Topias Liljegren

JPS algoritmin käyttö

Tietotekniikan Kandidaatin tutkielma

25. kesäkuuta 2024

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Topias Liljegren

Yhteystiedot: toplilje@jyu.fi

Työn nimi: JPS algoritmin käyttö

Title in English: Usage of the JPS algorithm

Työ: Kandidaatin tutkielma

Sivumäärä: 20+0

Tiivistelmä: Polunetsintä on keskeinen ongelma tietotekniikassa. Tämä tutkielma vertailee vanhoja polunetsimisalgoritmeja uudenpiin, kuten Jump Point Searchiin. Se luotiin vuonna 2011 ja se karsii turhia solmuja ja hyppii hyppypisteiden välillä, tehden siitä nopeamman ja muistitehokkaamman. Polunetsimisalgoritmeilla on monta käyttötarkoitusta, robotiikasta videopeleihin. Tämä tutkielma käy läpi JPS derivatiiveja, joilla on eri käyttötarkoituksia, sekä miten niitä käytetään. Vaikka JPS on julkaistu yli vuosikymmen sitten, on sen käyttö jäänyt ilmeisen vähäiseksi, vaikka sen suorituskyky on jopa kymmenkertainen klassisiin menetelmiin verrattuna. Tämä tutkielma päättyy reflekioihin potentiaalisista syistä, jotka voivat vaikuttaa siihen.

Avainsanat: JPS, polunetsintäalgoritmi, polunetsintä, algoritmi

Abstract: Pathfinding is an ever evolving problem in computer science. This thesis will compare old pathfinding algorithms to newer ones, like the Jump Point Search. It was created in 2011 and it prunes unnecessary nodes and jumps between jump points, making it much faster and memory efficient. Pathfinding algorithms have many use cases, ranging from robotics to video games. This thesis will talk about some JPS derivatives, which have more versatile use cases and why one should use them. Despite JPS being out for over a decade, it has no public show of use anywhere, while outperforming the industry standards up to tenfold. This thesis ends with reflections on potential reasons contributing to this.

Keywords: JPS, pathfindingalgorithm, pathfinding, algorithm

Kuviot

Sisällys

1 Johdanto

Polunetsintäalgoritmit ovat keskeinen osa monia sovelluksia, jotka vaativat tehokasta reitin löytämistä ja optimointia. Näitä algoritmejä käytetään muun muassa robotiikassa, pelien kehityksessä sekä logistiikassa. Tässä tutkielmassa tutkitaan vuonna 2011 kehitettyä JPS (engl. Jump Point Search) polunetsintäalgoritmia ja sen eri näkökulmia (Harabor ja Grastien 2011). JPS on merkittävä edistysaskel polunetsintäalgoritmien kehityksessä ja se toimii erityisesti ruutukartoissa polunetsinnässä. JPS vähentää tutkittavien solmujen (engl. node) määrää hyödyntämällä hyppypisteitä, jotka ovat tarkasti karsittuja solmuja. Näiden hyppypisteiden avulla voidaan hyppiä monen turhan solmun yli tehostaen reitinetsintää huomattavasti.

Tässä tutkielmassa tutkitaan JPS algoritmia sen alkuperäisessä muodossa, mutta sivutaan myös Dijkstran algoritmistä lähtien muutamia valittuja algoritmejä. Näihin kuuluu muun muassa A* algoritmi, joka on JPS suora edeltäjä, HPA* algoritmi (Foead ym. 2021), joka on A* algoritmin jatkokehitystä, sekä JPS+ algoritmi (Harabor ja Grastien 2014), joka on JPS algoritmin jatkokehitystä. Tarkoitus on kartoittaa kattavasti JPS algoritmin toimintaperiaatteita, erilaisia sovelluksia sekä jatkokehitystä.

Tässä tutkielmassa vertaillaan myös JPS algoritmia sen pääkäyttötarkoituksesa, sekä sovelletuissa käyttötarkoituksissa, muihin samankaltaisiin algoritmeihin. Tämän vertailun, sekä muiden algoritmien tutkimisen perusteella pyritään tutkimaan jatkokehityksen haasteita, sekä JPS algoritmin optimointia pääkäyttötarkoituksen ulkopuolelta. JPS algoritmilla ei kuitenkaan ole lähes yhtään näkyvää käyttöä, tehokkuudestaan huolimatta, ja sitä pohditaan tutkielman lopussa.

