Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Tuomo Karhu

Muistipohjaisen reititysalgoritmin luonti

Metropolia Ammattikorkeakoulu

Insinööri

Tietotekniikan koulutusohjelma

Opinnäytetyö

1.5.2019

Tekijä(t) Otsikko Sivumäärä Aika	Tuomo Karhu Muistipohjaisen reititysalgoritmin luonti 46 sivua + 0 liitettä 9.5.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tietotekniikan koulutusohjelma
Suuntautumisvaihtoehto	Ohjelmointi
Ohjaaja(t)	Osaamisaluepäällikkö Janne Salonen

Opinnäytetyössä oli tavoitteena kehittää uusi nopeampi reitinhakualgoritmi, joka toimisi nopeammin kuin A*-reitinhakualgoritmi. Algoritmia on tarkoitus käyttää ruutupohjaisessa pelikartassa. Tarkoitus oli kehittää reitinhakualgoritmi siten, että reittejä ei etsitä kartalta vaan ne haetaan suoraan muistista. Tämä säästää aikaa ja vähentää suorittimen työkuormaa. Muistipohjaisen reitinhakualgoritmin kehittämiseksi aluksi oli kuitenkin tutkittava, onko olemassa laitteita, jotka pitävät reittejä muistissa.

Tutkimme aluksi, miten Dijkstran-algoritmi ja sitä hyödyntävä A*-reitinhakualgoritmi toimii. Syvensimme tutkimusta selvittämällä, miten Dijkstran-algoritmia hyödynnetään tietoverkkotekniikassa. Tietoverkkotekniikassa reitittimet käyttävät useita eri reititinprotokollia, jotka hyödyntävät Dijkstran-algoritmia. Päädyimme tutkimaan OSPF-reititinprotokollaa, joka toimii hyvin samankaltaisesti kuin A*-reitinhakualgoritmi. OSPF-protokolla kuitenkin pitää kaikki reitit muistissa ja on siten haluamamme muistipohjaisen reitinhakualgoritmin kaltainen. OSPF-protokollan pohjalta kehitimme oman protokollan, jolla kartan ruudut eli noodit pystyivät vaihtamaan reittitietoja keskenään. Tämä oli avaintekniikka, jolla pystyimme luomaan muistipohjaisen reitinhakualgoritmin.

Muistinkäyttö osoittautui kuitenkin haastavaksi, joten tiivistimme reittilistaa alueilla. Jaoimme kartan alueisiin IP-osoitteiden verkon osoitteiden kaltaisesti. Tämä tekniikka vähensi muistinkäyttöä riittävästi, että reitinhakualgoritmi voitiin todeta toimivaksi.

Opinnäytetyön tuloksena valmistui reitinhakualgoritmi, joka on moninkertaisesti A*-reitinhakualgoritmia tehokkaampi. Ainoaksi heikkoudeksi osoittautui kartan latauksen hitaus. Reittien muodostus vie aikansa ja hidastaa siten kartan latausaikaa. Muistinkäyttö on kuitenkin aluejaon ansiosta sopivan pieni.

Avainsanat	reitinhaku, algoritmit, A*

Author(s) Title	Tuomo Karhu Creating memory-based pathfinding algorithm
Number of Pages Date	46 pages + 0 appendices 9 May 2019
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Specialisation option	Programming
Instructor	Janne Salonen, Head of School of ICT

The purpose of this final year project was to create a new memory-based pathfinding algorithm for 2D tile-based video games. The goal was that the new algorithm should be faster than the A*-pathfinding algorithm to be able to support more non-player characters. In a memory-based pathfinding algorithm all the routes should be stored in memory. This gives the advantage of needing less processing power since all the routes can be fetched from memory instead of being searched.

The memory-based algorithm was created by doing research on how Dijkstran algorithm was used in router technology. OSPF routing protocol was the primary researched technology that made it possible to create a route exchange protocol for the memory-based pathfinding algorithm.

As a result of this thesis, a new algorithm that was roughly 10 times faster than A* pathfinding algorithm was created. The only notable disadvantages were higher memory consumption and longer initial load time for the map.

In conclusion the new memory-based pathfinding algorithm created in this thesis is an excellent algorithm for 2D tile-based games that need to support a high number of non-player characters.

Keywords	pathfinding, algorithms, A*

Sisällys

1 Johdanto 1

2 Reitinhakualgoritmien tutkiminen 1

2.1 Dijkstran algoritmi 2
2.2 A*-reitinhakualgoritmi 3
2.3 Reittilista 3
2.4 Reititys tietoverkkotekniikassa 4
2.4.1 IP-osoite 5
2.4.2 OSPF reititysprotokolla 6
2.5 Reittilistan protokolla 7
2.6 Reittilistan reittien tiivistäminen 9
2.7 Aluetaulun protokolla 9
2.8 Testaus 10

3 Toteutus 10

3.1 Nykyinen toteutus A*-reitinhakualgoritmilla 10
3.1.1 Kartta A*-reitinhakualgoritmiin 10
3.1.2 Reitti-luokka 12
3.1.3 AstarPathFinder-luokka 13
3.1.4 ClosestHeurestic-luokka 15
3.1.5 A*-reitinhaun kulku 15
3.2 Uusi muistipohjainen reitinhaku 19
3.2.1 Noodien ja reittilistan toiminta 20
3.2.2 Noodien tilan muuttaminen 21
3.2.3 Aluelistan toiminta 22
3.2.4 Reitinhakualgoritmi 23
3.2.5 Node-luokka 24
3.2.6 Route-luokka 28
3.2.7 AbstractRouteList-luokka 29
3.2.8 RouteList-luokka 34
3.2.9 AreaList-luokka 35
3.3 Testaus 35
3.3.1 Testikartat 36
3.3.2 Testaamisen toteutus 39

4 Testaustulokset 40

4.1 Yhden reitinhaku 40

4.2 Kymmenen reitinhaku 41
4.3 Sadan reitinhaku 42
4.4 Tuhannen peräkkäisen reitinhaku 43
4.5 Testitulosten analysointi 44

5 Yhteenveto 44

1 Johdanto

Nykyisistä pelimaailmoissa käytetyistä reitinhakualgoritmeista A* tuntuu olevan ainoa vaihtoehto. Se on algoritmi, jossa lyhin reitti etsitään kartalta. Havaitsin kuitenkin, että pelissäni kaikki 2D-kartat olivat suhteellisen rajattuja, joten aloin miettimään reitin etsiminen joka kerta ole hirveän raskas operaatio. Mitä jos reitit laitettaisiin muistiin? Tämä mahdollistaisi yksinkertaisesti reitin noutamisen muistista, jolloin operaatio veisi huomattavasti vähemmän aikaa. Tämä auttaisi huomattavasti prosessorin kuorman vähentämisessä, joka vapauttaisi tehoja muihin toimintoihin. Normaalisti prosessorin tehot eivät peleissä ole niinkään ongelma, mutta koska pelissäni oli tarkoitus simuloida kymmeniä tuhansia ihmisiä erilaisissa avaruusasemissa ja avaruudessa, olisi kaikki säästöt tarpeen.

Opinnäytetyön tavoite on luoda uusi vaihtoehtoinen tehokkaampi reitinhakualgoritmi peliini jo toteutetun A*-reitinhakualgoritmin tilalle. A*-reitinhakualgoritmissa reitti etsitään tutkimalla alkuruudun ympärillä olevia ruutuja. Haluan kuitenkin, että kartan jokaisella ruudulla on valmiina tieto siitä, miten muihin ruutuihin päästään. Tämän pitäisi tehostaa reitinhakua moninkertaisesti, koska reitit ovat valmiina eikä ohjelman tarvitse tehdä etsimistä aina alusta. Muistin käyttöä tulee kuitenkin rajoittaa niin, että peliä on mahdollista pelata neljän gigatavun ram-muistilla.

Muistipohjaisen reitinhakualgoritmin kehittämiseksi tutkin reitittimien käyttämiä reititysprotokollia. Niiden pohjalta on tarkoitus kehittää uusi protokolla, jolla kartan noodit voivat luoda ja jakaa reittejä keskenään. Tällöin kehittäjän ei tarvitse itse generoida reittejä noodeille, vaan ne luodaan automaattisesti.

Työ toteutetaan Java-kielellä ja tehokkuus nykyiseen A*-reitinhakualgoritmiin vertaillaan Junit-testeillä eri kokokoisilla ja tyylisillä kartoilla.

2 Reitinhakualgoritmien tutkiminen

Reitinhakualgoritmien tarkoitus on löytää paras reitti alkupisteestä tavoitepisteeseen. Paras reitti voi olla nopein, lyhin, luotettavin tai minkä tahansa käyttäjän määrittämän kriteerin toteuttava reitti. Reitinhakualgoritmejä käytetään nykyään niin peleissä, navigaattoreissa kuin reitittimissä. Tässä työssä käsittelemme Dijkstran algoritmiin perustuvia reitinhakualgoritmejä, joihon pelissäni käyttämä A*-reitinhakualgoritmi perustuu.

2.1 Dijkstran algoritmi

Suurin osa nykyisin käytetyistä reitinhakualgoritmeistä perustuu Dijkstran algoritmiin. Dijkstran algoritmi on kehitetty etsimään paras reitti tutkimalla halvimmat reitit ensiksi. Algoritmista käytetään myös englanniksi nimitystä ”Shortest Path First” eli suomennettuna lyhin reitti ensiksi.

Algoritmi toimii reittien pisteytyksellä. Jokaisen noodien välisellä reitillä pitää olla hinta. Hinnoissa pienempi luku on parempi. Algoritmillä on avoin ja suljettu lista. Avoin lista sisältää tutkittavat noodit, ja suljettu lista sisältää jo tutkitut noodit. Avoimessa listassa tutkittavat noodit pidetään aina järjestyksessä halvimmasta kalleimpaan noodiin.

Algoritmi aloittaa parhaan reitin etsinnän tutkimalla kaikki aloituspisteeseen yhteydessä olevat noodit. Reitti laskee kullekin tutkitulle noodille hinnan lisäämällä noodin hinnan lähtöhintaan. Lähtöhinta on reitin hinta aloituspisteestä tutkittavan reitin alkupisteeseen. Aloituspisteessä kuitenkin lähtöhinta on nolla. Reittien kohdenoodit lisätään avoimeen listaan ja järjestetään hinta järjestykseen halvimmasta kalleimpaan. Tutkittava noodi lisätään tämän jälkeen suljettuun listaan. Seuraavaksi algoritmi valitsee avoimesta listasta ensimmäisen eli halvimman noodin tutkittavaksi. Algoritmi tutkii kaikki tähän noodin yhteydessä olevat noodit, laskee reittien hinnat ja lisää ne avoimeen listaan. Tutkittava noodi lisätään suljettuun listaan ja etsintäluuppi alkaa taas alusta.

Reitinhaku päättyy, kun kohdenoodi lisätään suljettuun listaan. Tämän jälkeen algoritmi valitsee halvimman reitin, joka on lopullinen reitti. Reitinhaku päättyy myös, jos avoin lista on tyhjä eli kaikki noodit on jo tutkittu. Tämä tarkoittaa sitä, että reittiä alkunoodin ja kohdenoodin välillä ei ole.

Dijkstran algoritmin ongelma on siinä, että se tutkii aina halvimmat reitit ensiksi. Tilanteessa, jossa etsittävä kohdenoodi on pelkästään kahden kalliin reitin takana, reittiä lähdetään hakemaan halvemmista eli vääristä suunnista ensiksi. Suurissa kartoissa tämä voi tarkoittaa suurtakin ajan tuhlausta.

2.2 A*-reitinhakualgoritmi

A*-reitinhakualgoritmi on korjaa Dijkstran ongelmat lisäämällä etäisyyden reittien hintaan. Djistran algoritmin tavoin A*-algoritmi käyttää ruutujen siirtomaksuja päättääkseen, kuinka hyvä reitti on. A*-algoritmi kuitenkin lisää siirtomaksuihin etäisyys hinnan, joka auttaa algoritmia hahmottamaan, ollaanko menossa kohti vai pois päin kohteesta. Tämä mahdollistaa sen, että halpoja reittejä ei tutkita liian pitkälle väärään suuntaan, koska etäisyys ja hinta kasvavat.

Niin siirtomaksut kuin etäisyysmaksut ovat muunneltavissa. Etäisyysmaksuksi voidaan muuttaa tarpeen mukaan eri heuristiikoilla. Yksinkertaisin heuristiikka, jota itse pelissä käytän, on tutkittavan noodin suora etäisyys kohdenoodista. Pelin ruutukartasta tämä saadaan yksinkertaisesti laskemalla neliöjuuri x:n kahteen potenssin ja y:n kahteen potenssin summasta. Etäisyysmaksua kuitenkin voidaan korottaa esimerkiksi kertomalla kahdella, jos halutaan suosia reittimaksua etäisyyden sijaan. Jakamalla kahdella taas suositaan siirtomaksuja, joten käyttäjällä on monta eri mahdollisuutta muuntaa algoritmia omaan käyttöön sopivaksi.

