Haaga-Helia
ammattikorkeakoulu Oy

# Tietokoneshakin ohjelmointi JavaScript-kielellä

Jarkko Kuusela

Opinnäytetyö
Tietojenkäsittelyn
koulutusohjelma
2017

# Tiivistelmä

Haaga-Helia ammattikorkeakoulu Oy

**Tekijä(t)**
Jarkko Kuusela

**Koulutusohjelma**
Tietojenkäsittelyn koulutusohjelma

**Opinnäytetyön otsikko**
Tietokoneshakin ohjelmointi JavaScript-kielellä

**Sivu- ja liitesivumäärä**
36 + 41

Shakki on suosittu kaksin pelattava strateginen lautapeli. Tietokoneshakin ohjelmointia on syytä tutkia, sillä täydellistä tietokoneshakkia ei ole vielä onnistuttu toteuttamaan liian suuren tietomäärän vuoksi. Nykypäivänä on vielä tarve tutkia, kuinka kehitetään algoritmeja ja tekniikoita paremmiksi. Tavoitteena on ohjelmoida mahdollisimman älykäs ja tehokas tietokoneshakkiohjelma, joka pystyy laskemaan mahdollisimman pitkälle saavuttaen parhaan mahdollisen lopputuloksen.

Tavoitteena projektissa oli tutkia, miten JavaScript-kielellä voidaan toteuttaa shakkipeli selainympäristöön. Peliin oli tavoitteena kehittää selkeä ja yksinkertainen käyttöliittymä. Tekoälyn oli kyettävä säännönmukaisiin siirtoihin järkevin perustein kohtuullisessa ajassa. Ohjelman vaatimuksiin kuului sääntöjen määrittely, laudan esitys, arviointifunktio ja pelipuun käsittely.

Tavoitteena ei ollut tehdä huippuluokan tekoälyä shakkipeliin projektissa. Käyttöliittymän ei tarvinnut olla yhtä monipuolinen kuin laadukkaammissa shakkiohjelmistoissa. Tavoitteena ei ollut kehittää varsinaista verkkopeliä. JavaScript-kirjastoja ei käytetty. Tietokantaominaisuudet rajattiin pois.

Teoriaosassa esiteltiin pelin säännöt, jossa käytiin läpi shakkilauta, nappulat, shakki, linnoitus, shakki ja matti, tasapelisäännöt ja nappuloiden arvot. Käytiin läpi, kuinka shakkilauta kannattaa esittää tietokoneohjelmassa ja käsiteltiin siirtojen tuottamista. Pohdittiin, kuinka arviointifunktio kannattaa toteuttaa. Pelipuuhun liittyvät minmax-algoritmi ja alpha-beta-karsinta on käsitelty. Lisäksi kuvattiin lyhyesti, mikä on JavaScript.

Ohjelmointi aloitettiin kesällä 2015. Projekti käynnistettiin virallisesti syksyllä 2015. Varsinainen raportointi aloitettiin syksyllä 2017.

Projektin aikana toteutettiin tietokoneshakkiohjelma, joka toimii valtaselaimilla JavaScript-kielellä. Tietokoneshakkiohjelmaan kuului pelikoodi, joka vastasi käyttöliittymän, pelisääntöjen ja tekoälyn yhdistämisestä. Sääntökoodi vastasi sääntöjen oikeudenmukaisuudesta. Tekoälykoodi laski siirtoja käyttäen minmax-algoritmia ja arviointifunktiota. Käyttöliittymästä tuli selkeä. Sillä pystyi pelaamaan säännönmukaisia siirtoja. Tekoälystä ei tullut huipputasoista, mutta harjoitusvastustajaksi aloitteleville pelaajille se kelpaisi.

**Asiasanat**
shakki, tekoäly, ohjelmointi, JavaScript

## Sisällys

# 1 Johdanto

Shakki on suosittu kaksin pelattava strateginen lautapeli, jota ihmiset ovat pelanneet jo yli tuhat vuotta. Pitkästä iästä huolimatta shakki on vielä suuressa suosiossa eri puolilla maailmaa. Kun ihmisen kyvyt eivät enää riitä, tietotekniikkaa voidaan käyttää apuna shakin analysoinnissa. Tietokoneshakin ohjelmointia on syytä tutkia, sillä täydellistä tietokoneshakkia ei ole vielä onnistuttu toteuttamaan liian suuren tietomäärän vuoksi. Nykypäivänä on vielä tarve tutkia, kuinka kehitetään algoritmeja ja tekniikoita paremmiksi. Tavoitteena on ohjelmoida mahdollisimman älykäs ja tehokas tietokoneshakkiohjelma, joka pystyy laskemaan mahdollisimman pitkälle saavuttaen parhaan mahdollisen lopputuloksen. Shakki on tärkeä osa tekoälyn tutkimista. Tekoälyn kehittäminen voi tulevaisuudessa avata uusia mahdollisuuksia yhteiskunnalle, ja esimerkiksi shakin avulla voidaan edistää tekoälytutkimuksia.

Tavoitteena tässä projektissa on tutkia, miten JavaScript-kielellä voidaan toteuttaa shakkipeli selainympäristöön. Peliin kehitetään selkeä ja yksinkertainen käyttöliittymä. Tekoälyn on kyettävä säännönmukaisiin siirtoihin järkevin perustein kohtuullisessa ajassa. Ohjelman vaatimuksiin kuuluu sääntöjen määrittely, laudan esitys, arviointifunktio ja pelipuun käsittely.

Ohjelmaan täytyy määritellä pelinappuloiden sijainti laudalla ja pelaajan siirtovuoro. Ohjelman täytyy löytää jokaisessa asemassa lailliset siirtomahdollisuudet. Tekoälyn täytyy rakentaa pelipuu. Täydellisen pelipuun rakentamisen aikavaativuus on liian suuri, joten tekoälyn täytyy lopettaa pelipuun rakentaminen jossakin vaiheessa jollakin perusteella. Keskeneräistä pelipuuta varten tarvitaan arviointifunktio, joka arvioi eri siirtovaihtoehtojen edullisuutta. Minmax-algoritmi suodattaa parhaan siirtovaihtoehdon pelipuussa olevien arvojen perusteella. Alpha-beta-karsinta nopeuttaa minmax-algoritmin toimintaa.

Lopputuloksena syntyy kirjallinen teoriaosuus, joka sisältää sekä yleistä tietoa tietokoneshakin ohjelmoinnista, että yksityiskohtaisempaa selostusta toteutetun tietokoneshakkiohjelman toiminnasta. Lopputulokseen kuuluu tietokoneshakkiohjelman kommentoitu lähdekoodi.

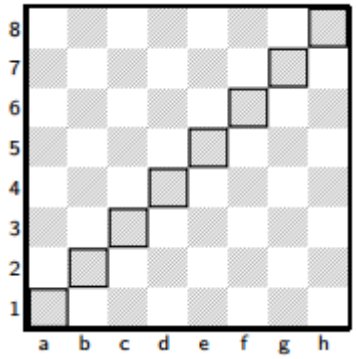
Tavoitteena ei ole tehdä huippuluokan tekoälyä shakkipeliin tässä projektissa. Käyttöliittymän ei tarvitse olla yhtä monipuolinen kuin laadukkaammissa shakkiohjelmistoissa. Vaikka pelissä voi pelata kaksi käyttäjää vastakkain, tavoitteena ei ole kehittää varsinaista verkkopeliä. JavaScript-kirjastoja ei käytetä, koska ilman niitä selvitään. Tietokantaominaisuudet rajataan pois, koska ne eivät ole välttämättömiä shakkipelin toiminnalle.

Shakkipelistä on hyötyä peruskäyttäjälle, joka voi pelata shakkipeliä selaimessa joko tekoälyä tai toista peruskäyttäjää vastaan harjoittelun ja ajanvietteen vuoksi. Projektia voi mahdollisesti käyttää pohjana jatkokehitykselle.

## 2 Shakin säännöt

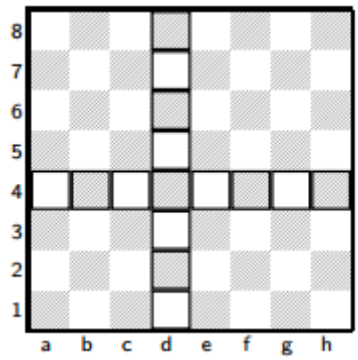
Shakki on kaksin pelattava peli, jota pelataan shakkilaudalla. Shakkilauta on ruutukuvioinen lauta, jossa on yhteensä 64 ruutua. Ruudut ovat vuoroin vaaleita ja tummia. Shakkilaudoissa voi olla laudan reunoilla numerot ja kirjaimet, joita kutsutaan koordinaateiksi. Koordinaattien avulla on mahdollista nimetä jokainen shakkilaudan ruutu. (Vainio 2005, 1.)

Kuviossa 1 on korostettu ääriviivoilla ruudut a1, b2, c3, d4, e5, f6, g7 ja h8. Tämän tyyppistä viistolinjaa kutsutaan shakissa usein diagonaaliksi. (Vainio 2005, 1.)



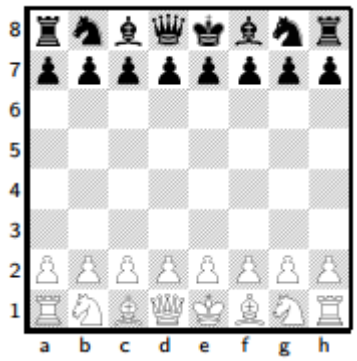

Kuvio 1. Diagonaali shakkilaudalla (Vainio 2005, 1)

Shakkilauta jakautuu kahdeksaan riviin ja kahdeksaan linjaan (Vainio 2005, 1-2). Rivejä nimitetään numeroilla ja linjoja kirjaimilla. Kuviossa 2 ovat korostettuina ääriviivoilla neljäs rivi ja d-linja. Neljänteen riviin kuuluvat kuvion 2 mukaan ruudut a4, b4, c4, d4, e4, f4, g4 ja h4. D-linja sisältää kuvion 2 mukaan ruudut d1, d2, d3, d4, d5, d6, d7 ja d8. (Vainio 2005, 2.)




Kuvio 2. Rivi ja linja shakkilaudalla (Vainio 2005, 1)

Kuviossa 3 shakkinappulat on aseteltu alkuasemaan. Shakkipeliä pelataan siis shakkinappuloilla shakkilaudalla. Shakkilaudalle asetetaan molemmille pelaajille kahdeksan sotilasta, kaksi tornia, kaksi ratsua, kaksi lähettiä, yksi kuningatar ja yksi kuningas. Valkoiset nappulat tulevat ensimmäiselle ja toiselle riville ja mustan nappulat asetetaan kahdeksannelle ja seitsemännelle riville. Sotilaat asetetaan toiselle ja seitsemännelle riville. Sen jälkeen asetetaan tornit nurkkiin, tornien viereen ratsut ja ratsujen viereen lähetit. Daamit asetetaan d-ruutuihin ja kuninkaat e-ruutuihin. Kuviossa 4 on nimetty shakkinappuloita vastaavat symbolit. (Vainio 2005, 2.)




Kuvio 3. Shakkinappulat shakkilaudalla (Vainio 2005, 2)

♙ sotilas.
♖ torni.
♘ ratsu eli hevonen.
♗ lähetti.
♕ kuningatar eli daami.
♔ kuningas.

Kuvio 4. Shakkinappulat nimettyinä (Vainio 2005, 2)

Shakkikuvioissa valkoisen puoli on kuvattu joka kerta alhaalla ja mustan ylhäällä. Lauta voidaan jakaa e- ja d-linjojen välistä keskeltä kahtia, jolloin vasenta puolta kutsutaan kuningatarsivustaksi ja oikeaa puolta kuningassivustaksi. Kaikkia nappuloita kutsutaan upseereiksi paitsi sotilaita. (Vainio 2005, 2.)

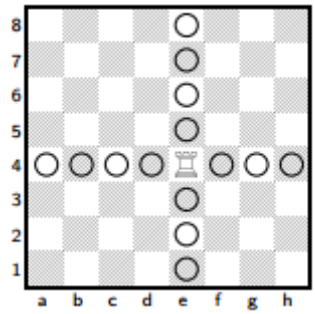
Peli alkaa, kun nappulat ovat alkuasemassa. Valkoisilla nappuloilla pelaava pelaaja tekee ensimmäisen siirron, jonka jälkeen pelaajat siirtävät nappuloitaan vuorotellen. Shakissa siirtovuoroa ei voi jättää pelaamatta, joten siirtovuorolla on tehtävä vain ja ainoastaan yksi siirto. (Vainio 2005, 2.)

### 2.1 Nappuloiden liikkeet

Pelaajan on tiedettävä ennen pelin aloittamista, kuinka shakkinappulat liikkuvat laudalla. Huomataan, kuinka sotilas, torni, ratsu, lähetti, daami ja kuningas liikkuvat eri tavalla. (Vainio 2005, 2-3.)

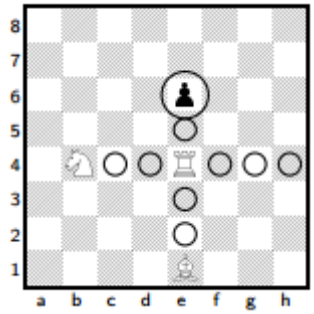
#### 2.1.1 Torni

Kuvion 5 mukaisesti torni liikkuu vaaka- ja pystysuoraan laudalla. Torni liikkuu niin monta ruutua kuin haluaa aina laudan laitoihin asti. Toisaalta torni ei pysty hyppäämään muiden nappuloiden yli. (Vainio 2005, 4.)




Kuvio 5. Tornin liikkeet (Vainio 2005, 4)

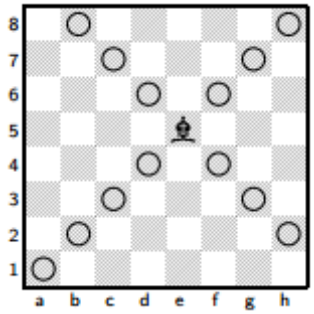
Kuviossa 6 torni voi liikkua vastustajan nappulan päälle ja syödä sen, jolloin kuvion 6 tapauksessa musta sotilas poistetaan shakkilaudalta. Toisaalta torni ei voi siirtyä omien nappuloiden päälle, jolloin omia nappuloita ei voi syödä. (Vainio 2005, 4.)




Kuvio 6. Torni muiden nappuloiden kanssa (Vainio 2005, 4)

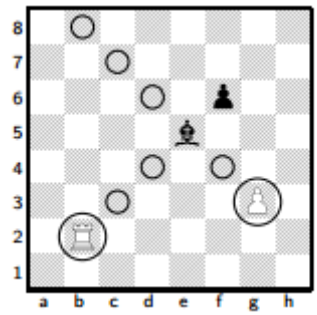
#### 2.1.2 Lähetti

Kuvion 7 mukaisesti lähetti liikkuu diagonaaleja pitkin laudalla. Lähetti liikkuu niin monta ruutua kuin haluaa aina laudan laitoihin asti. Toisaalta lähetti ei pysty hyppäämään muiden nappuloiden yli. (Vainio 2005, 4.)



Kuvio 7. Lähetin liikkeet (Vainio 2005, 4)

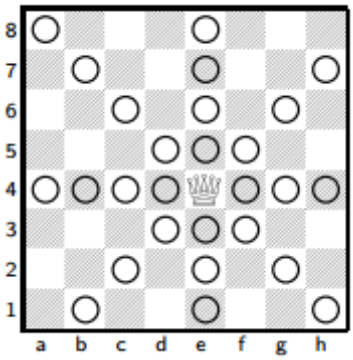
Kuviossa 8 lähetti voi liikkua vastustajan nappulan päälle ja syödä sen, jolloin kuvion 8 tapauksessa valkoinen torni tai valkoinen sotilas poistetaan shakkilaudalta. Toisaalta lähetti ei voi siirtyä omien nappuloiden päälle, jolloin omia nappuloita ei voi syödä. (Vainio 2005, 5.)



Kuvio 8. Lähetti muiden nappuloiden kanssa (Vainio 2005, 4)

### 2.1.3 Kuningatar

Kuvion 9 mukaisesti kuningatar liikkuu vaakasuoraan, pystysuoraan ja diagonaaleja pitkin laudalla. Kuningatar liikkuu niin monta ruutua kuin haluaa aina laudan laitoihin asti. Toisaalta kuningatar ei pysty hyppäämään muiden nappuloiden yli. (Vainio 2005, 5.)



Kuvio 9. Kuningattaren liikkeet (Vainio 2005, 5)

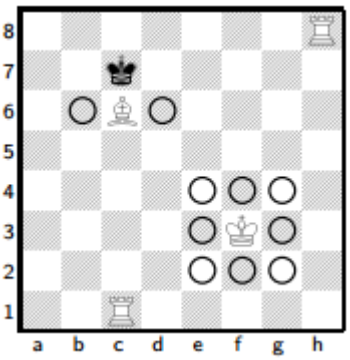
Kuningatar voi liikkua vastustajan nappulan päälle ja syödä sen, jolloin vastustajan nappula poistetaan shakkilaudalta. Toisaalta kuningatar ei voi siirtyä omien nappuloiden päälle, jolloin omia nappuloita ei voi syödä. (Vainio 2005, 5.)

### 2.1.4 Kuningas

Kuningas voi liikkua jokaiseen suuntaan, mutta vain yhden ruudun kerrallaan. Kuningas voi astua jokaiseen sen vieressä olevaan ruutuun tyhjällä laudalla. (Vainio 2005, 5.)

Kuningasta ei voida syödä pois laudalta. Tällöin on kiellettyä joutua tilanteeseen, jossa vastustajalla olisi mahdollisuus syödä kuningas pois laudalta. (Vainio 2005, 5.)

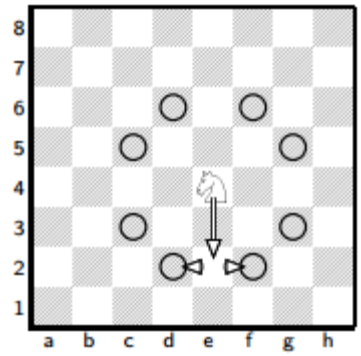
Kuviossa 10 on kaksi kuningasta. Jos valkoisilla pelaava olisi siirtovuorossa, hän voisi siirtää valkoista kuningastaan tavallisesti. Jos mustilla pelaava olisi siirtovuorossa, hänellä olisi vain kaksi siirtovaihtoehtoa, sillä kuusi muuta ruutua mustan kuninkaan ympärillä ovat uhattuja. (Vainio 2005, 5.)