2 Polunetsintäalgoritmien pohjustus

Polunetsintäalgoritmit ovat keskeinen osa tietojenkäsittelytiedettä ja graafiteoriaa. Niiden avulla etsitään reittejä eri solmujen välillä. Solmut voivat olla pisteitä kartalla ja polut reittejä solmujen välillä. Näiden algoritmien avulla voi löytää lyhyimmän, nopeimman tai halvimman reitin solmujen välillä.

2.1 Polunetsinnän termejä

2.1.1 Solmu

Solmu (engl. node) on piste koordinaatistossa, graafissa, ruudukossa tai muuten vain jollain kartalla. Polunetsinnässä kuljetaan solmujen välejä ja ne voivat kuvainnollistaa esimerkiksi bussipysäkkejä kaupungin sisällä, ruutuja pelilaudalla tai tietokoneita tietokoneverkossa. Solmusta voidaan puhua myös pisteenä tai paikkana.

2.1.2 Polku

Polku (engl. path) on reitti solmujen välillä jolla on lähtö- ja maalisolmu. On olemassa erinlaisia polkuja eri käyttötarkoituksiin. Joskus tarvitaan nopein tai lyhyin polku kahden solmun väliltä, joskus tarvitaan polku joka käy kaikkien solmujen kautta joko tasan kerran tai etsii lyhyimmän reitin kaikki käyttäen. Tähän liittyy monia avoimia ongelmia, kuten kauppamatkustajan ongelma. Polusta voidaan puhua myös reittinä tai tienä.

2.1.3 Algoritmi

Algoritmi on ohje joka kertoo miten jokin prosessi tulee suorittaa. Algoritmille on määritelty seuraavat ominaisuudet: sen pitää olla äärellisessä ajassa suoritettava, sen askeleiden pitää olla selkeästi määritelty, sillä on nolla tai enemmän syötettä, se tuottaa ulos yhden tai enemmän tulosteen ja sen operaatioiden pitää olla tarpeeksi yksinkertaisia (Knuth 1997). Algoritmit pohjautuvat matematiikkaan ja ovat yksinkertaisimmillaan kuin matematiikan funktiot.

2.1.4 Polunetsintäalgoritmi

Polunetsintäalgoritmit ovat osajoukko algoritmeistä, joiden tarkoitus on etsiä polku solmusta A solmuun B. Polkua voi joutua etsimään erilaisista kartoista ja graafeista eri ulottuvuuksissa. Yleisimpiä käyttökohteita polunetsintäalgoritmeille ovat muun muassa kartalla polun etsiminen tai painotetussa graafissa (engl. weigted graph) polun etsiminen. Koska reittejä voi olla monia erilaisia, joista läheskään kaikki eivät ole optimaalisia, on polunetsintäalgoritmeille myös yleistä yrittää etsiä nopein tai lyhin reitti solmujen välillä. Polunetsiminen muistuttaa läheisesti lyhimmän reitin ongelmaa (engl. shortest path problem) graafiteoriasta. Graafiteoria tutkii miten löytää kriteerien täyttämän reitin kahden pisteen välillä jonkin kaltaisessa verkostossa. Kriteeri voi olla löytää lyhin, halvin tai nopein reitti ja samoja kriteereitä noudattavat myös nykyiset polunetsintäalgoritmit. Polunetsintäalgoritmit ovat siis vain graafiteorian sovelluksia reittien etsinnässä.

2.2 Polunetsintäalgoritmien historia

Polunetsiminen ongelmana sai alkunsa aikaisin tietokoneiden kehityksessä ja Dijkstran algoritmi on sen ajan käytetyin ja tunnetuin polunetsintäalgoritmi. Sittemmin Dijkstran algoritmi on jäänyt tehokkaampien algoritmien jalkoihin, joista tunnetuin on A* algoritmi. A* algoritmi on kokenut saman kohtalon kuin Dijkstran algoritmi eikä ole käytössä sellaisenaan enää, vaan on saanut parempia jälkeläisiä. Tietokoneiden yleistyessä on tarve erilaisille polunetsintäalgoritmeille lisääntynyt ja täten eri algoritmit erikoistuvat eri tarpeisiin, eikä siksi ole yhtä algoritmia kaikkiin tarpeisiin.