Algoritmi etsii reitin aina kutsuttaessa uudestaan, vaikka samaa reittiä olisi juuri haettu. Tämä tekee algoritmista raskaan suorittimelle, mutta mahdollistaa reitin löytämisen aina myös muuttuvissa kentissä.

2.3 Reittilista

Työn tarkoitus on tehdä reitinhakumenetelmä, jossa jokaisella ruudulla on reittilista muistissa. Reittilistan avulla jokainen reitti on valmiiksi muistissa, joten sitä ei tarvitse luoda etsimällä. Tämä säästää suorittimen tehoa, kuitenkin muistin kustannuksella. Koska jokaisella ruudulla pitää olla muistissa reitti jokaiseen muuhun kuin omaan ruutuun, tulee muistinkäyttö olemaan valtava. Onkin parempi, että koko reitin sijaan reittilistassa on vain tieto seuraavasta ruudusta, jolla päästään kohti tavoiteruutua. Näin reitti voidaan luoda hakemalla aina seuraavan ruudun reittilistasta saatu tieto seuraavasta ruudusta. Kuitenkin on parempi, että reittilista sisältää vain kullekin ruudulle luodun uniikin tunnusnumeron eikä ruudun x- ja y-koordinaattia. Tämä puolittaa reittitaulun viemän tilanmäärän, vaikkakin tehokkuus kärsii hieman, koska tunnusnumerolla on haettava ruudun koordinaatit reittiä varten.

Muistinkäyttö tulee silti olemaan valtava kartan koon kasvaessa. Oletetaan, että reittitaulu sisältää vain objektin, jolla on kaksi muuttujaa. Näistä ensimmäinen on tavoite ruudun tunnus ja toinen on seuraavan ruudun tunnus. Kumpikin kokonaislukumuuttuja vie siten neljä tavua muistia (1). Yhteensä reittilista käyttää kahdeksan tavua muistia jokaista reittiä kohden.

Jokaisen reittilistan reittimäärän saamme seuraavalla kaavalla:

X * Y – 1

Kaavassa X on kartan ruutujen määrä leveyssuunnassa ja Y on kartan ruutujen määrä korkeussuunnassa. Kertomalla tämä määrä reitin muistikäyttömäärällä eli kahdeksalla ja ruutujen kokonaismäärällä saamme seuraavan kaavan koko kartan muistinkäyttöön:

X * Y * ((X * Y – 1) * 8)

Näillä tiedoilla tiedämme, että 100 ruudun levyisen ja 100 ruudun pituisen kartan jokaisella ruudulla tulee olemaan reittilista, jossa on 9999 reittiä. Jokainen ruutu tulisi siis viemään 8 * 9999 eli 79992 tavua muistia. Kokonaismuistimäärä kartalle tulisi olemaan 799 920 000 tavua eli noin 800 megatavua muistia. Käytettävä muistinmäärä kuulostaa liian suurelta toimiakseen, joten kenttien pitää olla hyvin pieniä, jotta tämä hakutapa on käytännöllinen. Tuhannen ruudun levyisellä ja tuhannen ruudun korkuisella kartalla muistin käyttö olisi jo 7999,992 gigatavua eli reilusti yli tavoitellun neljän gigatavun. Reittilistan tilanvientiä on liian suuri, joten reittien määrää on jotenkin tiivistettävä.

2.4 Reititys tietoverkkotekniikassa

Reititystaulu käsittää tietoverkkotekniikassa reittitaulun, jolla löydetään reitti tiettyyn osoitteeseen. Reititystauluja löytyy reitittimistä, jotka ohjaavat internetin liikennettä paikasta toiseen. Sen tärkein tehtävä on pitää yllä topologiaa lähiympäristön verkosta. Reititystaulu sisältyy tyypillisesti kolmesta eri tiedosta.

1. Verkon tunnus (network id)

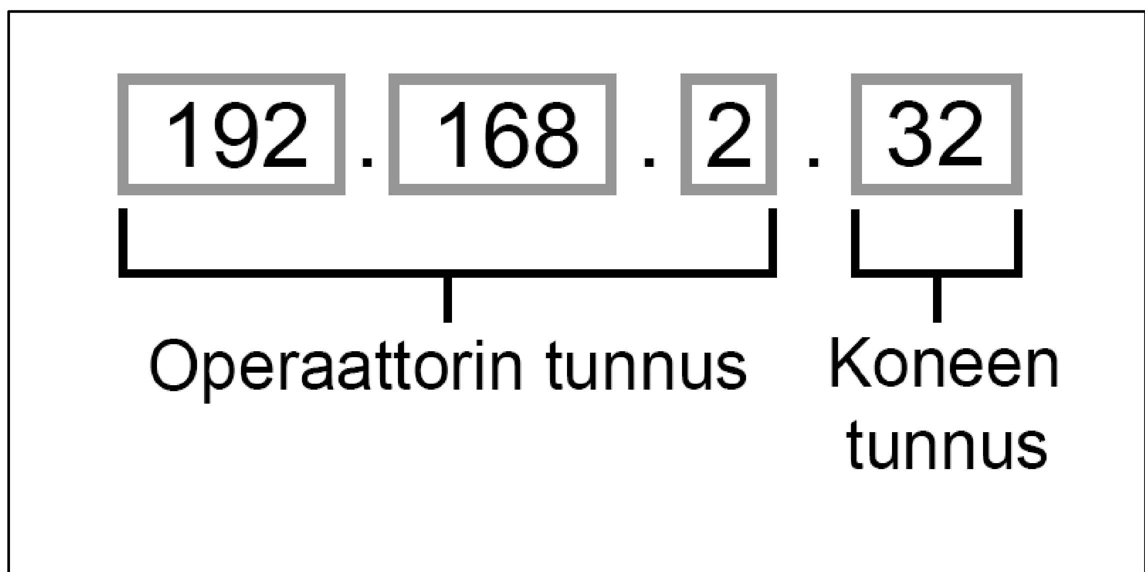
2. metriikka (metrics)

3. seuraava osoite (next hop).

Verkon tunnus sisältää verkon osoitteen ja verkon peitteen. Verkon tunnus niputtaa useamman osoitteen yhteen, jotta useampi IP-osoite voidaan ohjata koskemaan yhtä reititystaulun riviä. Metriikka on reitittimen aputieto reitityspäätöksen tekoa varten. Tämä voi sisältää esimerkiksi viiveen, jolloin reititin voi valita kahdesta samasta reititysosoitteesta nopeammin reitin. Metriikan muoto riippuu käytetystä reititysprotokollasta. Seuraava osoite on verkossa olevan seuraavan laitteen IP-osoite.

2.4.1 IP-osoite

Tällä hetkellä käytössä on kaksi eri Internetin protokollaosoitetta eli IP-osoitetta. Nämä ovat IP:n versio 4 ja versio 6. Käsittelen tässä kappaleessa vain IP:n neljättä versiota, joka tunnetaan myös nimeltä IPv4. IP-osoite muodostuu neljästä pisteellä erotetusta numerosarjasta (3). Näistä numerosarjoista ensimmäinen on väliltä 1-255 ja loput kolme ovat väliltä 0-255. Reitittimet kuitenkin käsittelevät näitä numeroita neljän kahdeksan bitin sarjana eli 32-bittisenä osoitteena.

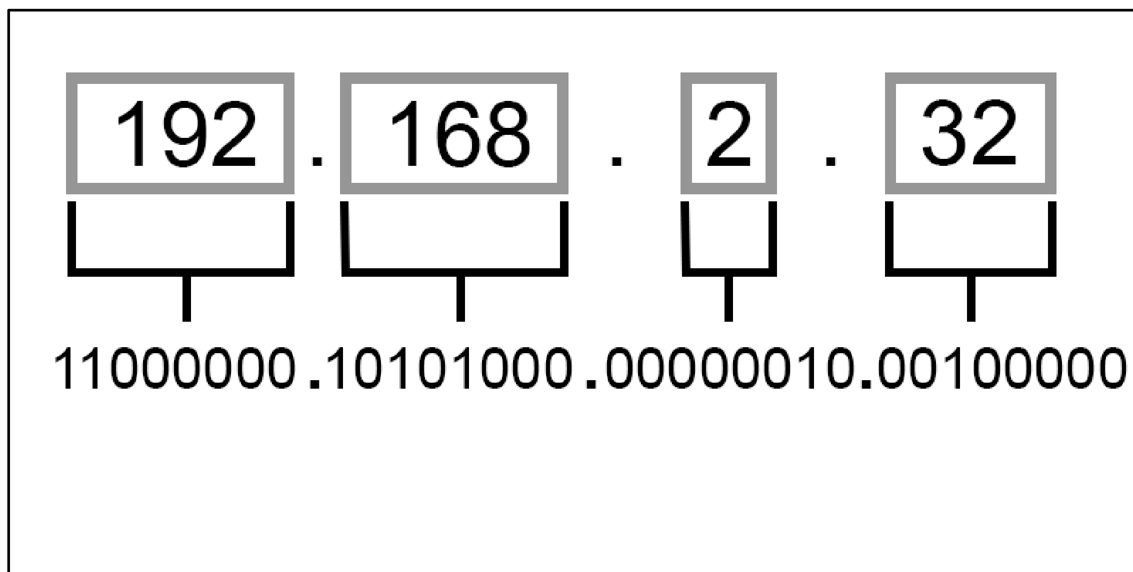


Kuva 1. IP-osoitteen koostumus

IP-osoitteen alkuosa on verkon osoite eli operaattorin tunnus ja loppuosa koneen tai verkon oma tunnus. Nämä kaksi osaa voidaan erotella toistaan verkon peitteellä. Verkon peite voidaan ilmaista kahdella eri tavalla. CIDR-notaatiossa verkon tunnus ilmoitetaan bitteinä. Ilmoitettujen bittien määrä vastaa verkko tunnuksen osuutta 32-bittisen osoitteen alkupäästä. Vastaavasti perinteisellä tavalla ilmoitettuna verkon peite ilmoitetaan aivan kuten IP-osoite neljänä eri desimaaleiksi käännettyinä kahdeksan bitin sarjoina,

Metropolia

jotka on eroteltu pisteillä. Kukin neljästä kahdeksan bitin osasta käännetään desimaaleiksi.



Kuva 2. IP-osoite desimaaleina ja bitteinä

Kaikkien verkon osoitteiden listaus olisi mahdotonta reitittimille, joten reitittimet niputtavat verkko-osoitteet verkon peitteiden avulla. Tämä mahdollistaa muistin huomattavan säästämisen, koska yhdellä rivillä voidaan ohjata useampaa osoitetta.

Verkon peitteen vastakohtana toimii aliverkko, joka kertoo verkossa olevien laitteiden määrän. Luku esitetään verkon peitteen ja IP-osoitteen tavoin neljänä kahdeksan bitin sarjana, jotka esitetään desimaaleina. Toisin kuin verkon peite aliverkko ilmoitetaan lopusta päin.

Aliverkko ei kuitenkaan välttämättä ole verkon tunnuksen käänteinen osoite, sillä yksi verkon tunnus voidaan jakaa useampaan aliverkkoon. Näin tehdään usein esimerkiksi yritysten eri osastojen välillä turvallisuuden parantamiseksi.

2.4.2 OSPF reititysprotokolla

Reititysprotokollat ovat reitittimien menetelmä jakaa tietoa tuntemistaan verkko-osoitteista keskenään. Ne osaavat päivittää tiedon verkon rakenteen muutoksesta automaattisesti. Tämä mahdollistaa sen, että uuden verkkoon kytketyn laitteen tiedot välittyvät

muille reitittimille. Tässä luvussa keskitymme OSPF eli Open Shortest Path First-reititysprotokollaan (2). Protokolla käyttää Dijkstran algoritmia löytääkseen lyhimmän reitin kohteeseen.

Toisiinsa kytketyt reitittimet muodostavat naapurikunnan. Kun reitittimet ovat naapureita, ne voivat vaihtaa tietoa tuntemistaan laitteista keskenään. Reitittimet kommunikoivat keskenään LSA-viesteillä, jotka sisältävät tietoa reitittimeen kytketyistä ja muista tuntemistaan laitteista. Reitittimet valitsevat parhaan reitin LSDB-tietokantaansa.

Kun OSPF-reititysprokollaa käyttävä reititin kytketään verkkoon suoraan toiseen reitittimeen, lähettää se tervehdysviestin kytkettyyn reitittimeen. Viestin vastaanottava reititin tunnistaa, että kyseessä on uuden reititin tunnuksen omaava reititin, joten se lisää reitittimen omaan muistiinsa naapuriksi. Tämän jälkeen se lähettää omat tietonsa takaisin kytketylle reitittimelle. Kytketty reititin lukee viestin ja lisää naapurireitittimen tiedot omaan muistiinsa. Lopuksi se vielä palauttaa kuitin naapurireitittimelle.

