Riku Ville Olavi Laitinen

Polunetsinnän algoritmit ja niiden tehokkuus

Tietotekniikan kandidaatintutkielma

25. toukokuuta 2021

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Riku Ville Olavi Laitinen

Yhteystiedot: riviolla@student.jyu.fi

Ohjaaja: Rossi Tuomo

Työn nimi: Polunetsinnän algoritmit ja niiden tehokkuus

Title in English: Pathfinding algorithms and their effectiveness

Työ: Kandidaatintutkielma

Opintosuunta: Tietotekniikka

Sivumäärä: 24+0

Tiivistelmä:

Tässä tutkimuksessa tutkitaan erilaisten polunetsinnän algoritmeja ja niiden tehokkuutta 2D-videopelimaailmassa. Tutkittavat algoritmit ovat A*-algoritmi, Dijkstran algoritmi, Breadth-first algoritmi, Depth-first algoritmi ja kuinka näitä algoritmeja voidaan hyödyntää hierarkkisen polunsuunnittelun kanssa.

Avainsanat: Algoritmit, Depth-first, Breadth-first, Dijkstran, A-star, Kandidaatintutkielma

Abstract:

In this thesis we research different pathfinding algorithms and their efficiency in 2D-video game environment. The algorithms used in this thesis are A-star algorithm, Dijkstra algorithm, Breadth-first algorithm, Depth-first algorithm and how these algorithms can be used in hierarchical path planning.

Keywords: Algorithms, Depth-first, Breadth-first, Dijkstran, A-star, Bachelor's Theses

Kuviot

Sisällys

1 Johdanto

Videopelit kasvavat vuosi vuodelta kooltaan suurimmiksi ja suorituskykyä vaativimmiksi. Enää ei kelpaa yksittäiset tietokoneen ohjaamat pelihahmot ja pienet kartat, vaan kaikkea täytyy olla enemmän ja kaiken täytyy olla suurempaa. Tämä aiheuttaa suuren rasitteen tietokoneiden suorituskyvylle, kun sekä liikkuvia entiteettejä lisätään ja karttoja kokoa kasvatetaan samaan aikaan.

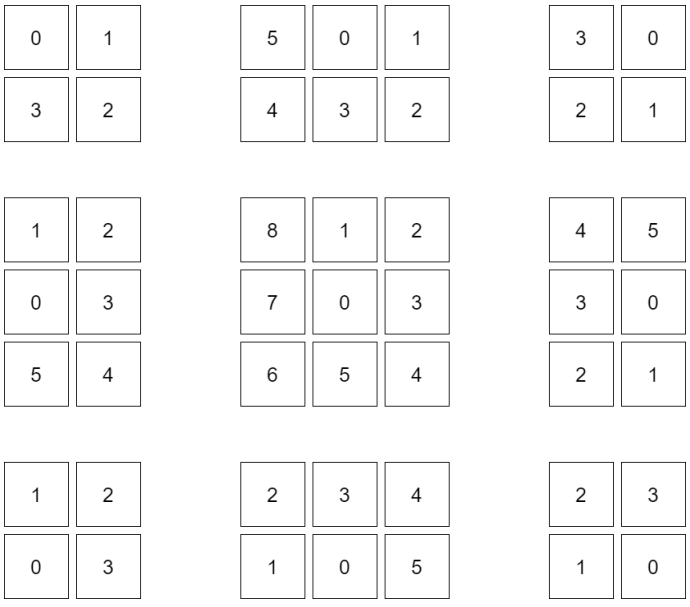
Vuosien varrella on kehitetty monenlaisia algoritmeja, joilla pystytään liikuttamaan kartalla olevia entiteettejä tehokkaasti paikasta a paikkaan b. Tosin jotkin näistä algoritmeista selviytyvät tehtävästään paremmin kuin toiset, mutta kaikilla näillä algoritmeilla on sama ongelma, kun kartan kokoa kasvatetaan tarpeeksi, joka on tehokkuus. Me emme pysty ennustamaan kuinka kauan, jokaisella algoritmilla menee löytää reitti kahden pisteen väliltä, mutta me voimme laskea niiden pahimman mahdollisen tehokkuuden.

Tässä tutkimuksessa tutkitaan neljän algoritmin tehokkuutta valtavissa ruudukkokartoissa, hyödyntäen hierarkkista polun suunnittelua ja ilman. Tutkimusta varten neljästä algoritmista kaksi hyödyntää solmujen välistä etäisyyttä ja kaksi ei.

2 Algoritmit ja niiden tehokkuus

Algoritmit ovat menetelmiä tai tekniikoita joilla ratkaistaan ongelmia askel askeleelta. Tässä kappaleessa tutustutaan ensiksi Depth-first search algoritmiin (DFS), Breadth-first search algoritmiin (BFS), Dijkstran algoritmiin (DA) ja viimeisenä A-star (A*) algoritmiin yleisellä tasolla. Jokaisen algoritmin kohdalla käydään läpi niiden toimivuus ja pohditaan niiden tehokkuutta suurissa kartoissa.

Tutkimuksessa käytetään neljää erikokoista ruudukkokarttaa 64 x 64, 128 x 128, 256 x 256 ja 512 x 512. Nämä ruudukkokartat koostuvat yksittäisistä ruuduista. Ruutuihin voi liikkua sivuttain, pystypäin ja viistossa tällöin jokaisella ruudulla on kolme, viisi tai kahdeksan kaarta riippuen sen sijainnista kartalla (ks. kuvio 1). 512 x 512 kokoisella kartalla on 262144 solmua ja (260 100 * 8 + 510 * 4 * 5 + 4 * 3) 2 091 012 kaarta. Kuviossa 1 näkyy kuinka solmulla 0 on kaari yhteydet, jokaiseen sen ympärillä olevaan solmuun 1-8.