2.2.1 Dijkstran algoritmi

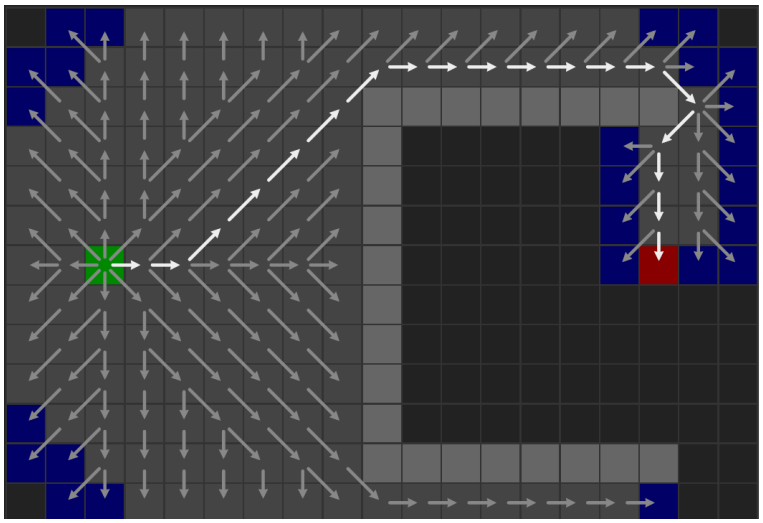
Dijkstran algoritmi on algoritmi joka etsii lyhimmän reitin kahden solmun välillä painotetussa graafissa. Sen kehitti Edsger W. Dijkstra vuonna 1956 ja se julkaistiin kolme vuotta myöhemmin. Dijkstran algoritmista on myös suositumpi versio, joka kiinnittää yhden solmun lähtösolmuksi ja laskee siitä kaikkiin solmuihin lyhimmän reitin, luoden lyhimpien reittien puun (engl. shortest-path tree). Dijkstran algoritmia voidaan käyttää tälläisenään esimerkiksi junaverkossa tai autoteillä nopeimman reitin etsimisessä. Koska ruudukkokartta on vain

säännöllisessä järjestyksessä oleva vakiopainoinen graafi, voidaan Dijkstran algoritmia käyttää myös perus kartoissa, joissa se leviää kaikkiin suuntiin tasaisesti, kunnes löytää maalin.

2.2.2 A* algoritmi

A* algoritmi on vuonna 1968 kehitetty jatko-osa Dijkstran algoritmiin, joka paranti sen nopeutta. A* algoritmi onnistuu tässä tunnistamalla suunnan johon liikutaan. Dijkstran algoritmi tallentaa solmukohtaisesti vain lyhimmän etäisyyden lähtösolmusta sekä edellisen solmun. A* algoritmi tallentaa näiden lisäksi myös solmukohtaisen arvioidun etäisyyden loppusolmuun linnuntietä pitkin ja tätä hyväksikäyttäen saa Dijkstran algoritmille suuntavaiston. Tämän ansiosta A* algoritmi onnistuu olemaan keskimäärin melkein kaksi kertaa nopeampi kuin Dijkstran algoritmi (Candra, Budiman ja Hartanto 2020). A* algoritmi ei kuitenkaan ole yhtä monikäyttöinen kuin Dijkstran algoritmi, sillä se etsii vain reittejä solmusta A solmuun B, eli sillä ei voi luoda esimerkiksi lyhimpien reittien puuta.

Kuvassa 1 vaaleanharmaat laatikot ovat seiniä, nuolien täyttämät harmaat ovat tarkistettuja solmuja, siniset ovat jonossa seuraavaksi tarkistettavia, jos maalia ei olisi vielä löytynyt. Vihreä laatikko on lähtösolmu ja punainen on maalisolmu ja vaalein nuolien muodostama reitti on lopulta löydetty reitti. Tämä kartta on visualisaatio A* algoritmin toiminnasta yksinkertaisessa kartassa, jossa on vain muutama seinä.



Kuvio 1. A* algoritmin visualisaatio zerowidth sivun sandboxissa (Witmer, n.d.).