Kuvio 10. Kuninkaan liikkeet muiden nappuloiden kanssa (Vainio 2005, 5)

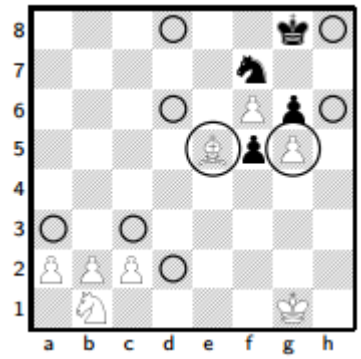
### 2.1.5 Ratsu

Kuvion 11 mukaisesti ratsua siirretään kaksi ruutua vaakasuoraan tai pystysuoraan ja sen jälkeen yksi ruutu sivulle. Ratsun liikkeet muistuttavat eri päin olevia L-kirjaimia. (Vainio 2005, 6.)



Kuvio 11. Ratsun liikkeet (Vainio 2005, 6)

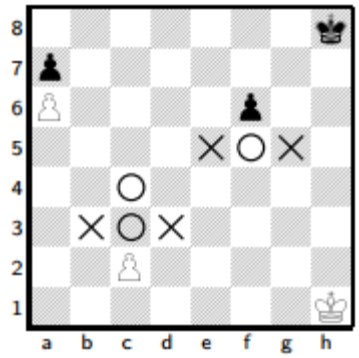
Kuviosta 12 näkee, että ratsu voi hypätä omien ja vastustajien nappuloiden yli. Ratsu voi liikkua vastustajan nappulan päälle ja syödä sen, jolloin vastustajan nappula poistetaan shakkilaudalta. Toisaalta ratsu ei voi siirtyä omien nappuloiden päälle, jolloin omia nappuloita ei voi syödä. (Vainio 2005, 6.)



Kuvio 12. Ratsun liikkeet muiden nappuloiden kanssa (Vainio 2005, 6)

### 2.1.6 Sotilas

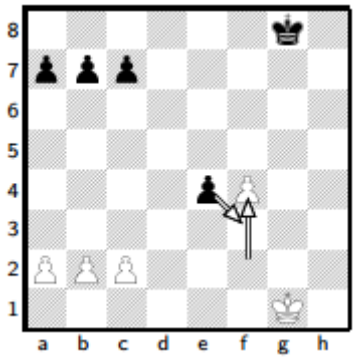
Kuviossa 13 on neljä sotilasta. Sotilaat a-linjalla eivät voi liikkua mihinkään, koska sotilas voi astua edessä olevaan ruutuunsa vain, jos kyseisessä ruudussa ei ole nappulaa. Lisäksi sotilas etenee vain eteenpäin. Sotilas ei koskaan liiku taaksepäin. Valkoisen c-sotilas voi astua joko yhden tai kaksi ruutua eteenpäin. Normaalisti sotilas voi edetä vain yhden ruudun kerralla, mutta kyseinen c-sotilas voi tehdä kaksoisaskeleen, koska se sijaitsee lähtöruudussa. Sotilaat voivat mennä yhden ruudun viistosti eteenpäin, jos kohderuudussa on vastustajan nappula. Tällöin vastustajan nappula tulee syödyksi ja poistetaan laudalta. (Vainio 2005, 6.)



Kuvio 13. Sotilaan liikkeet (Vainio 2005, 7)

Sotilas on korotettava, jos se saavuttaa laudan viimeisen rivin. Tällöin kyseinen sotilas poistetaan laudalta ja tilalle samaan ruutuun laitetaan oman värinen torni, ratsu, lähetti tai daami. (Vainio 2005, 7.)

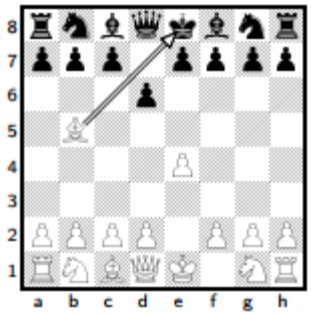
Kuviossa 14 pelaajalla on mahdollisuus tehdä ohestalyönti vastustajan sotilaan kaksoisaskeleen jälkeen. Kun vastustajan sotilas siirtyy kaksoisaskeleella pelaajan sotilaan viereiseen ruutuun, voi pelaaja tehdä ohestalyönnin siirtämällä sotilastaan viistosti juuri kaksoisaskeleen tehneen sotilaan taakse. Jos ohestalyönti tehdään, kyseinen vastustajan sotilas poistetaan laudalta. Oikeus ohestalyöntiin on ainoastaan heti vastustajan sotilaan kaksoisaskeleen jälkeen, jolloin sitä ei voi enää myöhemmillä siirroilla tehdä. Ohestalyönnissä sotilas voi syödä vain toisen sotilaan ja vain kaksoisaskeleen jälkeen. (Vainio 2005, 8.)



Kuvio 14. Ohestalyönti sotilaalla (Vainio 2005, 8)

### 2.2 Shakki – kuninkaan uhkaaminen

Shakkipeli on saanut nimensä peliin liittyvästä keskeisestä käsitteestä. Shakki siis tarkoittaa shakkipelissä tilannetta, jossa kuningas on uhattuna. Kuviossa 15 valkoinen lähetti uhkaa mustaa kuningasta. Uhkaavaa nappulaa kutsutaan shakkaavaksi nappulaksi. (Vainio 2005, 10.)



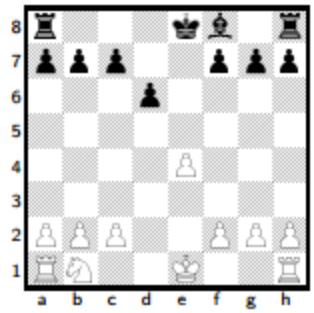
Kuvio 15. Kuningas shakissa (Vainio 2005, 10)

Shakki on aina torjuttava välittömästi. Shakki on mahdollista torjua syömällä shakkaava nappula, laittamalla oma nappula shakkaavan nappulan ja kuninkaan väliin tai väistymällä kuninkaalla turvalliseen ruutuun. (Vainio 2005, 10.)

### 2.3 Linnoitus

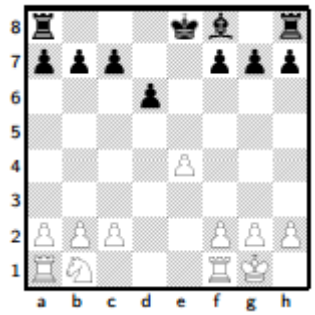
Linnoitus on shakin yksi erikoissiirroista. Linnoituksessa saa siirtää kuningasta ja toista tornia yhtä aikaa. Molempien nappuloiden tulee olla lähtöruuduissaan ja niiden välissä ei saa olla muita nappuloita. Linnoitus on mahdollista tehdä molemmille puolille lautaa. Linnoituksessa kuningas siirtyy kaksi ruutua sivulle tornia kohti, jonka jälkeen torni siirtyy kuninkaan yli kuninkaan viereiseen ruutuun. (Vainio 2005, 15.)

Kuviossa 16 valkoinen pystyy linnoittautumaan kuningassivustan puolelle. Linnoitus kuningassivustan puolelle tarkoittaa samaa kuin lyhyt linnoitus. (Vainio 2005, 15.)



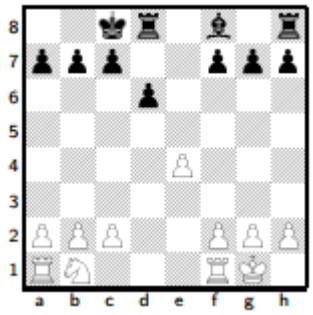
Kuvio 16. Tilanne ennen linnoituksia (Vainio 2005, 15)

Kuviossa 17 musta pystyy linnoittautumaan kuningatarsivustan puolelle. Linnoitus kuningatarsivustan puolelle tarkoittaa samaa kuin pitkä linnoitus. (Vainio 2005, 15.)



Kuvio 17. Lyhyt linnoitus suoritettu (Vainio 2005, 15)

Kuviossa 18 molemmat pelaajat ovat tehneet linnoituksen. Aina linnoitus ei onnistu näin helposti, sillä linnoitukseen liittyy lisää erikoissääntöjä. Linnoitusta ei saa tehdä, jos linnoitukseen osallistuvaa tornia tai kuningasta on jo siirtänyt aiemmin, linnoitukseen aikova kuningas on shakissa tai kuningas hyppäisi linnoituksessa uhatun ruudun yli. (Vainio 2005, 15-16.)

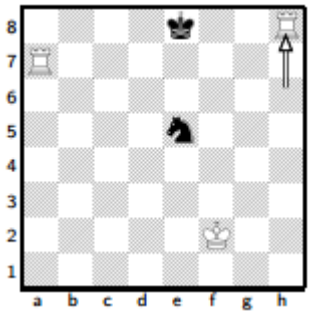


Kuvio 18. Pitkä linnoitus suoritettu (Vainio 2005, 15)

### 2.4 Shakki ja matti – pelin voittaminen

Jos kuningas on shakkipelissä shakissa, eikä pelaaja pysty torjumaan shakkia, hän on häviäjä. Tilanne on tällaisessa tapauksessa shakki ja matti. (Vainio 2005, 19.)

Kuviossa 19 valkoinen torni shakkaa mustaa kuningasta. Musta kuningas ei voi paeta shakkia, jolloin valkoinen on voittanut pelin. (Vainio 2005, 19.)



Kuvio 19. Kuningas shakissa ja matissa (Vainio 2005, 19)

Shakkipelin voi voittaa, jos vastustaja ilmoittaa luovuttavansa. Luovutus yleensä tapahtuu, kun huomataan, että oma kuningas joutuu shakkiin ja mattiin tulevaisuudessa pakosti. (Vainio 2005, 21.)

### 2.5 Tasapeli

Shakkipeli voi päättyä tasapeliin. Tasapeliin voidaan päätyä monin eri tavoin. (Vainio 2005, 23.)

Patti on pelitilanne, jossa pelaaja ei voi siirtää mitään nappulaa ja mikään vastustajan nappuloista ei shakkaa. Patti on siis tasapeli. (Vainio 2005, 23.)

Ikuinen shakki on tasapelin muoto, jossa peli ei etene kuninkaan joutuessa joka siirron jälkeen uudestaan shakkiin. Ikuinen shakki johtaa yleensä siirtojen toistumiseen. (Vainio 2005, 23.)

Jos pelilaudalla on toistunut kolme kertaa täysin sama asema, molemmilla pelaajilla on oikeus vaatia tasapeliä. Tätä tasapelin muotoa kutsutaan siirtojen toistumiseksi. (Vainio 2005, 24.)

50:n siirron säännön mukaan shakkipeli on tasapeli, jos peli ei ole edennyt viimeisen 50:n siirtoparin aikana. Shakkipelin katsotaan edenneen, jos toinen pelaajista liikuttaa sotilasta tai syö jonkun pelinappulan. Kun peli etenee, laskeminen aloitetaan alusta. (Vainio 2005, 24.)

Shakkipelin aikana pelaajat voivat ehdottaa tasapeliä missä vaiheessa tahansa. Jos vastapelaaja suostuu ehdotukseen, päättyy peli sopimustasapeliin. (Vainio 2005, 24.)

Jos kummallakaan pelaajalla ei ole shakin ja matin tekoon riittäviä nappuloita laudalla, peli päättyy tasapeliin (Vainio 2005, 24). Pelkällä lähetillä tai ratsulla ei pysty shakkia ja mattia tekemään (Vainio 2005, 25).

### 2.6 Nappuloiden arvot

Kuviossa 20 on laskettu jokaiselle nappulalle arvo. Sotilas on yhden sotilaan arvoinen, ratsu ja lähetti ovat kolmen sotilaan arvoisia, torni on viiden sotilaan arvoinen, kuningatar on yhdeksän sotilaan arvoinen ja kuningas on mittaamattoman arvokas. (Vainio 2005, 26.)

| Nappula | Arvo |
|---|---|
| ♙ | 1 |
| ♘ | 3 |
| ♗ | 3 |
| ♖ | 5 |
| ♕ | 9 |
| ♔ | ∞ |

Kuvio 20. Nappuloiden arvot (Vainio 2005, 26)

Kyseiset shakkinappuloiden arvot eivät ole absoluuttisia (Shannon 1950, 260). Nappuloiden arvot eivät varsinaisesti kuulu shakin sääntöihin, mutta ne helpottavat eri pelitilanteiden arviointia (Vainio 2005, 26).

# 3 Laudan representaatio ja sallittujen siirtojen tuottaminen

Siirtojen tuottaminen on oleellinen toteutettava osa tietokoneshakin ohjelmoinnissa. Keskeinen kysymys on, miten tietokone saadaan hahmottamaan lauta ja nappulat. Lisäksi on tiedettävä, miten kustakin asemasta tuotetaan kaikki sallitut siirrot. Siirtojen tuottaminen on merkittävä osa-alue tehokkuuden kannalta, sillä ohjelman miettiessä siirtoaan prosessoriajasta suuri osa kuluu siirtojen tuottamiseen. (Seppälä 2004, 7.)

Shakkilaudan voi esittää tietokoneelle monella eri tavalla. Toiset tavat sopivat paremmin aloitteleville ja toiset kehittyneimmille ohjelmoijille. (Seppälä 2004, 7.)

## 3.1 8x8 taulukko

Yksinkertaisin tapa laudan koodaamiseen tietokoneelle on 8x8 taulukko. Tietokoneshakkiohjelmassa taulukon arvoina ovat vastaavan ruudun nappulat tai ruudun ollessa tyhjä tyhjä-arvo. Mahdolliset arvot ovat esimerkiksi vS (valkea sotilas), vL (valkea lähetti), vR (valkea ratsu), vT (valkea torni), vD (valkea daami), vK (valkea kuningas), mS (musta sotilas), mL (musta lähetti), mR (musta ratsu), mT (musta torni), mD (musta daami), mK (musta kuningas) ja tyhjä-arvo. Tämän tavan etuna on materiaalin laskemisen helppous. Käydään silmukassa kaikki ruudut läpi ja summataan arvot. (Seppälä 2004, 7.)

Sallittujen siirtojen tuottaminen voi olla haasteellista tässä tavassa. Voi olla hankalaa havaita, milloin nappula on mennyt ulos laudalta. Tähän voidaan tarvita monimutkaisia testejä, jolloin kaikki eivät suosi tätä tapaa. (Seppälä 2004, 7.)

## 3.2 12x10 taulukko

Yleinen tapa laudan koodaamiselle on 12x10 taulukko. Shakkilaudan alapuolella ja yläpuolella on ylimääräiset kaksi riviä ja kummallakin sivulla yksi ylimääräinen linja (taulukko 1) (Seppälä 2004, 7-8). Nappulat voidaan esittää lukuina, jolloin 1 = valkea sotilas, 2 = valkea ratsu, 3 = valkea lähetti, 4 = valkea torni, 5 = valkea daami, 6 = valkea kuningas, -1 = musta sotilas, -2 = musta ratsu, -3 = musta lähetti, -4 = musta torni, -5 = musta daami, -6 = musta kuningas, 0 = tyhjä ruutu ja 7 = ei laudalla (taulukko 2) (Shannon 1950, 264).

Taulukko 1. Taulukko indekseineen (Seppälä 2004, 8)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 |
| 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 |
| 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 |
| 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 |
| 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 |
| 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 |
| 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Taulukko 2. Taulukko arvoineen (Seppälä 2004, 8)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| 7 | -4 | -2 | -3 | -5 | -6 | -3 | -2 | -4 | 7 |
| 7 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 7 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 |
| 7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 7 |
| 7 | 4 | 2 | 3 | 5 | 6 | 3 | 2 | 4 | 7 |
| 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |

Siirtojen tuottaminen on helppoa, kun shakkilaudan rajat voidaan havaita. Mahdolliset siirrot voidaan laskea plus- ja miinuslaskutoimituksina. Tietoina tarvitaan nappulan sijainti laudalla ja liikkumistapa. Mikään luku ei siis saa siirtyä luvulle 7. Vaikka tällä ohjelmointitavalla voidaan varmistaa, että nappula pysyy laudalla, pitää vielä varmistaa siirron laillisuus. Esimerkiksi täytyy tarkistaa, ettei syö omaa nappulaa. Tässä tapauksessa positiivinen luku ei saa siirtyä positiivisen eikä negatiivinen luku negatiivisen luvun päälle. Ohestalyönti ja linnoitus vaativat silti monimutkaisempaa ohjelmointia. (Seppälä 2004, 8.)

### 3.3 0x88

Moderni menetelmä on 0x88-systeemi. Tässä pelilaudan oikealla puolella on toinen lauta, jolloin ruutuja on 128. Käyttämättömän lisälaudan ansiosta voidaan yhdellä lauseella testata, ollaanko pelilaudalla vai ei. (taulukko 3.) (Seppälä 2004, 9.)

Taulukko 3. Vasemmalla pelilauta ja oikealla lisälauta (Seppälä 2004, 9)

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 |
| 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 |
| 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 |
| 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

Testilauseessa i on laillinen, jos ja vain jos (i&0x88) == 0. Testilause sisältää kaksi testiä. 0x80-testi testaa, ollaanko 128 ruudun alueella. 0x08-testi testaa, ollaanko pelilaudalla. (Seppälä 2004, 9.)

Jokainen ruutu ymmärretään kahdeksan bitin sanana. Ensimmäiset neljä bittiä kertovat rivin ja jälkimmäiset neljä bittiä linjan. Tärkeää on huomata neljän viimeisen bitin ensimmäinen bitti. Jos kyseinen bitti on yksi, ollaan pelilaudan ulkopuolella. (Seppälä 2004, 9.)