Kun reitittimet ovat löytäneet naapurinsa, lähettävät ne toisilleen DBD-viestin, joka sisältää kaikki tuntemansa verkot. Kun reititin löytää viestistä verkon, jota sillä ei ole listassa, pyytää se naapuriltaan tiedot tästä verkosta Link State Request -viestillä eli LSR-viestillä. Tähän naapuri vastaa Link State Update -viestillä eli LSU-viestillä. LSU-viesti sisältää kaikki tämän verkon tiedot. Kun reititin vastaanottaa LSU-viestin, se lisää siitä saamansa tiedot itselleen. Lopuksi reititin kuittaa naapurilleen Link State Acknowledge -viestillä eli LSAck-viestillä naapurilleen, että sai tiedot itselleen. Reitittimet tekevät saman viestiketjun kaikkia uusiksi tunnistamiaan verkkoja kohden.

Reititin voi olla myös kytkettynä useaan reitittimeen samanaikaisesti. Näissä tapauksissa on mahdollista, että reititin joutuu tilanteeseen, jossa se voi saada lähetettyä viestin vieraaseen verkkoon useampaa eri reititintä pitkin. Näissä tapauksissa reititin käyttää sitä reittiä, jossa on pienin hinta-arvo eli path cost. Hinta-arvo perustuu kytketyn linjan nopeuteen. Mitä pienempi hinta eli suurempi linjan nopeus sitä parempi reitti on. Hinta-arvo ei kuitenkaan perustu vain seuraavan reitin hinta-arvoon vaan koko reitin hinta-arvojen summaan.

2.5 Reittilistan protokolla

Tietoverkkotekniikasta ja varsinkin OSPF-protokolasta voimme oppia, miten reitinjakoprotokollan tulisi toimia reittilistalle. Reittilistan omistavan noodin tulisi ottaa yhteyttä

naapureina oleviin noodeihin ja lisätä näiden noodien tunnukset omaan reittilistaansa sekä seuraavana noodina että kohdenoodina.

Kun reittilista on päivitetty, noodin pitää pystyä kertomaan siitä naapurinoodeille päivitysilmoituksella. Naapurit voivat tämän jälkeen pyytää päivitetty reittilista naapureilta ja tarkistaa sen sisällön. Uudet noodit voidaan suoraan lisätä omaan reittilistaan, mutta jo olemassa olevien reittien kanssa tilanne on ongelmallisempi. Reittilista tarvitsee tavan päättää, onko jo olemassa oleva reitti parempi kuin uusi reitti. Tämä tarkoittaa sitä, että reittitauluun on lisättävä kolmas sarake, joka kertoo reitin paremmuuden. Reitin paremmuuden mittaamiseksi toimii parhaiten arvo reitin pituudesta. Tällöin naapurinoodit merkkaavat reitin arvoksi yksi ja naapureiden naapurit arvon kaksi.

Tähän voidaan vielä lisätä arvo noodin läpikulkumaksusta, jolloin arvo ei kasvakaan aina yhdellä vaan noodin läpikulkumaksulla. Tällöin naapurinoodi merkkaakin reitin arvoksi kohdenoodin läpikulkumaksun. Naapurin naapurinoodi taas saisi naapurinoodilta reitin, jossa on jo kohdenoodin läpikulkumaksu. Tähän lukuun sen pitää lisätä naapurin läpikulkumaksu saadakseen lopullisen reitinarvon.

Näillä tiedoilla reittitaulu pystyy vertailemaan olemassa olevia reittejä ja valitsemaan pienimmän hinnan reitin. Pelissä saattaa olla myös mekaniikka, jolloin esimerkiksi ovi on jumissa. Tämä tarkoittaa sitä, että noodi tulee merkata läpikulkemattomaksi. Tämän mahdollistamiseksi noodin tulee kyetä ilmoittamaan, että se on pois käytöstä. Tämä onnistuu ilmoittamalla naapurinoodeille tilan muutoksesta. Tilamuutoksen saadessaan naapurinoodin tulee poistaa reittitaulusta kaikki reitit, jotka kulkevat pois käytöstä olevan noodin läpi. Tämä aiheuttaa reittitaulun muutoksen, jolloin naapurin naapurin tulee verrata omia reittejään naapurin reitteihin. Huomatessa, että reitit ovat poistuneet reittitaulusta, tulee sen poistaa myös kaikki nämä reitit reittitaulusta. Protokollan pitäisi kyetä päivittämään reitti siten, että ovi kierretään, jos oven kiertävä reitti on mahdollinen.

Vaihtoehtoisen reitin päivitys reittilistaan pitää myös onnistua automaattisesti. Poistaessaan reitin omasta reittilista noodin on myös pyydettävä naapureita antamaan omat reittinsä. Jos se löytää poistetun reitin, joka ei kulje itsensä kautta, tulee sen lisätä tämä reitti omaan reittilistaansa. Tämän jälkeen tämän noodin naapurit kuulevat päivityksestä, ja uusi vaihtoehtoinen reitti lähtee leviämään.

2.6 Reittilistan reittien tiivistäminen

Todettuamme, että reittilistan reitit tulevat tarvitsemaan myös tiedon reitin hinnasta, on reittilistan muistinkäyttö arvioitava uudestaan. Jokainen reittilistan reitin tieto vaatii kolme tietoa, jotka ovat kohteen tunnus, seuraava noodi ja reitin hinta. Tämä tarkoittaa sitä, että jokainen reitti vie 12 tavua muistia (1). Nyt kokonaismuistinkäyttö 100x100-kartalla olisi noin 1200 megatavua edellisen laskelman noin 800 megatavun sijaan.

Reittitaulun reittien vähentämiseksi noodit on jaettava alueisiin. Noodeilla tulisi siis olla aluekoodi. Reittilista voisi muistinkäytön vähentämiseksi sisältää vain oman alueensa reitit ja naapurialueen noodit, jotka ovat oman alueen noodien suoria naapureita. Kunkin alueen noodeilla tulee olla yhteinen aluetaulu. Aluelistan pitää sisältää tieto siitä, minkä alueen kautta kuljetaan muille alueille. Alueiden tulisi ideaalissa tilanteessa olla yhteydessä vain yhdellä noodilla. Tällöin alueiden välillä siirtyminen toimisi helposti etsimällä reittilistasta se noodi, jonka alue on haluttu. Jos halutulle alueelle pystytään kulkemaan useamman noodin kautta, on haettava halvimman reittihinnan noodi. Tämä vertailu hidastaa hakua hiukan, mutta ei merkittävästi. Kartta kuitenkin kannattaa suunnitella siten, että alueita yhdistettäviä noodeja on mahdollisimman vähän.

2.7 Aluetaulun protokolla

Aluelistan tulisi käyttää reittilistan tavoin protokollaa naapurialueiden löytämiseksi ja tiedon välittämiseksi naapuri alueille. Kun reittilistaan lisätään noodi, jonka alue on muu kuin oma alue, pitää reittilistan välittää tieto aluetaululle. Jos alue ei ole vielä aluetaulussa, lisätään se uutena yhteytenä listaansa. Tämän jälkeen alue voi kertoa naapurialueelle tiedon yhteydestä.

Kun alueiden välinen yhteys on muodostettu, voivat alueet jakaa tietoa tuntemistaan alueista. Tämän jälkeen aluetaulu voi ilmoittaa reittilistan tavoin naapureilla aina kun omassa listassaan tapahtuu muutos, joko alueen lisäyksen tai poiston muodossa. Kun aluelista löytää naapurin aluetaulusta alueen, johon se ei ole suoraan yhteydessä, tulee sen pyytää naapurialuetaululta siirtymähinta. Siirtymähinnan tulee olla, reittitaulusta löydetyistä reiteistä, lyhimmän reitin reittimaksu. Aluetaulu voi tämän jälkeen lisätä alueen ja hinnan omaan listaansa.

2.8 Testaus

Molempien protokollien testaus toteutetaan käyttäen Junit-testausta. Junit on Javan oma testauskirjasto, jolla Java-luokkia voidaan testata. Tarkoitus on analysoida kummankin eri reitinhakualgoritmin paremmuus. Kriteereinä paremmuudelle ovat haun nopeus ja tarkkuus.

Testauksen reiluuden vuoksi kummankin reitinhaun menetelmän tulee testata reitinhakua samoissa kartoissa. Testikarttoja on kaksi kappaletta, joista ensimmäinen on lähes täysin avoin kenttä, jossa on vain sallittuja noodeja. Toinen testikenttä sisältää estettyjä noodeja toistuvassa muodostelmassa. Reitinhakua tullaan testaamaan lyhyen, keskipitkän ja pitkän matkan etäisyydeltä. Lyhyt matka tulee olemaan pituudeltaan alle 40 ruutua. Keskipitkän pituutena tulee olemaan noin 100 ruutua ja pitkän matkan pituutena noin 200 ruutua.

Hakunopeus tullaan mittaamaan ottamalla järjestelmän tarkka aika ennen hakua ja välittömästi haun jälkeen. Haku tarkkuuden tulee olla kummallakin menetelmällä täydellinen, jotta voimme todeta reitinhakualgoritmin toimivaksi.

3 Toteutus

3.1 Nykyinen toteutus A*-reitinhakualgoritmilla

A*-reitinhakualgoritmin tehtävänä on etsiä kartalta paras reitti. Haettu reitti sisältää koordinaatit alkupisteestä haettuun loppupisteeseen saakka. Algoritmi käyttää siirtomaksuja päättääkseen, kuinka hyvä reitti on. Algoritmi palauttaakin siirtopisteiltä halvimman reitin.

3.1.1 Kartta A*-reitinhakualgoritmiin

Kartta sisältää X * Y -määrän noodeja. Noodit ovat kartan ruutuja, joihin pelaaja voi liikkua kartalla. Jotta algoritmi toimisi, kartan pitää toteuttaa seuraavanlaiset Java-rajapinnan metodit:

```
public interface TileBasedMap {

        public int getWidthInTiles();
        public int getHeightInTiles();
        public void pathFinderVisited(int x, int y);
        public boolean blocked(Mover mover, int x, int y);
        public float getCost(Mover mover, int sx, int sy, int tx, int
ty);
```

Esimerkkikoodi 1. TileBasedMap-rajapinnan lähdekoodi

A*-reitinhakualgoritmi saa tarvittavat perustiedot kartasta rajapinnan vaatimien getWidtInTiles- ja getHeightInTiles-metodien avulla. Rajapinnan vaatiman getWidthInTiles-metodin tulee palauttaa kartan ruutujen määrä sivuttaissuunnassa. Vastaavasti getHeightInTiles-metodin tulee palauttaa kartan ruutujen määrä pystysuunnassa.

Itse haun toteuttamiseen rajapinta antaa myös vaatimuksen reitin kulkutiedoille. Rajapinnan pathFinderVisited-metodin tarkoitus on merkata kartan x- ja y-koordinaateista löytyvä ruutu tutkituksi. Tämän metodin avulla algoritmi estää saman ruudun tutkimisen uudelleen, joka voisi johtaa loputtomaan etsintäluuppiin. Rajapinnan blocked-metodi selvittää, onko liikkuminen x- ja y-koordinaateissa olevaan ruutuun sallittua. Mover on liikutettava hahmo, jota voidaan hyödyntää metodin toteutuksessa. Esimerkiksi jos siviilillä ei ole oikeutta olla vartioidulla alueella, voidaan metodin toteutuksessa estää liikkuminen ruutuun muille kuin vartijoille. Liikkuminen voidaan estää palauttamalla arvon epätosi ja sallia palauttamalla arvon tosi.

Rajapinnan getCost-metodi hakee siirtomaksun kahden ruudun liikkumisen välillä. Moverin eli hahmon nykyisen ruudun x-koordinaatti löytyy sx-muuttujasta ja y-koordinaatti sy-muutujasta. Vastaavasti tx-muuttuja sisältää kohderuudun x-koordinaatin ja ty-muuttuja kohderuudun y-koordinaatin. Metodi palauttaa arvon, joka ynnätään reitin hintaan. Pienempi arvo on aina parempi reitin kustannuksen kannalta, mutta muuten arvo on vapaasti valittavissa. Muistin käytön kannalta on kuitenkin hyvä pitää luku mahdollisimman pienenä. Itse käytin arvoa yksi normaaliin liikkumiseen ja arvoa kaksi ei toivottavaan liikkumiseen, kuten esimerkiksi tuolin sisältää ruudun läpi kulkemiseen. On hyvä huomioida, että myös negatiiviset arvot ja arvo nolla ovat sallittuja. Näitä voi käyttää muun muassa tietyn oven läpi kulkemisen suosimiseen. Jos huoneessa on kaksi ovea, esimerkiksi pääovi ja sivuovi, ja halutaan, että kulkureitit suosivat pääovea, niin on mahdollista

antaa pääovelle negatiivinen arvo, jotta sivu oven kautta kulkevat lyhyemmät reitit ovatkin arvoltaan pääoven kautta kulkevaa reittiä kalliimmat. Sama toteutus tietenkin toimisi antamalla sivuovien läpi liikkumiselle korkeampi hinta, joka on myös mahdollisesti koodin ymmärrettävyyden kannalta helpompilukuinen. Moverin eli hahmon avulla voidaan myös kustomoida siirtomaksua eri hahmotyypeille eli esim. ohjata hahmoja, joilla on eri ammatti, eri ovista.