2.2.3 A* derivatiiveja

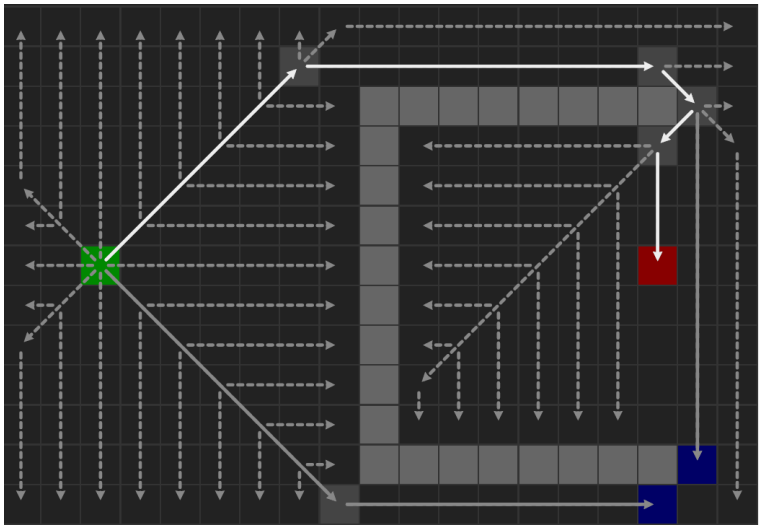
A* algoritmistä on vuosien varrella syntynyt useita eri derivatiiveja eri tarkoituksiin. Basic Theta* ja Phi* ovat A* kaltaisia polunetsintäalgoritmejä, mutta eroavat siten, että A* on ruudukossa 8-konnektiivinen, tarkoittaen että A* tutkii vain kahdeksaan suuntaan 45° väleillä, kun taas Basic Theta* ja Phi* tutkivat kaikilla kulmilla (Duchoň ym. 2014). Tämän ansiosta nämä algoritmit voivat löytää lyhyempiä reittejä kuin A*, mutta ovat myös magnituudin hitaampia. Jos on rajoitteita ajan tai muistin käytön suhteen, ei A* algoritmi ole sellaisenaan ole enään paras vaihtoehto, vaan on kehitetty erilaisia optimointeja, kuten bidirectional A*, IDA* (Iterative Deepening A*) ja HPA* (Hierarchical Path-Finding A*) (Foead ym. 2021). Joskus kuitenkin nopeuden puolesta saattaa joutua uhraamaan optimaalisen reitin kuten HPA* algoritmi välillä tekee. A* algoritmista on nykyään lukemattomia eri versioita, joilla on omat puolensa kullakin, joten vakio A* algoritmi ei näe enään käyttöä sellaisenaan ilman optimointia.

3 JPS

JPS (engl. Jump Point Search) on yksi polunetsintäalgoritmi, joka toimii ruutukartoissa. JPS on optimointi A* polunetsintäalgoritmille, joka on lisäosa Dijkstra algoritmiin. Dijkstra kehitettiin vuonna 1956 ja julkaistiin kolme vuotta sen jälkeen. Dijkstran tarkoitus oli etsiä lyhin reitti kahden solmun välillä ja se teki sen hitaasti, mutta varmasti. A* päivitti Dijkstran algoritmiä vuonna 1968, lisäämällä algoritmille kyvyn tunnistaa, onko se menossa suurinpiirtein oikeaan suuntaan, sen sijaan että se katsoisi kaikkiin suuntiin yhtä aikaa. Molemmat Dijkstra, sekä A* algoritmit löytävät aina lyhyimmän reitin, sekä painotetuissa graafeissa että ruudukoissa. JPS optimoi vielä A* algoritmia nopeuttamalla sitä huomattavasti.

JPS kehittäjien sanoin, meidän algoritmimme voidaan kuvata mikro-operaattoriksi, joka identifioi ja valikoivasti laajentaa vain tiettyjä solmuja ruudukossa, jota me kutsumme hyppypisteiksi (Harabor ja Grastien 2011). JPS nopeuttaa A* algoritmia magnituudin verran käyttämällä edellä mainittuja hyppypisteitä, jolloin algoritmin ei tarvitse ruutu kerrallaan tarkastella ruudukkoa, vaan se tarkastelee ruudun kaikkien suuntien suorat ja analysoi vain muutosten tapahtuessa, tarvitseeko suuntaa muuttaa.

Kuvassa 2 vaaleanharmaat laatikot ovat seiniä, yhtenäisten nuolien käännöskohdissa olevat vähän tummemmat harmaat laatikot ovat hyppypisteitä ja siniset laatikot ovat jonossa seuraavaksi tarkastettavia hyppypisteitä, ellei maalia olisi löytynyt. Vihreä on lähtösolmu ja punainen on maalisolmu ja yhtenäinen valkoinen nuolijono on löydetty reitti. Tämä kartta on visualisaatio JPS algoritmin toiminnasta yksinkertaisessa kartassa, jossa on vain muutama seinä ja jota voi verrata A* algoritmin suoritukseen samassa kartassa 1.



Kuvio 2. JPS algoritmin visualisaatio zerowidth sivun sandboxissa (Witmer, n.d.).