### 3.4 Bittikartat

Monesti pelilauta esitetään bittikarttoina 64-bittisten muuttujien avulla. 64-bittiset muuttujat soveltuvat pelilaudan esittämiseen hyvin, sillä pelilaudan koko on 64 ruutua. Jokaisella bitillä voi siis esittää yhden ruudun. Jos esimerkiksi sotilasbittikartassa ruudussa on sotilas, asetetaan kyseinen bitti ykköseksi, jolloin muissa ruuduissa bitit ovat nollia. (Timonen 2008, 78.)

Pelilauta ja nappulat esitetään kahdentoista 64-bittisen muuttujan listana. Lista voidaan jakaa kahtia, jolloin toisessa 64-bittisen muuttujan listassa on valkeat ja toisessa mustat nappulat. Jokainen bittikartta sisältää yhden nappulatyypin kaikki sijainnit pelilaudalla. (taulukko 4.) (Timonen 2008, 78.)

Taulukko 4. Pelilaudan esittäminen bittikarttojen avulla (Timonen 2008, 79)

| Valkoiset tornit | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Valkoiset ratsut | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Valkoiset lähetit | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Valkoiset daamit | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Valkoinen kuningas | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Valkoiset sotilaat | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| Mustat tornit | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

| Mustat ratsut | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |

| Mustat lähetit | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |

| Mustat daamit | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

| Musta kuningas | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

| Mustat sotilaat | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Kyseiset bittikartat ovat hyviä siirtojen generoimiseen, koska on mahdollista suorittaa loogisia operaatiota (AND, OR, NOT) bittikarttojen välillä. Nappulan sallittujen siirtojen selvittämiseksi luodaan ensin bittikartta kaikista kyseisen nappulan mahdollisista siirroista. Täten tehdään bittikartta kaikista pelaajan nappuloista vertaamalla kaikkia kuutta bittikarttaa OR-operaatiolla. Tehdystä bittikartasta otetaan komplementti (NOT). Täten verrataan komplementin tuloksena saatua bittikarttaa kyseisen nappulan kaikki siirrot sisältävään bittikarttaan AND-operaatiolla. Tämä bittikartta sisältää kaikki kyseisen nappulan sallitut siirrot. (Suh 2005.)

# 4 Arviointifunktio

Shakkitietokoneen ohjelmoijan on osattava määritellä shakkiohjelmalle arviointifunktio. Arviointifunktio arvioi kyseisen pelitilanteen edullisuutta. Mitä parempi arviointifunktio on, sitä laadukkaampaa tietokoneen pelaaminen on. (Ranta-aho 2001, 6.) Sen sijaan hidas arviointifunktio yleensä pilaa hyvän shakkimoottorin (Myyrä 2011, 51).

Arviointifunktio toimii esimerkiksi siten, että se palauttaa kokonaisluvun, joka kuvaa pelitilanteen edullisuutta. Arviointifunktio voi palauttaa johtotilanteessa positiivisen arvon. Vastaavasti arvo nolla tarkoittaa pelin johtamista tasapeliin ja negatiivinen arvo tarkoittaa tietokoneen mahdollista häviämistä, jos vastustaja tekee oikeat siirrot. (Timonen 2008, 79.)

Jos arviointifunktio halutaan hyväksi, sen tulee olla nopeatoiminen ja tulee mahdollisimman tarkasti kuvata todenmukaisia voitonmahdollisuuksia (Ranta-aho 2001, 7). Shakkitietokone voi arvioida shakin tilanteita käyttäen eri tekijöitä, joita ovat esimerkiksi pelaajien nappuloiden lukumäärä eli materiaali, nappuloiden muodostama muunnelma eli kehitys, nappuloiden liikkumavapaus ja kuninkaan suojaus. Jokainen näistä arviointitavoista on tärkeä koko pelin ajan, mutta niiden painoarvo vaihtelee pelin vaiheiden mukaan. Oikean arviointitavan määrittäminen tuo ohjelmoijalle lisähaasteita. (Timonen 2008, 80.)

Arviointifunktio muodostuu tekijöiden lisäksi kertoimista. Kertoimet ovat tekijöiden painokertoimia. Yksi tapa painokertoimien määrittämiseen on tehdä hyvä arvaus ja parantaa tätä arvausta käytännön kokemusten perusteella. Painokerroin voidaan määritellä systemaattisen testauksen avulla, jolloin arviointifunktio rakennetaan vaihe vaiheelta. Painokerroin on mahdollista määrittää automaattisesti ohjelmassa. (Seppälä 2004, 51-52.)

Shakkitietokoneelle pitää määritellä eri nappuloille eri arvot. Arvojen perusteella muodostetaan materiaalietufunktio, joka arvioi yksittäisen nappulan kerrallaan välittämättä muista nappuloista. Tällainen funktio on painotettu lineaarinen funktio. Funktiossa kuvataan pelitilanne, joka koostuu painokertoimista eli nappuloiden arvoista ja piirteistä eli nappuloiden lukumääristä. (Ranta-aho 2001, 7.)

Shakkiohjelmoijan täytyy huomioida, että shakkitietokoneen on osattava antaa useampi siirtovaihtoehto eri asemille. Jos tietokone käyttää aina tiettyä siirtoa aina tietyssä asemassa, voi ihmispelaaja harjoitella voittopelin ulkoa. Tällöin ihminen voisi aina voittaa tietokoneen aina samalla pelillä. Siksi tietokoneen täytyy tehdä tietyissä tilanteissa siirto eri tavalla. Vaihtoehtoisen siirron täytyy olla periaatteessa yhtä hyvä kuin tietokoneen ensin ehdottama siirto. (Ranta-aho 2001, 7.)

# 5 Pelipuu

Kun pelaajia on kaksi, voidaan pelistä luoda pelipuu. Kun rakennetaan pelipuuta, aloitustilanne asetetaan juurisolmuksi. Lapsisolmuiksi laitetaan kaikki ne pelitilanteet, joihin päästään, kun pelaaja tekee yhden siirron. Lapsisolmuille laitetaan edelleen lapsisolmuiksi kaikki vastapelaajan säännönmukaiset siirrot. Lapsisolmuja lisätään koko ajan niin kauan, kunnes kaikki pelipuun haarat johtavat lopputilanteeseen. Näin pelipuu sisältäisi kaikki mahdolliset siirrot ja voitaisiin päätellä, mikä siirto olisi paras aina missäkin tilanteessa. Arvioiden mukaan yhdellä siirtovuorolla on noin 35 erilaista siirtomahdollisuutta ja yksi peli kestää keskimäärin noin 100 siirtoa. Tällöin pelipuun läpikäytäviä haaroja olisi $35^{100}$. (Timonen 2008, 80.) Vaikka mahdollisia erilaisia pelitilanteita on vain $10^{40}$, voidaan samoihin tilanteisiin tulla monia eri reittejä pitkin (Ranta-aho 2001, 4).

Tietokannat voivat sisältää valmiiksi määriteltyjä pelipuita. Shakkitietokoneohjelmaan kannattaa liittää alku- ja loppupelien tietokantoja, jotta tietokoneen laskeminen sujuisi entistäkin nopeammin. Täten tietokone pystyy miettimättä pelaamaan alku- ja loppupelin. Alku- ja loppupelin voi rajata erilleen keskipelistä. Keskipelin pelipuun tutkimiseen voidaan siirtyä ohjelmassa, kun alkupelitietokannasta loppuvat valmiiksi määritetyt siirrot. Vastaavasti loppupelissä siirrytään käyttämään loppupelin tietokantoja. (Timonen 2008, 83.)

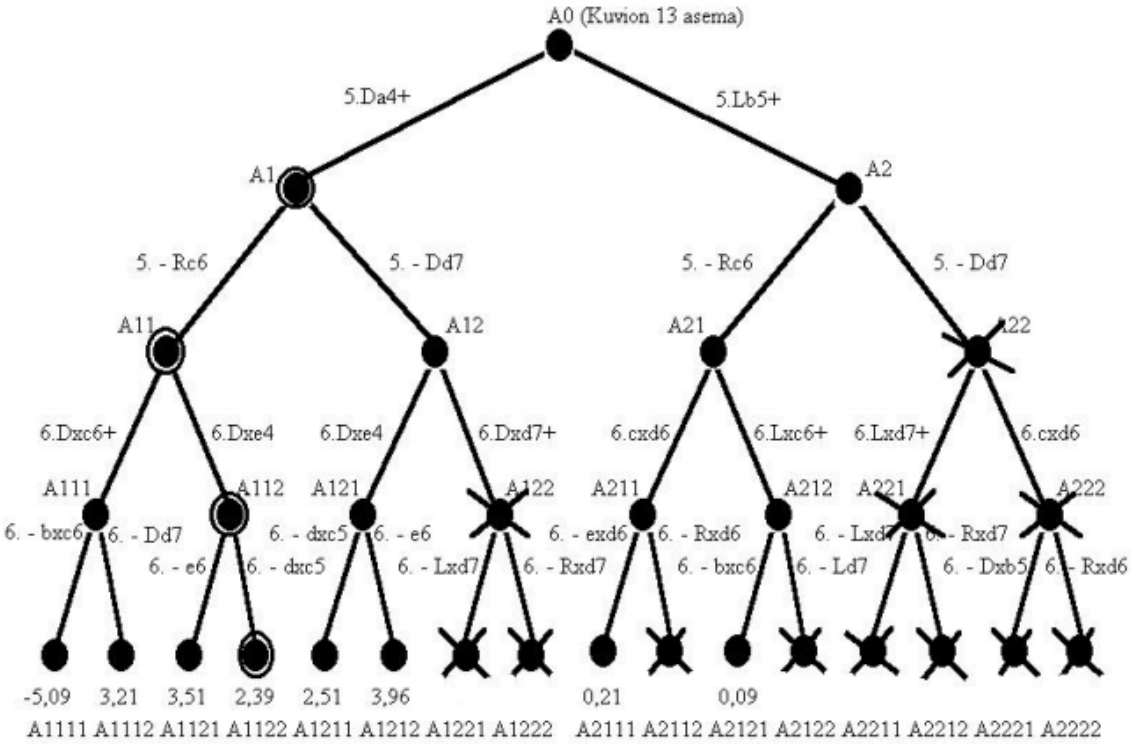
## 5.1 Minmax-algoritmi

Tietokoneshakin ohjelmoinnissa pelipuun luontiin ja läpikäyntiin käytetään esimerkiksi minmax-algoritmia. Minmax-algoritmi käy läpi rekursiivisesti kaikki mahdolliset siirrot ja valitsee parhaan mahdollisen siirron. Mahdolliset siirrot pitää käydä tiettyyn hakusyvyyteen asti, koska kaikkien siirtojen läpi käyminen nykypäivänä on liian hidasta. Minmax-algoritmi palauttaa luvuksi positiivisen luvun, jos pelitilanne on hyvä. Vastaavasti negatiivinen luku ilmaisee huonoa pelitilannetta. (Timonen 2008, 80.)

Pelin alussa ohjelma mahdollisesti seuraa alkupelitietokantaansa, mutta jossain vaiheessa ohjelma joutuu ulos alkupelitietokannastaan. Tässä vaiheessa ohjelman täytyy laskea siirrot, jolloin tarvitaan hakualgoritmia. Ensin hakualgoritmi rakentaa pelipuun generoimalla siirrot. Pelipuun syvyyden voi ohjelmoija määritellä. Hakualgoritmi kutsuu arviointifunktiota, joka arvioi pääteasemat. Lopuksi hakualgoritmi suorittaa tehtävänsä, joka on parhaan mahdollisen pelijatkon löytäminen pelipuusta. Hakualgoritmien perustana on yleensä minmax-algoritmi. Minmax-algoritmi perustuu periaatteelle, jonka mukaan kumpikin pelaaja valitsee kannaltaan parhaan siirtovaihtoehdon, kun pelipuuta käydään läpi alhaalta ylöspäin. (Seppälä 2004, 56.)

## 5.2 Alpha-beta-karsinta

Alpha-beta-karsintaa käytetään pelipuun karsintaan, jolloin pelin lopputulokseen vaikuttamattomat siirrot jätetään huomioimatta ja shakkitietokone pystyy laskemaan siirtoja huomattavasti nopeammin. Kyseisen karsinnan avulla voidaan tietyn solmun alipuu poistaa hausta kokonaan, jos puusta on jo muualta löydetty paremman tuloksen antava solmu. Lapsisolmujen tutkimisjärjestyksellä on väliä, sillä jos jatkuvasti kohdataan satunnaisesti entistä parempia arvoja, ei alipuita pysty karsimaan. Siksi ensin täytyy arvioida ne jälkeläiset, joista mahdollisesti saadaan parhaimmat arvot. (Kuvio 21.) (Ranta-aho 2001, 8.)




Kuvio 21. Esimerkki pelipuusta alpha-beta-karsinnan jälkeen. Rastilla merkittyjä asemia ei tutkita. (Seppälä 2004, 63)

Alpha-beta-karsinnan voi toteuttaa niin, että minmax-algoritmissa pidetään kirjaa kahdesta muuttujasta. Muuttuja alpha alustetaan negatiivisella äärettömällä ja beta positiivisella äärettömällä. Muuttujaan alpha tallennetaan suurin löydetty maksimiarvo ja vastaavasti muuttujaan beta tallennetaan pienin löydetty minimiarvo. (Timonen 2008, 82.)

# 6 JavaScript

JavaScript on tulkattava ohjelmointikieli, jossa on olioperustaisen ohjelmoinnin mahdollisuuksia. Syntaktisesti JavaScript-kielen perusta muistuttaa kieliä C, C++ ja Java, sillä näissä kaikissa on samoja ohjelmallisia käsitteitä, kuten esimerkiksi if-lause, while-silmukka ja &&-operaattori. JavaScript on tietotyyppivapaa kieli, jolloin muuttujilla ei tarvitse olla tyyppimäärittelyä. JavaScript-kielessä oliot yhdistävät ominaisuuksien nimet mielivaltaisiin ominaisuuksien arvoihin. Täten JavaScript-oliot muistuttavat enemmän hajautustaulukoita kuin Javan olioita. JavaScript-kielessä olioperustainen perimämekanismi on prototyyppiperustainen. (Flanagan 2006, 1.)

JavaScript tukee numeroita, merkkijonoja ja totuusarvoja. Se sisältää lisäksi sisäänrakennetun tuen taulukoille, päivämäärille ja säännöllisille lausekkeille. (Flanagan 2006, 1.)

JavaScript on yleisesti käytetty verkkoselaimissa. Sitä on laajennettu olioilla, jotka sallivat koodin olla vuorovaikutuksessa käyttäjän kanssa. Oliot voivat hallita selainta ja muuttaa dokumentin sisältöä, joka ilmestyy selainikkunassa. Tämä JavaScript-kielen sulautettu versio suorittaa koodia sulautetusti HTML-verkkosivuissa. Tätä on yleisesti kutsuttu asiakaspuolen JavaScript-kieleksi painottaen, että koodi suoritetaan asiakaskoneella eikä verkkopalvelimella. (Flanagan 2006, 1.)

JavaScript-kieli ja sen sisäänrakennetut tietotyypit ovat kansainvälisten standardien alaisia. Osa asiakaspuolen JavaScript-kielestä on muodollisesti standardisoitu, osa kielestä on käytännössä standardinmukaista ja osa koostuu selainkohtaisista laajennuksista. JavaScript-ohjelmoijien pitää ottaa huomioon selainten välinen yhteensopivuus. (Flanagan 2006, 1.)

# 7 Toteutus

Tavoitteena oli luoda tietokoneshakki JavaScript-kielellä selainympäristöön. Tarkoitus oli käyttää tavanomaisia tekniikoita teorian pohjalta yhdessä JavaScript-kielen kanssa. Toteutuskieleksi valittiin JavaScript, sillä se oli valmiiksi sulautettu HTML-kielen kanssa.

Ensin piti pohtia muutamia kysymyksiä. Kuinka shakkitietokone luotaisiin käyttäen tavanomaisia tekniikoita? Kuinka shakkiohjelma saataisiin sääntöjen rajoittamana laskemaan ja tulkitsemaan siirtoja? Kuinka esimerkiksi arviointifunktiot, minmax-algoritmi ja alpha-beta-karsinta saataisiin järkevästi toimivaksi koodissa? Miten haluttaisiin esittää laudan? Mitä arviointikriteereitä käytettäisiin arviointifunktiossa?

Oli oletettavaa, että muuttujien, ehtojen, toistojen, taulukoiden ja funktioiden monipuolisella käsittelyllä päästäisiin melko pitkälle, sillä shakkitietokoneohjelman ohjelmoinnin painopiste olisi pelilogiikan toteuttamisessa. Oletettiin, että tietokoneshakin olisi simuloitava siirtoja ennen varsinaista päätöksen tekoa, koska tietokoneen olisi kokeilun kautta huomattava siirtojen laillisuus ja arvo.

Ohjelmaa oli tarkoitus rakentaa pala palalta. Ensin toteutettaisiin yksinkertaisempia kokonaisuuksia ja viimeiseksi pohdittaisiin monimutkaisia aiheita. Toteutusjärjestys oli mietittävä. Ajateltiin, että ensin toteutettaisiin käyttöliittymä, seuraavaksi säännöt ja viimeiseksi tekoäly.

Oli mietittävä koodin rakennetta siten, että jos jotain jouduttaisiin tulevaisuudessa muuttamaan, ei koko koodi menisi rikki. Mietittiin, miten koodi olisi hyvä jaotella eri tiedostoihin. Oli mietittävä, mitä työkaluja käyttäisi. Oletettiin, että Notepad++-tekstieditori helpottaisi ohjelmointia. Oli tiedostettava valtaselaimet, joissa ohjelman olisi hyvä toimia.

