A* ALGORITM*

Ott Vaiknemets

This work is produced by The Connexions Project and licensed under the
Creative Commons Attribution License †

Abstract

A* algoritmi kasutamine lühima tee leidmiseks kaardil asuva kahe punkti vahel.

1 Algoritm "A*"

A* võib olla olla algajaile keeruline. Kuigi veebis on palju artikleid, mis selgitavad A* algoritmi, on enamus neist kirjutatud inimeste poolt kes on kursis põhialustega. Selle artikli sihtgrupp on algajad. Kuigi artikli lõpus on viited näitekoodile C++-s ja Blitz Basic-us, pole see artikkel programmeerimiskeele-spetsiifiline.

1.1 Otsingupiirkond

Oletame et keegi tahab minna punktist A punkti B ja nende kahe punkti vahel on sein. Pildil kujutab roheline punkt alguspunkti A ja punane lõpppunkti B, sinised ruudud nende vahel on sein.

*Version 1.2: Oct 14, 2008 3:40 pm -0500

†http://creativecommons.org/licenses/by/2.0/

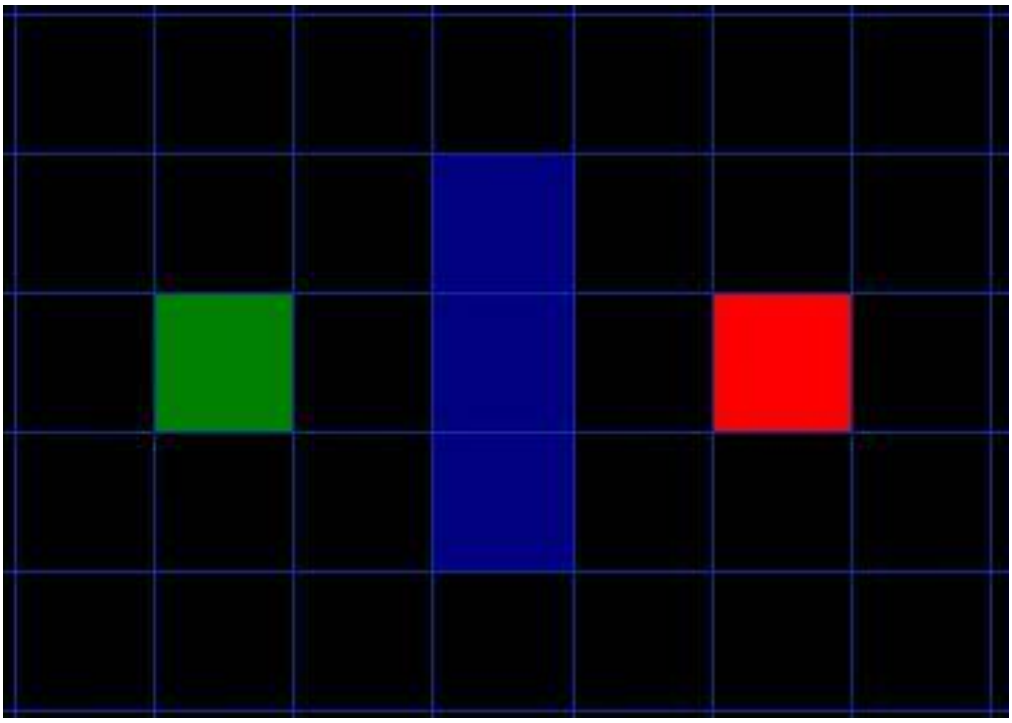

Figure 1

Paneme tähele, et otsingupiirkond on jaotatud ruudustikuks. Otsingupiirkonna lihtsustamine, mida me sellega tegime, on esimene samm teeleidmisel. See konkreetne võte lihtsustab meie otsipiirkonna lihtsaks kahemõõtmeliseks maatriksiks. Iga *ü*ksus selles maatriksis kujutab endast *ü*hte ruutu ruudustikust, mis on märgitud kas "läbitavaks" või "läbimatuks". Tee leidmiseks leitakse ruudud, mis tuleb läbida jõudmaks punktist A punkti B. Tee läbimisel liigutakse *ü*he ruudu keskkohast järgmise ruudu keskele kuni jõutakse sihtkohta.

Neid ruute kutsutakse sõlmedeks (node). Miks mitte kutsuda neid lihtsalt ruutudeks? Sellepärast, et võid piirkonna jaotada ka *ü*kskõik mis muudeks kujunditeks, näiteks kolmnurkadeks. Ja sõlmed ei pea asuma kujundi keskel, vaid võivad olla ka servades või *ü*kskõik kus. Siin kasutatakse ruute, kuna see on lihtsaim.

1.2 Alustamine

Peale otsipiirkonna lihtsustamist hallatavaks arvuks sõlmedeks on järgmiseks sammuks l*ü*hima tee leidmine. Selleks me alustame punktist A naaberruutudele liikumist kuni jõuame punkti B.

See toimub järgmiste sammudega:

1. Lisa alguspunkt A loendisse "avatud". See loendis hoiame ruute mis võivad jääda otsitavale teele ehk see on loend ruutudest, mis vajavad lähemat uurimist. Esialgu on seal vaid 1 ruut - A.
2. Lisa ka kõik naaberruudud, mis on märgitud "läbitavaks" (ehk mis eelmisel joonisel pole sinised), "avatud" loendisse. Iga lisatud ruudu kohta märgi temale eelnevaks ruuduks ruut A. Eelnevate ruutude meeles pidamine on oluline kui tahame hiljem teed läbida. Seda vaatleme lähemalt allpool.
3. Eemalda alguspunkt A "avatud" loendist ja lisa ta "suletud" loendisse. Selles loendis on ruudud mida praegu pole vaja rohkem uurida.

Selleks hetkeks on sul olemas info, mida illustreerib järgnev joonis. Keskmine roheline ruut on alguspunkt, heledama raamiga tähistame, et ta on kantud "suletud" loendisse. Kõik piirnevad ruudud on "avatud" loendis, seda tähistab roheline raam. Iga uurimisel ruut sisaldab viidet eelmisele ruudule, joonisel tähistab seda hall joon mis on suunatud alguspunktile (ehk ruudule, kust jõuti noolega ruudule).