3.1.2 Reitti-luokka

Reitti-luokka on reitinhakualgoritmissa käytettävä haettava reitti. Reitti sisältää tiedot haettavasta reitistä. Reitti on askeleita sisältävä ArrayList, johon voidaan lisätä haettuja askeleita. Askeleet ovat Step-luokan objekteja. Step-luokan objektit sisältävät vain x- ja y-koordinaatin sekä metodit niiden noudolle. Lisäksi luokka sisältää equals-metodin, jolla voidaan verrata kahden askeleen sijainteja ja päätellä, ovatko ne samat.

```
public class Step {
		private int x;
		private int y;

		public Step(int x, int y) {
			this.x = x;
			this.y = y;
		}

		public int getX() {
			return x;
		}

		public int getY() {
			return y;
		}

		public int hashCode() {
			return x*y;
```

Esimerkkikoodi 2. Step-luokan lähdekoodi

Reittiin voidaan lisätä askeleita sekä alkuun appendStep-metodilla että perään prependStep-metodilla. Metodilla getLength voidaan hakea reitin pituus, ja getStep-metodi hakee tietystä indeksistä löytyvän askeleen. Tietyn indeksin askeleen x- ja y-koordinaatit

voidaan hakea getX- ja getY-metodilla. Contains-metodi palauttaa totuusmuuttuja-arvona tiedon siitä, onko reitissä x- ja y-koordinaattia sisältävää askelta.

```
public class Path {
		//List of steps building the path
		private ArrayList<Step> steps = new ArrayList<Step>();

		public Path() {

		}

		public int getLength() {
			return steps.size();
		}

		public Step getStep(int index) {
			return (Step) steps.get(index);
		}

		public int getX(int index) {
			return getStep(index).x;
		}

		public int getY(int index) {
			return getStep(index).y;
		}
```
Esimerkkikoodi 3. Path-luokan lähdekoodi

3.1.3 AstarPathFinder-luokka

AstarPathFinder-luokka on A*-reitinhakualgoritmin pääluokka. Tämän luokan tarkoitus on toteuttaa reitin haku. Luokka lähdekoodi näyttää seuraavalta:

```
public class AStarPathFinder implements IPathFinder {

	private ArrayList closed = new ArrayList();
	private SortedList open = new SortedList();
	private TileBasedMap map;
	private int maxSearchDistance;
	private Node[][] nodes;
	private boolean allowDiagMovement;
	private AStarHeuristic heuristic;

	public AStarPathFinder(TileBasedMap map, int maxSearchDis-
tance, boolean allowDiagMovement) {
		this(map, maxSearchDistance, allowDiagMovement,
new ClosestHeuristic());
	}

	public AStarPathFinder(TileBasedMap map, int maxSearchDis-
tance, boolean allowDiagMovement, AStarHeuristic heuristic) {
		this.heuristic = heuristic;
		this.map = map;
		this.maxSearchDistance = maxSearchDistance;
		this.allowDiagMovement = allowDiagMovement;
```

Esimerkkikoodi 4. AStarPathFinder-luokan alkuosa

Luokka sisältää ArrayList-tyyppisen closed-muutujan sekä SortedList-tyyppisen open-muuttujan. Closed-muuttuja pitää kirjaa siitä, mitkä noodit on jo tutkittu. Open-muuttuja sisältää kaikki tutkimatta olevat noodit. TileBasedMap-tyyppinen map-muuttuja annetaan luokan alustuksessa. Tämä muuttuja sisältää koko tutkittavan kartan.

Kokonaislukumuuttuja maxSearchDistance annetaan niin ikään alustettaessa, ja se sisältää suoritettavan haun maksimietäisyyden. Tällä luvulla estetään tuloksettomien hakujen suoritintehon vientiä. Koko kartta käydään läpi, jos tätä arvoa ei ole tai se on suurempi kuin kartan leveys ja korkeus yhteensä. Tämä tarkoittaa sitä, että tuloksettomat haut vievät valtavasti resursseja, sillä haku päättyy vasta, kun kaikki ruudut on käyty läpi. Jos arvo on liian pieni, voi se taas estää reitin löytymisen, vaikka sellainen on olemassa. Oikean suuruisen arvon löytäminen tarvitsee kartan ja sen käyttötarkoituksen tuntemisen.

Node[][]-tyyppinen nodes-muuttuja alustetaan, kun luokka luodaan. Se sisältää Node-luokan objektin kartan jokaisesta ruudusta. Nodes-muuttujasta voidaan hakea x- ja y-koordinaateilla noodi muodossa nodes[x][y]. Totuusarvomuuttujalla allowDiagMovement sallitaan tai kielletään viistottainen liikkuminen. AstarHeuristic-tyyppinen heurestic-muuttuja annetaan luokan alustuksessa. Tätä muuttujaa käytetään päättämään, mikä ruutu tutkitaan seuraavaksi.

3.1.4 ClosestHeurestic-luokka

Omaan toteutukseen valitsin AstarHeurestic-rajapinnan toteuttamiseen lähin noodi ensimmäiseksi toteuttavan ClosestHeurestic-luokan.

```
public class ClosestHeuristic implements AStarHeuristic {

public float getCost(TileBasedMap map, Mover mover, int x, int y, int
tx, int ty) {
                        float dx = tx - x;
                        float dy = ty - y;

                        float          result          =          (float)
(Math.sqrt((dx*dx)+(dy*dy)));
```

Esimerkkikoodi 5. ClosestHeurestic-luokan lähdekoodi

Luokka toteuttaa AstarHeuristic-rajapinnan vaatiman getCost-metodin palauttamalla kohteen x:n ja y:n sekä aloituspisteen x:n ja y:n etäisyyden hypotenuusan pituuden.

3.1.5 A*-reitinhaun kulku

Reitin haku käynnistetään kutsumalla AstarPathFinder-luokan findPath(Mover mover, int sx, int sy, int tx, int ty)-metodia. Metodi palauttaa löydetyn reitin Path-luokan objektina tai null-arvon jos reittiä ei löydy. Mover sisältää Mover-rajapinnan toteuttavan hahmon, jota liikutetaan. Kokonaislukumuuttujat sx, sy, tx ja ty sisältävät aloituspisteen x- ja y-koordinaatit sekä kohteen x- ja y-koordinaatit vastaavasti.

```
            if (map.blocked(mover, tx, ty)) {
                        return null;
            }
            nodes[sx][sy].cost = 0;
            nodes[sx][sy].depth = 0;
            closed.clear();
            open.clear();
            open.add(nodes[sx][sy]);
```

Esimerkkikoodi 6. AstarPathFinder-luokan reitinhaun lähdekoodin ensimmäinen osa

Aivan aluksi tutkitaan, onko haettu ruutu sallittu vai ei. Haku päättyy palauttamalla arvon null, jos ruutuun liikkuminen ei ole sallittua. Lähtöruudun hinnaksi ja syvyydeksi alustetaan arvo nolla. Seuraavaksi closed- ja open-muuttujien listat tyhjennetään ja open-muuttujan listalle annetaan lähtöruutu avoimeksi hakukohteeksi. Kohteen noodin vanhemman arvoksi annetaan arvo null. Seuraavaksi alustetaan currentDepth-muuttuja arvolla nolla. Muuttuja kertoo haun syvyyden. Tämän jälkeen hakua aletaan toteuttaa while-luupissa.

```
while ((currentDepth < maxSearchDistance) && (open.size() != 0)) {
            Node current = getFirstInOpen();
            if (current == nodes[tx][ty]) {
                        break;
            }
```

Esimerkkikoodi 7. AstarPathFinder-luokan reitinhaun lähdekoodin toinen osa

Luuppia käydään siihen asti, kunnes asetettu maksimihakusyvyys saavutetaan tai vaihtoehtoisesti enää ei ole avoimia hakukohteita. Ensimmäisenä haetaan nykyiseksi noodiksi ensimmäinen noodi open-muuttujan listasta. Haku päättyy, jos nykyinen noodi on kohdenoodi. Tämän jälkeen nykyinen noodi poistetaan tutkittavien noodien listalta eli open-muuttujan listasta. Nykyinen noodi lisätään käsiteltyihin noodien listalle eli closed-muuttujan listalle. Seuraavaksi aletaan käydä läpi nykyisen noodin ympärillä olevia noodeja.

```
        for (int x=-1;x<2;x++) {
                for (int y=-1;y<2;y++) {
                        if ((x == 0) && (y == 0)) {
                                continue;
                        }

                        if (!allowDiagMovement) {
                                if ((x != 0) && (y != 0)) {
                                        continue;
```

Esimerkkikoodi 8. AstarPathFinder-luokan reitinhaun lähdekoodin kolmas osa

For-luupilla tutkitaan kaikki nykyisen noodin ympärillä olevat noodit aloittaen noodista nykyisen noodin x – 1 ja nykyisen noodin y – 1. Luuppia suoritetaan aina nykyisen noodin x + 1 ja nykyisen noodin y + 1.

Nykyinen noodi ohitetaan luupissa ensimmäisessä if-tarkistuksessa. Jälkimmäisessä tarkistuksessa estetään viistoittainen liikkuminen, jos se ei ole sallittua. Tämä tehdään tarkistamalla, ovatko molemmat x- ja y-koordinaatit nollasta poikkeavia. Lopuksi tarkistettavan noodin sijainti tiedot tallennetaan kahteen kokonaislukumuuttujaan. Muuttuja xp sisältää nykyisen x:n ja yp-muuttuja sisältää nykyisen y:n.

```
            if (isValidLocation(mover,sx,sy,xp,yp)) {
                        float  nextStepCost  =  current.cost  +  getMove-
mentCost(mover, current.x, current.y, xp, yp);
                        Node neighbour = nodes[xp][yp];
                        map.pathFinderVisited(xp, yp);

                        if (nextStepCost < neighbour.cost) {
                                    if (inOpenList(neighbour)) {
                                                removeFromOpen(neighbour);
                                    }
                                    if (inClosedList(neighbour)) {
                                                removeFromClosed(neigh-
bour);
                                    }
                        }

                        if     (!inOpenList(neighbour)     &&     !(in-
ClosedList(neighbour))) {
                                    neighbour.cost = nextStepCost;
                                    neighbour.heuristic   =   getHeuristic-
Cost(mover, xp, yp, tx, ty);
                                    currentDepth  =  Math.max(currentDepth,
neighbour.setParent(current));
                                    addToOpen(neighbour);
                        }
            }
```

Esimerkkikoodi 9. AstarPathFinder-luokan reitinhaun lähdekoodin neljäs osa

Seuraavaksi tarkistetaan, onko liikkuminen tarkistettavaan noodin sallittua isValidLocation-metodin avulla. Tämän jälkeen haetaan siirtomaksu tarkistettavaan noodiin nextStepCost-muuttujaan. Siirtosumman arvo on nykyisen noodin siirtosumman ja tarkistettavaan noodiin liikkumisen siirtomaksun summa. Tarkistettavasta noodista haetaan Node-luokan objekti nodes[][]-taulukosta neighbour-muuttujaan ja noodi merkataan kartalla tarkistetuksi. Jos nextStepCost-muuttujaan tallennettu siirtomaksu on pienempi kuin haetulla Node-luokan objektille tallennettu siirtomaksu, olemme löytäneet lyhemmän reitin noodille. Tämän takia poistamme tarkistettavan noodin open- ja closed-muutujien listoista. Tämä mahdollistaa noodin arvioinnin uudelleen.

Jos tarkistettavaa noodia ei löydy open- ja closed-muutujien listoista, noodille on löydetty uusi lyhin reitti. Tarkistettavalle noodille lisätään siirtomaksuksi nextStepCost-muuttujaan tallennettu arvo. Heurestic-muuttujalta haetaan myös tarkistettavuuden kannattavuusarvo, joka tallennetaan neighbour-muutujalle. Nykyinen noodi lisätään tarkistettavan noodin vanhemmaksi noodiksi. Tämän jälkeen haun syvyys päivitetään tarkistettavan noodin syvyydellä. Lopuksi tarkistettava noodi lisätään avoimien noodien eli open-muuttujan listaan. Tämän jälkeen alkaa while-silmukan seuraava iteraatio.