## 7.1 Pelikoodi

Tässä esitellään JavaScript-dokumentti (game.js), joka vastaa käyttöliittymän, pelisääntöjen ja tekoälyn yhdistämisestä. Ensin määritellään päämuuttujat. Muuttuja n tarkoittaa laudan sivun pituutta, mainBoardLog tulee sisältämään siirtohistorian, ja mainLegalPositions tulee sisältämään kaikki sallitut siirrot. Muuttuja promotion tulee sisältämään tiedon, miksi nappulaksi sotilas korotetaan.

```
n=8;
mainBoardLog=[];
mainLegalPositions=[];
```

```
promotion;
```

CreateStartPosition-funktio asettaa nappulat alkuasetelmaan laudalle. Lautaa esittää tavallinen yksiulotteinen taulukko, johon voi sijoittaa 64 arvoa. Kun nappulat ja tyhjät ruudut on saatu asetettua siirtohistoriaan, etsitään vielä lailliset siirrot. Siirtohistoriaa ja tietoa pelivuorosta käytetään avuksi laillisten siirtojen etsimisessä. Lopuksi tulostetaan lauta pelaajan nähtäväksi.

```
function createStartPosition(){
// Asetetaan lauta.
board=[];
for(i=0;i<n*n;i++){
board[i]=0;
}
board[0]=-4;
board[1]=-2;
board[2]=-3;
board[3]=-5;
board[4]=-6;
board[5]=-3;
board[6]=-2;
board[7]=-4;
for (i=n;i<n*2;i++){
board[i]=-1;
}
for (i=n*6;i<n*7;i++){
board[i]=1;
}
board[n*7]=4;
board[n*7+1]=2;
board[n*7+2]=3;
board[n*7+3]=5;
board[n*7+4]=6;
board[n*7+5]=3;
board[n*7+6]=2;
board[n*7+7]=4;
mainBoardLog[mainBoardLog.length]=board.slice(0);
// Asetetaan säännöt.
boardLog=mainBoardLog.slice(0);
color=getColor();
legalPositions=[];
findLegalPositions();
mainLegalPositions=legalPositions.slice(0);
// Tulostetaan lauta.
printBoard();
}
```

PrintBoard-funktio tulostaa laudan. Laudan asemaksi määräytyy siirtohistorian viimeinen asema. Laudan lisäksi tulostetaan korotusvalikko. Asetetaan oletukseksi daami sekä graafisesti että ohjelmallisesti. Korotusvalikossa olevat nappulat vaihtavat väriä vuoron mukaan.

SelectSquare-funktio vastaa ruudun valinnasta. Lautaan ei voi koskea, jos tekoäly on päällä. Jos laudalla on entuudestaan valittu ruutu ja valitaan uusi ruutu, testataan, onko siirtoyritys laillinen. Jos siirtoyritys on laillinen, nollataan varmuudenvuoksi tekoäly, lisätään siirto siirtohistoriaan, etsitään seuraavat lailliset siirrot ja tulostetaan uusi lauta. Lisäksi katsotaan, jatkuuko peli vai onko peli matti tai patti. Jos siirtoyritys ei ollut laillinen, valinta vain poistuu. Jos entuudestaan ei ollut valittua ruutua tai siirto oli laiton, valitaan napsautettu ruutu, jos siinä on oma nappula.

SelectPromotion-funktio vastaa siitä, miksi nappulaksi sotilas korotetaan. Korotusvalikkoon ei voi koskea, jos tekoäly on päällä. Valikossa napsautettu ruutu valitaan ja valintaa vastaava arvo talletetaan sille tarkoitettuun muuttujaan.

CpuSelect-funktio luo tekoälyvalikon. JavaScript-kielessä on verkkotyöläinen (web worker), joka kykenee suorittamaan koodia taustalla vaikuttamatta itse sivun toimintaan. Valikko luodaan, jos selain tukee verkkotyöläistä. Oletuksena tekoäly pelaa mustilla nappuloilla.

SelectCpu-funktio vastaa tekoälyvalikon valinnoista. Joko tekoäly ei pelaa, pelaa toisilla nappuloilla tai pelaa molemmilla nappuloilla.

Ohjelma tarkistaa seuraavan koodin avulla sekunnin välein, onko tekoäly valittu. Jos on ja peli ei ole ohi, luodaan uusi verkkotyöläinen, jos sellaista ei ole alun perin luotu. Työläinen käyttää koodia, joka vastaa tekoälystä. Kyseinen työläinen lähettää viestinä kyseiselle koodille laudan sivun pituuden, siirtohistorian ja sallitut siirrot. Jäädään odottamaan koodin tuottamaa siirtoa. Kun siirto saapuu paluuviestinä, työläinen nollataan ja suoritetaan siirtotoimenpiteet tavallisesti.

### 7.2 Sääntökoodi

Tässä esitellään JavaScript-dokumentti (rules.js), joka vastaa sääntöjen oikeudenmukaisuudesta. GetColor-funktio vastaa siitä, kumman vuoro on. Funktio palauttaa positiivisen ykkösen, jos on valkean vuoro. Funktio palauttaa negatiivisen ykkösen, jos on mustan vuoro. Siirtovuoro määräytyy siirtohistorian mukaan.

```
function getColor(){
if((mainBoardLog.length-1)%2==0){
return 1;
}
return -1;
```

```
}
```

FindLegalPositions-funktio etsii lailliset siirrot. Tämä on pääfunktio, jossa on alifunktioina eri nappuloiden siirtotestausfunktiot. Etsitään laudalta vuorossa olevan pelaajan nappulat ja testataan siirrot.

```
function findLegalPositions(){
for(i=0;i<n*n;i++){
if(board[i]==color){
tryMovesPawn();
}else if(board[i]==2*color){
tryMovesKnight();
}else if(board[i]==3*color){
tryMovesBishop();
}else if(board[i]==4*color){
tryMovesRook();
}else if(board[i]==5*color){
tryMovesBishop();
tryMovesRook();
}
}
for(i=0;i<n*n;i++){
if(board[i]==6*color){
tryMovesKing();
}
}
}
```

Siirtotestausfunktiot testaavat kaikki mahdolliset kyseisen nappulan siirrot. Funktioissa testataan, meneekö nappula laudan ulkopuolelle. Lisäksi funktioissa testataan, onko kohderuudussa vuorossa olevan pelaajan nappula. Jos nappula pysyy laudan sisäpuolella eikä syö samanväristä nappulaa, testataan siirtoa. Kaikissa näissä funktioissa käytetään apuna kopiolautaa siirtojen määrittelyssä, koska ei haluta sotkea alkuperäistä tilannetta. Kopiolaudassa yleisesti kohderuutu saa arvoksi lähtöruudun arvon ja lähtöruutu muutetaan nollaksi. Lopuksi testataan, onko vuorossa olevan pelaajan kuningas jäänyt syötäväksi. Jos ei ole, siirto lisätään laillisten siirtojen joukkoon.

Vaikka lauta on tavallinen yksiulotteinen taulukko, siirrot tapahtuvat laudalla kaksiulotteiseen tyyliin. Laudan sivun pituuden avulla tiedetään, mihin riviin siirrytään. Toisin sanoen alas mentäessä kasvatetaan lähtöarvoa käyttäen sivun pituutta kertoimena. Ylös mentäessä vähennetään lähtöarvoa käyttäen sivun pituutta kertoimena. Oikealle mentäessä kasvatetaan lähtöarvoa ja vasemmalle mentäessä vähennetään lähtöarvoa. Jos nappula lentää ulos laudalta yläkautta, on arvo negatiivinen. Jos nappula lentää ulos laudalta alakautta, on arvo suurempi tai yhtä suuri kuin sivun pituuden neliö. Jakojäännöksen avulla voidaan selvittää, hyppääkö nappula laudalta pois oikealta tai vasemmalta puolelta.

IsCheck-funktio tarkistaa, onko oma kuningas shakissa. Otetaan ylös vastustajan hallitsemat ruudut kontrollilautaan. Etsitään oma kuningas kopiolaudalta. Jos kopiolaudan ruudussa on oma kuningas ja vastaavassa ruudussa kontrollilaudalla on suurempi arvo kuin nolla, on tilanne shakki.

```
function isCheck(){
controlBoard=[];
color=-color;
setControl(1,1,1,1,1,1);
color=-color;
for(k=0;k<n*n;k++){
if(copyBoard[k]==6*color){
if(controlBoard[k]>0){
return true;
}
break;
}
}
return false;
}
```

SetControl-funktio asettaa arvot kontrollilautaan. Tämä on funktio, jossa on alifunktioina eri nappuloiden hallinta-arvoja asettavat funktiot. Kontrollilaudassa ruudussa oleva luku kertoo, kuinka moni tietyn värinen nappula hallitsee kyseistä ruutua. Eri nappuloille voi antaa lisäksi eri hallinta-arvot tarpeen mukaan. Ensin kontrollilauta alustetaan nollilla. Etsitään kopiolaudalta tietyn väriset nappulat ja asetetaan hallinta-arvot ruutuihin.

```
function setControl(pv,nv,bv,rv,qv,kv){
for(k=0;k<n*n;k++){
controlBoard[k]=0;
}
for(k=0;k<n*n;k++){
if(copyBoard[k]==color){
setControlPawn(pv);
}else if(copyBoard[k]==2*color){
setControlKnight(nv);
}else if(copyBoard[k]==3*color){
setControlBishop(bv);
}else if(copyBoard[k]==4*color){
setControlRook(rv);
}else if(copyBoard[k]==5*color){
setControlBishop(qv);
setControlRook(qv);
}else if(copyBoard[k]==6*color){
setControlKing(kv);
}
}
}
```

### 7.3 Tekoälykoodi

Tässä esitellään JavaScript-dokumentti (cpu.js), joka vastaa tekoälyn toiminnasta. Tekoäly käyttää hyödyksi valmiiksi määriteltyjä sääntöjä. Otetaan vastaan viesti pelin perustietoineen ja etsitään mahdollisimman hyvä siirto.

```
importScripts('rules.js');
onmessage = function(e){
n=e.data[0];
mainBoardLog=e.data[1].slice(0);
mainLegalPositions=e.data[2].slice(0);
findBestMove();
};
```

Arvotetaan ruudut arviointifunktiota varten valmiiksi. Mitä keskemmällä lautaa ollaan, sen arvokkaampi ruutu on. Ruutujen arvot ovat seuraavat:

1, 2, 3, 4, 4, 3, 2, 1,

2, 4, 6, 8, 8, 6, 4, 2,

3, 6, 9, 12, 12, 9, 6, 3,

4, 8, 12, 16, 16, 12, 8, 4,

4, 8, 12, 16, 16, 12, 8, 4,

3, 6, 9, 12, 12, 9, 6, 3,

2, 4, 6, 8, 8, 6, 4, 2,

1, 2, 3, 4, 4, 3, 2 ja 1.

#### 7.3.1 Pelipuu

FindBestMove-funktio etsii niin hyvän siirron kuin pystyy. Luodaan pelipuu tiettyyn hakusyvyyteen asti. Samalla arvioidaan asemat. Valmis pelipuu analysoidaan, jolloin saadaan parhaat siirrot selville. Lopuksi parhaista siirroista arvotaan yksi siirroksi.

Taulukkoon sijoitetaan kaikki mahdolliset pelijatkot. Jokaisella jatkolla on yksilöivä avain ja peliarvo. Hakusyvyydeksi on asetettu tässä tapauksessa kolme puolisiirtoa. Hakuaika on kaksi sekuntia ja aluksi ei anneta ylimääräistä aikaa.

```
mainlines=[];
keys=[];
values=[];
hm=3;
time=2000;
extraTime=false;
```

Luodaan pohja jatkoille asettamalla ensimmäiset siirrot taulukkoon. Avaimet määräytyvät silmukan indeksin mukaan. Jatkon alku arvioidaan.

```
for(m=0;m<mainLegalPositions.length;m++){
mainline=[];
mainline[0]=mainLegalPositions[m].slice(0);
mainlines[mainlines.length]=mainline.slice(0);
keys[keys.length]=''+m;
valuePosition();
}
```

Seuraavaksi rakennetaan itse pelipuu. Otetaan aluksi muistiin aloitusaika. Itse pelipuukoodissa johdetaan taulukon ensimmäisestä jatkosta kaikki lailliset siirrot. Laillisten siirtojen perusteella asetetaan uudet pidemmälle luodut jatkot taulukon perälle. Näille jatkoille määritetään avain käyttäen pohjana alkuperäisen jatkon avainta. Uusille jatkoille annetaan lisäksi arvo. Täten alkuperäinen jatko poistetaan taulukon alusta, kun siitä on johdettu kaikki jatkot. Samoin alkuperäisen jatkon avain ja arvo poistetaan. Tätä toistetaan niin kauan, kunnes kaikki jatkot ovat yhtä pitkiä. Toisaalta jos jatkoa ei pysty pidentämään, on tilanne joko matti tai patti. Jos tekoäly on tekemässä mattia, annetaan arvoksi maksimaalinen arvo, josta vähennetään hakusyvyys, jolloin aikaisempi matti on arvokkaampi. Jos tekoäly on joutumassa mattiin, arvo äskeiseen verrattuna on vastakkainen. Pattitilanne on arvoltaan nolla. Jatko, avain ja arvo siirretään taulukon perälle. Kun kaikki jatkot on saatu käsiteltyä, aloitetaan sama jatkojen käsittely uudestaan, jos haluttua hakusyvyyttä ei ole saavutettu. Jos tavoiteltu hakusyvyys on saavutettu ja aikaa on vielä jäljellä, jatkoja yritetään vielä pidentää niin paljon kuin ehtii.

AnalyzeGameTree-funktio toteuttaa minmax-algoritmin ideaa. Analysoidaan juuri rakennettu pelipuu avainten avulla. Tällä kertaa pelipuun kasvattamisen sijaan kutistetaan avainten määrää vaihe vaiheelta latvasta lähtien. Otetaan käsittelyyn ensimmäinen avain, josta poistetaan viimeinen luku. Jos tämä yhdellä luvulla lyhennetty avain on sama kuin viimeinen avain, viittaavat molemmat avaimet samaan asemaan. Tällöin arvoja voidaan verrata keskenään. Jos käsiteltävä hakusyvyys on parillinen, täytyy arvoa minimoida mahdollisuuden mukaan. Jos käsiteltävä hakusyvyys on pariton, täytyy arvoa maksimoida mahdollisuuden mukaan. Jos arvoa voidaan päivittää, korvataan viimeistä avainta vastaava arvo uudella arvolla. Ensimmäinen avain ja arvo poistetaan. Toisaalta jos tämä yhdellä luvulla lyhennetty avain on eri kuin viimeinen avain, siirretään tämä ensimmäinen avain ja sen arvo viimeiseksi. Tätä kiertoa jatketaan niin kauan, kunnes kaikki avaimet, jotka on voitu lyhentää, on lyhennetty yhdellä luvulla. Tätä avainten lyhentämisprosessia jatketaan niin kauan, kunnes kaikissa avaimissa on vain yksi luku.

```
function analyzeGameTree(){
for(h=hm;h>1;h--){
g=keys.length;
for(i=0;i<g;i++){
key=keys[0].split(' ');
if(key.length==h){
// Avaimesta poistetaan viimeinen luku.
key.splice(key.length-1,1);
key=key.join(' ');
if(key==keys[keys.length-1]){
if(h%2==0){
// Minimoidaan arvo.
if(values[values.length-1]>values[0]){
values[values.length-1]=values[0];
}
}else{
// Maksimoidaan arvo.
if(values[values.length-1]<values[0]){
values[values.length-1]=values[0];
}
}
}else{
// Ensimmäinen avain lisätään loppuun.
keys[keys.length]=key;
values[values.length]=values[0];
}
// Ensimmäinen avain poistetaan.
keys.splice(0,1);
values.splice(0,1);
}else{
// Avain siirretään perälle ilman käsittelyä.
keys[keys.length]=keys[0];
values[values.length]=values[0];
keys.splice(0,1);
values.splice(0,1);
}
}
}
}
```

Nyt tiedetään jokaisen laillisen siirron arvo. Parhaista siirroista arvotaan yksi varsinaiseksi siirroksi. Siirto lähetetään tekoälyn siirrettäväksi.

```
bestMoves=[];
for(m=0;m<values.length;m++){
if(values[m]==Math.max.apply(Math,values)){
bestMoves[bestMoves.length]=mainLegalPositions[m].slice(0);
}
}
postMessage(bestMoves[Math.floor(Math.random()*best-
Moves.length)].slice(0));
```

### 7.3.2 Arviointifunktio

Seuraavaksi esitellään arviointifunktio. Funktio arvioi jatkon viimeisen aseman perusteella.

Ensin summataan materiaali yhteen. Omasta nappulasta yhteisarvo kasvaa ja vastustajan nappulasta yhteisarvo vähenee. Sotilaan arvo on 1, ratsun 3, lähetin 3, tornin 5 ja daamin 9.

```
materialValue=0;
for(j=0;j<n*n;j++){
if(board[j]==getColor()){
// Oma sotilas löytyi.
materialValue+=1;
}else if(board[j]==-getColor()){
// Vastustajan sotilas löytyi.
materialValue-=1;
}else if(board[j]==2*getColor()){
// Oma ratsu löytyi.
materialValue+=3;
}else if(board[j]==2*-getColor()){
// Vastustajan ratsu löytyi.
materialValue-=3;
}else if(board[j]==3*getColor()){
// Oma lähetti löytyi.
materialValue+=3;
}else if(board[j]==3*-getColor()){
// Vastustajan lähetti löytyi.
materialValue-=3;
}else if(board[j]==4*getColor()){
// Oma torni löytyi.
materialValue+=5;
}else if(board[j]==4*-getColor()){
// Vastustajan torni löytyi.
materialValue-=5;
}else if(board[j]==5*getColor()){
// Oma daami löytyi.
materialValue+=9;
}else if(board[j]==5*-getColor()){
// Vastustajan daami löytyi.
materialValue-=9;
}
}
```

Seuraavaksi luodaan oma kontrollilauta. Omassa kontrollilaudassa ruudussa oleva luku kertoo, kuinka moni oma nappula hallitsee kyseistä ruutua. Eri nappuloille on annettu lisäksi eri hallinta-arvot. Toisin sanoen nyt sotilas lisää ruutuun 9, ratsu 7, lähetti 7, torni 5, daami 1 ja kuningas 6 pistettä. Vaikka kuningas on äärettömän arvokas, sen hallinta-arvo on tornin ja kevyen upseerin välissä (lähetti ja ratsu). Idea on se, että mitä heikompi nappula saadaan hyötykäytettyä, sitä parempi asema on. Ainakin tämä estää esimerkiksi turhan aktiivisen daamin käytön. Esimerkiksi avauksessa daamia ei kannata kehittää

yleensä avausteorian mukaan. Kontrollilauta kuvaa liikkuvuutta, kehitystä ja keskustan hallintaa. Lauta kuvaa liikkuvuutta, koska nappula hallitsee melkein samalla tavalla kuin liikkuu. Mitä enemmän nappulat hallitsevat, sitä kehittyneempiä ne todennäköisesti ovat. Mitä keskemmällä nappula on lautaa, sitä enemmän se hallitsee.

```
color=getColor();
copyBoard=board.slice(0);
controlBoard=[];
setControl(9,7,7,5,1,6);
ownControlBoard=controlBoard.slice(0);
```

Täytyy luoda lisäksi vastustajan kontrollilauta. Nyt katsotaan, kuinka paljon vastustaja hallitsee.

```
color=-color;
copyBoard=board.slice(0);
controlBoard=[];
setControl(9,7,7,5,1,6);
opponentControlBoard=controlBoard.slice(0);
```

Luodaan oma kuningaslauta. Mitä lähempänä ruutu on omaa kuningasta, sitä suurempi luku ruudussa on. Jos kuningas olisi laudan nurkassa, olisi kuningaslauta seuraavan näköinen:
16, 14, 12, 10, 8, 6, 4, 2,
14, 14, 12, 10, 8, 6, 4, 2,
12, 12, 12, 10, 8, 6, 4, 2,
10, 10, 10, 10, 8, 6, 4, 2,
8, 8, 8, 8, 8, 6, 4, 2,
6, 6, 6, 6, 6, 6, 4, 2,
4, 4, 4, 4, 4, 4, 4, 2,
2, 2, 2, 2, 2, 2, 2 ja 2.