Kuvio 1. Vektorikartan solmut

Me tulemme käyttämään yleisiä määritelmiä verkkoteoriasta, jossa vektorikartta on $G = (V, E)$ missä V on kartan solmut ja E on kaaret. Dijkstar ja A* algoritmeilla on paino funktio w, jolla annetaan jokaiselle kaarelle positiivinen paino.

2.1 Depth-first algoritmi

Depth-first tai toisella nimellä tunnettu takaperin etsintä on tekniikka, jolla etsitään "syvemmältä" kaaviosta (ks. kuvio 2). Algoritmissa valitaan aloitussolmusta yksi kaari, josta siirrytään toiseen solmuun ja merkitään se käydyksi. Jos solmussa on useampi kuin yksi kaari, valitaan satunnaisesti jokin niistä ja toistetaan niin kauan, kunnes tullaan solmuun, jossa ei ole enää kaaria. Solmussa jossa kaaria ei ole, siirrytään takaisin aikaisempaan solmuun ja tarkistetaan ne kaaret joissa algoritmi ei ole vielä käynyt. Tätä toistetaan kunnes tullaan takaisin sellaiseen solmuun, jossa on käymättömiä kaaria. Tilanteessa jossa päädytään sellaiseen solmuun, jossa algoritmi on käynyt, toteutetaan sama takaisin kulkeminen. Algoritmi päättyy kunnes palataan solmuun, josta aluksi lähdettiin ja algoritmin tehtävän on silloin tehty. (Tarjan 1972; Cormen ym. 2009a)

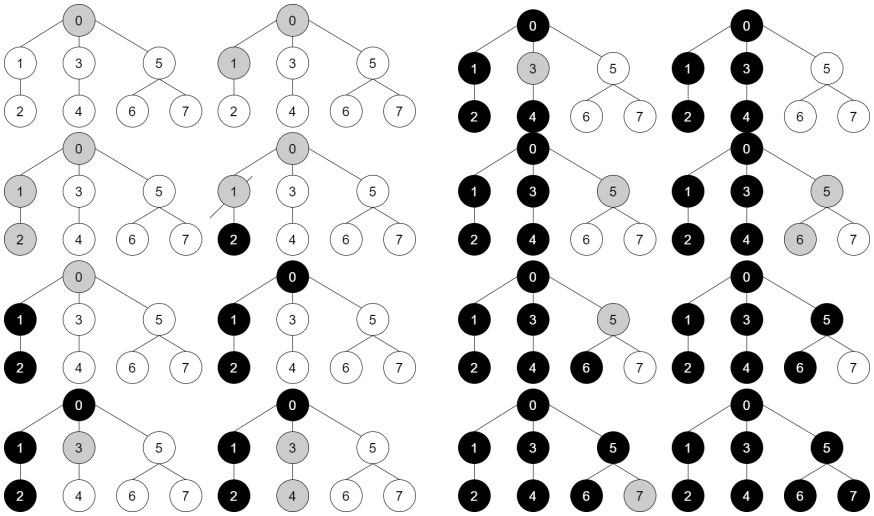
Seuraava pseudokoodi on yksinkertainen depth-first etsintäalgoritmi, joka on mukaelma (Cormen ym. 2009b) pseudokoodista.

2.1:

```
DFS(G, s)
    G -> Graafi
    s -> Aloitussolmu
begin
    s.kayty = true
    for each vertex in G
        if s.kayty == false
            DFS(G, vertex)
end

init() {
    for each s in G
        s.kayty = false
    DFS(G, s)
}
```

Pseudokoodissa merkataan aluksi silmukassa jokaisen G-graafin solmun arvoksi epätosi, jonka jälkeen kutsutaan DFS-funktiota. DFS-funktiossa asetetaan aloitussolmun s.kayty arvoksi tosi ja käydään jokainen G:n solmun läpi. Jos G:stä löytyy solmu jossa ei ole käyty, DFS-funktiota kutsutaan uudestaan rekursiolla. Jokainen G:n solmu on käyty läpi kun ohjelma päättyy.

Depth-first algoritmin tehokkuudeksi tulee $O(V+E)$, missä V on silmukoiden määrä ja E on kaarten määrä. Tehokkuuden laskemisessa oletetaan, että pahimmassa tapauksessa jokainen solmu ja kaari joudutaan tutkimaan. (Cormen ym. 2009c)



Kuvio 2. Esimerkki DFS-algoritmista

2.2 Breadth-first algoritmi

Breadth-first etsintäalgoritmi on yksinkertaisimpia algoritmeja, joilla etsitään kaavioiden solmuja. Yleensä sitä käytetään testaamaan vektorikartan yhtenäisyyttä tai laskemaan vektorikartan lyhin reitti (Beamer, Asanovic ja Patterson 2012).

Algoritmissa valitaan jokin satunnainen silmukka, jonka kaikki viereiset silmukat käydään ensiksi läpi. Sama prosessi toteutetaan kaikille silmukoille, jotka algoritmi etsii kunnes kaikki solmut on löydetty (ks. kuvio 3). Breadth-first algoritmi käyttää solmujen tutkimisessa jonotus metodia, jossa se laittaa tutkittavan silmukan naapurit jonoon ja käy järjestyksessä näiden silmukoiden naapurit joissa se ei ole vielä käynyt. Lopuksi algoritmi lisää ne jonon

perälle ja jatkaa tätä prosessia kunnes kaikki solmut ovat jonossa ja niissä on käyty. (Beamer, Asanovic ja Patterson 2012; Cormen ym. 2009d).