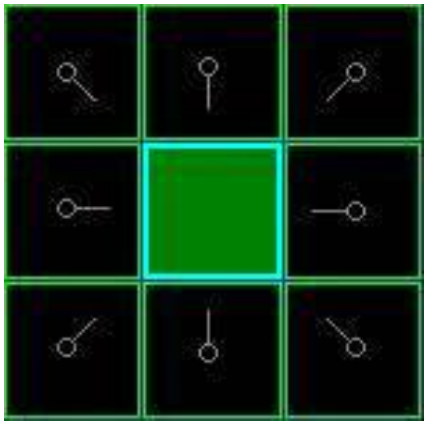

Figure 2

Järgmiseks valime *ü*he ruudu "avatud" loendist ja kordme eelnevaid samme. Valiku tegemiseks leiame "avatud" loendi ruutudele "hinna", selle valem onF = G + HF tähistab hinda, G tähistab praeguseks hetkeks leitud tee pikkust alguspunktist sellesse konkreetsesse punkti, H on oletuslik teepikkus sellest punktist lõpp-punkti B. Seda on tihti nimetatud heuristiliseks, mis võib olla pisut eksitav. Põhjus selle tee nimetamiseks heuristiliseks on, et me ei tea veel tegelikku teepikkust punkti B (me alles otsime seda ja pole jõudnud kõigi takistusteni). On erinevaid võimalusi H arvutamiseks, neist tuleb juttu allpool.Tee leitakse, korrates eelnavalt toodud t*öö*tlussamme, valides järgmiseseks ruuduks "avatud" loendist sellise, mille hind F on väikseim.Nagu *ü*leval *ö*eldud, on G teepikkus alguspunktist antud ruutu. Käesolevas näites määrame igale horisontaal- või vertikaal-suunas liikumise sammule hinnaks 10 ja diagonaal-suunas liikumise sammule 14. Hind 14 tuleneb sellest, et teepikkus *ü*hikruudu diagonaali pidi liikumisel on ruutjuur 2-st ehk ca. 1,414 korda pikem kui horisontaalis või vertikaalis liikudes. K*ü*mnega on need *ü*hikud korrutatud vaid lihtsustamise huvides. (Pealegi arvutab arvuti täisarvudega kiiremini, tee leidmine võib teinekord olla väga t*öö*mahukas ettevõtmine.)Seni läbitud teepikkuse G leidmiseks tuleb eelmise ruudu G-le liita 14 või 10 vastavalt sellele, kas eelmine ruut on diagonaalsuunas või mitte. H leidmiseks on mitmeid võimalusi. Siin me kasutame Manhattani meetodit, kus me leiame sammude kogusumma vertikaalselt ja horisontaalselt liikudes, ignoreerides diagonaalset liikumist ja ignoreerides võimalikke takistusi mis võivad olla teel. Ehk H=x koordinaatide erinevuse ja y koordinaatide erinevuse summa korrutatud 10-ga (k*ü*lgepidi liikumise sammu hind). Nimetus "Manhattani meetod" tulenevat sellest, et taoline loogika kehtib ka ruudustiku-kujuliste kvartalitega linnas liikudes (mil kvartalit ei saa läbida diagonaalselt).

Mida täpsemini me järelejäänud teelõigu pikkust ennustada suudame, seda kiiremini leiab ka algoritm lahenduse. Kui me *ü*lehindame vahemaad, siis pole garanteeritud et lahenduseks on l*ü*him tee. Sel juhul tegeleb meie algoritm "sobimatu heuristikaga" (inadmissible heuristic). Tehniliselt meie näites on Manhattani meetod sobimatu kuna *ü*lehindab pisut järgijäänud distantsi. Kuid me kasutame seda ikkagi kuna on lihtsalt mõistetav ja kuivõrd ta põhjustab vaid väikese *ü*lehindamise. Harvadel juhtudel, kui tulemuseks tagastatav tee ka pole l*ü*heim võimalik, on see kindlasti *ü*ks l*ü*hemaist. Täiendavat lugemist teepikkuse ennustamiseks saab siit[1] .

[1] http://www.policyalmanac.org/games/heuristics.htm

F leitakse G ja H liitmisega. Esimese sammu tulemus on näha järgneval joonisel. F, G ja H tulemused on kirjutatud igasse ruutu. Nagu on näidatud rohelisest parempoolsel ruudul, F on *ü*leval, G all vasakul ja H all paremal.