Tehdään kuningaslauta vastustajalle. Nyt lauta määräytyy vastustajan kuninkaan sijainnin perusteella.

Lopuksi lasketaan kontrollilautojen ruutujen arvot yhteen. Oman laudan arvot kasvattavat yhteisarvoa ja vastustajan vähentävät sitä. Lisäksi ruudun arvolla on kertoimia. Ruudun arvo kertautuu sen mukaan, mitä lähempänä ruutu on keskustaa ja vastustajan kuningasta.

```
controlValue=0;
for(j=0;j<n*n;j++){
```

```
controlValue+=ownControlBoard[j]*squareValues[j]*opponentKing-
Board[j];
controlValue-=opponentControlBoard[j]*squareValues[j]*ownKing-
Board[j];
}
```

Seuraava koodi kannustaa sotilaita etenemään ehkä mahdollista korotusta varten. Mitä lähempänä oma sotilas on laudan vastakkaista riviä, sitä enemmän pisteitä. Jos vastustaja on lähellä korotusta, saadaan miinuspisteitä.

```
pawnPromotion=0;
for (j=1;j<n-1;j++){
for (k=n*j;k<n*(j+1);k++){
if(board[k]==1){
if(getColor()==1){
pawnPromotion+=n-2-j;
}else{
pawnPromotion-=n-2-j;
}
}else if(board[k]==-1){
if(getColor()==1){
pawnPromotion-=j-1;
}else{
pawnPromotion+=j-1;
}
}
}
}
```

Viimeiseksi asetetaan jatkolle varsinainen kokonaisarvo, joka on tässä tapauksessa materiaalin, hallinnan ja korotusmahdollisuuden summa. Hallinnalle ja korotusmahdollisuudelle on yritetty antaa sopiva kerroin, jotta ne olisivat hyvässä suhteessa materiaaliin. Lopuksi pyöristetään arvo yhden desimaalin tarkkuudelle, jotta samanarvoisia siirtoja olisi vaihtoehtona enemmän.

```
values[values.length]=Math.round((materialValue+controlValue/10000+pawnPromo-
tion/20)*10)/10;
```

### 7.4 Yhteenveto

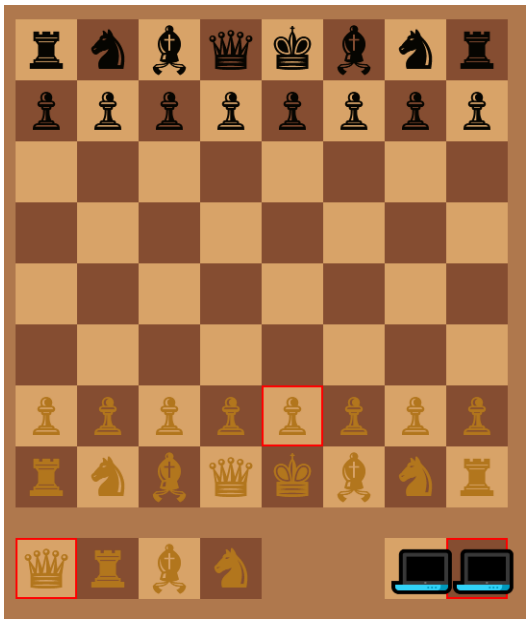
Shakkiohjelmasta tuli toimiva kokonaisuus. Shakin suurmestaria siitä ei tullut, mutta se noudatti sääntöjä ja teki järkevän näköisiä siirtoja kohtuullisessa ajassa. Ohjelma ainakin osasi tehdä yhden siirron matin ja torjua sellaisen. Avaussiirrot näyttivät sujuvan avausteorian kaltaisesti yllättävän usein. Tekijäänsä parempaa pelaajaa siitä ei tullut.

Oltaisiin ehkä voitu tehdä pelipuun käsittelystä kevyempi alpha-beta-karsinnan avulla. Tyydyttiin joka tapauksessa perinteiseen minmax-algoritmiin. Keksittiin, että kontrollilautoja voi käyttää hyödyksi aseman arvioinnissa.

Sääntökoodista tuli yllättävän pitkä. Olisi varmasti ollut yksinkertaisempia tapoja toteuttaa säännöt. Toisaalta idea oli ymmärrettävissä. Tasapelisäännöt jäivät toteuttamatta, mutta onneksi ne eivät ole tärkeitä pelin sujuvuuden kannalta. Näkyvä peliloki olisi ollut lisäksi järkevää toteuttaa, mutta sekään ei kuulu tietokoneshakkiohjelmoinnin tärkeimpiin osa-alueisiin.

Onnistuttiin yhdistämään ohjelman eri osa-alueet järkevästi toimivaksi kokonaisuudeksi. Käyttöliittymä toimi hyvin tekoälyn ja sääntökoodin kanssa. Käyttöliittymästä tuli yksinkertainen mutta selkeä. Käyttöliittymän luomiseen käytettiin apuna CSS-kieltä. HTML-kielellä tehty indeksitiedosto toimi pohjana koko koodille. Ymmärrettiin käyttää verkkotyöläistä siirtojen miettimiseen, sillä muuten selain olisi voinut jäätyä.

Saatiin tehtyä kuvion 22 mukainen käyttöliittymä. Nappulasymbolit saatiin laudalle Unicode-merkkejä käyttäen.



Kuvio 22. Tietokoneshakki selaimessa

Ohjelma toimi parhaiten Mozilla Firefox -selaimella. Firefox oli nopeampi kuin Google Chrome. Chrome-selaimella ohjelma ei aluksi toiminut. Ratkaisu oli --allow-file-access-from-files -komento selaimen käynnistyksen yhteydessä. Verkkotyöläinen siis käytti paikallista tiedostoa apuna. Palvelimella Chrome toimi ongelmitta, vaikka oli jostain syystä

melko hidas. Ohjelma toimi lisäksi Microsoft Edge -selaimessa. Ohjelmaa testattiin pelaamalla tietokonetta vastaan ja kokeilemalla erilaisia asemia. Poikkeuksellista toimintaa ei ole havaittu.

# 8 Pohdinta

Projektin aikana toteutettiin tietokoneshakkiohjelma, joka toimii valtaselaimilla JavaScript-kielellä. Tietokoneshakkiohjelmaan kuului pelikoodi, joka vastasi käyttöliittymän, pelisääntöjen ja tekoälyn yhdistämisestä. Sääntökoodi vastasi sääntöjen oikeudenmukaisuudesta. Tekoälykoodi laski siirtoja käyttäen minmax-algoritmia ja arviointifunktiota. Käyttöliittymästä tuli selkeä. Sillä pystyi pelaamaan säännönmukaisia siirtoja. Tekoälystä ei tullut huipputasoista, mutta harjoitusvastustajaksi aloitteleville pelaajille se kelpaisi.

Alpha-beta-karsinta olisi ollut oleellinen lisäys minmax-algoritmin tueksi. Sillä olisi tekoälyn laskeminen saatu nopeammaksi. Itse asiassa sellainen tehtiinkin, mutta sitä ei ehditty raportoida. Kyseisellä karsinnalla oltaisiin päästy pelipuussa yksi siirto syvemmälle samassa ajassa. Jopa tasapelisäännöt, siirtopöytäkirja ja syötyjen nappuloiden näyttö toteutettiin, mutta niitä ei ehditty raportoida.

Tietokoneshakin ohjelmointi opetti, että vahvan tekoälyn luominen ei olekaan niin yksinkertaista. Vaadittaisiin paljon enemmän optimointia ja soveltamista, jos haluttaisiin luoda haastava tekoäly. Mahdollisesti on nopeampia tapoja tuottaa siirrot. Arviointifunktiota voisi kehittää tarkemmaksi ja monipuolisemmaksi. Soveltavilla tekniikoilla olisi siirtojen hakua voinut ohjata optimaalisempaan suuntaan. Mahdollisesti JavaScript ei ole paras vaihtoehto, jos halutaan mahdollisimman tehokas tekoäly. Tehokkuus vaihteli eri selaimilla, joten koodin suoritusnopeus on paljon riippuvainen selaimesta.

Nykyiset shakinpeluualgoritmit ovat vielä kehitysvaiheessa ja niitä kehitetään edelleen. Lisäksi koko ajan yritetään keksiä uusia keinoja pelin laadun parantamiseksi. Shakkitietokoneista yritetään tehdä entistä inhimillisempiä, jolloin peli ei perustuisi yhtä paljon kaavoihin ja laskemiseen. Toisaalta herää kysymys, onko shakkia tarkoitus pelata inhimillisesti. Onko inhimillisyys mahdollisuus vai rajoite? Jos shakkipelistä pystyisi rakentamaan täydellisen pelipuun, täytyisi pelin merkitys miettiä uudelleen. Jos kuitenkin täydellinen pelipuu saataisiin valmiiksi, olisi tekniikka silloin niin kehittynyt, että maailmassa pystyttäisiin ratkaisemaan muitakin haastavia ongelmia. Kasvomme edessä ovat kaikki ratkaisut. Ne pitää vain etsiä.

## Lähteet

Flanagan, D. 2006. JavaScript: The Definitive Guide. 5. painos. O'Reilly Media, Inc. Yhdysvallat. Luettavissa: https://books.google.fi/books?hl=fi&lr=&id=k0CbAgAAQBAJ&oi=fnd&pg=PT6&dq=javascript&ots=O3mrepiypR&sig=ceoWfTGOveiwQCGDSQaAgQ64AA&redir_esc=y#v=onepage&q=javascript&f=false. Luettu: 4.11.2017.

Myyrä, T. 2011. Tietokoneshakki funktionaalisella ohjelmoinnilla. Amk-opinnäytetyö. Metropolia Ammattikorkeakoulu. Luettavissa: https://publications.theseus.fi/bitstream/handle/10024/29841/tietokoneshakki.pdf?sequence=2. Luettu: 9.11.2017.

Ranta-aho, R. 2001. Tietokoneshakki. TKK, Tietoliikenneohjelmistojen ja multimedian laboratorio. Luettavissa: http://www.tml.tkk.fi/Opinnot/Tik-111.590/2001/paperit/ranta-aho.pdf. Luettu: 9.11.2017.

Seppälä, A. 2004. Tekoälyn ohjelmointi shakkipelille. Tietotekniikan Pro gradu -tutkielma. Jyväskylän yliopisto. Jyväskylä. Luettavissa: https://jyx.jyu.fi/dspace/bitstream/handle/123456789/12488/URN_NBN_fi_jyu-20054.pdf. Luettu: 9.11.2017.

Shannon, C. 1950. Programming a computer for playing chess. Philosophical Magazine. Vol. 41.

Suh, E. 2005. Introduction to Chess Board Representation. AI Horizon -verkkosivusto. Luettavissa: http://www.aihorizon.com/essays/chessai/boardrep.htm. Luettu: 6.11.2017

Timonen, K. 2008. Tekoäly kortti- ja lautapeleissä. Helsingin yliopisto. Helsinki. Luettavissa: http://docplayer.fi/9758857-Tietokoneshakki-kari-timonen-kari-timonen-cs-helsinki-fi.html. Luettu: 9.11.2017.

Vainio, V. 2005. Shakin säännöt. Hannunniitun shakkikerho. Turku. Luettavissa: http://www.shakki.net/koulut/hannunniittu/shakkikoulu/shakin_saannot.pdf. Luettu: 6.11.2017.

# Liitteet

## Liite 1. Indeksitiedosto

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8"
/>
<title>Chess</title>
<link rel="stylesheet" href="style.css" type="text/css"  />
</head>
<body onload="createStartPosition();cpuSelect()">
<div id="board"></div>
<div id="option">
<div id="promotion"></div>
<div id="cpu"></div>
</div>
</body>
<script type="text/javascript" src="game.js"></script>
<script type="text/javascript" src="rules.js"></script>
</html>
```

**Liite 2. Pelikoodi**

```
/*
Tässä esitellään JavaScript-dokumentti (game.js), joka vastaa
käyttöliittymän, pelisääntöjen ja tekoälyn yhdistämisestä.
Ensin määritellään päämuuttujat.
Muuttuja n tarkoittaa laudan sivun pituutta, mainBoardLog tulee
sisältämään siirtohistorian, ja mainLegalPositions tulee sisältä-
mään kaikki sallitut siirrot.
Muuttuja promotion tulee sisältämään tiedon, miksi nappulaksi so-
tilas korotetaan.
*/
n=8;
mainBoardLog=[];
mainLegalPositions=[];
promotion;
/*
CreateStartPosition-funktio asettaa nappulat alkuasetelmaan lau-
dalle.
Lautaa esittää tavallinen yksiulotteinen taulukko, johon voi si-
joittaa 64 arvoa.
Kun nappulat ja tyhjät ruudut on saatu asetettua siirtohistoriaan,
etsitään vielä lailliset siirrot.
Siirtohistoriaa ja tietoa pelivuorosta käytetään avuksi laillisten
siirtojen etsimisessä.
Lopuksi tulostetaan lauta pelaajan nähtäväksi.
*/
function createStartPosition(){
// Asetetaan lauta.
board=[];
for(i=0;i<n*n;i++){
board[i]=0;
}
board[0]=-4;
board[1]=-2;
board[2]=-3;
board[3]=-5;
board[4]=-6;
board[5]=-3;
board[6]=-2;
board[7]=-4;
for (i=n;i<n*2;i++){
board[i]=-1;
}
for (i=n*6;i<n*7;i++){
board[i]=1;
}
board[n*7]=4;
board[n*7+1]=2;
board[n*7+2]=3;
board[n*7+3]=5;
board[n*7+4]=6;
board[n*7+5]=3;
board[n*7+6]=2;
board[n*7+7]=4;
```

```
mainBoardLog[mainBoardLog.length]=board.slice(0);
// Asetetaan säännöt.
boardLog=mainBoardLog.slice(0);
color=getColor();
legalPositions=[];
findLegalPositions();
mainLegalPositions=legalPositions.slice(0);
// Tulostetaan lauta.
printBoard();
}
/*
PrintBoard-funktio tulostaa laudan.
Laudan asemaksi määräytyy siirtohistorian viimeinen asema.
Laudan lisäksi tulostetaan korotusvalikko.
Asetetaan oletukseksi daami sekä graafisesti että ohjelmallisesti.
Korotusvalikossa olevat nappulat vaihtavat väriä vuoron mukaan.
*/
function printBoard(){
board=mainBoardLog[mainBoardLog.length-1].slice(0);
// Luodaan lauta.
table='<table>';
for (i=0;i<n;i++){
table+='<tr>';
for (j=0;j<n;j++){
// Määrää, minkä värinen ruutu on.
if((i%2==0&&j%2==0)||(i%2!=0&&j%2!=0)){
table+='<td onclick="selectSquare(this.id)" class="light"
id="'+(n*i+j)+'">';
} else{
table+='<td onclick="selectSquare(this.id)" class="dark"
id="'+(n*i+j)+'">';
}
// Määrää, minkä värinen nappula on.
if(board[n*i+j]>0){
table+='<span class="white">';
}else if(board[n*i+j]<0){
table+='<span class="black">';
}
// Määrää, mikä nappula on.
if(Math.abs(board[n*i+j])==1){
table+='&#9823;';
}else if(Math.abs(board[n*i+j])==2){
table+='&#9822;';
}else if(Math.abs(board[n*i+j])==3){
table+='&#9821;';
}else if(Math.abs(board[n*i+j])==4){
table+='&#9820;';
}else if(Math.abs(board[n*i+j])==5){
table+='&#9819;';
}else if(Math.abs(board[n*i+j])==6){
table+='&#9818;';
}
table+='</span>';
table+='</td>';
}
table+='</tr>';
}
```

```
table+='</table>';
document.getElementById('board').innerHTML=table;
// Luodaan korotusvalikko.
promotion=5;
table='<table>';
table+='<tr>';
table+='<td onclick="selectPromotion(this.id)" class="light high-
light" id="p5">';
if(getColor()==1){
table+='<span class="white">';
}else{
table+='<span class="black">';
}
table+='&#9819;';
table+='</span>';
table+='</td>';
table+='<td onclick="selectPromotion(this.id)" class="dark"
id="p4">';
if(getColor()==1){
table+='<span class="white">';
}else{
table+='<span class="black">';
}
table+='&#9820;';
table+='</span>';
table+='</td>';
table+='<td onclick="selectPromotion(this.id)" class="light"
id="p3">';
if(getColor()==1){
table+='<span class="white">';
}else{
table+='<span class="black">';
}
table+='&#9821;';
table+='</span>';
table+='</td>';
table+='<td onclick="selectPromotion(this.id)" class="dark"
id="p2">';
if(getColor()==1){
table+='<span class="white">';
}else{
table+='<span class="black">';
}
table+='&#9822;';
table+='</span>';
table+='</td>';
table+='</tr>';
table+='</table>';
// Tulostetaan lauta.
document.getElementById('promotion').innerHTML=table;
}
/*
SelectSquare-funktio vastaa ruudun valinnasta.
Lautaan ei voi koskea, jos tekoäly on päällä.
Jos laudalla on entuudestaan valittu ruutu ja valitaan uusi ruutu,
testataan, onko siirtoyritys laillinen.
```

