,1765	6940,8039	13234,9608
Tid (sek)	0,1367	1,6118	3,8035	6,3357
Pladsforbrug (MB)	1,9804	24,8499	60,0022	97,1719
Knuder i endelige vej	15,098	17,3137	19,3725	20,451
Opdateringer på vejen	1	11,6275	29,6471	48,1176
Lang rute	FF = 0, P = 38	FF = 10, P = 38	FF = 30, P = 38	FF = 50, P = 38
Dage	75,2368	87,1053	100,2895	121,3947
Besøgte knuder	1517,5789	9871,1842	28970,3158	48075,6316
Tid (sek)	0,2187	2,612	7,1594	12,6209
Pladsforbrug (MB)	2,2683	35,3479	97,0838	170,8343
Knuder i endelige vej	24,6842	27,5526	30,8947	36,0789
Opdateringer på vejen	1	16,1842	45,9474	82
Alle ruter	FF = 0, P = 100	FF = 10, P = 100	FF = 30, P = 100	FF = 50, P = 100
Dage	47,1181	54,9442	63,7970	73,5338
Besøgte knuder	696,7271	4353,3930	12230,9187	20802,3187
Tid (sek)	0,1549	1,5803	4,0906	7,0069
Pladsforbrug (MB)	2,0654	22,8930	59,7809	100,9930
Knuder i endelige vej	15,6547	17,5312	19,6042	21,6009
Opdateringer på vejen	1,0000	10,3918	28,6830	49,0392

Tabel 10: Tovejs Dijkstra resultater. FF = fjende frekvens både 10, 30 og 50, P = antal veje. Alle data er gennemsnit af de ovenstående gennemsnit (kun de dynamiske).

Dynamiske ruter	FF = 10/30/50, P = 300
Dage	64,0917
Besøgte knuder	12462,2101
Tid (sek)	4,2259
Pladsforbrug (MB)	61,2223
Knuder i endelige vej	19,5788
Opdateringer på vejen	29,3713

Tabel 11: Tovejs A* resultater. FF = fjende frekvens i procent, P = antal veje.
Alle data er gennemsnit.

Kort rute	FF = 0, P = 11	FF = 10, P = 11	FF = 30, P = 11	FF = 50, P = 11
Dage	21,0	23,1818	29,4545	37,4545
Besøgte knuder	84,4545	260,4545	894,1818	1908,2727
Tid (sek)	0,1332	0,6781	1,9889	4,0902
Pladsforbrug (MB)	2,2535	10,8404	30,2838	61,1566
Knuder i endelige vej	7,1818	7,5455	8,9091	10,1818
Opdateringer på vejen	1	3,8182	12,4545	26,1818

Mellem rute	FF = 0, P = 51	FF = 10, P = 51	FF = 30, P = 51	FF = 50, P = 51
Dage	45,1176	54,2353	63,6078	71,3333
Besøgte knuder	479,8627	2857,3725	7279,3333	12600,6078
Tid (sek)	0,1614	1,6762	4,8919	8,4428
Pladsforbrug (MB)	2,2863	26,301	69,3045	114,5277
Knuder i endelige vej	15,098	17,3922	19,3529	20,5294
Opdateringer på vejen	1	10,5686	29,549	49,3333

Lang rute	FF = 0, P = 38	FF = 10, P = 38	FF = 30, P = 38	FF = 50, P = 38
Dage	75,2368	88,1053	104,0	119,6842
Besøgte knuder	1515,5263	9867,6316	28518,3158	46007,1842
Tid (sek)	0,2422	2,917	9,5341	15,9562
Pladsforbrug (MB)	2,574	40,9905	120,9036	193,5028
Knuder i endelige vej	24,6842	27,8947	31,4737	34,9211
Opdateringer på vejen	1	16,3684	50,2632	80,9737

Alle ruter	FF = 0, P = 100	FF = 10, P = 100	FF = 30, P = 100	FF = 50, P = 100
Dage	47,1181	55,1741	65,6874	76,1573
Besøgte knuder	693,2812	4328,4862	12230,6103	20172,0216
Tid (sek)	0,1789	1,7571	5,4716	9,4964
Pladsforbrug (MB)	2,3713	26,0440	73,4973	123,0624
Knuder i endelige vej	15,6547	17,6108	19,9119	21,8774
Opdateringer på vejen	1,0000	10,2517	30,7556	52,1629

Tabel 12: Tovejs A* resultater. FF = fjende frekvens både 10, 30 og 50, P = antal veje. Alle data er gennemsnit af de ovenstående gennemsnit (kun de dynamiske).

Dynamiske ruter	FF = 10/30/50, P = 300
Dage	65,6730
Besøgte knuder	12243,7060
Tid (sek)	5,5750
Pladsforbrug (MB)	74,2012
Knuder i endelige vej	19,8000
Opdateringer på vejen	31,0567

Tabel 13: Bellman-Ford resultater. FF = fjende frekvens i procent, P = antal veje. Alle data er gennemsnit.

Kort rute	FF = 0, P = 1	FF = 10, P = 1	FF = 30, P = 1	FF = 50, P = 1
Dage	34	34	63	67
Besøgte knuder	3925	11775	94200	133450
Tid (sek)	31,3843	89,9929	782,6032	1171,6937
Pladsforbrug (MB)	0,1656	0,4920	3,9196	5,5530
Knuder i endelige vej	10	10	16	17

Mellem rute	FF = 0, P = 1	FF = 10, P = 1	FF = 30, P = 1	FF = 50, P = 1
Dage	65	79	94	118
Besøgte knuder	3925	58875	176625	274750
Tid (sek)	31,0663	459,6326	1597,5412	2507,2910
Pladsforbrug (MB)	0,1657	2,4503	7,3473	11,4274
Knuder i endelige vej	21	25	28	35

Lang rute	FF = 0, P = 1	FF = 10, P = 1	FF = 30, P = 1	FF = 50, P = 1
Dage	88	88	118	156
Besøgte knuder	3925	31400	172700	384650
Tid (sek)	30,3112	244,4209	1603,6280	3064,3054
Pladsforbrug (MB)	0,1657	1,3084	7,1853	16,0021
Knuder i endelige vej	30	30	39	48

Alle ruter	FF = 0, P = 3	FF = 10, P = 3	FF = 30, P = 3	FF = 50, P = 3
Dage	62,3333	67,0000	91,6667	113,6667
Besøgte knuder	3925,0000	34016,6667	147841,6667	264283,3333
Tid (sek)	30,9206	264,6821	1327,9241	2247,7634
Pladsforbrug (MB)	0,1657	1,4169	6,1507	10,9942
Knuder i endelige vej	20,3333	21,6667	27,6667	33,3333

Tabel 14: Bellman-Ford resultater. FF = fjende frekvens både 10, 30 og 50, P = antal veje. Alle data er gennemsnit af de ovenstående gennemsnit (kun de dynamiske).

Dynamiske ruter	FF = 10/30/50, P = 300
Dage	90,7778
Besøgte knuder	148713,8889
Tid (sek)	1280,1232
Pladsforbrug (MB)	6,1873
Knuder i endelige vej	27,5556