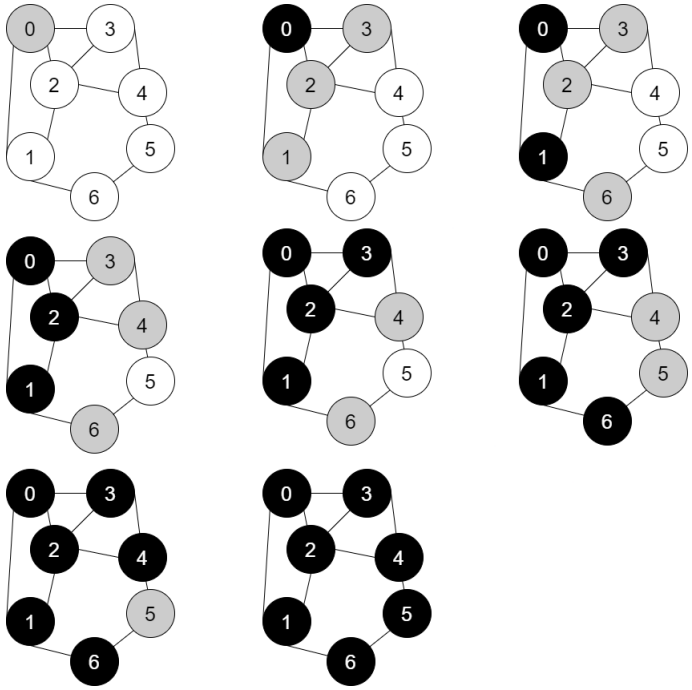
Seuraava pseudokoodi on yksinkertainen breadth-first etsintäalgoritmi, joka on mukaelma (Cormen ym. 2009e) pseudokoodista.

```
BFS(G, s)
    G -> Graafi
    s -> Aloitussolmu

begin
    let q[] //taulukko johon solmut tallennetaan
    q.lisaa(s) //Sijoita lähdesolmu taulukkoon

    mark s as visited
    while(q != empty)
    //Poista solmu, jonka vierekkäisiä solmuja käsitellään
        n = q.poista()
        for all edges from n to w in G
            if(w != visited) q.poista(w)
        mark w as visited
end
```

Pseudokoodissa luodaan ensimmäiseksi tyhjä taulukko q, johon solmuja tallennetaan. Aloitussolmu s lisätään taulukkoon q ja merkitään käydyksi, josta ohjelma jatkaa while-silmukkaan. While-silmukkaa toistetaan niin kauan, kun q ei ole tyhjä. While-silmukassa ensin sijoitetaan n-muuttujaan q-taulukon huipulla oleva solmu. Silmukka jatkuu for-silmukkaan, jossa kaikki n-solmun vierekkäiset solmut tarkistetaan ja katsotaan, onko niissä käyty. While-silmukan lopuksi w-solmu merkitään käydyksi ja aloitetaan while-silmukan prosessi alusta, kunnes q-taulukko on tyhjä.

Breadth-first algoritmin tehokkuudeksi tulee $O(V+E)$, missä V on silmukoiden määrä ja E on kaarien määrä. Tehokkuuden laskemisessa oletetaan, että pahimmassa tapauksessa jokai-

nen solmu ja kaari joudutaan tutkimaan. (Cormen ym. 2009f)



Kuvio 3. Esimerkki BFS-algoritmista

2.3 Dijkstran algoritmi

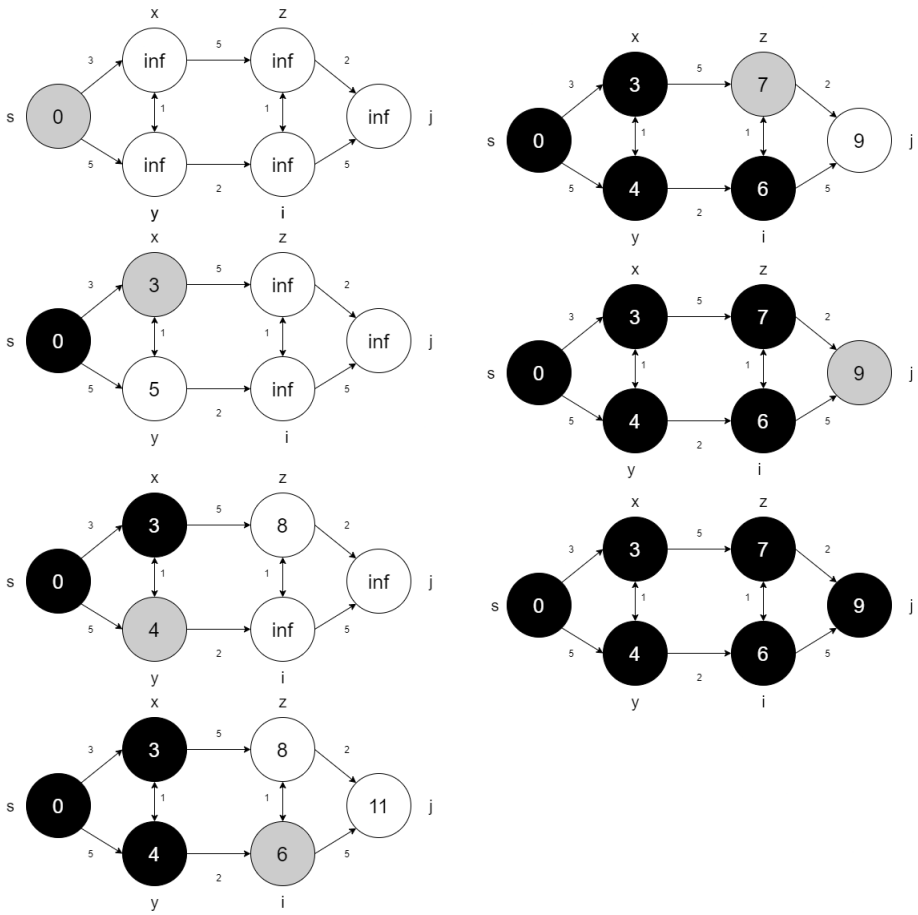
Dijkstran algoritmi toimii etsimällä lyhimmän reitin pituuden graafin kahden eri solmun väliltä (Cormen ym. 2009g). Reitin etsintä tapahtuu valitsemalla ensiksi lähtösolmu, jolle annetaan arvo 0. Algoritmi vertaa sen vieressä olevia solmuja ja valitsee sen solmun, johon sillä on lyhin matka (ks. kuvio 4). Algoritmin toimintaa varten tarvitaan tyhjä joukko *S*-solmuja, johon tallennetaan lyhimmän reitin solmu sen löytyessä. Prosessia jatketaan lisäämällä joukkoon aina se solmu, johon on lyhin matka. Jokaiselle lisätylle solmulle annetaan arvo, joka on matkan pituus lisättynä lähdetyn solmun arvoon. Lisäysprosessia toistetaan, kunnes määränpääsolmu on joukossa *S*.