```
Jos siirtoyritys on laillinen, nollataan varmuudenvuoksi tekoäly,
lisätään siirto siirtohistoriaan, etsitään seuraavat lailliset
siirrot ja tulostetaan uusi lauta.
Lisäksi katsotaan, jatkuuko peli vai onko peli matti tai patti.
Jos siirtoyritys ei ollut laillinen, valinta vain poistuu.
Jos entuudestaan ei ollut valittua ruutua tai siirto oli laiton,
valitaan napsautettu ruutu, jos siinä on oma nappula.
*/
function selectSquare(j){
// Kun tekoäly on päällä, ei tehdä mitään.
if((document.getElementById('+1').className.search('highlight')!=-
1&&getColor()==1)||(document.getElementById('-1').class-
Name.search('highlight')!=-1&&getColor()==-1)){
return;
}
board=mainBoardLog[mainBoardLog.length-1].slice(0);
// Etsitään valittu ruutu.
for(i=0;i<n*n;i++){
if(document.getElementById(i).className.search('highlight')!=-1){
// Verrataan sallituja asemia nykyiseen asemaan.
for(k=0;k<mainLegalPositions.length;k++){
position=mainLegalPositions[k].slice(0);
if((board[i]==position[j]&&posi-
tion[i]==0&&board[j]*getColor()<=0)||(board[i]==getColor()&&posi-
tion[j]==promotion*getColor()&&posi-
tion[i]==0&&board[j]*getColor()<=0)){
if(typeof(w)!="undefined") {
w.terminate();
w=undefined;
}
// Tehdään seuraava siirto.
board=position.slice(0);
mainBoardLog[mainBoardLog.length]=board.slice(0);
boardLog=mainBoardLog.slice(0);
color=getColor();
legalPositions=[];
findLegalPositions();
mainLegalPositions=legalPositions.slice(0);
printBoard();
// Katsotaan, onko peli päättynyt.
if(mainLegalPositions.length==0){
copyBoard=board.slice(0);
if(isCheck()){
alert('Checkmate');
}else{
alert('Stalemate');
}
}
return;
}
}
// Poistetaan valinta.
document.getElementById(i).className = document.getEle-
mentById(i).className.replace(' highlight', '');
break;
}
}
```

```
// Valitaan ruutu, jos siinä on oma nappula.
if(board[j]*getColor()>0){
document.getElementById(j).className+=' highlight';
}
}
/*
SelectPromotion-funktio vastaa siitä, miksi nappulaksi sotilas ko-
rotetaan.
Korotusvalikkoon ei voi koskea, jos tekoäly on päällä.
Valikossa napsautettu ruutu valitaan ja valintaa vastaava arvo
talletetaan sille tarkoitettuun muuttujaan.
*/
function selectPromotion(j){
// Kun tekoäly on päällä, ei tehdä mitään.
if((document.getElementById('+1').className.search('highlight')!=-
1&&getColor()==1)||(document.getElementById('-1').class-
Name.search('highlight')!=-1&&getColor()==-1)){
return;
}
// Etsitään valittu ruutu.
for(i=5;i>=2;i--){
if(document.getElementById('p'+i).className.search('highlight')!=-
1){
// Poistetaan valinta.
document.getElementById('p'+i).className = document.getEle-
mentById('p'+i).className.replace(' highlight', '');
// Valitaan ruutu.
document.getElementById(j).className+=' highlight';
promotion=j.substring(1);
break;
}
}
}
/*
CpuSelect-funktio luo tekoälyvalikon.
JavaScript-kielessä on verkkotyöläinen (web worker), joka kykenee
suorittamaan koodia taustalla vaikuttamatta itse sivun toimintaan.
Valikko luodaan, jos selain tukee verkkotyöläistä.
Oletuksena tekoäly pelaa mustilla nappuloilla.
*/
function cpuSelect(){
if(typeof(Worker) === "undefined") {
return;
}
table='<table>';
table+='<tr>';
table+='<td>';
table+='</td>';
table+='<td>';
table+='</td>';
table+='<td onclick="selectCpu(this.id)" class="light" id="+1">';
table+='<span class="white">';
table+='&#128187;';
table+='</span>';
table+='</td>';
table+='<td onclick="selectCpu(this.id)" class="dark highlight"
id="-1">';
```

```
table+='<span class="black">';
table+='&#128187;';
table+='</span>';
table+='</td>';
table+='</tr>';
table+='</table>';
document.getElementById('cpu').innerHTML=table;
}
/*
SelectCpu-funktio vastaa tekoälyvalikon valinnoista.
Joko tekoäly ei pelaa, pelaa toisilla nappuloilla tai pelaa molem-
milla nappuloilla.
*/
function selectCpu(j){
if(document.getElementById(j).className.search('highlight')==-1){
document.getElementById(j).className+=' highlight';
}else{
document.getElementById(j).className = document.getEle-
mentById(j).className.replace(' highlight', '');
}
}
/*
Ohjelma tarkistaa seuraavan koodin avulla sekunnin välein, onko
tekoäly valittu.
Jos on ja peli ei ole ohi, luodaan uusi verkkotyöläinen, jos sel-
laista ei ole alun perin luotu.
Työläinen käyttää koodia, joka vastaa tekoälystä.
Kyseinen työläinen lähettää viestinä kyseiselle koodille laudan
sivun pituuden, sivuhistorian ja sallitut siirrot.
Jäädään odottamaan koodin tuottamaa siirtoa.
Kun siirto saapuu paluuviestinä, työläinen nollataan ja suorite-
taan siirtotoimenpiteet tavallisesti.
*/
var w;
setInterval(function(){
if((document.getElementById('+1').className.search('highlight')!=-
1&&getColor()==1)||(document.getElementById('-1').class-
Name.search('highlight')!=-1&&getColor()==-1)){
if(mainLegalPositions.length==0){
return;
}
// Luodaan uusi verkkotyöläinen, jos sellaista ei ole alun perin
luotu.
if(typeof(w)=="undefined") {
w=new Worker("cpu.js");
// Lähetetään tiedot.
w.postMessage([n,mainBoardLog,mainLegalPositions]);
result=undefined;
}
// Otetaan vastaan vastaus.
w.onmessage=function(e) {
result=e.data;
};
if(result!=undefined){
// Tehdään seuraava siirto.
board=result;
w.terminate();
```

```
w=undefined;
mainBoardLog[mainBoardLog.length]=board.slice(0);
boardLog=mainBoardLog.slice(0);
color=getColor();
legalPositions=[];
findLegalPositions();
mainLegalPositions=legalPositions.slice(0);
printBoard();
if(mainLegalPositions.length==0){
copyBoard=board.slice(0);
if(isCheck()){
alert('Checkmate');
}else{
alert('Stalemate');
}
}
}
}
}, 1000);
```

**Liite 3. Sääntökoodi**

```
/*
Tässä esitellään JavaScript-dokumentti (rules.js), joka vastaa
sääntöjen oikeudenmukaisuudesta.
GetColor-funktio vastaa siitä, kumman vuoro on.
Funktio palauttaa positiivisen ykkösen, jos on valkean vuoro.
Funktio palauttaa negatiivisen ykkösen, jos on mustan vuoro.
Siirtovuoro määräytyy siirtohistorian mukaan.
*/
function getColor(){
if((mainBoardLog.length-1)%2==0){
return 1;
}
return -1;
}
/*
FindLegalPositions-funktio etsii lailliset siirrot.
Tämä on pääfunktio, jossa on alifunktioina eri nappuloiden siirto-
testausfunktiot.
Etsitään laudalta vuorossa olevan pelaajan nappulat ja testataan
siirrot.
*/
function findLegalPositions(){
for(i=0;i<n*n;i++){
if(board[i]==color){
tryMovesPawn();
}else if(board[i]==2*color){
tryMovesKnight();
}else if(board[i]==3*color){
tryMovesBishop();
}else if(board[i]==4*color){
tryMovesRook();
}else if(board[i]==5*color){
tryMovesBishop();
tryMovesRook();
}
}
for(i=0;i<n*n;i++){
if(board[i]==6*color){
tryMovesKing();
}
}
}
/*
Siirtotestausfunktiot testaavat kaikki mahdolliset kyseisen nappu-
lan siirrot.
Funktioissa testataan, meneekö nappula laudan ulkopuolelle.
Lisäksi funktioissa testataan, onko kohderuudussa vuorossa olevan
pelaajan nappula.
Jos nappula pysyy laudan sisäpuolella eikä syö samanväristä nappu-
laa, testataan siirtoa.
Kaikissa näissä funktioissa käytetään apuna kopiolautaa siirtojen
määrittelyssä, koska ei haluta sotkea alkuperäistä tilannetta.
```

```
Kopiolaudassa yleisesti kohderuutu saa arvoksi lähtöruudun arvon
ja lähtöruutu muutetaan nollaksi.
Lopuksi testataan, onko vuorossa olevan pelaajan kuningas jäänyt
syötäväksi.
Jos ei ole, siirto lisätään laillisten siirtojen joukkoon.

Vaikka lauta on tavallinen yksiulotteinen taulukko, siirrot tapah-
tuvat laudalla kaksiulotteiseen tyyliin.
Laudan sivun pituuden avulla tiedetään, mihin riviin siirrytään.
Toisin sanoen alas mentäessä kasvatetaan lähtöarvoa käyttäen sivun
pituutta kertoimena.
Ylös mentäessä vähennetään lähtöarvoa käyttäen sivun pituutta ker-
toimena.
Oikealle mentäessä kasvatetaan lähtöarvoa ja vasemmalle mentäessä
vähennetään lähtöarvoa.
Jos nappula lentää ulos laudalta yläkautta, on arvo negatiivinen.
Jos nappula lentää ulos laudalta alakautta, on arvo suurempi tai
yhtä suuri kuin sivujen neliö.
Jakojäännöksen avulla voidaan selvittää, hyppääkö nappula laudalta
pois oikealta tai vasemmalta puolelta.
*/
/*
TryMovesPawn-funktio testaa kaikki mahdolliset sotilaan siirrot.
Testataan sotilaan perusliikkeet, korotusmahdollisuudet ja ohesta-
lyönti.
Ensin tarkistetaan, kumman vuoro on.
Tämä on tiedettävä, koska sotilaan liikkumissuunta määräytyy vuo-
ron perusteella.
Tarkistetaan, onko sotilaan edessä oleva ruutu tyhjä.
Jos on, tarkistetaan, millä rivillä sotilas on.
Jos sotilas on lähtörivillä, testataan tavallinen askel ja kak-
soisaskel.
Jos sotilas on rivillä, josta on mahdollisuus korottua, testataan
korotussiirrot.
Ensin kohderuudun arvo muutetaan ratsuksi, seuraavaksi lähetiksi,
torniksi ja lopulta daamiksi.
Jos sotilas on muulla rivillä, testataan perusaskel.

Testataan sotilaan viistoliikkeet.
Jos kohderuudussa on vastustajan nappula, jatketaan testaamista.
Jos sotilas on rivillä, josta on mahdollisuus korottua, testataan
korotussiirrot.
Jos sotilas on muulla rivillä, testataan perusviistoaskel.
Jos kohderuudussa ei ole vastustajan nappulaa, mutta sotilaan vie-
ressä on vastustajan sotilas, testataan ohestalyönti.
Jos sotilas on ohestalyöntiin oikeutetulla rivillä, otetaan muis-
tiin siirtohistorian toiseksi viimeinen asema.
Samalla vastustajan sotilasta siirretään kaksi askelta taaksepäin
kopiolaudalla.
Jos kopiolauta ja muistiinotettu asema ovat samat, testataan var-
sinainen ohestalyönti.
Muistetaan poistaa kopiolaudasta syötävä sotilas.
*/
function tryMovesPawn(){
if(color==1){
if(board[i-n]==0){
if(i>=n*6&&i<n*7){
```

```
copyBoard=board.slice(0);
copyBoard[i-n]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
if(board[i-n*2]==0){
// double move
copyBoard=board.slice(0);
copyBoard[i-n*2]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}else if(i>=n&&i<n*2){
// promotion
copyBoard=board.slice(0);
copyBoard[i-n]=2;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i-n]=3;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i-n]=4;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i-n]=5;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}else{
copyBoard=board.slice(0);
copyBoard[i-n]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
if((i-1)%n!=(n-1)%n){
if(board[i-n-1]<0){
if(i>=n&&i<n*2){
// promotion
copyBoard=board.slice(0);
copyBoard[i-n-1]=2;
copyBoard[i]=0;
```

```
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i-n-1]=3;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i-n-1]=4;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i-n-1]=5;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}else{
copyBoard=board.slice(0);
copyBoard[i-n-1]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}else if(board[i-1]==-1){
if(i>=n*3&&i<n*4){
position=boardLog[boardLog.length-2].slice(0);
copyBoard=board.slice(0);
copyBoard[i-n*2-1]=copyBoard[i-1];
copyBoard[i-1]=0;
for(j=0;j<n*n;j++){
if(position[j]!=copyBoard[j]){
break;
}
if(j==n*n-1){
// en passant
copyBoard=board.slice(0);
copyBoard[i-n-1]=copyBoard[i];
copyBoard[i-1]=0;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
}
}
}
if((i+1)%n!=0){
if(board[i-n+1]<0){
if(i>=n&&i<n*2){
// promotion
```

```
copyBoard=board.slice(0);
copyBoard[i-n+1]=2;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i-n+1]=3;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i-n+1]=4;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i-n+1]=5;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}else{
copyBoard=board.slice(0);
copyBoard[i-n+1]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}else if(board[i+1]==-1){
if(i>=n*3&&i<n*4){
position=boardLog[boardLog.length-2].slice(0);
copyBoard=board.slice(0);
copyBoard[i-n*2+1]=copyBoard[i+1];
copyBoard[i+1]=0;
for(j=0;j<n*n;j++){
if(position[j]!=copyBoard[j]){
break;
}
if(j==n*n-1){
// en passant
copyBoard=board.slice(0);
copyBoard[i-n+1]=copyBoard[i];
copyBoard[i+1]=0;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
}
}
}
}else{
```

```
if(board[i+n]==0){
if(i>=n&&i<n*2){
copyBoard=board.slice(0);
copyBoard[i+n]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
if(board[i+n*2]==0){
// double move
copyBoard=board.slice(0);
copyBoard[i+n*2]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}else if(i>=n*6&&i<n*7){
// promotion
copyBoard=board.slice(0);
copyBoard[i+n]=-2;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i+n]=-3;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i+n]=-4;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i+n]=-5;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}else{
copyBoard=board.slice(0);
copyBoard[i+n]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
if((i-1)%n!=(n-1)%n){
if(board[i+n-1]>0){
if(i>=n*6&&i<n*7){
// promotion
copyBoard=board.slice(0);
```

```
copyBoard[i+n-1]=-2;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i+n-1]=-3;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i+n-1]=-4;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i+n-1]=-5;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}else{
copyBoard=board.slice(0);
copyBoard[i+n-1]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}else if(board[i-1]==1){
if(i>=n*4&&i<n*5){
position=boardLog[boardLog.length-2].slice(0);
copyBoard=board.slice(0);
copyBoard[i+n*2-1]=copyBoard[i-1];
copyBoard[i-1]=0;
for(j=0;j<n*n;j++){
if(position[j]!=copyBoard[j]){
break;
}
if(j==n*n-1){
// en passant
copyBoard=board.slice(0);
copyBoard[i+n-1]=copyBoard[i];
copyBoard[i-1]=0;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
}
}
}
if((i+1)%n!=0){
if(board[i+n+1]>0){
```

```
if(i>=n*6&&i<n*7){
// promotion
copyBoard=board.slice(0);
copyBoard[i+n+1]=-2;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i+n+1]=-3;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i+n+1]=-4;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
copyBoard=board.slice(0);
copyBoard[i+n+1]=-5;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}else{
copyBoard=board.slice(0);
copyBoard[i+n+1]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}else if(board[i+1]==1){
if(i>=n*4&&i<n*5){
position=boardLog[boardLog.length-2].slice(0);
copyBoard=board.slice(0);
copyBoard[i+n*2+1]=copyBoard[i+1];
copyBoard[i+1]=0;
for(j=0;j<n*n;j++){
if(position[j]!=copyBoard[j]){
break;
}
if(j==n*n-1){
// en passant
copyBoard=board.slice(0);
copyBoard[i+n+1]=copyBoard[i];
copyBoard[i+1]=0;
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
}
}
```

```
}
}
}
/*
Ratsulla on maksimissa kahdeksan siirtovaihtoehtoa.
*/
function tryMovesKnight(){
if(i-n*2-1>=0&&(i-n*2-1)%n!=(n-1)%n){
if(board[i-n*2-1]*color<=0){
copyBoard=board.slice(0);
copyBoard[i-n*2-1]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
if(i-n*2+1>=0&&(i-n*2+1)%n!=0){
if(board[i-n*2+1]*color<=0){
copyBoard=board.slice(0);
copyBoard[i-n*2+1]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
if(i-n-2>=0&&(i-n-2)%n!=(n-1)%n&&(i-n-2)%n!=(n-2)%n){
if(board[i-n-2]*color<=0){
copyBoard=board.slice(0);
copyBoard[i-n-2]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
if(i-n+2>=0&&(i-n+2)%n!=0&&(i-n+2)%n!=1){
if(board[i-n+2]*color<=0){
copyBoard=board.slice(0);
copyBoard[i-n+2]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
if(i+n-2<n*n&&(i+n-2)%n!=(n-1)%n&&(i+n-2)%n!=(n-2)%n){
if(board[i+n-2]*color<=0){
copyBoard=board.slice(0);
copyBoard[i+n-2]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
```