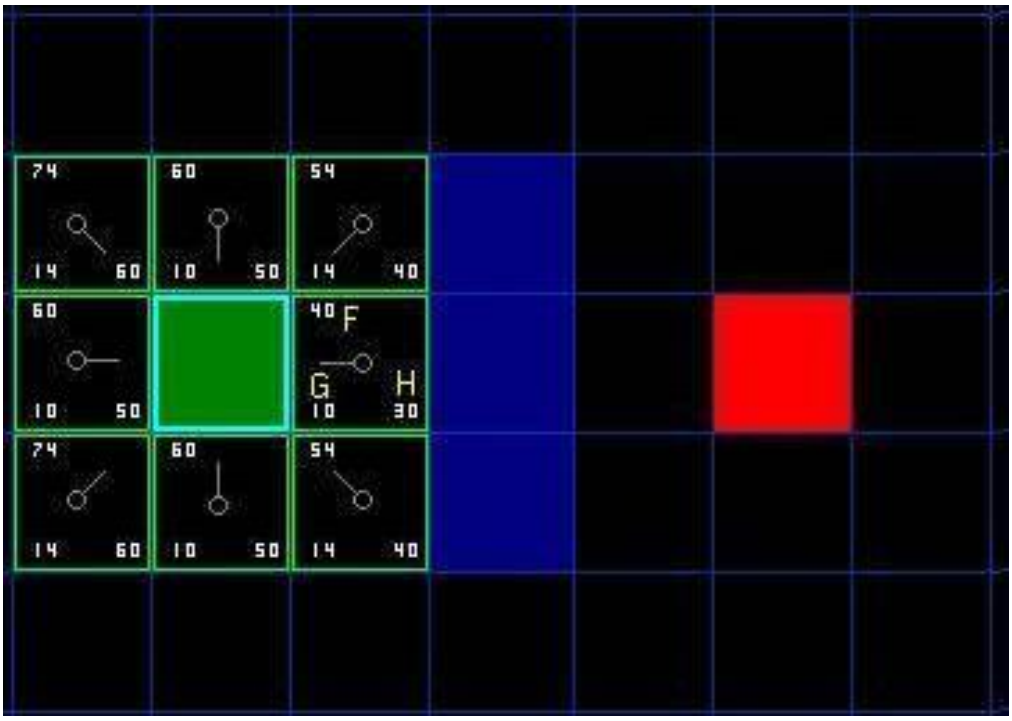

Figure 3

Tähtedega ruudul on G=10 ehk kaugus alguspunktist on 1 horisontaalsamm. Rohelisest alguspunktist diagonaalsuunda jäävatel ruutudel on G=14. H on arvutatud kui eeldatav kaugus punase punktini, kasutades Manhattani meetodit, liikudes ainult horisontaalselt ja vertikaalselt ning ignoreerides ruutude läbitavust. Seetõttu ei olegi rohelisest diagonaalsuunas asuva ruudu H võrdne 34-ga (mis oleks 1 diagonaalsamm + 2 horisontaalsammu punase punktini), vaid 40 (ehk punasesse punkti liikumiseks peaks kuluma 3 horisontaalsammu ja 1 vertikaalsamm). Ja F on lihtsalt G ja H summa.

1.3 Jätkamine

Järgmise sammuga me lihtsalt valime väikseima F-ga ruudu "avatud" loendist, ning:

1. Eemaldame ta loendist "avatud" ja lisame loendisse "suletud".
2. Kõik selle ruudu naaberruudud, mis pole loendites "avatud" ega "suletud" ning pole märgitud mitteläbitavaiks, lisada loendisse "avatud", märkides *ü*htlasi iga*ü*hele eelmiseks ruuduks selle ruudu, millelt siia jõuti (ja mis eelmises sammus tõsteti "suletud" loendisse).
3. Kui mõni naaberruutudest on juba loendis "avatud", siis kontrolli, kas käesoleva ruudu kaudu sinna minnes oleks tee l*ü*hem kui see teepikkus, mis on praegu märgitud sellele ruudule (G). Teisisõnu kontrolli, kas selle naaberruudu G on suurem kui käesoleva ruudu G+sammu pikkus sellele naaberruudulesel juhul muudame ka naaberruudu ruudu G-d, määrates selleks käesoleva ruudu G+sammu pikkus (mis meie näites saab olla 10 või 14) ja märgime muudetud ruudu eellaseks käesoleva ruudu. Samuti arvutame uuesti muudetud naaberruudu F.

Vaatame lähemalt, kuidas see toimib. Meie senivaadeldud *ü*heksast ruudust on meil 8 "avatud" loendis ja algusruut tõstetud "suletud" loendisse. Neist väikseima F-iga on parempoolne keskmine, mille F=40. Seega valime me selle ruudu järgmiseks sammuks. See on märgistatud helesinise raamiga alljägneval illustratsioonil:

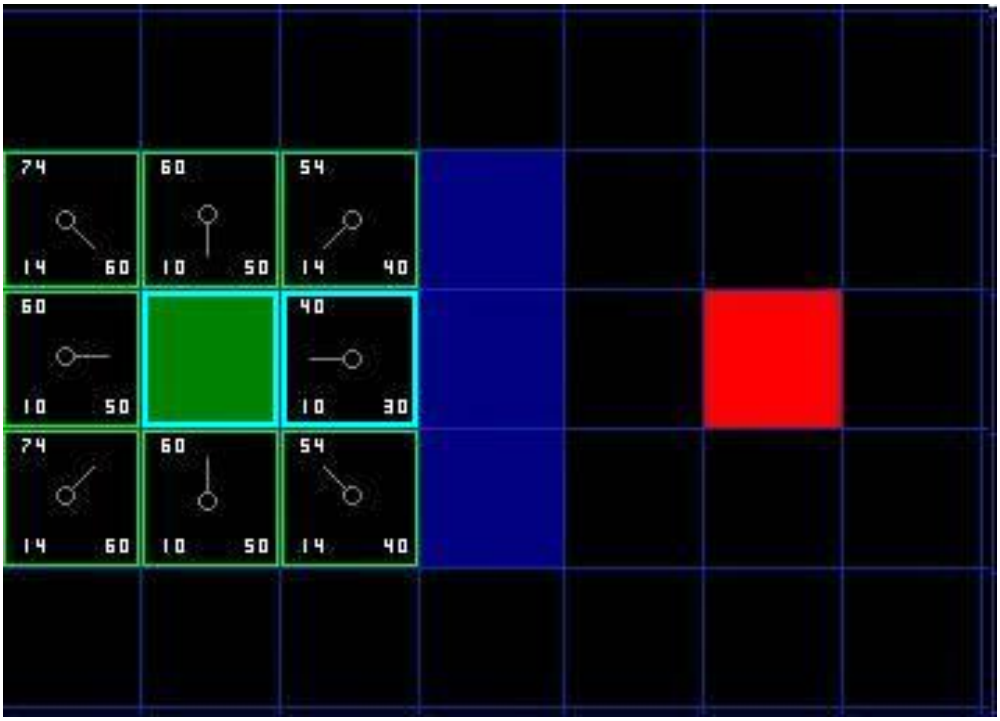

Figure 4

Esiteks me tõstame selle ruudu "suletud" loendisse (joonisel on kõik helesinise raamiga ruudud vaadeldud loendis). Seejärel vaatleme naaberruute. Kõik parempoolsed kujutavad kaardil m*üü*ri ehk pole läbitavad ja seetõttu ignoreerime neid. Vasakul keskel on algusruut, mis on tõstetud "vaadeldud" loendisse, seega ignoreerime ka seda. Ülejäänud neli naabreruutu on juba "vaatlemisel" loendis, seega peame vaid *ü*le vaatama nende teepikkused, et kas mõnda neist on käesoleva ruudu kaudu minnes l*ü*hem tee. Ülemise ruudu G=14. Käesoleva ruudu kaudu liikudes oleks see 20 (käesoleva ruudu G + sammu pikkus = 10 + 10). Seega ei leidnud l*ü*hemat teed. Kordame seda seda protsessi ka *ü*lejäänud 3 naabri korral ja näeme, et *ü*helgi korral ei ole läbi käesoleva ruudu minnes l*ü*hem tee, seega ei muuda me midagi. Võime võtta järgmise väikseima F-iga ruudu avatud loendis. N*üü*d on meil vaatlemisel loendis 7 ruutu, milledest kahel F=54. Võime neist võtta suvalise, kuid kiiruse huvides võib olla kasulikum valida ruut mis lisati viimati "vaatlemisel" loendisse. See põhineb eeldusel, et liikudes algusruudust lõppruudu poole on hiljem vaatlusesse lisatud ruudud suurema tõenäosusega otsitaval teel kui varemlisatud. Kuid ei pruugi olla- näiteks meie kasutatava kaardi korral pole. (Erinevused samahindeliste alade valimisel ongi põhjuseks, miks erinevad A* realisatsioonid saavad leida erinevaid teid.) Valime siis alumise ruudu nagu on näidatud joonisel:

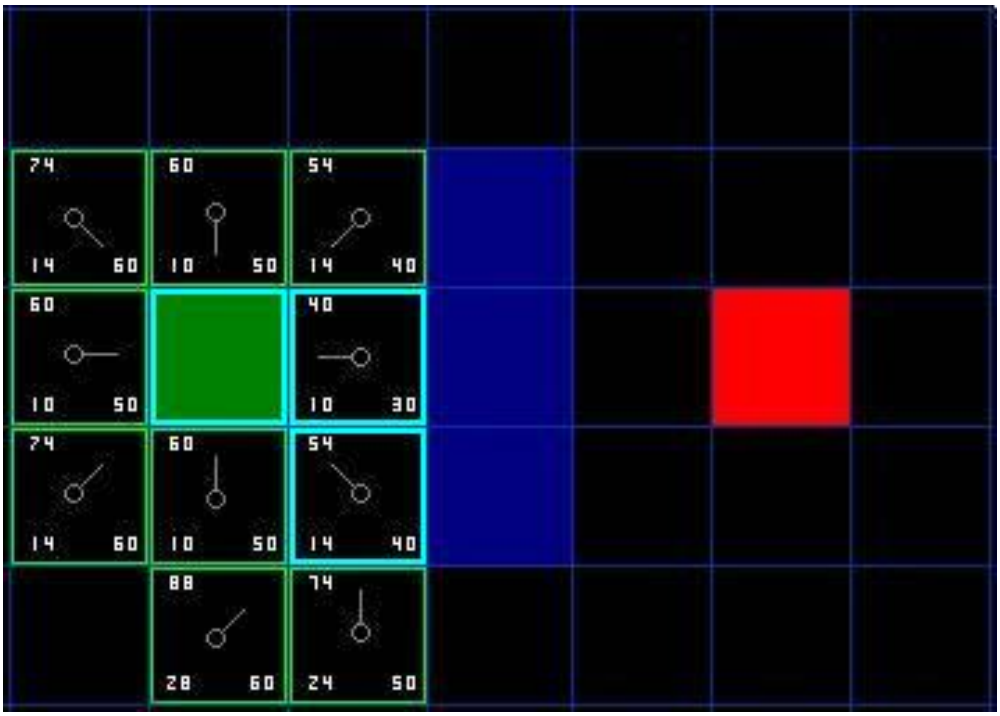

Figure 5

Seekord on 2 naabrit seinad, milliseid ignoreerime. Samuti ignoreerimine ruutu mis on otse m*üü*ri all. Miks? Sellepärast, et sellel ruudule ei saa diagonaalis liikuda ilma lõikamata kõrvalolevat "m*üü*ri". Kõigepealt tuleb liikuda alla ja siis saab liikude sellele ruudule, liikudes *ü*mber seina. (Märkus: see reegel pole kohustuslik. See sõltub eelkõige sellest, kuidas sõlmed on paigutatud.). 2 naaberruutu pole loendisse lisatud, seega lisame nad, määrates jooksva ruudu nende eellaseks. Ülejäänud 3-st ruudust 2 on juba "suletud" loendis, seega neid ei vaatle *ü*ldse, 1 on "avatud" loendis, selle korral veendume, et sealne G on väiksem kui jooksva ruudu kaudu minnes oleks. Sellega on selle ruudu t*öö*tlemine lõppenud ja saame valida järgmise väikseima F-ga ruudu "avatud" loendist.Kordame seda protsessi kuni oleme lisanud "suletud" listi sihtmärgiks oleva ruudu:

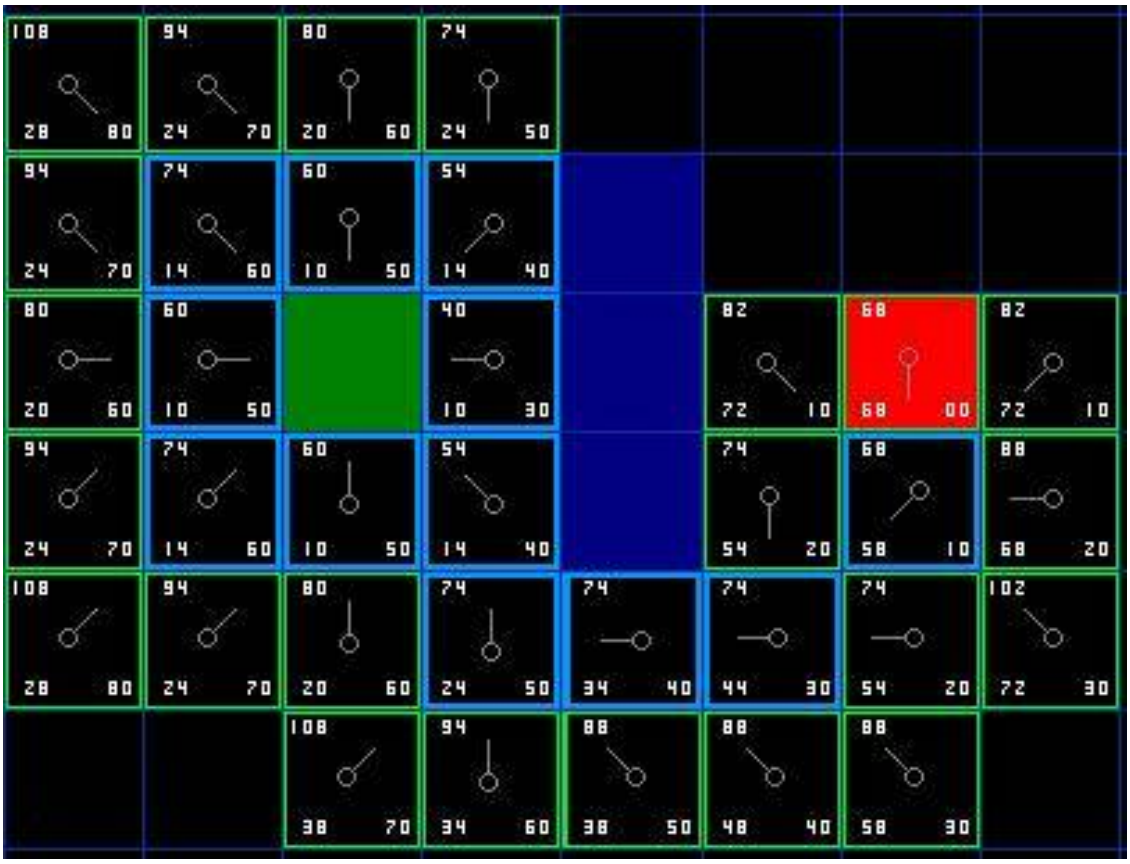

Figure 6

Pane tähele, et algusruudust kaks ruutu allpool oleva ruudu eellane on muutunud võrreldes eelmise joonisega. Enne oli ta G=28, n*üü*d 20. See juhtus siis kui t*öö*deldi ruutu tema kohal (misjärel läks see ruut "suletud" loendisse). Meie näite korral ei oma see suuremat mõju, kuid mõne teistsuguse kaardi korral võib sellisest *ü*mberarvutavusest hargneda uus tee.

1.4 Lõpetamine

Niisiis, kuidas määrata tee? Lihtne, alusta punasest lõppruudust ja liigu eelmise ruudu viiteid pidi tagasi alguspunkti. Läbitud ruudud ongi tee (vastupidises järjekorras), vt järgmist joonist. Liikudes algusruudust A sihtruutu B on lihtsalt liikumine iga ruudu (sõlme) keskpunktist järgmise ruudu (sõlme) keskpunkti kuni jõuame sihtmärgini.

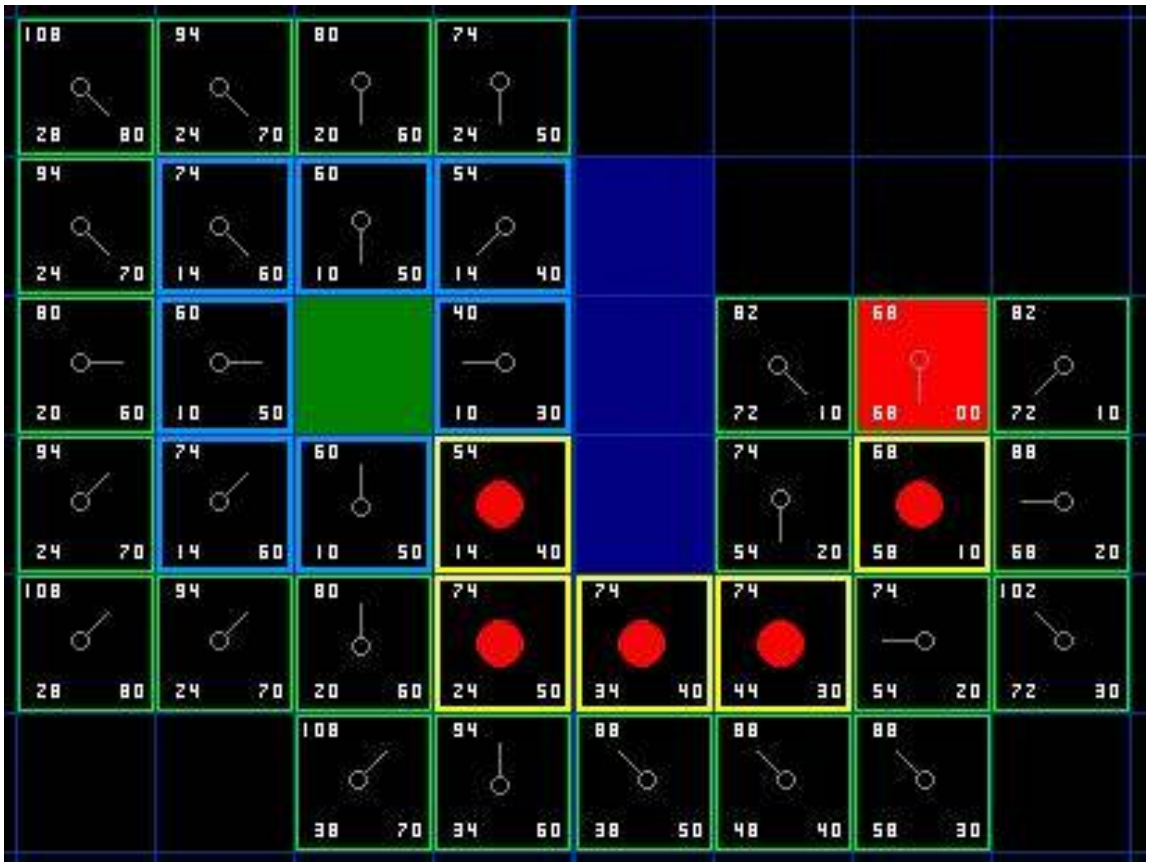

Figure 7

2 Kokkuvõte

N*üü*d uuesti A* algoritm võimalikult lakooniliselt:

1. Lisa algussõlm "avatud" loendisse.
2. Korda järgnevaid samme:
   - Otsi väikseima F-ga sõlm "avatud" loendist. Viitame sellele kui jooksvale ruudule.
   - Tõsta see "suletud" loendisse.
   - Iga jooksva ruudu naabri kohta...
     - Kui see sõlm pole läbitav või on "suletud" loendis siis ignoreeri seda.Vastasel juhul tee alljärgnevat:
     - Kui see pole "avatud" loendis, lisa see sinna. Jooksev ruut märgi selle eellaseks. Arvuta F, G ja H.
     - Kui see on juba "avatud" loendis, siis kontrolli, kas tee jooksva ruudu kaudu oleks l*ü*hem kui praegune tee selle ruuduni. Selleks tuleb vaadata, kas G tuleks jooksva ruudu kaudu minnes väiksem kui on praegune. Kui jah, siis määrata selle sõlme eellaseks jooksev sõlm ja arvutada uuesti G ja F. Juhul, kui hoiad "avatud" listi sorteerituna F-i järgi (kiirendamaks väikseima F-iga ruudu leidmist), sorteeri uuesti.

- Lõpeta kui sa:
  - Lisad sihtmärgi "suletud" loendisse, sel hetkel on tee leitud (vt märkust allpool), või
  - "Avatud" loend on tühi ja sihtmärki pole jõudnud. Sellele juhul teed ei eksisteeri.

3. Salvesta tee. Tee on ruudud, mis läbitakse sihtsülmest algsõlme liikumisel viidete "eelmine ruut" kaudu liikudes.

Märkus: Lõpetades töötlemise hetkel, kui sihtmärk on lisatud "avatud" loendisse on pisut kiirem ja annab peaaegu alati sama tulemuse kui lõpetamisel "suletud" loendisse sattumisel, kuid mitte alati. Tulemus võib sõltuvalt kaardist erineda oluliselt (nt "jõe" korral, kui hakkatakse valet kallast pidi lähenema). Erinevates foorumites toimuvates arutelused A* meetodi kohta võib kohata viiteid programmikoodile mis tegelikult pole A*. A* kindlateks komponentideks on "avatud" ja "suletud" loend ning hinnete F, G ja H kasutamine. On ka palju muid teeleidmise algoritme, kuid üldiselt on A* tunnustatud parimaks. Bryan Stout on arutlenud sel teemal artiklis mis on toodud lõpus, tuues välja nii eelised kui ka puudused. Mõned alternatiivid on paremad teatud situatsioonides, kuid sel juhul peab eelnema kaardile vastav analüüs.

3 Lisakommentaarid

Alljärgnevalt on juttu asjaoludest, millised tuleks rakendust kirjutades läbi mõelda.

3.1 1. Liikuvad objektid (kollisioonidest hoidumine)

Juhul, kui tee leidmisega kaasneb kellegi/millegi liigutamine mööda seda teed, võib olla oluline ka arvestada teiste kaardil toimetavate üksustega. Lisatud näidiskood ignoreerib täielikult muid objekte kaardil. Sisuliselt tähendab see eeldust, et nende olemasolu korral saab teed otsiv üksus ja muud objektid üksteist läbistada / asuda samas ruudus. Sõltuvalt mängust võib see olla aktsepteeritav või mitte. Kui te tahate arvestada teisi objekte teeleidmise algoritmis ja need on liikuvad, siis soovitan arvestada ainult objektidega mis on seiskunud või teed otsiva objekti kõrval sel hetkel kui teed otsitakse (sel juhul tuleks sõlm lugeda läbimatuks kui seal paikneb teine objekt). Liikuvate ja naabruses asuvate objektide korral saate kokkupõrget ära hoida mitte tööldes ruute mis asuvad nende (eeldataval) teel. (Lähemalt kirjeldatud osas #2.)Kui otsustate arvestade teiste objektidega mis liiguvad ega asu teed otsiva (ehk teie juhitava) objekt naabruses, siis peate välja tööta meetodi ennustamaks, kus nad võivad paikneda mingil ajahetkel. Vastasel juhul tekib tulemuseks veider sik-sakiline tee mis arvestab objektide asukohtadega kus neid tegelikult enam pole. Samuti on vajalik välja arendada kokkupõrke tuvastus, vähemalt juhul kui teiste objektide liikumine pole 100% kindel. Kui toimub kokkupõrge tuleb tee uuesti leida või liikuva objekti korral oodata, kuni ta on liikunud mujale.Need näpunäited on piisavad alusamiseks. Täiendava informatsiooni jaoks võib abi olla alljärgnevatest materjalidest:

- Steering Behavior for Autonomous Characters[2] : Craig Reynold'i tööjuhtimisest erineb pisut teeleidmisest, kuid seda saab integreerida teeleidmisega loomaks paremat liikumise ja kokkupõrgetest hoidumise süsteemi.
- The Long and Short of Steering in Computer Games[3] : Huviatv uuring juhtimise ja teeledsise alasest kirjandusest. (pdf-fail)
- Coordinated Unit Movement[4] : Esimene osa kaheosalisest artiklist gruppide formeerimisest ja liikumisest. Autoriks on "Age of Empires" disainer Dave Pottinger.
- Implementing Coordinated Movement[5] : Teine osa Dave Pottinger'i artiklist.