3.1 Miksi JPS toimii

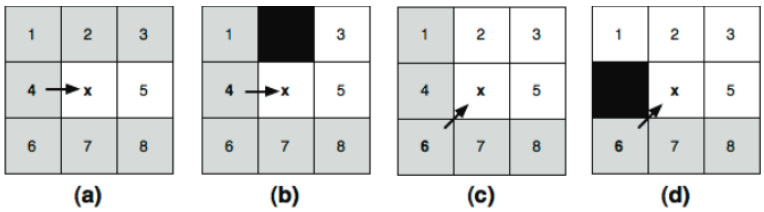
JPS tunnistaa solmukohtaisesti itselleen listan jatkajia funktiolla TUNNISTA. TUNNISTA tarvitsee tiedoksi solmut x: nykyinen solmu, a: lähtö ja b:maali. Lähtösolmu tarkoittaa koko ruudukon lähtösolmua ja vastaavasti maalisolmu on koko ruudukon maalisolmu. Pseudokoodina TUNNISTA funktio:

```
func TUNNISTA(x, a, b):
    jatkajat = tyhjä lista
    naapurit = karsittu lista x:n viereisistä solmuista
    for each n in naapurit:
        hypätty = HYPPÄÄ(x, suunta(x, n), a, b)
        jatkajat.lisää(hypätty)
    return jatkajat
```

TUNNISTA funktio kutsuu silmukassaan HYPPÄÄ funktiota, joka tarvitsee x: nykyinen solmu, d: suunta, a: lähtö ja b: maali. Pakotettu naapuri tarkoittaa että solmu n ei ole solmun x karsittu naapuri ja reitti solmun x vanhemmasta solmuun n on lyhyempi ilman solmua x. Kuva 3 visualisoi, miten valitaan onko naapurisolmu karsittava vai ei. Valkoiset ruudut ovat tutkimattomia soluja, nuoli kertoo mistä mennään ja mihin, tummemmat ruudut ovat pois karsittuja ja mustat ovat seiniä. JPS on diagonaalisesti ensin -polun (engl. diagonal first

path) etsivä, joten esimerkiksi kohdassa a), kun liikutaan solusta 4 soluun x, ovat solut 3 ja 9 jo karsittu pois, koska ensin diagonaalisesti kulkien niihin olisi päässyt 4 -> 2 -> 3 ja 4 -> 8 -> 9. Seinän läpi ei voi kulkea, joten niiden takaa ei ole voinut etukäteen karsia kohdissa b) ja d). Pseudokoodina HYPPÄÄ funktio:

```
func HYPPÄÄ(x, d, a, b):
    n = astu(x, d)
    if n on suljettu solmu tai ruudukon ulkopuolella:
        return null
    if n löytyy listasta naapurit, niin että n on pakotettu:
        return n
    if d on diagonaali:
        for each suunta in {ylös, alas, oikea, vasen}:
            if HYPPÄÄ(n, suunta, a, b) ei ole null:
                return n
    return HYPPÄÄ(n, d, a, b)
```



Kuvio 3. JPS naapurisolujen karsiminen (Harabor ja Grastien 2011)

3.2 JPS derivatiiveja

On olemassa myös muunneltuja versioita JPS algoritmista joilla, kuten A* algoritmin eri versioilla, on omat vahvuutensa. Kolme vuotta JPS julkaisun jälkeen, julkaisivat sen alkuperäiset kehittäjät uuden paperin, jossa puhutaan useammasta parannellusta versiosta JPS algoritmille (Harabor ja Grastien 2014). Tässä yhteydessä kutsun alkuperäistä JPS algoritmia nimellä JPS 2011. Paperissa puhutaan ja verrataan JPS 2011, JPS (B), JPS (B+P), JPS+ ja JPS+ (P) algoritmeistä. JPS (B) karsii monia solmuja yhtä aikaa muuttamalla kartan bittimatriisiksi, joka kertoo onko mikäkin solmu kulkukelpoinen. JPS+ esilaskee ruudukon hyp-

pypisteet valmiiksi, jolloin kun tarvitaan reitti solmusta A solmuun B, voidaan vain käyttää esilaskettuja hyppypisteitä ja täten saada yli magnituudin nopeammin laskettua reitin. JPS+ haittapuoli puolestaan on, että jos ruudukko muuttuu, joudutaan kaikki hyppypisteet laskemaan uudestaan ainakin lokaalisti ja toiseksi se vie enemmän muistia, sillä jokaiselle solmulle pitää tallentaa kahdeksaan suuntaan seuraava hyppypiste. JPS (B+P) ja JPS+ (P) lisäävät edellämainittuihin vielä parannellun karsinnan tunnistamalla hyppypisteet kahden erilaisen väliltä; solmut joilla on vähintään yksi suljettu naapuri vieressä ja niihin joilla ei ole. Solmut joilla ei ole suljettuja naapureita ovat vain solmuja, joissa suunta saattaa vaihtua, joten sen solmun voi jättää huomiotta ja tutkia vain sen mahdolliset kolme seuraavaa hyppypistettä. Nämä kaikki optimoinnit nopeuttavat JPS 2011 algoritmia yhden tai jopa kahden magnituudin verran.