```
if(i+n+2<n*n&&(i+n+2)%n!=0&&(i+n+2)%n!=1){
if(board[i+n+2]*color<=0){
copyBoard=board.slice(0);
copyBoard[i+n+2]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
if(i+n*2-1<n*n&&(i+n*2-1)%n!=(n-1)%n){
if(board[i+n*2-1]*color<=0){
copyBoard=board.slice(0);
copyBoard[i+n*2-1]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
if(i+n*2+1<n*n&&(i+n*2+1)%n!=0){
if(board[i+n*2+1]*color<=0){
copyBoard=board.slice(0);
copyBoard[i+n*2+1]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
}
function tryMovesBishop(){
for(j=n+1;j<n*n;j+=n+1){
if(i-j>=0&&(i-j)%n!=(n-1)%n){
if(board[i-j]==0){
copyBoard=board.slice(0);
copyBoard[i-j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}else if(board[i-j]*color<0){
copyBoard=board.slice(0);
copyBoard[i-j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
break;
}else{
break;
}
}else{
break;
}
}
for(j=n-1;j<n*n;j+=n-1){
```

```
if(i-j>=0&&(i-j)%n!=0){
if(board[i-j]==0){
copyBoard=board.slice(0);
copyBoard[i-j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}else if(board[i-j]*color<0){
copyBoard=board.slice(0);
copyBoard[i-j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
break;
}else{
break;
}
}else{
break;
}
}
for(j=n-1;j<n*n;j+=n-1){
if(i+j<n*n&&(i+j)%n!=(n-1)%n){
if(board[i+j]==0){
copyBoard=board.slice(0);
copyBoard[i+j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}else if(board[i+j]*color<0){
copyBoard=board.slice(0);
copyBoard[i+j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
break;
}else{
break;
}
}else{
break;
}
}
for(j=n+1;j<n*n;j+=n+1){
if(i+j<n*n&&(i+j)%n!=0){
if(board[i+j]==0){
copyBoard=board.slice(0);
copyBoard[i+j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}else if(board[i+j]*color<0){
```

```
copyBoard=board.slice(0);
copyBoard[i+j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
break;
}else{
break;
}
}else{
break;
}
}
}
function tryMovesRook(){
for(j=n;j<n*n;j+=n){
if(i-j>=0){
if(board[i-j]==0){
copyBoard=board.slice(0);
copyBoard[i-j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}else if(board[i-j]*color<0){
copyBoard=board.slice(0);
copyBoard[i-j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
break;
}else{
break;
}
}else{
break;
}
}
for(j=1;j<n;j++){
if((i-j)%n!=(n-1)%n){
if(board[i-j]==0){
copyBoard=board.slice(0);
copyBoard[i-j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}else if(board[i-j]*color<0){
copyBoard=board.slice(0);
copyBoard[i-j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
break;
```

```
}else{
break;
}
}else{
break;
}
}
for(j=1;j<n;j++){
if((i+j)%n!=0){
if(board[i+j]==0){
copyBoard=board.slice(0);
copyBoard[i+j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}else if(board[i+j]*color<0){
copyBoard=board.slice(0);
copyBoard[i+j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
break;
}else{
break;
}
}else{
break;
}
}
for(j=n;j<n*n;j+=n){
if(i+j<n*n){
if(board[i+j]==0){
copyBoard=board.slice(0);
copyBoard[i+j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}else if(board[i+j]*color<0){
copyBoard=board.slice(0);
copyBoard[i+j]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
break;
}else{
break;
}
}else{
break;
}
}
}
function tryMovesKing(){
```

```
if(i-n-1>=0&&(i-n-1)%n!=(n-1)%n){
if(board[i-n-1]*color<=0){
copyBoard=board.slice(0);
copyBoard[i-n-1]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
if(i-n>=0){
if(board[i-n]*color<=0){
copyBoard=board.slice(0);
copyBoard[i-n]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
if(i-n+1>=0&&(i-n+1)%n!=0){
if(board[i-n+1]*color<=0){
copyBoard=board.slice(0);
copyBoard[i-n+1]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
if((i-1)%n!=(n-1)%n){
if(board[i-1]*color<=0){
copyBoard=board.slice(0);
copyBoard[i-1]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
if((i+1)%n!=0){
if(board[i+1]*color<=0){
copyBoard=board.slice(0);
copyBoard[i+1]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
if(i+n-1<n*n&&(i+n-1)%n!=(n-1)%n){
if(board[i+n-1]*color<=0){
copyBoard=board.slice(0);
copyBoard[i+n-1]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
```

```
}
}
}
if(i+n<n*n){
if(board[i+n]*color<=0){
copyBoard=board.slice(0);
copyBoard[i+n]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
if(i+n+1<n*n&&(i+n+1)%n!=0){
if(board[i+n+1]*color<=0){
copyBoard=board.slice(0);
copyBoard[i+n+1]=copyBoard[i];
copyBoard[i]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
// castle
if(color==1){
for(j=boardLog.length-1;j>=0;j--){
position=boardLog[j].slice(0);
if(position[n*7+4]!=6){
break;
}
if(j==0){
for(j=boardLog.length-1;j>=0;j--){
position=boardLog[j].slice(0);
if(position[n*7]!=4){
break;
}
if(j==0){
if(board[n*7+1]!=0||board[n*7+2]!=0||board[n*7+3]!=0){
break;
}
copyBoard=board.slice(0);
if(isCheck()){
break;
}
copyBoard[n*7+3]=copyBoard[n*7+4];
copyBoard[n*7+4]=0;
if(isCheck()){
break;
}
copyBoard[n*7+2]=copyBoard[n*7+3];
copyBoard[n*7+3]=copyBoard[n*7];
copyBoard[n*7]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
```

```
for(j=boardLog.length-1;j>=0;j--){
position=boardLog[j].slice(0);
if(position[n*7+7]!=4){
break;
}
if(j==0){
if(board[n*7+5]!=0||board[n*7+6]!=0){
break;
}
copyBoard=board.slice(0);
if(isCheck()){
break;
}
copyBoard[n*7+5]=copyBoard[n*7+4];
copyBoard[n*7+4]=0;
if(isCheck()){
break;
}
copyBoard[n*7+6]=copyBoard[n*7+5];
copyBoard[n*7+5]=copyBoard[n*7+7];
copyBoard[n*7+7]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
break;
}
}
}else{
for(j=boardLog.length-1;j>=0;j--){
position=boardLog[j].slice(0);
if(position[4]!=-6){
break;
}
if(j==0){
for(j=boardLog.length-1;j>=0;j--){
position=boardLog[j].slice(0);
if(position[0]!=-4){
break;
}
if(j==0){
if(board[1]!=0||board[2]!=0||board[3]!=0){
break;
}
copyBoard=board.slice(0);
if(isCheck()){
break;
}
copyBoard[3]=copyBoard[4];
copyBoard[4]=0;
if(isCheck()){
break;
}
copyBoard[2]=copyBoard[3];
copyBoard[3]=copyBoard[0];
copyBoard[0]=0;
```

```
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
for(j=boardLog.length-1;j>=0;j--){
position=boardLog[j].slice(0);
if(position[7]!=-4){
break;
}
if(j==0){
if(board[5]!=0||board[6]!=0){
break;
}
copyBoard=board.slice(0);
if(isCheck()){
break;
}
copyBoard[5]=copyBoard[4];
copyBoard[4]=0;
if(isCheck()){
break;
}
copyBoard[6]=copyBoard[5];
copyBoard[5]=copyBoard[7];
copyBoard[7]=0;
if(!isCheck()){
legalPositions[legalPositions.length]=copyBoard.slice(0);
}
}
}
break;
}
}
}
}
/*
IsCheck-funktio tarkistaa, onko oma kuningas shakissa.
Otetaan ylös vastustajan hallitsemat ruudut kontrollilautaan.
Etsitään oma kuningas kopiolaudalta.
Jos kopiolaudan ruudussa on oma kuningas ja vastaavassa ruudussa
kontrollilaudalla on suurempi arvo kuin nolla, on tilanne shakki.
*/
function isCheck(){
controlBoard=[];
color=-color;
setControl(1,1,1,1,1,1);
color=-color;
for(k=0;k<n*n;k++){
if(copyBoard[k]==6*color){
if(controlBoard[k]>0){
return true;
}
break;
}
}
return false;
```

```
}
/*
SetControl-funktio asettaa arvot kontrollilautaan.
Tämä on funktio, jossa on alifunktioina eri nappuloiden hallinta-
arvoja asettavat funktiot.
Kontrollilaudassa ruudussa oleva luku kertoo, kuinka moni tietyn
värinen nappula hallitsee kyseistä ruutua.
Eri nappuloille voi antaa lisäksi eri hallinta-arvot tarpeen mu-
kaan.
Ensin kontrollilauta alustetaan nollilla.
Etsitään kopiolaudalta tietyn väriset nappulat ja asetetaan hal-
linta-arvot ruutuihin.
*/
function setControl(pv,nv,bv,rv,qv,kv){
for(k=0;k<n*n;k++){
controlBoard[k]=0;
}
for(k=0;k<n*n;k++){
if(copyBoard[k]==color){
setControlPawn(pv);
}else if(copyBoard[k]==2*color){
setControlKnight(nv);
}else if(copyBoard[k]==3*color){
setControlBishop(bv);
}else if(copyBoard[k]==4*color){
setControlRook(rv);
}else if(copyBoard[k]==5*color){
setControlBishop(qv);
setControlRook(qv);
}else if(copyBoard[k]==6*color){
setControlKing(kv);
}
}
}
function setControlPawn(v){
if(color==1){
if((k-n-1)%n!=(n-1)%n){
controlBoard[k-n-1]+=v;
}
if((k-n+1)%n!=0){
controlBoard[k-n+1]+=v;
}
}else{
if((k+n-1)%n!=(n-1)%n){
controlBoard[k+n-1]+=v;
}
if((k+n+1)%n!=0){
controlBoard[k+n+1]+=v;
}
}
}
function setControlKnight(v){
if(k-n*2-1>=0&&(k-n*2-1)%n!=(n-1)%n){
controlBoard[k-n*2-1]+=v;
}
if(k-n*2+1>=0&&(k-n*2+1)%n!=0){
controlBoard[k-n*2+1]+=v;
```

```
}
if(k-n-2>=0&&(k-n-2)%n!=(n-1)%n&&(k-n-2)%n!=(n-2)%n){
controlBoard[k-n-2]+=v;
}
if(k-n+2>=0&&(k-n+2)%n!=0&&(k-n+2)%n!=1){
controlBoard[k-n+2]+=v;
}
if(k+n-2<n*n&&(k+n-2)%n!=(n-1)%n&&(k+n-2)%n!=(n-2)%n){
controlBoard[k+n-2]+=v;
}
if(k+n+2<n*n&&(k+n+2)%n!=0&&(k+n+2)%n!=1){
controlBoard[k+n+2]+=v;
}
if(k+n*2-1<n*n&&(k+n*2-1)%n!=(n-1)%n){
controlBoard[k+n*2-1]+=v;
}
if(k+n*2+1<n*n&&(k+n*2+1)%n!=0){
controlBoard[k+n*2+1]+=v;
}
}
function setControlBishop(v){
for(l=n+1;l<n*n;l+=n+1){
if(k-l>=0&&(k-l)%n!=(n-1)%n){
controlBoard[k-l]+=v;
if(copyBoard[k-l]!=0){
break;
}
}else{
break;
}
}
for(l=n-1;l<n*n;l+=n-1){
if(k-l>=0&&(k-l)%n!=0){
controlBoard[k-l]+=v;
if(copyBoard[k-l]!=0){
break;
}
}else{
break;
}
}
for(l=n-1;l<n*n;l+=n-1){
if(k+l<n*n&&(k+l)%n!=(n-1)%n){
controlBoard[k+l]+=v;
if(copyBoard[k+l]!=0){
break;
}
}else{
break;
}
}
for(l=n+1;l<n*n;l+=n+1){
if(k+l<n*n&&(k+l)%n!=0){
controlBoard[k+l]+=v;
if(copyBoard[k+l]!=0){
break;
}
```

```
}else{
break;
}
}
}
function setControlRook(v){
for(l=n;l<n*n;l+=n){
if(k-l>=0){
controlBoard[k-l]+=v;
if(copyBoard[k-l]!=0){
break;
}
}else{
break;
}
}
for(l=1;l<n;l++){
if((k-l)%n!=(n-1)%n){
controlBoard[k-l]+=v;
if(copyBoard[k-l]!=0){
break;
}
}else{
break;
}
}
for(l=1;l<n;l++){
if((k+l)%n!=0){
controlBoard[k+l]+=v;
if(copyBoard[k+l]!=0){
break;
}
}else{
break;
}
}
for(l=n;l<n*n;l+=n){
if(k+l<n*n){
controlBoard[k+l]+=v;
if(copyBoard[k+l]!=0){
break;
}
}else{
break;
}
}
}
function setControlKing(v){
if(k-n-1>=0&&(k-n-1)%n!=(n-1)%n){
controlBoard[k-n-1]+=v;
}
if(k-n>=0){
controlBoard[k-n]+=v;
}
if(k-n+1>=0&&(k-n+1)%n!=0){
controlBoard[k-n+1]+=v;
}
```

```
if((k-1)%n!=(n-1)%n){
controlBoard[k-1]+=v;
}
if((k+1)%n!=0){
controlBoard[k+1]+=v;
}
if(k+n-1<n*n&&(k+n-1)%n!=(n-1)%n){
controlBoard[k+n-1]+=v;
}
if(k+n<n*n){
controlBoard[k+n]+=v;
}
if(k+n+1<n*n&&(k+n+1)%n!=0){
controlBoard[k+n+1]+=v;
}
}
```

**Liite 4. Tekoälykoodi**

```
/*
Tässä esitellään JavaScript-dokumentti (cpu.js), joka vastaa teko-
älyn toiminnasta.
Tekoäly käyttää hyödyksi valmiiksi määriteltyjä sääntöjä.
Otetaan vastaan viesti pelin perustietoineen ja etsitään mahdolli-
simman hyvä siirto.
*/
importScripts('rules.js');
onmessage = function(e){
n=e.data[0];
mainBoardLog=e.data[1].slice(0);
mainLegalPositions=e.data[2].slice(0);
findBestMove();
};
/*
FindBestMove-funktio etsii niin hyvän siirron kuin pystyy.
Luodaan pelipuu tiettyyn hakusyvyyteen asti.
Samalla arvioidaan asemat.
Valmis pelipuu analysoidaan, jolloin saadaan parhaat siirrot sel-
ville.
Lopuksi parhaista siirroista arvotaan yksi siirroksi.
*/
function findBestMove(){
/*
Taulukkoon sijoitetaan kaikki mahdolliset pelijatkot.
Jokaisella jatkolla on yksilöivä avain ja peliarvo.
Hakusyvyydeksi on asetettu tässä tapauksessa kolme puolisiirtoa.
Hakuaika on kaksi sekuntia ja aluksi ei anneta ylimääräistä aikaa.
*/
mainlines=[];
keys=[];
values=[];
hm=3;
time=2000;
extraTime=false;
/*
Arvotetaan ruudut arviointifunktiota varten valmiiksi.
Ruutujen arvot ovat seuraavat:

1,2,3,4,4,3,2,1,
2,4,6,8,8,6,4,2,
3,6,9,12,12,9,6,3,
4,8,12,16,16,12,8,4,
4,8,12,16,16,12,8,4,
3,6,9,12,12,9,6,3,
2,4,6,8,8,6,4,2,
1,2,3,4,4,3,2,1

Mitä keskemmällä lautaa ollaan, sen arvokkaampi ruutu on.
Tämä liittyy lähinnä arviointifunktioon.
*/
squareValues=[];
for(i=0;i<n/2;i++){
```

```
squareValues[i]=i+1;
squareValues[n-i-1]=i+1;
squareValues[n*7+i]=i+1;
squareValues[n*n-i-1]=i+1;

squareValues[n+i]=(i+1)*2;
squareValues[n*2-i-1]=(i+1)*2;
squareValues[n*6+i]=(i+1)*2;
squareValues[n*7-i-1]=(i+1)*2;

squareValues[n*2+i]=(i+1)*3;
squareValues[n*3-i-1]=(i+1)*3;
squareValues[n*5+i]=(i+1)*3;
squareValues[n*6-i-1]=(i+1)*3;

squareValues[n*3+i]=(i+1)*4;
squareValues[n*4-i-1]=(i+1)*4;
squareValues[n*4+i]=(i+1)*4;
squareValues[n*5-i-1]=(i+1)*4;
}
/*
Luodaan pohja jatkoille asettamalla ensimmäiset siirrot tauluk-
koon.
Avaimet määräytyvät silmukan indeksin mukaan.
Jatkon alku arvioidaan.
*/
for(m=0;m<mainLegalPositions.length;m++){
mainline=[];
mainline[0]=mainLegalPositions[m].slice(0);
mainlines[mainlines.length]=mainline.slice(0);
keys[keys.length]=''+m;
valuePosition();
}
/*
Seuraavaksi rakennetaan itse pelipuu.
Otetaan aluksi muistiin aloitusaika.
Itse pelipuukoodissa johdetaan taulukon ensimmäisestä jatkosta
kaikki lailliset siirrot.
Laillisten siirtojen perusteella asetetaan uudet pidemmälle luodut
jatkot taulukon perälle.
Näille jatkoille määritetään avain käyttäen pohjana alkuperäisen
jatkon avainta.
Uusille jatkoille annetaan lisäksi arvo.
Täten alkuperäinen jatko poistetaan taulukon alusta, kun siitä on
johdettu kaikki jatkot.
Samoin alkuperäisen jatkon avain ja arvo poistetaan.
Tätä toistetaan niin kauan, kunnes kaikki jatkot ovat yhtä pitkiä.
Toisaalta jos jatkoa ei pysty pidentämään, on tilanne joko matti
tai patti.
Jos tekoäly on tekemässä mattia, annetaan arvoksi maksimaalinen
arvo, josta vähennetään hakusyvyys, jolloin aikaisempi matti on
arvokkaampi.
Jos tekoäly on joutumassa mattiin, arvo äskeiseen verrattuna on
vastakkainen.
Pattitilanne on arvoltaan nolla.
Jatko, avain ja arvo siirretään taulukon perälle.
```