While-silmukan päätteeksi tarkistetaan, onko tavoitetulla noodilla vanhempaa. Jos vanhempaa ei löydy, niin reittiä haun tavoite noodiin ei löytynyt. Vanhemman löytyessä aletaan reittiä alkunoodista tavoitenoodiin rakentaa seuraavalla koodilla:

```
Path path = new Path();
        Node target = nodes[tx][ty];
        while (target != nodes[sx][sy]) {
                path.prependStep(target.x, target.y);
                target = target.parent;
        }
        path.prependStep(sx,sy);
```

Esimerkkikoodi 10. AstarPathFinder-luokan reitinhaun lähdekoodin viides osa

Reitti luodaan noodien vanhempien avulla lopusta alkuun päin. Reitin luonti käynnistetään lisäämällä tavoitenoodi reittiin. Tämän jälkeen reitin alkuun lisätään tavoitenoodin vanhempi. Vanhempien lisäystä jatketaan, kunnes olemme päässeet aloitusnoodiin asti. Tämän jälkeen reitti palautetaan, ja haku päättyy.

3.2 Uusi muistipohjainen reitinhaku

Muistipohjainen reitinhaku perustuu siihen, että jokainen kartan noodi tietää, miten toiseen noodin päästään. Tämä on paljon nopeampi tapa saada reitti haettua kuin edellisen luvun suoritinpohjainen reitinhakuun.

3.2.1 Noodien ja reittilistan toiminta

Noodit ovat kartan yksittäisiä ruutuja. Jokaisella noodi voi olla sallitussa tai estetyssä tilassa. Sallitussa tilassa olevat noodit sallivat noodiin tai sen läpi kulkemisen, kun taas estetyssä tilassa noodiin tai sen läpi ei voi kulkea. Kartan alustusvaiheessa jokaisen noodin vieressä olevat noodit merkataan noodin naapureiksi. Jos naapurinoodi on sallitussa tilassa, niin noodi luo reitin naapurinoodiin omaan reittilistaansa. Reitin hinnaksi tulee naapurinoodin kulkuhinta. Tämän jälkeen noodi ilmoittaa muille noodeille, että se on löytänyt uuden reitin.

Reittien lisäyssäännöt

1. Reittiä ei löydy listasta.
2. Uusi reitti on halvempi kuin olemassa oleva reitti.
3. Noodi ei ole itse reitin kohde.

Noodi kyselee jokaisella pelin luupilla naapurinoodeilta heidän tilaa. Jos se huomaa, että naapurinoodi on lisännyt uuden reitin, se pyytää naapurinoodilta listan juuri lisätyistä reiteistä. Noodi käy jokaisen lisätyn reitin läpi ja etsii, onko sillä jo olemassa oleva reitti kohteeseen. Jos olemassa olevaa reittiä ei ole, niin se lisää reitin reittilistaan ja lisää oman reitin hintaan naapurinoodin siirtomaksun. Jos reitti löytyy jo omasta reittilistasta, niin noodi vertailee reittien hintaa. Jos uusi reitti on edullisempi, niin reitti poistetaan listasta ja uusi reitti lisätään listaan. Kummassakin tapauksessa noodi ilmoittaa naapurinoodeille, että se on joko lisännyt tai lisännyt ja poistanut reittejä.

Reittien poistosäännöt:

1. Reitti löytyy reittilistasta.
2. Reitin seuraava askel kulkee reitin poistaneen noodin lävitse.

Noodin saadessa tiedon naapurinoodin reitin poistosta pyytää se naapurinoodilta listan poistetuista reiteistä. Tämän jälkeen se vertailee listan reittejä oman reittilistan reitteihin. Jos se löytää omasta reittilistasta poistetun reitin ja huomaa, että se kulkee kyseisen

naapurinoodin läpi, se poistaa reitin omasta reittilistasta. Koska tässä tapauksessa naapurinoodi lisäsi uuden reitin samaan kohteeseen omaan listaan, lisää noodi myös reitin takaisin omaan listaansa uudella hinnalla.

Näillä kahdella reittien jakomenetelmällä kartan jokainen noodi saa loppuen lopuksi tiedon jokaisen muun noodin sijainnista. Aluksi noodit lisäävät reitit vain naapureihin. Seuraavalla päivityssyklillä noodit lisäävät reitit naapureiden naapureihin. Tästä eteenpäin noodien reittien määrä tuplaantuu jokaisella syklillä, kunnes loppuen lopuksi noodilla on reitti jokaiseen kartan muuhun noodiin.

3.2.2 Noodien tilan muuttaminen

Jokainen noodi voi olla sallitussa tai estetyssä tilassa. Sallitussa tilassa noodi jakaa ja ylläpitää reittilistaa normaalisti. Estetyssä tilassa noodi ei pidä reittilistaa eikä siten myöskään jaa reittitietoja naapureille. Jos noodin tila muuttuu sallitusta estetyksi, niin se tiputtaa oman reittilistan ja ilmoittaa naapurinoodeille tilan muuttumisisesta.

Kun noodi saa tiedon naapurinoodin tilan muuttumisesta estetyksi, se tarkistaa omat reittinsä. Se poistaa reittilistasta kaikki reitit, joiden seuraava askel on tilansa estetyksi muuttanut noodi. Tämän jälkeen noodi ilmoittaa naapureille reittien poistosta. Naapurinoodit tutkivat tämän jälkeen normaalisti poistosääntöjen mukaan, poistavatko poistetut reitit reittilistasta vai eivät. Jos noodi huomaa, että sillä on hallussaan reitti samaan kohteeseen kuin poistettu reitti, jonka seuraava askel ei ole reitin poistanut noodi, niin se muuttaa tilansa löytötilaan.

Noodin löytötilan aktivointi johtavat tilanteet:

1. Noodi löytää omasta reittilistasta reitin, joka kulkee eri kautta kuin naapurinoodista poistettu reitti.

2. Noodi muuttaa tilansa estetystä sallituksi.

Kun noodi huomaa, että naapurinoodi on löytötilassa, niin se pyytää naapurinoodilta listan kaikista sen reiteistä. Se vertailee kaikkia naapurinoodin reittejä omaansa ja etsii niistä sekä uusia että halvempia reittejä. Näin mahdollistetaan reittien jakautuminen uudelleen kaikille noodeille. Tällä menetelmällä loppuen lopuksi jokaisella noodilla on taas uusi reitti jokaiseen kartan sallitussa-tilassa olevaan noodiin.

Kun noodi muuttaa tilaansa estetystä sallituksi, niin se lisää välittömästi omaan reittilistaansa reitit naapureihin ja muuttaa tilansa löytötilaan. Tämän jälkeen noodi tekee täyden reittien vertailun kaikkien naapureiden kanssa saadakseen näiltä parhaat reitit kaikkiin muihin noodeihin. Noodin naapurit lisäävät tämän jälkeen reitin aktivoituneeseen naapurinoodiin. Lisäksi ne tekevät täyden reittilistojen vertailun aktivoituneen noodin kanssa ja päivittävät omat reittinsä, jotka voivat nyt kulkea aktivoituneen noodin lävitse. Näin lopulta kaikkien noodien reitit ottavat huomioon myös aktivoituneen noodin.

3.2.3 Aluelistan toiminta

Aluelistan tarkoitus on jakaa noodit ryhmiin. Jokaiselle noodille annetaan alustaessa aluekoodi, joka kertoo, mihin alueeseen noodi kuuluu. Noodit vaihtavat reittitietoja vain oman alueensa noodien kanssa. Noodi kuitenkin lisää reitin naapurinoodiin, jos se kuuluu erialueeseen. Tämä mahdollistaa kulun oman alueen ulkopuolelle. Eri alueeseen kuuluva noodi lisätään myös aluelistan naapurinoodiksi. Aluelista luo tämän jälkeen reittilistaansa reitin naapurinoodin alueeseen.

Kun aluelista on lisännyt tiedon erialueeseen kuuluvasta naapurinoodista, se alkaa vaihtaa tietoa alueiden välisistä reiteistä naapurinoodin aluelistan kanssa. Toimintaperiaate on täysin sama kuin noodien reittilistassa, mutta noodien välisten kulkutietojen sijaan reitit sisältävät tietoa alueiden välisistä reiteistä.

Kun aluelista saa tiedon, että naapurialueen listaan on lisätty uusi reitti, pyytää se aluelistalta tiedon lisätystä reitistä. Jos sillä ei ole omassa listassa lisättyä reittiä, se lisää saman reitin listaansa, jos jokin reittilistasta tutuista säännöistä on totta.

Reittien lisäyssäännöt aluelistalle:

1. Reittiä ei löydy listasta.
2. Uusi reitti on halvempi kuin olemassa oleva reitti.
3. Alue ei ole itse reitin kohde.

Kun alue on lisännyt uuden reitin listaansa, se välittää tiedon lisäyksestä naapurialueille. Tämän jälkeen naapurialueet vertailevat lisättyä reittiä omiin reitteihin, ja näin tiedot alueiden välisistä reiteistä välittyvät eteenpäin. Loppuen lopuksi jokaisella alueella on reittitiedot jokaiseen muuhun alueeseen kulkemista varten.

On huomattava, että toisin kuin noodien reittilistassa, alue ei voi koskaan poistua kokonaan käytöstä. Koska alue on tarkoitettu kattamaan satoja eri noodeja, en lisännyt protokollaan mahdollisuutta kytkeä koko aluetta pois päältä. Aluelistalla ei täten ole mahdollisuutta poistaa reittejä omasta reittilistastaan muissa tapauksissa kuin jos se löytää uuden halvemman reitin. Tämä tarkoittaa sitä, että kun aluelistalla on tiedossa reitti jokaiseen muuhun alueeseen kulkemista varten, reittilista on lopullinen.

3.2.4 Reitinhakualgoritmi

Reitti haetaan kahden noodin välillä. Aluksi algoritmi tarkistaa, ovatko kummatkin noodit sallitussa tilassa. Jos ne ovat, haku voi jatkua. Muussa tapauksessa haku päättyy, koska reitti ei ole mahdollinen. Aluksi reitinhakualgoritmi tarkistaa, kuuluvatko kummatkin noodit samaan alueeseen. Jos ne kuuluvat samaan alueeseen, suoritetaan reitin haku noodien välillä suoraan reittilistan avulla. Muussa tapauksessa jatketaan reitinhakuun aluelistalla.

Kun etsitään reittiä alueiden välillä, aloitusnoodin aluelistalta etsitään reitti kohdealueeseen. Reitistä kerätään tieto seuraavasta alueesta. Tämän jälkeen noodin reittilistalta haetaan halvin reitti seuraavan alueen noodiin. Reitistä otetaan muistiin tieto alueen kohdenoodista ja tämän jälkeen reitin haku aloitusnoodin ja alueen kohdenoodin kanssa voi alkaa.

Reitti muodostetaan lisäämällä aluksi aloitusnoodin koordinaatit muodostettavaan Path-luokan reittiin. Tämän jälkeen noodin reittilistalta haetaan tieto seuraavasta askeleesta kohti kohdenoodia. Reitistä saadaan seuraavan askeleen noodin tunnusluku, jolla haetaan kartalta Node-luokan olio. Noodin koordinaateilla lisätään jälleen uusi askel Path-luokan reittiin. Reitin haku etenee tällä tavoin noodi kerrallaan, kunnes tavoitenoodi on lisätty Path-luokan reittiin askeleeksi.

Kun alueen kohdenoodi on saavutettu, tarkistetaan, onko uusi alue haunkohdenoodin alue. Jos ei ole etsitään, uuden alueen kohdenoodi aluelistalta, jonka jälkeen edetään jälleen reitinhakuun reittilistan avulla. Kun kohdealue on saavutettu, haetaan reittilistalta

suoraan kohdenoodi, jonka jälkeen reittilistasta haetaan reitti kohdenoodiin. Reitti on valmis, kun kohdenoodi on lisätty askeleeksi Path-luokan reittiin. Tämän jälkeen valmis Path-luokan reitti palautetaan haun suorittajalle. Jos reittiä kohdenoodiin ei löydetä, esimerkiksi tapauksissa, joissa se on estettyjen noodien ympäröimänä, haku palauttaa arvon null.

3.2.5 Node-luokka

Käytän tässä opinnäytetyössä kartan ruuduista nimitystä noodi. Node-luokka sisältää kaikki perustiedot noodia varten. Kummatkin tässä työssä käytetyt reitinhakumenetelmät käyttävät samaa luokkaa, mutta suurin osa luokan käyttämistä muuttujista ja metodeista on tarkoitettu vain muistipohjaista reitinhakua varten.

```
private int id;
private int x, y;
private int cost;
private boolean isBlocked;
private int areaCode;
private RouteList routeList;
```

Esimerkkikoodi 11. Node-luokan muuttujien lähdekoodi

Käydään aluksi läpi muuttujat, joita molemmat reitinhakumenetelmät käyttävät. Kokonaislukumuuttuja id on noodin ainutlaatuinen tunnusluku. Tunnuslukua käytetään reittitiedoissa koordinaattien sijaan muistitilan säästämiseksi. Kokonaislukumuuttujat x ja y sisältävät tiedon noodin sijainnista eli x- ja y-koordinaateista. Muuttuja-cost sisältää tiedon siirtohinnasta reitille. Hinnalla voidaan vaikuttaa reittien kulkuun. Korkeampi hinta vähentää reittien kulkua noodin läpi, kun taas pienempi hinta suosii noodin läpikulkua. Omissa testeissäni kuitenkin kaikilla noodeilla on hintana arvo yksi. Totuusarvomuuttuja isBlocked sisältää tiedon siitä, onko noodi estetty vai sallittu. Noodille kulku on estetty arvolla tosi, kun taas arvolla epätosi kulku on sallittu. Luokan muut muuttujat on tarkoitettu ainoastaan muistipohjaisen reitinhakualgoritmin käytettäviksi.