Seuraava pseudokoodi on yksinkertainen Dijkstra etsintäalgoritmi, joka on mukaelma (Cormen ym. 2009h) pseudokoodista.

```
RELAX(u, v, w)
begin
    if (d[u] + w(u,v) < d[v]) //Jos tosi niin päivitä
        update d[v] to[u] + w(u,v)
end

Dijkstra (G, w)
begin
let S[] //Tyhjä taulukko
let Q = V[G]
while (Q != empty) do
        u = extract_min(Q)
        S = S union {u}
        for each vertex v in Adj[u]
            do RELAX(u, v, w)
end
```

Dijkstra algoritmin pseudokoodissa luodaan tyhjä talukko S ja annetaan muuttujalle Q kaikki solmut. Koodi jatkuu while-silmukkaan, jota käydään kunnes Q on tyhjä. Muuttujalle u annetaan solmu Q:sta, joka on lähimpänä muuttujaa u ja se lisätään joukkoon S. RELAX-funktio selvittää lyhimmän reitin laskemalla solmun u ja v painon w ja jos se on vähemmän kuin $d[v]$ niin $[u]+w(u,v)$ on uusi $d[v]$

Dijkstran algoritmin tehokkuudeksi tulee $O(|E|+|V|log|V|)$, missä $|V|$ on solmujen määrä ja $|E|$ on reunojen määrä. Tehokkuuden laskemisessa oletetaan, että pahimmassa tapauksessa jokainen silmukka ja kaari joudutaan tutkimaan. Tehokkuuden nopeuttamiseksi algoritmi toteutetaan Fibonacci-kekoa käyttäen, miksi tehokkuudeksi tulee $O(|E|+|V|log|V|)$ eikä $O(V^2)$ (Cormen ym. 2009i).



Kuvio 4. Esimerkki Dijkstran algoritmista

2.4 A-star algoritmi

A*-algoritmi (A-tähti) etsii reitin käyttämällä best-first etsintää jolla etsitään halvin mahdollinen reitti aloitussolmun ja määränpään väliltä (Nosrati, Karimi ja Hasanvand 2012). Algoritmi käyttää heuristista funktiota, jolla se määrittelee missä järjestyksessä se käy ruudukossa olevat solmut (Noori ja Moradi 2015).

Algoritmi käyttää kolmea arvoa, joilla se määrittää lyhimmän reitin lähtöpisteestä maaliin (ks. kuvio 5). g(n)-arvo on solmujen välinen etäisyys (paino), h(n)-arvo on solmun heuristinen arvo ja f(n)-arvo on g(n) ja h(n) yhteenlaskettu määrä, siis arvio lyhimmästä reitistä. Lyhimmän matkan määrittämiseksi algoritmi laskee aloitussolmun vieressä olevien solmujen g(n)-,h(n)- ja f(n)-arvon ja siirtyy solmuun, jolla on pienin f(n)-arvo. Algoritmi toistaa saman toimenpiteen kuin aikaisemmassa solmussa ja valitsee taas solmun, jolla on pienin

f(n)-arvo kunnes saapuu maaliin (Randria ym. 2007; Nosrati, Karimi ja Hasanvand 2012).