```
Kun kaikki jatkot on saatu käsiteltyä, aloitetaan sama jatkojen
käsittely uudestaan, jos haluttua hakusyvyyttä ei ole saavutettu.
Jos tavoiteltu hakusyvyys on saavutettu ja aikaa on vielä jäl-
jellä, jatkoja yritetään vielä pidentää niin paljon kuin ehtii.
*/
start=performance.now();
for(h=1;h<hm;h++){
g=mainlines.length;
for(m=0;m<g;m++){
if((mainlines[0].length==h&&performance.now()-start<time&&ex-
traTime)||(mainlines[0].length==h&&!extraTime)){
// Asetetaan säännöt.
mainline=mainlines[0].slice(0);
board=mainline[mainline.length-1].slice(0);
boardLog=mainBoardLog.slice(0);
for(i=0;i<mainline.length;i++){
boardLog[boardLog.length]=mainline[i].slice(0);
}
if((boardLog.length-1)%2==0){
color=1;
}else{
color=-1;
}
legalPositions=[];
findLegalPositions();
// Katsotaan, onko peli päättynyt.
if(legalPositions.length==0){
// Ensimmäinen jatko lisätään taulukon loppuun.
mainlines[mainlines.length]=mainline.slice(0);
keys[keys.length]=keys[0];
copyBoard=board.slice(0);
if(isCheck()){
if(color!=getColor()){
values[values.length]=Math.pow(2,53)-h;
}else{
values[values.length]=-Math.pow(2,53)+h;
}
}else{
values[values.length]=0;
}
// Poistetaan ensimmäinen jatko.
mainlines.splice(0,1);
keys.splice(0,1);
values.splice(0,1);
}
// Jos peli ei ole päättynyt, luodaan uudet jatkot.
for(i=0;i<legalPositions.length;i++){
mainline=mainlines[0].slice(0);
mainline[mainline.length]=legalPositions[i].slice(0);
mainlines[mainlines.length]=mainline.slice(0);
keys[keys.length]=keys[0]+' '+i;
valuePosition();
// Poistetaan ensimmäinen jatko lopuksi.
if(i==legalPositions.length-1){
mainlines.splice(0,1);
keys.splice(0,1);
values.splice(0,1);
```

```
}
}
}else{
// Jatko siirretään taulukon perälle ilman käsittelyä.
mainlines[mainlines.length]=mainlines[0].slice(0);
keys[keys.length]=keys[0];
values[values.length]=values[0];
mainlines.splice(0,1);
keys.splice(0,1);
values.splice(0,1);
}
}
// Jos tavoiteltu hakusyvyys on saavutettu ja aikaa on vielä jäl-
jellä, lisätään hakusyvyyttä ja mennään lisäajalle.
if(performance.now()-start<time&&h==hm-1){
hm++;
extraTime=true;
}
}
// Analysoidaan pelipuu.
analyzeGameTree();
/*
Nyt tiedetään jokaisen laillisen siirron arvo.
Parhaista siirroista arvotaan yksi varsinaiseksi siirroksi.
Siirto lähetetään tekoälyn siirrettäväksi.
*/
bestMoves=[];
for(m=0;m<values.length;m++){
if(values[m]==Math.max.apply(Math,values)){
bestMoves[bestMoves.length]=mainLegalPositions[m].slice(0);
}
}
postMessage(bestMoves[Math.floor(Math.random()*best-
Moves.length)].slice(0));
}
/*
AnalyzeGameTree-funktio toteuttaa minmax-algoritmin ideaa.
Analysoidaan juuri rakennettu pelipuu avainten avulla.
Tällä kertaa pelipuun kasvattamisen sijaan kutistetaan avainten
määrää vaihe vaiheelta latvasta lähtien.
Otetaan käsittelyyn ensimmäinen avain, josta poistetaan viimeinen
luku.
Jos tämä yhdellä luvulla lyhennetty avain on sama kuin viimeinen
avain, viittaavat molemmat avaimet samaan asemaan.
Tällöin arvoja voidaan verrata keskenään.
Jos käsiteltävä hakusyvyys on parillinen, täytyy arvoa minimoida
mahdollisuuden mukaan.
Jos käsiteltävä hakusyvyys on pariton, täytyy arvoa maksimoida
mahdollisuuden mukaan.
Jos arvoa voidaan päivittää, korvataan viimeistä avainta vastaava
arvo uudella arvolla.
Ensimmäinen avain ja arvo poistetaan.
Toisaalta jos tämä yhdellä luvulla lyhennetty avain on eri kuin
viimeinen avain, siirretään tämä ensimmäinen avain ja sen arvo
viimeiseksi.
Tätä kiertoa jatketaan niin kauan, kunnes kaikki avaimet, jotka on
voitu lyhentää, on lyhennetty yhdellä luvulla.
```

```
Tätä avainten lyhentämisprosessia jatketaan niin kauan, kunnes
kaikissa avaimissa on vain yksi luku.
*/
function analyzeGameTree(){
for(h=hm;h>1;h--){
g=keys.length;
for(i=0;i<g;i++){
key=keys[0].split(' ');
if(key.length==h){
// Avaimesta poistetaan viimeinen luku.
key.splice(key.length-1,1);
key=key.join(' ');
if(key==keys[keys.length-1]){
if(h%2==0){
// Minimoidaan arvo.
if(values[values.length-1]>values[0]){
values[values.length-1]=values[0];
}
}else{
// Maksimoidaan arvo.
if(values[values.length-1]<values[0]){
values[values.length-1]=values[0];
}
}
}else{
// Ensimmäinen avain lisätään loppuun.
keys[keys.length]=key;
values[values.length]=values[0];
}
// Ensimmäinen avain poistetaan.
keys.splice(0,1);
values.splice(0,1);
}else{
// Avain siirretään perälle ilman käsittelyä.
keys[keys.length]=keys[0];
values[values.length]=values[0];
keys.splice(0,1);
values.splice(0,1);
}
}
}
}
/*
Seuraavaksi esitellään arviointifunktio.
Funktio arvioi jatkon viimeisen aseman perusteella.
*/
function valuePosition(){
board=mainline[mainline.length-1].slice(0);
/*
Ensin summataan materiaali yhteen.
Omasta nappulasta yhteisarvo kasvaa ja vastustajan nappulasta yh-
teisarvo vähenee.
Sotilaan arvo on 1, ratsun 3, lähetin 3, tornin 5 ja daamin 9.
*/
materialValue=0;
for(j=0;j<n*n;j++){
if(board[j]==getColor()){
```

```
// Oma sotilas löytyi.
materialValue+=1;
}else if(board[j]==-getColor()){
// Vastustajan sotilas löytyi.
materialValue-=1;
}else if(board[j]==2*getColor()){
// Oma ratsu löytyi.
materialValue+=3;
}else if(board[j]==2*-getColor()){
// Vastustajan ratsu löytyi.
materialValue-=3;
}else if(board[j]==3*getColor()){
// Oma lähetti löytyi.
materialValue+=3;
}else if(board[j]==3*-getColor()){
// Vastustajan lähetti löytyi.
materialValue-=3;
}else if(board[j]==4*getColor()){
// Oma torni löytyi.
materialValue+=5;
}else if(board[j]==4*-getColor()){
// Vastustajan torni löytyi.
materialValue-=5;
}else if(board[j]==5*getColor()){
// Oma daami löytyi.
materialValue+=9;
}else if(board[j]==5*-getColor()){
// Vastustajan daami löytyi.
materialValue-=9;
}
}
/*
Seuraavaksi luodaan oma kontrollilauta.
Omassa kontrollilaudassa ruudussa oleva luku kertoo, kuinka moni
oma nappula hallitsee kyseistä ruutua.
Eri nappuloille on annettu lisäksi eri hallinta-arvot.
Toisin sanoen nyt sotilas lisää ruutuun 9, ratsu 7, lähetti 7,
torni 5, daami 1 ja kuningas 6 pistettä.
Vaikka kuningas on äärettömän arvokas, sen hallinta-arvo on tornin
ja kevyen upseerin välissä (lähetti ja ratsu).
Idea on se, että mitä heikompi nappula saadaan hyötykäytettyä,
sitä parempi asema on.
Ainakin tämä estää esimerkiksi turhan aktiivisen daamin käytön.
Esimerkiksi avauksessa daamia ei kannata kehittää yleensä avaus-
teorian mukaan.
Kontrollilauta kuvaa liikkuvuutta, kehitystä ja keskustan hallin-
taa.
Lauta kuvaa liikkuvuutta, koska nappula hallitsee melkein samalla
tavalla kuin liikkuu.
Mitä enemmän nappulat hallitsevat, sitä kehittyneempiä ne todennä-
köisesti ovat.
Mitä keskemmällä nappula on lautaa, sitä enemmän se hallitsee.
*/
color=getColor();
copyBoard=board.slice(0);
controlBoard=[];
setControl(9,7,7,5,1,6);
```

```
ownControlBoard=controlBoard.slice(0);
/*
Täytyy luoda lisäksi vastustajan kontrollilauta.
Nyt katsotaan, kuinka paljon vastustaja hallitsee.
*/
color=-color;
copyBoard=board.slice(0);
controlBoard=[];
setControl(9,7,7,5,1,6);
opponentControlBoard=controlBoard.slice(0);
/*
Luodaan oma kuningaslauta.
Mitä lähempänä ruutu on omaa kuningasta, sitä suurempi luku ruu-
dussa on.

16,14,12,10,8,6,4,2,
14,14,12,10,8,6,4,2,
12,12,12,10,8,6,4,2,
10,10,10,10,8,6,4,2,
8,8,8,8,8,6,4,2,
6,6,6,6,6,6,4,2,
4,4,4,4,4,4,4,2,
2,2,2,2,2,2,2,2

Jos kuningas olisi laudan nurkassa, olisi kuningaslauta edellisen
näköinen.
*/
ownKingBoard=[];
for(j=0;j<n*n;j++){
if(board[j]==6*getColor()){
ownKingBoard[j]=n*2;
for(k=1;k<n;k++){
for(l=0;l<=k;l++){
if(j-n*k-l>=0&&(j-n*k-l)%n<=j%n){
ownKingBoard[j-n*k-l]=n*2-k*2;
}else{
break;
}
}
for(l=0;l<=k;l++){
if(j-n*k+l>=0&&(j-n*k+l)%n>=j%n){
ownKingBoard[j-n*k+l]=n*2-k*2;
}else{
break;
}
}
for(l=0;l<=k;l++){
if(j-k-n*l>=0&&(j-k-n*l)%n<j%n){
ownKingBoard[j-k-n*l]=n*2-k*2;
}else{
break;
}
}
for(l=0;l<=k;l++){
if(j-k+n*l<n*n&&(j-k+n*l)%n<j%n){
ownKingBoard[j-k+n*l]=n*2-k*2;
}else{
```

```
break;
}
}
for(l=0;l<=k;l++){
if(j+k-n*l>=0&&(j+k-n*l)%n>j%n){
ownKingBoard[j+k-n*l]=n*2-k*2;
}else{
break;
}
}
for(l=0;l<=k;l++){
if(j+k+n*l<n*n&&(j+k+n*l)%n>j%n){
ownKingBoard[j+k+n*l]=n*2-k*2;
}else{
break;
}
}
for(l=0;l<=k;l++){
if(j+n*k-l<n*n&&(j+n*k-l)%n<=j%n){
ownKingBoard[j+n*k-l]=n*2-k*2;
}else{
break;
}
}
for(l=0;l<=k;l++){
if(j+n*k+l<n*n&&(j+n*k+l)%n>=j%n){
ownKingBoard[j+n*k+l]=n*2-k*2;
}else{
break;
}
}
}
}
}
/*
Tehdään kuningaslauta vastustajalle.
Nyt lauta määräytyy vastustajan kuninkaan sijainnin perusteella.
*/
opponentKingBoard=[];
for(j=0;j<n*n;j++){
if(board[j]==6*-getColor()){
opponentKingBoard[j]=n*2;
for(k=1;k<n;k++){
for(l=0;l<=k;l++){
if(j-n*k-l>=0&&(j-n*k-l)%n<=j%n){
opponentKingBoard[j-n*k-l]=n*2-k*2;
}else{
break;
}
}
for(l=0;l<=k;l++){
if(j-n*k+l>=0&&(j-n*k+l)%n>=j%n){
opponentKingBoard[j-n*k+l]=n*2-k*2;
}else{
break;
}
}
```

```
for(l=0;l<=k;l++){
if(j-k-n*l>=0&&(j-k-n*l)%n<j%n){
opponentKingBoard[j-k-n*l]=n*2-k*2;
}else{
break;
}
}
for(l=0;l<=k;l++){
if(j-k+n*l<n*n&&(j-k+n*l)%n<j%n){
opponentKingBoard[j-k+n*l]=n*2-k*2;
}else{
break;
}
}
for(l=0;l<=k;l++){
if(j+k-n*l>=0&&(j+k-n*l)%n>j%n){
opponentKingBoard[j+k-n*l]=n*2-k*2;
}else{
break;
}
}
for(l=0;l<=k;l++){
if(j+k+n*l<n*n&&(j+k+n*l)%n>j%n){
opponentKingBoard[j+k+n*l]=n*2-k*2;
}else{
break;
}
}
for(l=0;l<=k;l++){
if(j+n*k-l<n*n&&(j+n*k-l)%n<=j%n){
opponentKingBoard[j+n*k-l]=n*2-k*2;
}else{
break;
}
}
for(l=0;l<=k;l++){
if(j+n*k+l<n*n&&(j+n*k+l)%n>=j%n){
opponentKingBoard[j+n*k+l]=n*2-k*2;
}else{
break;
}
}
}
}
}
/*
Lopuksi lasketaan kontrollilautojen ruutujen arvot yhteen.
Oman laudan arvot kasvattavat yhteisarvoa ja vastustajan vähentä-
vät sitä.
Lisäksi ruudun arvolla on kertoimia.
Ruudun arvo kertautuu sen mukaan, mitä lähempänä ruutu on keskus-
taa ja vastustajan kuningasta.
*/
controlValue=0;
for(j=0;j<n*n;j++){
controlValue+=ownControlBoard[j]*squareValues[j]*opponentKing-
Board[j];
```

```
controlValue-=opponentControlBoard[j]*squareValues[j]*ownKing-
Board[j];
}
/*
Seuraava koodi kannustaa sotilaita etenemään ehkä mahdollista ko-
rotusta varten.
Mitä lähempänä oma sotilas on laudan vastakkaista riviä, sitä
enemmän pisteitä.
Jos vastustaja on lähellä korotusta, saadaan miinuspisteitä.
*/
pawnPromotion=0;
for (j=1;j<n-1;j++){
for (k=n*j;k<n*(j+1);k++){
if(board[k]==1){
if(getColor()==1){
pawnPromotion+=n-2-j;
}else{
pawnPromotion-=n-2-j;
}
}else if(board[k]==-1){
if(getColor()==1){
pawnPromotion-=j-1;
}else{
pawnPromotion+=j-1;
}
}
}
}
/*
Viimeiseksi asetetaan jatkolle varsinainen kokonaisarvo, joka on
tässä tapauksessa materiaalin, hallinnan ja korotusmahdollisuuden
summa.
Hallinnalle ja korotusmahdollisuudelle on yritetty antaa sopiva
kerroin, jolloin ne olisi hyvässä suhteessa materiaaliin.
Lopuksi pyöristetään arvo yhden desimaalin tarkkuudelle, jotta sa-
manarvoisia siirtoja olisi vaihtoehtona enemmän.
*/
values[values.length]=Math.round((materialValue+con-
trolValue/10000+pawnPromotion/20)*10)/10;
}
```

**Liite 5. Tyylitiedosto**

```
@charset "utf-8";
/* CSS Document */

* {
          margin:0;
          padding:0;
}
body {
          background-color:#AF784D;
          /*background-color:#AE6631;*/
          /*background-color:#FFFF00;*/
          /*background-color:#404040;*/
          /*background-color:#966652;*/
    /**-webkit-touch-callout: none;
    -webkit-user-select: none;
    -khtml-user-select: none;
    -moz-user-select: none;
    -ms-user-select: none;
    user-select: none;**/
}
.dark{
          background-color:#854d31;
          /*background-color:#6D1B05;*/
          /*background-color:#FF8000;*/
          /*background-color:#3c0000;*/
          /*background-color:#6b4636;*/
}
/*.dark:nth-child(4n-3){
          background-color:#FF0000;
}
.dark:nth-child(4n-2){
          background-color:#00FF00;
}
.dark:nth-child(4n-1){
          background-color:#0000FF;
}*/
.light{
          background-color:#d8a368;
          /*background-color:#EFB15C;*/
          /*background-color:#FFC000;*/
          /*background-color:#cbfffd;*/
          /*background-color:#f3dcba;*/
}
/*.light:nth-child(4n-3){
          background-color:#808080;
}
.light:nth-child(4n-2){
          background-color:#00FFFF;
}
.light:nth-child(4n-1){
          background-color:#FF00FF;
}*/
.white{
```

```
		/*color:#00bbb4;*/
		color:#b2761d;
}
.black{
		/*color:#c30101;*/
		color:#060501;
}
table {
		display:inline-block;
		border-collapse:collapse;
}
td{
		display:inline-block;
		font-family:MS Mincho, sans-serif;
		font-size:56px;
		cursor:default;
		text-align:center;
		width:64px;
		height:64px;
		line-height: 120%;
}
/*td:nth-child(4n-3){
		font-family:Arial, sans-serif;
}
td:nth-child(4n-2){
		font-family:Meiryo, sans-serif;
}
td:nth-child(4n-1){
		font-family:DejaVu Sans, sans-serif;
}*/
span{
		/*display:inline-block;*/
		/*vertical-align:top;*/
		position:relative;
		top:0px;
}
.highlight{
		-webkit-box-shadow:inset 0px 0px 0px 2px #FF0000;
    -moz-box-shadow:inset 0px 0px 0px 2px #FF0000;
    box-shadow:inset 0px 0px 0px 2px #FF0000;
}
#promotion{
		float:left;
}
#board{
		position:relative;
		top:32px;
		left:32px;
		width:512px;
		height:512px;
}
#option{
		position:relative;
		top:64px;
		left:32px;
		width:512px;
		height:64px;}
```