4 Käyttötarkoitus

JPS algoritmi on osoittautunut tehokkaaksi ja optimaaliseksi ratkaisuksi polunetsinnässä ruutukartoilla. JPS kehittäjät ovat todenneet sen olevan ylivoimainen verrattuna klassisiin algoritmeihin, kuten A* tai HPA*. JPS algoritmia voi hyödyntää monella alalla sen yksinkertaisuuden, vähäisen muistinkäytön, nopeuden ja aina optimaalisen reitin löytämisen takia.

4.1 JPS peleissä

JPS kehittäjät todistivat julkaisussaan (Harabor ja Grastien 2011), että JPS algoritmi löytää aina optimaalisen reitin ruutukartassa. He myös vertasivat sen toimivuutta Swamps, HPA* sekä A* algoritmejä vastaan neljässä eri skenaariossa. Näissä skenaarioissa JPS oli nopeudeltaan ylivoimainen Swamps ja A* algoritmia vasten, ollen jopa magnituudin verran nopeampi. HPA* oli ainut, joka pystyi nopeudeltaan kamppailemaan JPS vastaan, ollen joko yhtä nopea tai hieman hitaampi. HPA* algoritmin heikkous on, ettei se löydä optimaaleja reittejä, mutta se on usein valittu algoritmi peleihin, kun tarvitaan nopea algoritmi. Tämän perusteella JPS algoritmin käyttö pelien polunetsintäalgoritmina on optimaali ratkaisu ja yksinkertaisimmissa tapauksissa jopa liioittelua.

Kehittäjien sanoin, se on yksinkertainen, silti todella tehokas; se ylläpitää optimaalisuutta, eikä tarvitse ylimääräistä muistia; se on todella nopea, eikä tarvitse prosessointia. He jatkavat vielä, se on suurelta osin ortogonaalinen kirjallisuuden kilpaileviin nopeutustekniikoihin nähden ja helposti yhdistettävissä niihin. Näistä positiivisista huolimatta moni käyttää yhä A* algoritmia peleissään, eikä JPS käytöstä ole mainintaakaan (Rafiq, Kadir ja Ihsan 2020), vaikka on selvästi todettu, että A* algoritmilla on ongelmia, jotka JPS voisi ratkaista. Jää siis vielä nähtäväksi JPS todellinen kyky pelien optimoinnissa.

4.2 Muita käyttötarkoituksia

JPS algoritmi voi olla hyvä vaihtoehto moneen eri tarkoitukseen, joihin sitä ei ole vielä keritty käyttää, tai josta ei löydy lähteitä. Robottien liikkuminen on yksi vaihtoehto, kunhan

ympäristön kartoittaminen onnistuu. JPS ei vaadi tietoa siitä, mitä seuraavan kulman takana on, vaan laskee reitin vain näkemänsä perusteella. Tämän ansiosta ei ole väliä hyödyntääkö robotti SLAM (engl. Simultaneous Localization and Mapping) tekniikkaa, vai onko alue kartoitettu jo etukäteen. Toinen hyvä käyttötarkoitus muistuttaa jo valmiiksi pelien polunetsinnän optimointia, sillä karttasovellukset voidaan esittää ruutukartassa. JPS voi siis esittää lyhimmän reitin solmujen välillä maastossa tai urbaanilla alueella, mutta ei voi etsiä lyhintä reittiä esimerkiksi junaverkossa, joka sisältää vaihtoja. Tähän tarkoitukseen tarvittaisiin painotettu graafi, johon JPS ei pysty.

5 Kehitysmahdollisuudet

JPS algoritmi on vahvimmillaan kaksiulotteisissa ruutukartoissa, mutta on vielä kehitysvaiheessa 3D-avaruuksissa. Eri tahojen puolesta on koitettu kehittää JPS-3D algoritmejä ja tulokset ovat olleet lupaavia jopa verrattaessa A*-3D algoritmiin. JPS algoritmia voi lähteä kehittämään myös erilaisiin kaksiulotteisiin karttoihin, kuten hexagonaalisiin ruudukoihin. Kyky käsitellä kulmia alle 45° intervallein voi myös lyhentää optimaalisen reitin pituutta, mutta on hitaampi.