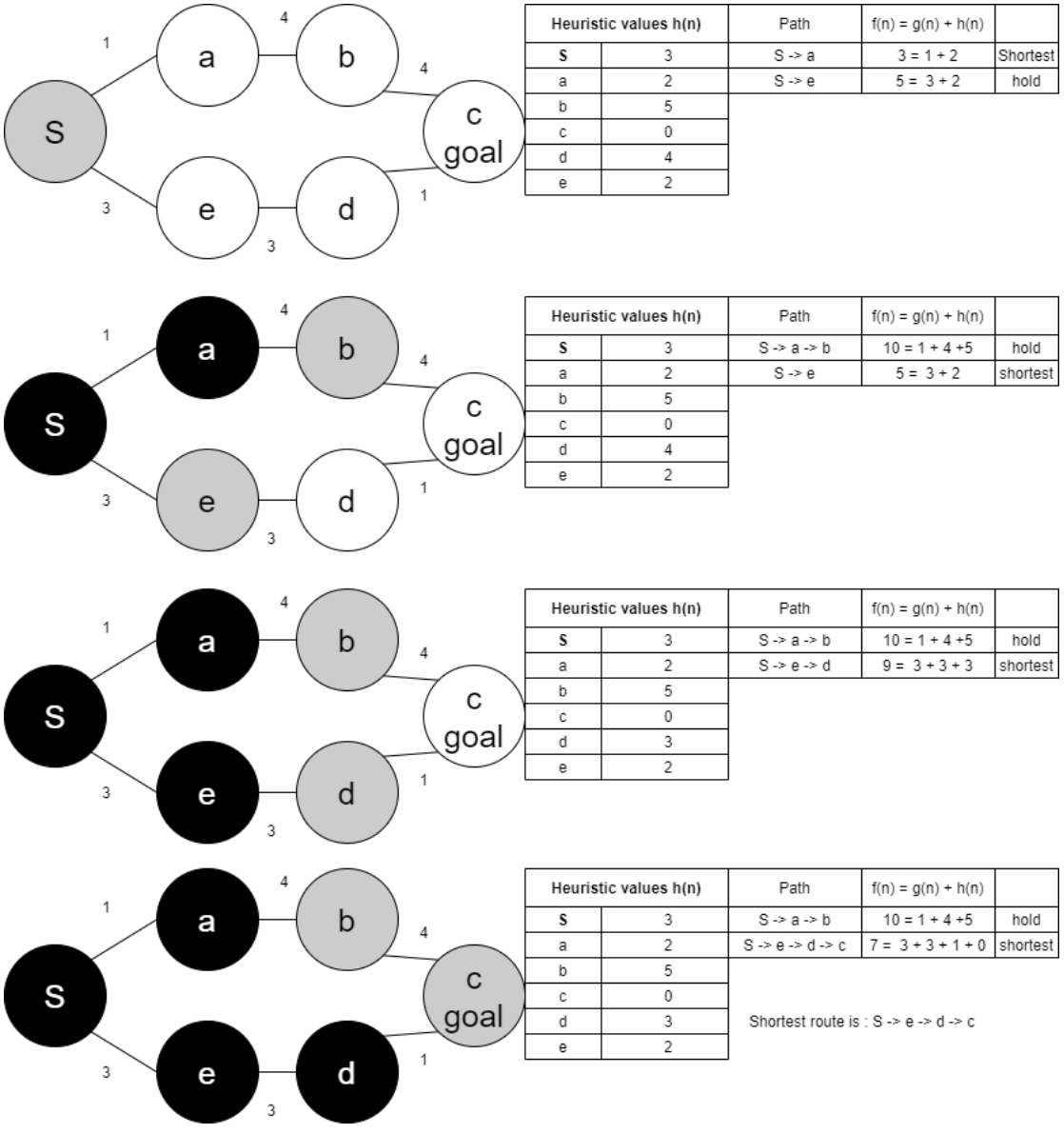
Seuraava pseudokoodi on yksinkertainen A* etsintäalgoritmi, joka on mukaelma (Martell ja Sandberg 2016) pseudokoodista.

```
//s = aloitussolmu, w = solmujen välimatka
//h(n) = arvio solmusta n määränpäähän
A_Star(s,w, h)
begin
    let S[] //Tyhjä taulukko jossa kaikki tunnetut solmut
    S[] = s //Lisää aloitussolmu s tyhjään taulukkoon
    let H[] //halvimman tunnetun reitin tyhjä taulukko
    let F[] // f(n) = h(n) + g(n)
    while( S != empty)
        //Se solmu jolla on pienin f-arvo
        solmu = extract_min(S(f(n))
        //Poistetaan solmu S-taulukosta
        S.remove(solmu)
        for each saavutettava from solmu
            //Jos saavutettava on taulukossa hypätään sen yli
            if(saavutettava  != S[])
                S[saavutettava] = solmu
                //Tallennetaan halvin reitti
                H[saavutettava] =
                    S[solmu] +
                    distance(solmu, saavutettava)
                F[saavutettava] =
                    H[saavutettava] + h(saavutettava)
            else()
                S.add(solmu)
```

```
end
```

Pseudokoodissa aloitussolmu s lisätään ensiksi tyhjään joukkoon S. Se solmu jolla on pienin f-arvo, poistetaan S-taulukosta ja sijoitetaan se muuttujaan solmu. Sijoittamisen jälkeen käydään kaikki ne muuttujan saavutettava solmut läpi, jotka on yhteydessä muuttujan solmu kanssa. Tarkistetaan, löytyykö saavutettava S-joukosta ja jos se löytyy S-joukosta, hypätään sen yli. Muussa tapauksessa sijoitetaan S-joukkoon muuttuja solmu, H-joukkoon heuristinen arvo ja F-joukkoon solmujen välinen etäisyys lisättynä heuristinen arvo. Toistetaan kunnes S-joukko on tyhjä (Zarembo ja Kodors 2013).

A* algoritmin tehokkuudeksi tulee sama kuin DA:lle $O(|E|+|V|log|V|)$, missä V on solmujen määrä ja E on reunojen määrä. Tämä johtuu siitä, että jos heuristinen arvo h(n) jätetään laskematta tai se on nolla, se palautuu DA:ksi

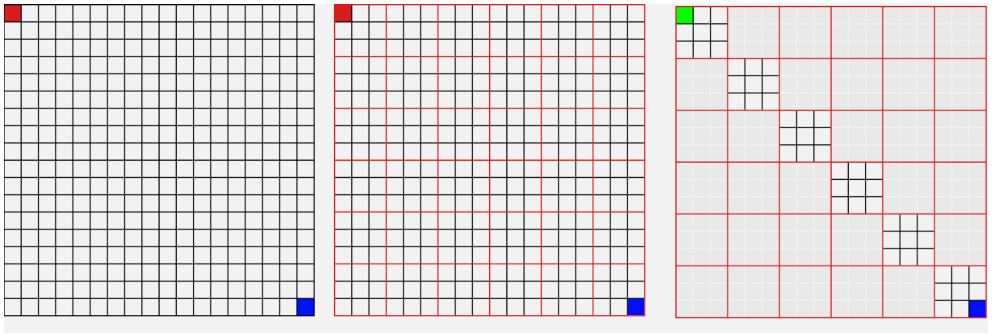


Kuvio 5. Esimerkki A* algoritmista

2.5 Hierarkkinen polun suunnittelu

Hierarkkisessa polun suunnittelussa (HPS) isossa kaaviossa olevat solmut jaetaan suurempiin klustereihin, jolloin luodaan "korkeampi kerros" (Botea, Müller ja Schaeffer 2004). Tällöin korkeammalla kerroksella on vähemmän solmuja, kuin alkuperäisessä matalammassa kaaviossa (ks. kuvio 6).