Kokonaislukumuuttuja areaCode sisältää kokonaislukuarvona aluenumeron. Noodit on jaettu eri ryhmiin aluenumeroilla. Noodit tietävät tarkat reittitiedot vain oman alueensa noodeille. Alueet mahdollistavat reittilistan muistitilankäytön vähentämisen. Noodin reittilista löytyy RouteList-luokasta, ja eri alueiden tiedot löytyvät AreaList-luokasta. Jokaisella alueella on valittuna yksi noodi, joka ylläpitää aluelistaa. Tällä noodilla totuusarvomuuttuja isAreaMaster sisältää arvon tosi. Kaikilla muilla alueen noodeilla tämä muuttuja sisältää arvon epätosi.

Totuusmuuttuja-arvo statusChanged sisältää arvon tosi, kun noodin tila on muuttunut estetystä tilasta sallituksi tai toisinpäin. Totuusmuuttuja-arvo changedToBlocked sisältää arvon tosi vain, kun noodin tila on muuttunut sallitusta tilasta estetyksi. Kumpaakin arvoa käytetään reittilistan hallinnassa.

```
public Node(int id, int x, int y, int cost, AreaList areaList,
                      boolean isBlocked, boolean isAreaMaster) {
           this.id = id;
           this.x = x;
           this.y = y;
           this.areaCode = areaList.getAreaCode();
           this.cost = cost;
           this.routeList = new RouteList(this);
           this.areaList = areaList;
           this.isBlocked = isBlocked;
           this.isAreaMaster = isAreaMaster;
           this.statusChanged = false;
           this.changedToBlocked = false;
```

Esimerkkikoodi 12. Node-luokan konstruktorin lähdekoodi

Konstruktori vaatii tiedot id, x, y, cost, areaList, isBlocked ja isAreaMaster muuttujille. Luokan muuttujista vain routeList (reittilista) generoidaan automaattisesti konstruktorissa. Tämän lisäksi totuusarvomuuttujille statusChanged ja changedToBlocked asetetaan automaattisesti arvo epätosi.

```
		public int getId() {
			return id;
		}

		public int getX() {
			return x;
		}

		public int getY() {
			return y;
		}

		public int getAreaCode() {
			return areaCode;
		}

		public int getCost() {
			return cost;
		}

		public RouteList getRouteList() {
			return routeList;
		}

		public AreaList getAreaList() {
			return areaList;
		}
```

Esimerkkikoodi 13. Node-luokan metodien lähdekoodin ensimmäinen osa

Luokka sisältää muuttujan haku metodit muuttujille id, x, y, areaCode, cost, routeList, areaList, isBlocked, statusChanged ja changedToBlocked.

```
public void addNeighbour(Node node) {
            routeList.addNeighbour(node);
            if(areaCode != node.getAreaCode()) {
                        areaList.addNeighbour(node);
            `
```

Esimerkkikoodi 14. Node-luokan addNeighbour-metodin lähdekoodi

Noodille lisätään naapurinoodi metodilla addNeighbour(Node node). Naapurinoodin kanssa noodi voi jakaa tietoa tuntemistaan reiteistä. Metodissa kutsutaan reittilistan samannimistä metodia, jolla luodaan reitti naapurinoodiin. Reitin muodostusta käsitellään tarkemmin seuraavassa luvussa. Jos naapurinoodilla on eri aluekoodi kuin itsellä, niin noodi lisätään myös naapuriksi aluelistaan.

```
public void changeBlockedStatus() {
            this.isBlocked = !isBlocked;
            this.statusChanged = true;
```

Esimerkkikoodi 15. Node-luokan changeBlockedStatus-metodin lähdekoodi

Noodin tilaa voidaan muuttaa changeBlockedStatus-metodilla. Metodilla voidaan vaihtaa noodi sallitun ja estetyn tilan välillä. Estetyssä tilassa noodille kulku on kielletty, kun taas sallitussa tilassa kulku sallitaan.

```
public void update() {
        if(changedToBlocked) {
                changedToBlocked = false;
        }
        if(!isBlocked) {
                routeList.update();
                if(statusChanged) {
                        statusChanged = false;
                }
        } else if(statusChanged) {
                routeList.update();
                statusChanged = false;
```

Esimerkkikoodi 16. Node-luokan update-metodin lähdekoodi

Luokan update-metodilla päivitetään noodin ja reittien tilaa. Update-metodia kutsutaan kartan jokaisella päivityssyklillä. Totuusmuuttuja-arvo changedToBlocked muutetaan aluksi epätosi-tilaan, jos se on tosi-tilassa. Jos noodi ei ole estetyssä tilassa, niin reittilista päivitetään ja totuusmuuttuja-arvo statusChanged saa arvon epätosi, jos se on tilassa tosi. Jos noodi on sallitussa tilassa ja noodin tila on muuttunut, niin reittilista päivitetään. Tila on muuttunut vain, kun noodi on juuri muuttunut estettyyn tilaan. Tämä tieto välitetään reittilistalle kutsumalla tämän update-metodia. Reittilista tietää tämän jälkeen, että reittitiedot on tyhjennettävä. Samassa if-haarassa statusChanged-totuusmuuttujaarvo muutetaan tilaan epätosi ja hangedToBlocked-totuusmuuttuja-arvo muutetaan arvoon tosi. Tämä totuusmuuttuja-arvo välittää naapurinoodeille tiedon siitä, että noodi on muuttanut tilansa estetyksi. Lopuksi vielä aluetta hallitseva noodi päivittää aluelistan kutsumalla sen update-metodia.

3.2.6 Route-luokka

Reitin kaikki tiedot säilytetään route-luokassa. Route-luokan on tarkoitus olla mahdollisimman pieni muistitilan säästämiseksi. Tästä syystä route-luokka sisältää ainoastaan kolme kokonaislukumuuttujaa. Muuttujat eivät ole tarkoitettu ainoastaan noodien tietoja varten, koska reittejä käytetään myös alueiden välisten kulkutietojen välityksessä. Noodien reittilistassa muuttujat sisältävät tietoa noodeista, kun taas aluelistassa muuttujat sisältävät tietoa aluekoodeista.

```
public class Route {
		private int targetId;
		private int nextStepId;
		private int cost;

		public Route(int targetId, int nextStepId, int cost) {
			this.targetId = targetId;
			this.nextStepId = nextStepId;
			this.cost = cost;
		}

		public int getTargetId() {
			return targetId;
		}
```

Esimerkkikoodi 17. Route-luokan lähdekoodi

Muuttuja targetId sisältää arvona kohdenoodin tai alueen tunnuksen. Kohdenoodi tai alue on reitin lopullinen kohde. Muuttuja nextStepId sisältää tiedon, mihin noodiin tai alueeseen on seuraavaksi liikuttava, jotta päästään kohti kohdetta. Muuttuja cost sisältää tiedon koko matkan hinnasta. Arvoa käytetään eri reittien paremmuuden vertailuun. Kun vertaillaan kahta reittiä, niin pienemmän hinnan reitti valitaan parempana reittinä.

Luokka vaatii konstruktorissa jokaisen kolmen muuttujan arvot. Kaikille arvoille on myös oma tiedonhakumetodi.

3.2.7 AbstractRouteList-luokka

AbstractRouteList-luokka on abstrakti luokka, jolla alustetaan pohja reitti- ja aluelistalle. Luokka sisältää kaikki yhteiset metodit ja muuttujat kummallekin listalle.

```
protected List<Route> routes, removedRoutes, addedRoutes;
protected List<Route> tempRemovedRoutes, tempAddedRoutes;
protected List<Node> neighbours, activatedNeighbours;
```

Esimerkkikoodi 18. AbstractRouteList-luokan muuttujien lähdekoodi

Luokalla on viisi erilaista listaa, jotka sisältävät Route-luokan olioita, ja kolme erilaista totuusmuuttuja-arvoa. Lisäksi luokalla on kaksi erilaista Node-luokan olioista koostuvaa listaa. Näistä naapurilistoista ensimmäinen neighbours-niminen lista sisältää kaikki naapurit. Tämä lista sisältää aina kaikki naapurit riippumatta siitä, ovatko ne aktiivisia vai eivät. Toinen, activatedNeighbours-niminen, naapurilista sisältää tilapäisesti kaikki naapurit, joille tulee lisätä reitti. Reittien lisäys käsitellään reitti- ja aluelistan osalta seuraavissa luvuissa.

Abstraktin luokan tärkein toiminnallisuus on lisätä reittejä reitti tauluun. Aktiiviset reitit pidetään routes-nimisessä listassa. Tämän listan tietoihin päästään käsiksi kolmella eri metodilla.

```
public List<Route> getRoutes() {
			return routes;
}

public boolean containsRoute(int targetId) {
		for(Route route : routes) {
			if(route.getTargetId() == targetId) {
					return true;
			}
		}
		return false;
}

public Route getRouteTo(int targetId) {
		for(Route route : routes) {
			if(route.getTargetId() == targetId) {
				return route;
			}
		}
		return null;
}
```

Esimerkkikoodi 19. AbstractRouteList-luokan lähdekoodin ensimmäinen osa

GetRoutes-metodilla saadaan haettu koko reittilista. Tätä käytetään silloin, kun naapurinoodi haluaa tiedon kaikista reiteistä. ContainsRoute-metodi tarvitsee parametrinä kohteen tunnuksen ja palauttaa tosi- tai epätosi-totuusmuuttuja-arvon riippuen siitä, löytyykö reittilistasta reittiä kohteeseen vai ei. GetRoute-metodi tarvitsee niin ikään parametrinä kohteen tunnuksen ja palauttaa reitin Route-luokan oliona, jos reittilista sisältää reitin kohteeseen. Muussa tapauksessa palautetaan arvo null.

Juuri lisätyt reitit säilytetään yhden päivityssyklin ajan addedRoutes-nimisessä listassa, josta naapurinoodit voivat helposti nähdä mitä reittejä kyseiselle noodille on juuri lisätty. Vastaavasti poistettuja reittejä säilytetään removedRoutes-nimisessä listassa. Myös poistettuja reittejä säilytetään yhden päivityssyklin ajan listassa, että naapurinoodit voivat helposti nähdä, mitä reittejä on poistettu.

Reittien lisäystä varten on kaksi päämetodia.

```
protected void addRouteTo(int targetId, int nextHopId, int cost) {
		tempAddedRoutes.add(new Route(targetId, nextHopId, cost));
}

protected void removeRouteTo(int targetId) {
		Route removedRoute = getRouteTo(targetId);
		if(removedRoute != null) {
			tempRemovedRoutes.add(removedRoute);
		}
}
```

Esimerkkikoodi 20. AbstractRouteList-luokan lähdekoodin toinen osa

Metodi addRouteTo luo tilapäiseen tempAddedRoutes-nimiseen listaan uuden lisättävän reitin. Metodi tarvitsee parametreinä tiedon reitin kohteesta, seuraavasta askeleesta ja hinnasta. Vastaavasti removeRouteTo-metodi etsii aluksi getRouteTo-metodin avulla reitin kohteeseen ja lisää reitin tilapäiseen tempRemovedRoutes-listaan. Reittien lisääminen ja poistaminen reittilistaan kuitenkin vaatii reittien lisäys- ja poisto-metodien kutsua.

```
protected void removeRemovedRoutes() {
		for(Route route : tempRemovedRoutes) {
			routes.remove(route);
			removedRoutes.add(route);
			routesUpdated = true;
		}
		tempRemovedRoutes.clear();
}

protected void addAddedRoutes() {
		for(Route route : tempAddedRoutes) {
			routes.add(route);
			addedRoutes.add(route);
			routesUpdated = true;
		}
		tempAddedRoutes.clear();
}
```

Esimerkkikoodi 21. AbstractRouteList-luokan lähdekoodin kolmas osa

RemoveRemovedRoutes-metodissa käydään läpi kaikki tempRemovedRoutes-muuttujan listasta löytyvät poistoa odottavat reitit. Reitit poistetaan routes-muuttujan listasta ja lisätään removedRoutes-muuttujan listaan. Tämän jälkeen totuusmuuttuja-arvo routesUpdated merkataan todeksi, jotta muut noodit voivat saada tiedon reittilistan muutoksesta. Lopuksi tilapäinen tempRemovedRoutes-muuttujan lista tyhjennetään. AddAddedRoutes-metodi tekee vastaavan toimenpiteen, mutta poistamisen sijaan se lisää reitit sekä routes-muuttujan aktiivisiin reitteihin että addedRoutes-muuttujan listaan.