5.1 3D optimisaatio

JPS toimii erinomaisesti sen alkuperäiseen tarkoitukseen, mutta on hyvin vielä hyvin keskeneräinen, jos verrataan vaikka polunetsimiseen 3D-avaruuksissa. On monia yksittäisten tahojen tekemiä JPS-3D algoritmejä, jotka toimivat vaihtelevalla onnistumisella. Yhteistä kuitenkin näille on, että JPS-3D algoritmi toimii lähes yhtä hyvin kuin A*-3D algoritmi (Ranttila 2019), tai jopa paremmin, riippuen ympäristöstä (Nobes ym. 2022), esimerkiksi unityä hyväksikäyttäen (Medina, Gittabao ja Agustin, n.d.). Nämä yksittäiset tuotokset ovat vielä hyvin tuoreita ja kaikissa todetaan, että vielä tarvitaan lisää tutkintaa JPS-3D algoritmien kehitykseen ja on näytetty, että sen yleistyminen on mahdollista.

5.2 JPS eri kartoissa

JPS julkaisupaperin lopussa mainittin, että voisi olla mielenkiintoista kokeilla erilaisissa kartoissa kuin neliö-ruudukoissa, esimerkiksi hexagonaalisessa ruudukossa. Eri muotoisilla kartoilla voi olla omat puolensa, mutta suurin osa polunetsintäalgoritmia tarvitsevista tarkoituksista ovat neliöpohjaisia, joten niihin on panostettu ensisijaisesti.

Toinen tapa muuttaa kulmia JPS algoritmilla on saada se vertaamaan kulmia alle 45° intervallein. Vakiona JPS on 8-konnektiivinen, tarkoittaen että kustakin ruudusta voidaan mennä vain kahdeksaan suuntaan, eli vain viereisiin ruutuihin. Basic Theta* ja Phi* tutkivat kaikilla kulmilla mahdollisia polkuja, mikä voi lyhentää matkaa, mutta tekee laskemisesta myös

magnituudin hitaampaa (Duchoň ym. 2014). Yksi vaihtoehto on aluksi luoda normaali JPS polku ja sen jälkeen tarkistaa solmukohtaisesti, voiko osan solmuista jättää pois. Oletetaan että on solmut P1, P2 ja P3 ja P1-P2 välillä on viiva on vaakatasossa vasemmalta oikealle ja P2-P3 välillä on viiva on 45° ylös oikealle. Jos mitään esteitä ei ole tiellä, voidaan P2 poistaa ja vetää viiva suoraan P1-P3 välille ja se on silloin lyhyempi. Tämä tapa voi lyhentää muutamia prosentteja alkuperäisestä JPS polusta ja on myös vain muutaman prosentin hitaampi (Lee, Jia ja Song 2022), mutta ei luo yhtä hyviä reittejä kuin Basic Theta* ja Phi*.

5.3 Epätodennäköiset kehitysmahdollisuudet

Jos palataan polunetsintäalgoritmien juurilla, tässä tapauksessa Dijkstra ja A* algoritmeihin, huomataan että ne toimivat myös painotetuissa graafeissa, toisin kuin JPS. Neliöruudukot ovat tavallaan painotettuja graafeja, joissa jokaisella solmulla on 8 yhteyttä ja jokaisen paino on sama. Tämän takia JPS algoritmia voidaan käyttää tässä ja hypätä ruutujen yli, mikä ei toimi erilaisissa graafeissa. JPS on fundamentaalisti kyvytön käsittelemään erilaisia graafeja ja on hyvin epätodennäköistä että se tulee tulevaisuudessakaan olemaan. Painotettujen graafien ja perusruudukon välimuotona toimii luonnosta löytyvä maasto, joka voi olla ylhäältäpäin katsottuna vain perusruudukko-ongelma, jossa puut ja rakennukset ovat ainoat esteet. Todellisuudessaa maastossa on kohoamia, portaita ja muita hidasteita, jotka eivät itsessään estä liikkumista, mutta nopeampaa voisi olla esimerkiksi kiertää kukkula. Tälläiset muutokset saavat tämän muistuttamaan enemmän painotettua ruudukkoa, joten JPS ei tälläisenään löydä välttämättä nopeinta reittiä.