Esimerkiksi 512 *x* 512 ruudun kokoinen kartta sisältää 262 144 ruutua (solmua), jolloin tehokkaillakin algoritmeilla voi mennä paljon aikaa reitin etsimiselle. Tällöin kartta voidaan jakaa isoihin ryhmiin, jotka koostuvat pienemmistä solmuista. 512 *x* 512 kokoinen kartta voidaan jakaa 64 klusteriin, joilla on 64 *x* 64 kokoinen ruudukko. Reitinetsintä algoritmi etsii ensiksi 64 klusterista reitin ja sen jälkeen käy pienemmät solmut niissä klustereissa, joista reitti löytyi. Tätä prosessia kutsutaan "reitin jalostamiseksi" (Duc, Sidhu ja Chaudhari 2008). Reitin etsintä nopeutuu huomattavasti, koska suurinta osaa kartan alemmista solmuista ei tarvitse käydä läpi.



Kuvio 6. Esimerkki HPS 18 *x* 18 kokoinen kartasta jaetaan 12 isompaan 3 *x* 3 kokoiseen klusteriin

3 Algoritmit ja hierarkkinen polun suunnittelu

Tässä osassa tutkimusta yhdistämme hierarkkisen polun suunnittelua ja algoritmeja keskenään. Vertaamme jokaista algoritmia kuinka ne suoriutuvat reitinetsinnästä, 64 x 64, 128 x 128, 256 x 256 ja 512 x 512 kokoisesta kartasta ilman hierarkkista reitin suunnittelua ja hierarkkisen suunnittelun kanssa. Oletuksena on, että kaikilla algoritmeilla reitin etsiminen nopeutuu huomattavasti sillä tarvittavia solmuja tarvitsee käydä huomattavasti vähemmän. Toisena oletuksena on, että kartassa ei ole yhtään esteitä, joten reitti löytyy aina ja reitti on kaikilla algoritmeilla sama (ks. kuvio 6).

Hierarkkisessa polun suunnittelussa kartat jaetaan vain kahteen kerrokseen, yhteen matalaan tasoon ja yhteen korkeaan tasoon. Matala taso on alkuperäisen kartan kokoinen ja korkea taso koostuu useammasta klusterista joissa jokaisessa on useampi ruutu. Kaikilla neljällä kartalla on saman kokoinen ylempi kerros, joka on 4 x 4. Tällöin korkeamman tason kartassa on 16 solmua ja 84 kaarta (ks. kuvio 1).

3.1 Depth-first, Breadth-first ja hierarkkinen polun suunnittelu

DFS- ja BFS-algoritmin tehokkuus on $O(V+E)$ ja koska HPS:n avulla meillä on kaksi kerrosta, ensimmäiselle kerrokselle tehokkuudeksi tulee $O(16+84)$ riippumatta kartan koosta. Ylemmän kerroksen klustereita on 4 kappaletta joista jokainen sisältää $(kartansivu/4)$ x $(kartansivu/4)$ ruutua (solmua). Palautetaan kartta takaisin matalaan tasoon ja lasketaan tehokkuus $O(V+E)$. Saatuun tehokkuuteen lisätään vielä ylemmän tason tehokkuus ja lopulliseksi tehokkuudeksi HPS:n avulla DFS- ja BFS-algoritmeille on laskettu (ks. kuvio 7).

Molemmat BFS ja DFS ovat raa'an voiman etsintä algoritmeja ja niiden pahin mahdollinen tehokkuus eroaa huomattavasti DA:n ja A* tehokkuudesta. Katsomalla kuviota 7 huomataan selvä ero, kun käytetään HPS:a. Vaikka solmujen määrä kasvaa eksponentiaalisesti kartan koon kasvaessa, putoaa tarvittavien solmujen läpikäynti neljänneksellä, kun HPS:a käytetään. Tarvittavien solmujen pudotessa putoaa myös pahin mahdollinen suoritusaika.

Zarembon ja Kodorsin (2013) tutkimuksessa BFS ajettiin saman kokoisissa kartoissa käy-

Kartan Koko	Algoritmien tehokkuus HPS:n kanssa
64 x 64	O(16 + 84) + O(1024 + 8001)
128 x 128	O(16 + 84) + O(4096 + 32385)
256 x 256	O(16 + 84) + O(16384 + 130305)
512 x 512	O(16 + 84) + O(65536 + 522753)
Kartan Koko	Algoritmien tehokkuus ilman HPS:a
64 x 64	O(4096 + 32004)
128 x 128	O(16384 + 129540)
256 x 256	O(65536 + 521220)
512 x 512	O(262144 + 2091012)

Kuvio 7. BFS ja DFS tehokkuudet ilman HPS:a ja HPS:a kanssa

tännössä ja kuviossa 8 näkyy BFS:n suoritusaika ilman HPS:a. Kuviosta huomaa suoritusajan kasvavan eksponentiaalisesti ja kuten Zarembo ja Kodors (2013) toteavat “512x512 ja 1024x1024 kokoisessa ruudukossa suoritusajan vievän liian kauan aikaa, että BFS algoritmia tuskin tullaan käyttämään reaaliajan ratkaisuna suurissa ruudukkokartoissa.” Tutkimuksessa ei valitettavasti ole mainintaa DFS tai BFS -algoritmista, jossa olisi hyödynnetty HPS:a. Tosin jos HPS:a olisi hyödynnetty, samat suoritusajat olisivat mahdollisesti huomattavasti pienempiä, pienemmän solmu määrän takia (ks. kuvio 7).