Lisäksi luokka sisältää kaksi apumetodia reitti- ja aluelistalle. Näistä ensimmäisellä foundCheaperRoute-metodilla voidaan tarkistaa, onko uusi reitti edullisempi kuin olemassa oleva reitti.

```
protected boolean foundCheaperRoute(Route route, int addedCost) {
		Route currentRoute = getRouteTo(route.getTargetId());
		int cost = addedCost + route.getCost();
		if(currentRoute.getCost() > cost) {
			return true;
		} else {
			return false;
		}
}
```

Esimerkkikoodi 22. AbstractRouteList-luokan foundCheaperRoute-metodin lähdekoodi

Metodi hakee getRouteTo-metodilla olemassa olevan reitin ja vertaa reitin hintaa parametrinä saadun reitin hintaan. Parametrissä on myös lisähinta, joka on uuden reitin hintaan lisättävä hinta, kun reittiä jatketaan metodin omaan noodiin. Metodi palauttaa totuusmuuttuja-arvon tosi tai epätosi riippuen siitä, onko uusi reitti halvempi. Jos reitit ovat saman hintaiset, palauttaa metodi arvon epätosi.

```
public void reactivateNeighbours() {
		for(Node node : neighbours) {
			if(!node.isBlocked()) {
				activatedNeighbours.add(node);
			}
		}
}
```

Esimerkkikoodi 23. AbstractRouteList-luokan reactivateNeighbours-metodin lähdekoodi

Toinen reactivateNeighbours-niminen apumetodi lisää kaikki sallitussa tilassa olevat naapurinoodit neighours-muuttujan listasta aktivoitaviin activatedNeighbours-muuttujan listaan. Tätä metodia hyödynnetään, kun halutaan uudelleen aktivoida reitit estetyssä tilassa olleeseen noodiin.

3.2.8 RouteList-luokka

RouteList-luokka jatkaa AbstractRouteList-luokkaa. Sen ainoa oma muuttuja on Node-luokan owner-muuttuja. RouteList-luokkaan lisätään naapurinoodit addNeighbour-metodilla. Tämä lisää naapurit vain listaan eikä vielä luo reittejä. Reitit naapureihin luodaan addRoutesToActivatedNeighbours-metodilla, joka luo reitit kaikille niille naapureille, jotka eivät olleet estetyssä tilassa. Metodi myös aktivoi lähetystilan, joka välittää viestin naapureille, että uudet reitit on löydetty.

Reittilista lisää kaikki naapurinoodin lisäämät uudet reitit omaan listaansa findAddedRoutes-metodilla. Vastaavasti poistettujen reittien poisto taas toimii removeRemovedRoutes-metodilla. Kummatkin toimenpiteet tehdään luvussa 2.2.1 mainittujen ehtojen mukaisesti. Kummatkin metodit suoritetaan updateRoutesFromNode-metodilla.

Lähetystilaa varten on lisäksi koko reittilistan tutkimista varten oma findNewRoute-metodi. Metodi käy koko naapurinoodin reittilistan läpi ja katsoo, toteuttaako ne luvussa 2.2.1 mainitut lisäämisehdot. Lähetystilan poistoa varten on myös oma updateCurrentRoutes-metodi. Tämä metodi tutkii omia reittejä ja etsii kaikki reitit, jotka kulkevat naapurinoodin lävitse. Näille reiteille tehdään hinta tarkastus, ja jos naapurilla ei enää ole reittiä, niin reitti poistetaan. Kummatkin metodit suoritetaan performDiscoveryWithNode-metodilla. Tämä metodi lisää myös reitin naapuriin, jos sellaista ei ole entuudestaan. Näin voi käydä tilanteessa, jossa noodi vaihtaa tilaansa estetystä sallituksi.

Reittilistalla on vielä lisäksi kaksi apumetodia. Näistä ensimmäisellä clearRoutesTo-metodilla poistetaan kaikki reitit, joiden seuraava askel olisi parametrinä saatu naapurinoodi. Metodia käytetään tilanteissa, jossa naapurinoodi on vaihtanut tilansa estetyksi. Toinen findClosestNode-metodi palauttaa parametrinä saadun aluekoodin avulla alueeseen kuuluvan halvimman reitin omaavan noodin. Tämä metodi on erityisesti aluelistaa varten kehitetty.

Luokan update-metodissa suoritetaan kaikki perustoiminnot. Aivan aluksi mahdollinen lähetystila nollataan. Tämän jälkeen lisätään kaikki aktivoidut naapurit ja suoritetaan näiden reittilistojen täysi tarkastus performDiscoveryWithNode-metodilla. Lisäksi mahdollinen reittien päivityksestä kertova routesUpdate-totuusmuuttujaarvo muutetaan epätodeksi ja lisättyjen sekä poistettujen reittienlistat tyhjennetään. Tämän jälkeen tarkistetaan, onko omistajanoodi muuttanut tilaansa. Jos se on nyt estetyssä tilassa, niin kaikki reitit tyhjennetään. Muutoin naapurinoodi aktivoidaan reactivateNeighbours-metodilla.

Lopuksi suoritetaan kaikkien naapureiden kanssa reittien päivitys. Lähetystilassa olevien noodien kanssa tehdään täysi tarkastus ja muiden kanssa normaalit tarkastukset.

3.2.9 AreaList-luokka

AreaList-luokalla eli aluelistalla on vain yksi oma muuttuja, kokonaislukumuuttuja areaCode eli aluekoodi. Tämä arvo annetaan alustus vaiheessa ja sisältää alueen tunnuksen. Koska aluelistat eivät voi muuttua estettyyn tilaan, AreaList-luokalla on vain kaksi metodia reittien lisäykseen ja poistoon. Lisäys tehdään findAddedRoutes-metodilla, joka toimii luvussa 2.2.3 kuvatulla tavalla. Poistoja joudutaan tekemään, jos naapurialue löytää edullisemman reitin kohteeseen. Näissä tapauksissa tehdään oman alueen reittien tutkinta updateCurrentRoutes-metodilla. Kummatkin metodit suoritetaan updateRoutesFromAreaList-metodilla.

Naapurialueiden lisäys tehdään reittilistan tavoin addNeighbours-metodilla. Reittien lisäys naapureille tehdään samoin addRoutesToActivatedNeighbours-metodilla. Metodilla getNodesOfArea voidaan hakea kaikki parametrinä annettuun alueeseen kuuluvat noodit. Lisäksi naapurialueiden olemassaolo voidaan tarkistaa hasNeighbours-metodilla.

Luokalla on myös update-metodi. Metodi toimii reittilistan tavoin perustoimintojen suorittamiseen. Aluksi neutralisoidaan mahdollinen lähetystila. Tämän jälkeen lisätään aktivoidut naapurit. Reittilistan tavoin mahdollinen reittien päivityksestä kertova routesUpdate-totuusmuuttujaarvo muutetaan epätodeksi ja lisättyjen sekä poistettujen reittienlistat tyhjennetään. Lopuksi suoritetaan kaikille naapurialueille reittilistojen vertailu updateRoutesFromAreaList-metodin avulla.

3.3 Testaus

Molempien protokollien testaus toteutetaan käyttäen Junit-testausta. Junit on Javan oma testauskirjasto, jolla Java luokkia voidaan testata. Tarkoitus on analysoida kummankin eri reitinhakualgoritmin paremmuus. Kriteereinä paremmuudelle ovat haun nopeus ja tarkkuus.

3.3.1 Testikartat

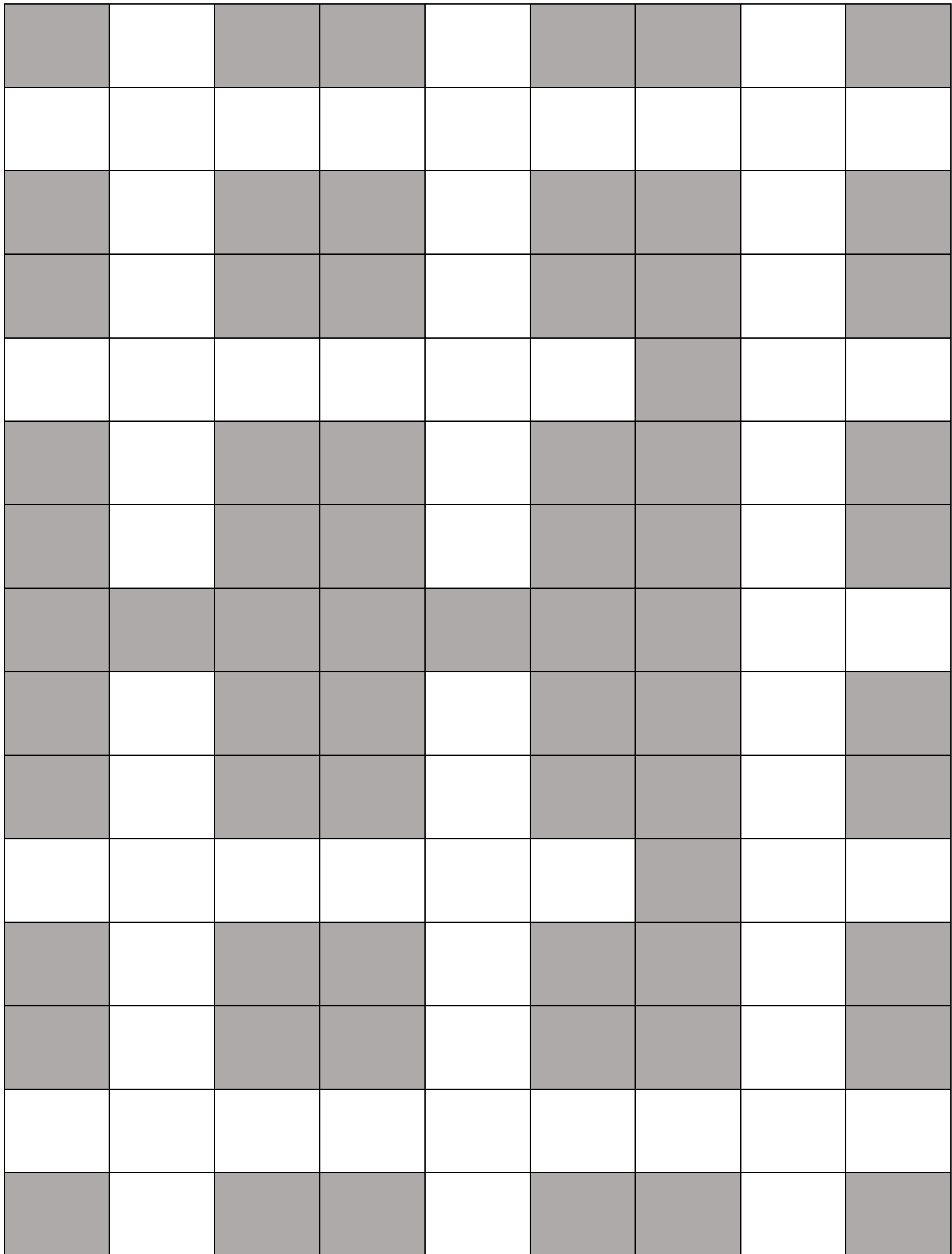
Testausta varten halusin luoda kolme erilaista karttaa. Ensimmäisessä kartassa kartan jokainen noodi on sallitussa tilassa. Tämä mahdollistaa liikkumisen kartan mistä tahansa noodista toiseen noodiin. Kartassa tullaan testaamaan liikkumista vasemmasta ylänurkasta oikeaan alanurkkaan. Kartasta käytetään kolmea eri kokoluokkaa. Pienin kartta on 20 noodia leveä ja 20 noodia korkea. Keskikokoinen kartta on 100 noodia leveä ja 100 noodia korkea. Iso kartta on 200 noodia leveä ja 200 noodia korkea.

Toinen testikartta sisältää ristimuodostelman sallituista ja estetyistä alueista. Tämä muodostelma vähentää sallittujen noodien määrää.

Kuva 3. Toisen testikartan ristimuodostelman kuviointi

Kartan harmaat noodit ovat estettyjä ja valkoiset sallittuja. Tämä vähentää muistipohjaisen reitinhakualgoritmin reittilistojen reittien määrää. A*-reitinhakualgoritmilla on tässä muodostelmassa vähemmän tutkittavaa, joten myös sen pitäisi olla tässä muodostelmassa nopeampi reitin etsinnässä. Kartassa tullaan ensimmäisen kartan tavoin testaamaan liikkumista vasemmasta ylänurkasta oikeaan alanurkkaan. Kartasta käytetään kolmea eri kokoluokkaa. Pienin kartta on 20 noodia leveä ja 20 noodia korkea. Keskikokoinen kartta on 100 noodia leveä ja 100 noodia korkea. Iso kartta on 200 noodia leveä ja 200 noodia korkea.