6 Johtopäätökset

JPS on sellaisenaan jo hyvin kykenevä ja kilpailukykyinen. Sillä ei ole oikeaa kilpailua, joka voisi syrjäyttää sen johtoaseman. Silti sen ohi menee hitaammat ja epäoptimaalit algoritmit. Koska mitään syytä tähän ei ole löytynyt mistään, voidaan vain spekuloida syitä tähän. Yksinkertaisimmillaan voi olettaa, että kukaan ei vain yksinkertaisesti ole kuullut JPS polunetsintäalgoritmista. Se ei ole ohjelmistokehityksessä mahdotonta, sillä uutta teknologiaa ja ohjelmistoa tulee kokoajan enemmän, jolloin pakostikin osa jää huomiotta. Tämän väitteen syrjäyttää Google Scholar, jossa JPS julkaisupaperilla on melkein puoli tuhatta viittausta, joten se ei voi olla tuntematonta tekniikkaa. Toisaalta pelikehittäjät eivät mitä luultavimmin lue ylimääräisiä papereita ja käyttävät ”tarpeeksi hyvin”-toimivaksi todettua algoritmia. Pelikehityksessä on usein kova kiire, eikä ole aikaa optimoida jokaista aspektia pelin sisällä, jolloin helposti tulee vain kopioitua jo valmis kirjasto, mikä saattaa olla vanhentunut. Näihin vanhentuneisiin ja epäoptimeihin kuuluu monen muun A* derivatiivin ohella etenkin HPA* algoritmi, mikä on alan johtava standardi pelikehityksessä (Foead ym. 2021).

JPS algoritmilla on useampia derivatiiveja, niin ”virallisista” eli alkuperäisen kehittäjän luomia, kuin ”epävirallisia” eli yksittäisten tahojen luomia. JPS (B), JPS (B+P), JPS+ ja JPS+ (P) nopeuttavat jo ennestään nopeaa JPS algoritmia (Harabor ja Grastien 2014), kun taas eri JPS-3D algoritmit lisäävät JPS:n tutkittavaksi yhden uuden ulottuvuuden (Nobes ym. 2022).

JPS on osoittanut olevansa monipuolinen, todistaen sen jatkokehityksen olevan mahdollista. Tämän takia sillä on vielä paljon lukittua potentiaalia ja monia jatkokehityksen mahdollisuuksia. Näistä kaikkia ei ole edes vielä osattu keksiä yrittää, kun taas toisia on spekuloitu ilman toteutusta, kuten esimerkiksi JPS hexa-ruudukossa (Harabor ja Grastien 2011).

Lähteet

Candra, Ade, Mohammad Andri Budiman ja Kevin Hartanto. 2020. “Dijkstra’s and a-star in finding the shortest path: a tutorial”. Teoksessa *2020 International Conference on Data Science, Artificial Intelligence, and Business Analytics (DATABIA),* 28–32. IEEE.

Duchoň, František, Andrej Babinec, Martin Kajan, Peter Beňo, Martin Florek, Tomáš Fico ja Ladislav Jurišica. 2014. “Path planning with modified a star algorithm for a mobile robot”. *Procedia engineering* 96:59–69.

Foead, Daniel, Alifio Ghifari, Marchel Budi Kusuma, Novita Hanafiah ja Eric Gunawan. 2021. “A systematic literature review of A* pathfinding”. *Procedia Computer Science* 179:507–514.

Harabor, Daniel ja Alban Grastien. 2011. “Online graph pruning for pathfinding on grid maps”. Teoksessa *Proceedings of the AAAI Conference on Artificial Intelligence,* 25:1114–1119. 1.

———. 2014. “Improving jump point search”. Teoksessa *Proceedings of the International Conference on Automated Planning and Scheduling,* 24:128–135.

Knuth, Donald E. 1997. *The Art of Computer Programming: Fundamental Algorithms, volume 1.* Addison-Wesley Professional.

Lee, Sihao, Qingxuan Jia ja Jinzhou Song. 2022. “An optimization algorithm for path generated by Jump point search”. Teoksessa *2022 IEEE International Conference on Mechatronics and Automation (ICMA),* 1574–1580. IEEE.

Medina, Raymond Carlos S, Hannah Shane B Gittabao ja Vivien A Agustin. n.d. “Modifying JPS Algorithm Using Navmesh Data Structure Applied in 3D Using Unity”.

Nobes, Thomas K, Daniel Harabor, Michael Wybrow ja Stuart DC Walsh. 2022. “The jps pathfinding system in 3d”. Teoksessa *Proceedings of the International Symposium on Combinatorial Search,* 15:145–152. 1.

Rafiq, Abdul, Tuty Asmawaty Abdul Kadir ja Siti Normaziah Ihsan. 2020. “Pathfinding Algorithms in game development”. Teoksessa *IOP Conference Series: Materials Science and Engineering,* 769:012021. 1. IOP Publishing.

Ranttila, Pertti. 2019. “JPS Algorithm Adaptation and Optimization to Three-dimensional Space”.

Witmer, Nathan. n.d. “A Visual Explanation of Jump Point Search”. Viitattu 14. toukokuuta 2024. https://zerowidth.com/2013/a-visual-explanation-of-jump-point-search/.