TABLE I

ALGORITHM BFS EXECUTION TIME

Grid size (nodes)	Execution time (ms)
64x64	150
128x128	2803
256x256	48313
512x512	821598
1024x1024	13962457

Kuvio 8. BFS:n suoritusaika ilman HPS:a (Zarembo ja Kodors 2013)

3.2 Dijkstran, A* ja hierarkkinen polun suunnittelu

Dijkstran ja A* algoritmien tehokkuus on $O(|E|+|V|log|V|)$ ja HPS:n avulla ensimmäiselle kerrokselle tulee tehokkuudeksi $O(|84|+|16|log|16|)$. Ylemmän kerroksen klustereita on 4 kappaletta kuten DFS:llä ja BFS:llä joista jokainen sisältää $(kartansivu/4)$ x $(kartansivu/4)$ ruutua (solmua). Lasketaan ylempi ja alempi kerros yhteen ja lopulliseksi tehokkuudeksi HPS:n avulla DA:lle ja A*:lle on laskettu (ks. kuvio 9).

Kartan Koko	Algoritmien tehokkuus HPS:n kanssa
64 x 64	O(|84| + |16| log|16|) + O(|8001| + |1024| log|1024|)
128 x 128	O(|84| + |16| log|16|) + O(|32385| + |4096| log|4096|)
256 x 256	O(|84| + |16| log|16|) + O(|130305| + |16384| log|16384|)
512 x 512	O(|84| + |16| log|16|) + O(|522753| + |65536| log|65536|)
Kartan Koko	**Algoritmien tehokkuus ilman HPS:a**
64 x 64	O(|32004| + |4096| log|4096|)
128 x 128	O(|129540| + |16384| log|16384)
256 x 256	O(|521220| + |65536| log|65536|)
512 x 512	O(|2091012| + |262144| log|262144|)

Kuvio 9. DA ja A* tehokkuudet ilman HPS:a ja HPS:a kanssa

Kuten DFS ja BFS -algoritmeille DA:n ja A*:n pahin mahdollinen vaadittava tehokkuus putoaa neljännekseen, kun HPS:a käytetään (ks. kuviot 9). Suoritusaika toisaalta on huomattavasti nopeampi molemmilla algoritmeilla riippumatta karttojen koosta verrattavissa BFS:n suoritusaikaan (ks. kuviot 8, 10). Suuria eroja näkyy myös DA:lla ja A*:lla varsinkin, kun verrataan suurempia karttoja, joissa A* suoriutuu tehtävästään lähes puolet nopeammin kuin DA.

A* suoritusaika lyhenee vielä entisestään, kun A* suoritetaan HPS:n kanssa (ks. kuvio 11). DA:n suoritusaika oletetusti putoaisi myös, jos HPS:a käytettäisiin, pienemmän solmu määrän takia.

TABLE II

DIJKSTRA'S ALGORITHM EXECUTION TIME

Grid size (nodes)	Execution time (ms)
64x64	6
128x128	25
256x256	120
512x512	515
1024x1024	2362

TABLE III

ALGORITHM A* EXECUTION TIME

Grid size (nodes)	Execution time (ms)
64x64	4
128x128	16
256x256	77
512x512	265
1024x1024	1148

Kuvio 10. DA ja A* suoritusaika ilman HPS:a (Zarembo ja Kodors 2013)

TABLE V

ALGORITHM HPA* EXECUTION TIME

Grid size (nodes)	Execution time (ms)
64x64	3
128x128	14
256x256	52
512x512	190
1024x1024	775

Kuvio 11. A* HPS:n kanssa (Zarembo ja Kodors 2013)

4 Vertailua

DFS- ja BFS-algoritmin tehokkuudet ovat samat $O(V+E)$. Algoritmien pahin mahdollinen tehokkuus putoaa lähes neljännekseen, kun käytetään HPS:a (ks. luku 3.1). Luvussa 3.2 A* ja HPA* välillä nähdään selvä ero suoritusajoilla, mitä suurimmiksi ruudukkokartat kasvoivat. Tästä voidaan vetää osviittaa siitä, että myös DFS ja BFS suoritusajat pienenisivät HPS:a avulla.

DA:n ja A* tehokkuuden parantumista on huomattavasti monimutkaisempaa verrata, sillä DA ja A* toimivat kaarilla joissa on paino. A* hyödyntää myös heuristiikkaa reitin selvittämisessä, ja sillä on merkittävä vaikutus siihen kuinka nopeasti algoritmi reitin löytää. Tutkimuksessa oletetaan, että algoritmit joutuvat käymään kartan kaikki solmut läpi löytääkseen reitin ilman, että kartassa olisi yhtään esteitä. Oletamme myös, että A* heuristinen arvo on nolla, kun laskemme pahinta mahdollista tehokkuutta. Näillä oletuksilla verrataan hypoteettisia pisimpiä aikoja, joita algoritmit saattavat joutua käymään.