Kolmas kartta käyttää suljettua ristimuodostelmaa. Se on suunniteltu olemaan mahdollisimman huono A*-reitinhakualgoritmille. Siinä kartta on jaettu kahtia siten, että kartan keskiosa on rivin aivan loppua lukuun ottamatta estetty. Tämän lisäksi kartan kunkin rivin neljänneksi viimeinen ruutu on kielletty ensimmäistä ja viimeistä kolmea riviä lukuun ottamatta. Kartassa tullaan testaamaan vasemmasta ylänurkasta liikkumista vasempaan alanurkkaan, joka pakottaa reitinhakualgoritmin löytämään reitin kartan ympäri. Kartasta käytetään vain yhtä kokoluokkaa, joka on 20 noodia leveä ja 20 noodia korkea.



Kuva 4. Kolmas testikartta

3.3.2 Testaamisen toteutus

Testaaminen toteutetaan JUnit-testeillä. Testeissä testataan kummankin hakualgoritmin käyttämä aika reitin etsimiseen eri kartoissa.

Hakunopeus tullaan mittaamaan ottamalla järjestelmän tarkka aika ennen hakua ja välittömästi haun jälkeen. Hakutarkkuuden tulee olla kummallakin menetelmällä täydellinen, jotta voimme todeta reitinhakualgoritmin toimivaksi. Tarkkuus mitataan ennakkoon lasketulla lyhimmän reitin pituudella, jota verrataan reitin todelliseen pituuteen JUnitin assertEquals-metodilla

```
private static int REPEAT_COUNT = 100;
@Test
public void testMemoryFindPathSmall() {
            Map testMap = new Map(20, 20, 10);
            testMap.initMultiAreaNodes();
            testMap.findNeighbours();
            for(int i = 0; i<40; i++) {
                        testMap.update();
            }
            MemoryBasedPathFinder memoryPathFinder = new MemoryBased-
            PathFinder(testMap);
            Path path = null;
            long startTime = System.currentTimeMillis();
            for(int i = 0; i < REPEAT_COUNT; i++) {
                        path = memoryPathFinder.findPath(1, 1, 19, 19);
```

Esimerkkikoodi 24. Esimerkki JUnit-testin lähdekoodista

Jokainen testi suoritetaan siten, että testataan haun nopeus eri määrillä. Aluksi testataan yksittäisen haun nopeus. Tämän jälkeen testataan kymmeneen, sataan ja tuhanteen peräkkäiseen hakuun käytetty aika.

4 Testaustulokset

4.1 Yhden reitinhaku

Yhden reitinhaun testaus suoritettiin JUnit-testeillä siten, että yhdeltä kartalta haetaan yksi reitti yhdelle hahmolle. Tulokset antavat viittaa siitä, kuinka hyvin haku toimii pelaajan hahmolle. Tuloksissa halutaan, että käytetty aika on mahdollisimman pieni.

Kartan tyyppi	Kartan koko	Algoritmi	Käytetty aika (ms)
Avoin	20x20	Muistipohjainen	0
Avoin	20x20	A*	0
Avoin	100x100	Muistipohjainen	3
Avoin	100x100	A*	64
Avoin	200x200	Muistipohjainen	14
Avoin	200x200	A*	657
Ristimuodostelma	20x20	Muistipohjainen	0
Ristimuodostelma	20x20	A*	0
Ristimuodostelma	100x100	Muistipohjainen	8
Ristimuodostelma	100x100	A*	10
Ristimuodostelma	200x200	Muistipohjainen	12
Ristimuodostelma	200x200	A*	377
Suljettu risti	20x20	Muistipohjainen	0
Suljettu risti	20x20	A*	0

Taulukko 1. Yhden reitinhaun JUnit-testien tulokset

4.2 Kymmenen reitinhaku

Kymmenen reitinhaun testaus suoritettiin JUnit-testeillä siten, että yhdeltä kartalta haetaan kymmenen reittiä yhdelle hahmolle. Tulokset antavat viittaa siitä, kuinka hyvin haku toimii, kun tekoälyn ohjaamia hahmoja on pienen ryhmän verran. Tuloksissa halutaan, että käytetty aika on mahdollisimman pieni.

Kartan tyyppi	Kartan koko	Algoritmi	Käytetty aika (ms)
Avoin	20x20	Muistipohjainen	0
Avoin	20x20	A*	1
Avoin	100x100	Muistipohjainen	18
Avoin	100x100	A*	379
Avoin	200x200	Muistipohjainen	135
Avoin	200x200	A*	5896
Ristimuodostelma	20x20	Muistipohjainen	0
Ristimuodostelma	20x20	A*	0
Ristimuodostelma	100x100	Muistipohjainen	13
Ristimuodostelma	100x100	A*	99
Ristimuodostelma	200x200	Muistipohjainen	142
Ristimuodostelma	200x200	A*	1506
Suljettu risti	20x20	Muistipohjainen	0
Suljettu risti	20x20	A*	1

Taulukko 2. Kymmenen reitinhaun JUnit-testien tulokset

4.3 Sadan reitinhaku

Sadan reitinhaun testaus suoritettiin JUnit-testeillä siten, että yhdeltä kartalta haetaan sata reittiä yhdelle hahmolle. Tulokset antavat viittaa siitä, kuinka hyvin haku toimii suurella kuormalla. Sataan samanaikaiseen hakuun päästäkseen käytön pitää olla hyvin raskasta, joten tulokset ovat hyödyllisiä, kun halutaan tehdä raskasta simulointia. Tuloksissa halutaan, että käytetty aika on mahdollisimman pieni.

Kartan tyyppi	Kartan koko	Algoritmi	Käytetty aika (ms)
Avoin	20x20	Muistipohjainen	1
Avoin	20x20	A*	18
Avoin	100x100	Muistipohjainen	153
Avoin	100x100	A*	3860
Avoin	200x200	Muistipohjainen	1244
Avoin	200x200	A*	57532
Ristimuodostelma	20x20	Muistipohjainen	1
Ristimuodostelma	20x20	A*	7
Ristimuodostelma	100x100	Muistipohjainen	108
Ristimuodostelma	100x100	A*	974
Ristimuodostelma	200x200	Muistipohjainen	1247
Ristimuodostelma	200x200	A*	13807
Suljettu risti	20x20	Muistipohjainen	1
Suljettu risti	20x20	A*	4

Taulukko 3. Sadan reitinhaun JUnit-testien tulokset

4.4 Tuhannen peräkkäisen reitinhaku

Tuhannen reitinhaun testaus suoritettiin JUnit-testeillä siten, että yhdeltä kartalta haetaan tuhat reittiä yhdelle hahmolle. Testit kehitettiin äärimmäistä rasitusta varten. Näiden testien oli tarkoitus ylikuormittaa hakualgoritmit ja siten nähdä suuret erot eri algoritmien välillä. Käytännössä tuhannen samanaikaisen reitinhaku tarkoittaisi lähes satojen tuhansien hahmojen simulointia. Tuloksissa halutaan, että käytetty aika on mahdollisimman pieni.

Kartan tyyppi	Kartan koko	Algoritmi	Käytetty aika (ms)
Avoin	20x20	Muistipohjainen	12
Avoin	20x20	A*	180
Avoin	100x100	Muistipohjainen	1338
Avoin	100x100	A*	37992
Avoin	200x200	Muistipohjainen	12520
Avoin	200x200	A*	576346
Ristimuodostelma	20x20	Muistipohjainen	10
Ristimuodostelma	20x20	A*	60
Ristimuodostelma	100x100	Muistipohjainen	1288
Ristimuodostelma	100x100	A*	9766
Ristimuodostelma	200x200	Muistipohjainen	12741
Ristimuodostelma	200x200	A*	136805
Suljettu risti	20x20	Muistipohjainen	11
Suljettu risti	20x20	A*	65

Taulukko 4. Tuhannen reitinhaun JUnit-testien tulokset

4.5 Testitulosten analysointi

Testituloksista näemme, että muistipohjainen reitinhakualgoritmi on huomattavasti A*-reitinhakualgoritmia nopeampi. Koska pelin tulisi päivittää ruutua vähintään 60 kertaa sekunnissa, saamme toimivan reitinhaun maksimiajaksi 1s/60 eli 16,7 millisekuntia. Pienissä 20x20-kartoissa haut olivat kuitenkin nopeita kummallakin algoritmilla. Tuhannen haun testissä A*-reitinhakualgoritmi oli hitaimmillaan avoimessa kartassa, mutta hakuaika oli silti vain 180 millisekuntia. Tämä on kuitenkin yli haun maksimiajan ja siten käyttökelvoton. Muistipohjaisella haulla vastaavaan hakuun meni ainoastaan 12 millisekuntia, joka on vain 6,7 prosenttia A*-reitinhakualgoritmin käyttämästä ajasta. Tämä aika on vielä reilusti alle maksimiajan.

A*-reitinhakualgoritmille vaikeimpia karttoja tutkittavaksi olivat avoimet kartat. Varsinkin iso (200x200) kartta aiheutti algoritmille suuria vaikeuksia. Jo yksittäinen haku tässä kartassa vei aikaa 657 millisekuntia, joka aiheuttaisi pelissä suuren katkon. Vastaava hakuaika muistipohjaiselle algoritmille oli 14 millisekuntia. Hakuaika on alle maksimiajan, mutta kymmenen haun testissä hakuaika kasvoi jo 135 millisekuntia, joka on reilusti yli maksimiajan. Käytännössä tämä tarkoittaa sitä, että pelaaja voi etsiä yksittäisen pitkän reitin, mutta tekoälyn tulisi välttää pitkiä reittejä.

Testituloksista voimme päätellä, että suoritettavien hakujen pitää olla lyhyitä, että voimme ohjata samanaikaisesti tuhansia tekoälyn hahmoja. Ristimuodostelma tulee olemaan lähempänä todellista kenttää. A*-reitinhakualgoritmillä pystymme siis todellisuudessa tukemaan noin 200 yhtäaikaista reitinhakua, jos haettavan reitin pituus on alle 50 ruutua. Arvio saadaan kertomalla sadan kerran ristimuodostelma kartan testituloksen 7 millisekuntia kahdella. Tämä luku eli 14 millisekuntia on vielä reilusti alle maksimiajan. Vastaavasti muistipohjainen reitinhakualgoritmi pystyy suorittamaan yli tuhat vastaavaa hakua, sillä tuhannen haun testissä tulos hakuaika oli vain 10 millisekuntia.

5 Yhteenveto

Muistipohjainen reitinhakualgoritmin suurin vahvuus on sen reitinhakunopeus. Se osoittautui huomattavasti A*-reitinhakualgoritmia nopeammaksi kaikissa hakutilanteissa. Heikkoutena kuitenkin on se, että kartan noodien reittilistat pitää luoda kartan käynnistysvaiheessa. Testauksessa havaitsin, että 200x200-suuruisen kartan käynnistämiseen meni noin 15 sekuntia aikaa. Tämän jälkeen hitautta ei ole havaittavissa. Ainoastaan

noodin sammuttaminen aiheuttaa ylimääräistä kuormaa suorittimelle, kun reitit on laskettava uudelleen. Tämä ei kuitenkaan aiheuta suurta hitautta, joten reittienlevitysprotokolla voidaan todeta toimivaksi.

Aluelistojen ansioista muistinkäyttö on melko pieni. Ison kartan reittien alustusvaiheessa oli havaittavissa, että muistin käyttö oli noin 800 megatavun luokkaa. Alustuksen jälkeen muistinkäyttö on kuitenkin vain alle 30 megatavua. Alueiden koolla voidaan vaikuttaa reittilistojen reittien määrään, mikä vähentää muistinkäyttöä huomattavasti. Liian pieni alueen koko kuitenkin aiheuttaa sen, että aluelistan reittien määrä kasvaa liian suureksi, jolloin muistinkäyttö ei vähenekään.

Muistipohjainen reitinhakualgoritmi on kuitenkin toimiva ja huomattavasti A*-reitinhakualgoritmia nopeampi tapa hakea useita reittejä. Hyvin harva peli kuitenkaan vaatii näin monen reitin yhtäaikaista hakemista. A*-reitinhakualgoritmi on huomattavasti helpompi ottaa käyttöön eikä vaadi suurta kartan latausaikaa. Silti muistipohjainen reitinhakualgoritmi tarjoaa mahdollisuuden tehdä useampia reitinhakuja yhtäaikaisesti. Se tarjoaa hyvän vaihtoehdon A*-reitinhakualgoritmille, kun suurempi muistinkäyttö ei ole ongelma.

Lähteet

1. Primitive Data Types. Verkkoaineisto. Oracle Corporation. <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>. Luettu 9.5.2019.

2. IP Addressing: IPv4 Addressing Configuration Guide, Cisco IOS XE Release 3S. Verkkoaineisto. Cisco Systems, Inc. <https://www.cisco.com/c/en/us/support/docs/ip/open-shortest-path-first-ospf/7039-1.html>. Luettu 25.4.2019.

3. OSPF Design Guide. Verkkoaineisto. Cisco Systems, Inc. <https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipaddr_ipv4/configuration/xe-3s/ipv4-xe-3s-book/configuring_ipv4_addresses.html#GUID-F558773E-5292-4E14-883D-4BBCED76E508>. Luettu 25.4.2019