Pahin mahdollinen DA:n ja A* tehokkuus on $O(|E|+|V|log|V|)$. Algoritmien pahimmalle mahdolliselle tehokkuudelle tapahtuu sama vaadittavan maksimi ajan putoaminen, mitä DFS- ja BFS-algoritmeille (ks. luku 3.2). Kuviosta 10 näkee DA:n suoritusajan kasvavan melkein kaksinkertaiseksi verrattuna A* suoritusaikaan. Aivan yhtä suurta ero ei kuitenkaan synny A* ja HPA* suoritusajalle (ks. kuvio 11), mutta huomattava ero kuitenkin kun ruudukkokartan koko kasvaa. Nopeammin löytyvä reitti tulee kuitenkin sillä hinnalla, että löydettyjen reittien optimaalisuus ei ole taattu.

5 Yhteenveto

Tutkimuksen alussa tutustuimme neljän eri algoritmien perusteisiin ja niiden pahimpiin mahdollisiin tehokkuuksiin. Kävimme yleisesti läpi näiden algoritmien toiminnallisuutta ja tehokkuutta. Algoritmien jälkeen kävimme yleisesti hierarkkisen polun suunnittelun ja lopuksi vertasimme neljän algoritmin toimintaa HPS:n kanssa ja ilman.

Tutkimuksessa huomaa, kuinka hierarkkinen polun suunnittelu parantaa huomattavasti kaikkien tässä tutkimuksessa käytettyjen algoritmien pahinta mahdollista tehokkuutta, vähentämällä ruutujen määrää kartoissa. Varsinkin niiden algoritmien joilla ei ole solmujen välillä painoa, Depth-first ja Breadth-first. Vertailussa kävi myös ilmi, että painolliset algoritmit hyötyvät myös HPS:sta, mutta vain silloin kun tutkimuksessa olevat oletukset ovat voimassa. Tutkimuksessa on oletettu, että

1. Kartta ei sisällä esteitä, joihin algoritmit eivät voisi kulkea.
2. HPS:lla ei tehdä kuin kaksi kerrosta.
3. Algoritmit joutuvat käymään jokaisen solmun ja kaaren löytääkseen reitin.
4. Jokaisen kaaren paino on 1.

HPS:n hyöty on hyvin selvä algoritmeille, jotka eivät hyödynnä solmujen välisiä etäisyyksiä reitin etsinnässä. Siksi tutkimuksen heikkoudeksi jää se, että kaikille algoritmeille ei ole suoritusaikaa käytynä käytännössä. Ainoastaan A* on ajettu HPS:n kanssa, joka jättää muut algoritmit spekulaation varaa kuinka hyödyllinen HPS on niille käytännössä.

Tällä hetkellä tutkimuksesta on selvää, että HPS vähentää tarvittavien ruutujen tutkimista kaikilla algoritmeilla. Lisää tutkimusta tarvitaan selvittämään, mikä on HPS:n todellinen hyöty suoritusaikojen parantamisessa DFS:n, BFS:n ja DA:n kanssa käytännössä.

Lähteet

Beamer, Scott, Krste Asanovic ja David Patterson. 2012. “Direction-optimizing breadth-first search”. Teoksessa *SC’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis,* 1–10. IEEE.

Botea, Adi, Martin Müller ja Jonathan Schaeffer. 2004. “Near optimal hierarchical path-finding.” *J. Game Dev.* 1 (1): 8.

Cormen, Thomas H, Charles E Leiserson, Ronald L Rivest ja Clifford Stein. 2009a. *Introduction to algorithms.* Luku 22.3, 540. MIT press.

———. 2009b. *Introduction to algorithms.* Luku 22.3, 541–542. MIT press.

———. 2009c. *Introduction to algorithms.* Luku 22.3, 543. MIT press.

———. 2009d. *Introduction to algorithms.* Luku 22.2, 531–532. MIT press.

———. 2009e. *Introduction to algorithms.* Luku 22.2, 532. MIT press.

———. 2009f. *Introduction to algorithms.* Luku 24.3, 534. MIT press.

———. 2009g. *Introduction to algorithms.* Luku 24.3, 595. MIT press.

———. 2009h. *Introduction to algorithms.* Luku 24.3, 595. MIT press.

———. 2009i. *Introduction to algorithms.* Luku 24.3, 599. MIT press.

Duc, Le, Amandeep Sidhu ja Narendra Chaudhari. 2008. “Hierarchical Pathfinding and AI-Based Learning Approach in Strategy Game Design”. *International Journal of Computer Games Technology* 2008 (tammikuu): 5–6. https://doi.org/10.1155/2008/873913.

Martell, Victor, ja Aron Sandberg. 2016. *Performance Evaluation of A* Algorithms.*

Noori, Azad, ja Farzad Moradi. 2015. “Simulation and Comparison of Efficency in Pathfinding algorithms in Games”. *Ciência e Natura* 37:230–238.

Nosrati, Masoud, Ronak Karimi ja Hojat Allah Hasanvand. 2012. “Investigation of the*(star) search algorithms: Characteristics, methods and approaches”. *World Applied Programming* 2 (4): 251–256.

Randria, I., M. M. Ben Khelifa, M. Bouchouicha ja P. Abellard. 2007. “A Comparative Study of Six Basic Approaches for Path Planning Towards an Autonomous Navigation”. Teoksessa *IECON 2007 - 33rd Annual Conference of the IEEE Industrial Electronics Society,* 2730–2735. https://doi.org/10.1109/IECON.2007.4460164.

Tarjan, Robert. 1972. “Depth-first search and linear graph algorithms”. *SIAM journal on computing* 1 (2): 147.

Zarembo, Imants, ja Sergejs Kodors. 2013. “pathfinding algorithm efficiency analysis in 2d grid”. Teoksessa *ENVIRONMENT. TECHNOLOGIES. RESOURCES. Proceedings of the International Scientific and Practical Conference,* 2:46–50.