[2] http://www.red3d.com/cwr/steer/
[3] http://ducati.doc.ntu.ac.uk/uksim/uksim'04/Papers/Simon%20Tomlinson-%2004-20/paper04-20%20CR.pdf
[4] http://www.gamasutra.com/features/game_design/19990122/movement_01.htm
[5] http://www.gamasutra.com/features/19990129/implementing_01.htm

3.2 2. Maastiku arvestamine

Käesolevas juhendis kaardil on kasutusel vaid kahte liiki maastikku: läbitav ja läbimatu. Kuid kuidas käituda siis, kui ruut on läbitav, kuid pikema aja jooksul? Sood, mäed, trpid koobastikes jne- näited maastikest, mis on läbitavad, kuid nõuavad seejuures rohkem energiat ja aega kui tasane maastik. Läbimise kulu võib ka väiksem olla- nt kui tee läheb allamäge. See probleem on kergelt lahendatav muutes hinna G arvutamise algoritmi, et ta arvestakse konkreetse sõlme maastikuga ning liidaks või korrutaks G vastava maastiku koefitsiendiga. A* algoritm ei leia tegelikult mitte niivõrd kõige lühemat, kuivõrd kõige "odavamat" teed (lihtsalt tavaliselt on lühim ka "odavaim").Sarnast lähenemist kasutab "mõju kaardistamine" (influence mapping) käigus luuakse täiendav punktisüsteem mida kasutatakse hindamisel. Oletame et meil on kaart koos hulga üksustega mis takistavad mingit teelõiku. Iga kord kui meie juhitav üksus üritab seda läbida, nabitakse ta kinni. Mõju kaardistamisel märgitakse ruudud, kus juhitavaid üksusi kinni on nabitud, täiendava koefitsiendiga. Iga kord kui programm saadab kellegi teele ja ta kinni nabitakse, tõstetakse vastavale ruudu koefitsienti ja muudetakse seekaudu kasutatavat teed.Kolmas võimalus on tõsta nende sõlmede koefitsienti mis on vaenulike üksuste läheduses. Üks A* nõrkusi on, et kui grupp üksusi püüab leida teed sihtpunkti, siis suur hulk üksusi (või suisa kõik, sõltub algpositsioonidest) valib sama tee. Lisades trahviühikuid sõlmedele teiste grupikaaslaste juba valitud sõlmedele hajutab gruppi ja vähendab kokkupõrkeohtu. Tuleb aga jälgida, et suure grupi korral ei läheks trahvikoefitsient liiga suureks (nt kõrgemaks kui nt vaenlase lisatav koefitsient). Samuti tuleks koefitsienti tõsta vaid neil sõlmedel, mis on teed otsiva üksuse lähedal, vältimaks hajutamist grupiliikmetest mis on nagunii eemal. Samuti on mõtet seda "trahvikoefitsienti" arvestade vaid läbimata tee kohta.

3.3 3. Avastamata ala

Mõnes arvutimängus võib täheldada, et arvuti juhitav vastane teab alati milline tee valida, kuigi kaart on veel uurimata. Sõltuvalt mängust võib selline teeleidmine paista ebarealistlik. Õnneks pole see kuigi keeruline probleem. Lahenduseks on "teadaolev läbitavuse" massiiv iga mängija ja arvuti kohta (ehk mitte üksikute objektide jaoks, vaid iga mängijate kõigi üksuste jaoks). Iga massiiv sisaldab informatsiooni mängija poolt avastatud pindade kohta, eeldades, et avastamata piirkond on läbitav seni, kuni pole kogetud vastupidist. Selle lähenemise korral üksused uitavad tupikutesse kuni plats on õpitud. Teised üksused enam järgmiste vigu ei korda.

3.4 4. Ühtlasemad teed

Kuigi A* tagastab lühima ehk odavaima tee, et tagasta ta ühtlaseimat teed. Vaadake teed meie näite lõpptulemusel:

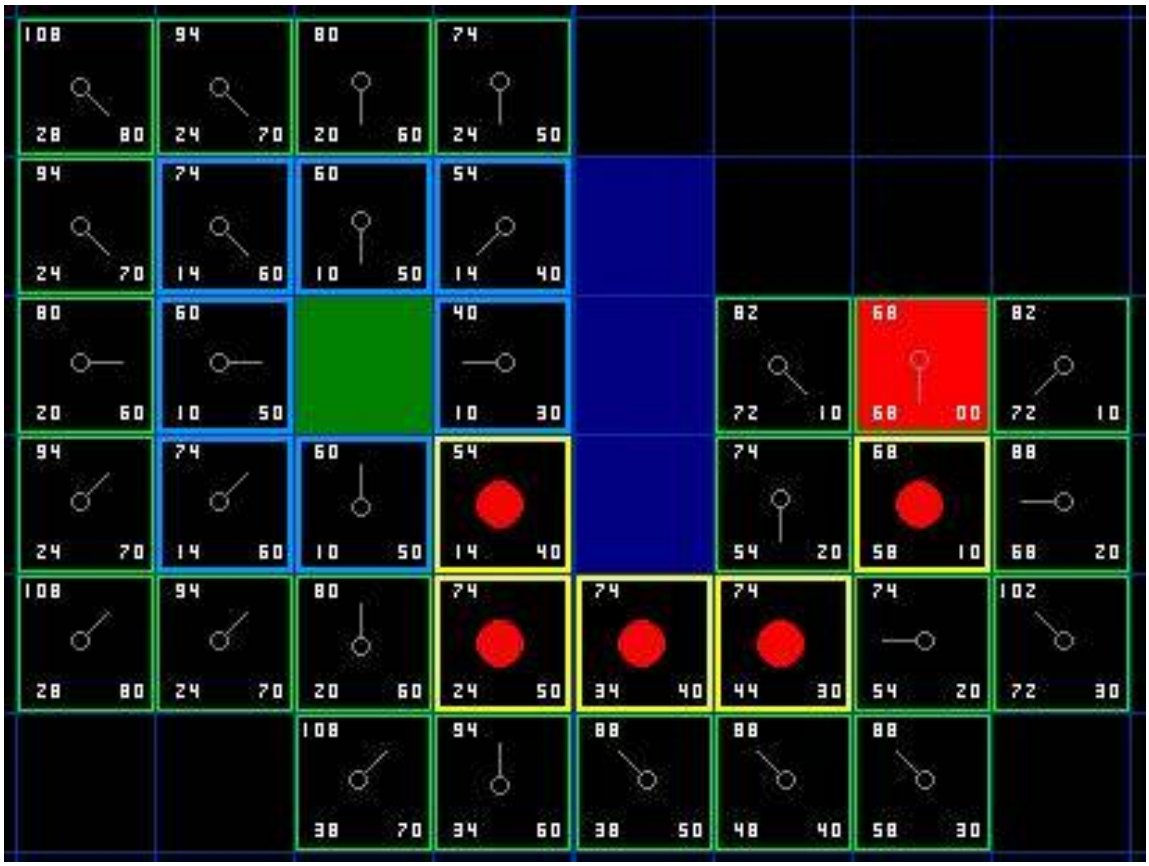

Figure 8

Sellel teel esimene samm on alla paremale. Kas poleks tee tee *ü*htlasem kui esimene samm oleks otse alla? On mitmeid teid selle probleemi menetlemiseks. Arvutades teed võite lisakoefintsiendi lisada teedele mille tulemusena tuleks suunda muuta, lisades selle nende G-skoorile. Või uurida teed peale selle leidmist, leidmaks kohti kus naabersõlme valimisel paistaks tee parem. Lähemalt saab sel teemal lugeda Toward More Realistic Pathfinding[6] , Marco Pinter'i arikkel Gamasutra.com-is (lugemiseks vajalik registreerumine).

3.5 5. Mitte-ruudukujulised sõlmed

Eelnavalt on kasutatud lihtsat 2-mõõtmelist ruudustikku, kuid selle asemel võib kasutada ka muid kujundeid. On olemas lauamäng "Risk", mille kaart kujutab endast tavalist poliitilist maakaarti. Sel juhul saab luua tabeli riikide naabruste salvestamiseks ja G leidmiseks liikumisel *ü*hest riigist teise. Samuti on vajalik leida algritm H leidmiseks. Kõik edasine toimuks sarnaselt eelkirjeldatule, lihtsalt naaberruutude asemel tuleks käsitleda naaberriike. Ehk maatriksi asemel kasutatakse graafi ja seetõttu on naabersõlmede arv, millised iga sõlme t*öö*deldes vaja *ü*le vaadata, erinev. Naabriteks (kellede hulgast lisatakse sõlmi avatud loendisse) on kõik sõlmed, kelledega on otse*ü*hendus (ehk siis naabreid võib olla suvaline hulk, mitte vaid 8 nagu ruudustiku korral).

Sarnaselt saab luua "peatuspunktide" s*ü*steemi teadaolevate teede jaoks kaardil. Nn "peatuspunktid" on *ü*ldiselt teede ristumiskohad, hargnemised, aga ka nt *ü*mberpõike kurvid- ehk kohad, kus saab või on ratsion-

[6] http://www.gamasutra.com/features/20010314/pinter_01.htm

aalne teha valikuid, mis suunas edasi liikuda. Nt tunneli algus- tunneli keskel suunda muuta pole mõttekas, ainult ümber pööra, mis on selgelt ebaratsionaalne ja aja raiskamine. Mängu disainides on tavaliselt kasulik sellised punktid eelnevalt defineerida arvuti juhitava mängija jaoks. Kaks peatuspunkti loetakse naabriteks kui nende vahel saab liikuda sirgjooneliselt (ehk nende vahel pole takistusi). Eelnavalt toodud maakaardi näites tuleks salvestada informatsioon naabruse kohta, seda kasutatakse G ja H arvutamiseks (liites lõikude hindasi/pikkusi peatuspunktide ehk sõlmede vahel). Ehk luuakse graaf, kus sõlmedeks on peatuspunktid, kaarte hinnad on teada (=kaarte pikkused) ning tee leidmisel leitakse tee graafi kahest sõlme vahel, milledest esimene on üksuse lähedal ja teine eesmärgi lähedal.

Amit Patel on kirjutanud ülevaatliku artikli[7] mõnedest alternatiividest.Üht võimalikku viisi isomeetrilise RPG kaardi kasutamisest mitte-ruudustikulisel kaardil on kirjeldatud artiklis[8] kahekihilisest teeleidmisest.

3.6 6. Optimeerimine

Arendades oma A* programmi või kohandades mõnda teist, leiate lõpuks et teeleidmine võib kasutada palju protsessori ressurssi, sõltuvalt teed otsivate üksuste arvust ja kaardi suurusest. Mõned ideed kuidas tööt lust optimeerida:

- Vähenda kaardi suurust või üksuste arvu.
- Teeotsimist rakenda korraga (ühel sammul) vaid väheste üksuste jaoks. Kasuta ootejärjekorda.
- Kasuta suuremaid (seega ka vähem) sõlmi. Veel parem, kasuta kahte kaarti- suuresõlmelist pikemate distantside jaoks ja väiksesõlmelist väiksemates piirkondades või sihtmärgi lähedal tegutsedes. Lähemalt saab sellest tehnikast lugeda siit:Two-Tiered A* Pathfinding[9] .
- Pikemate vahemaade korral võib kasutada eelarvutatud teid ja sõltuvalt sihtmärgi kaugusest neid kasutada.
- Leia eelnevalt kaardi piirkonnad, kuhu ei saa minna, nn "saared". Üks A* nõrkusi on, et kui sihtmärk juhtub olema kohal, kuhu ei saa liikuda, siis tööldakse läbi kogu kaardi pindala selle tulemuse leidmiseks. Tuleks tähistada mingi samasamasuguse tunnusega sõlmed, millede vahel on liikumine võimalik ja A* otsingut täiendada seda arvestavaks.
- Labürindi-taolises keskkonnas kaalu tupikteede märgistamist. Need võivad olla eelnevalt tähistatud kaardi looja poolt või paremal juhul olla tuvastatud kaardi eeltööt luse käigus. Kõik tupikud tuleks tähistada unikaalse numbriga, tööt luse käigus tuleks ignoreerida kõiki tupikuid, välja arvatud juhul kui sihtmärk paikneb sama identifikaatoriga tupikus.

3.7 7. Avatud loendi haldamine

See on üks ajakulukamaid osi A* algoritmis. Iga kord, kui pöördute loendi poole, on vaja leida sõlm väikseima F-iga. On mitmeid teid selle tegemiseks. Võib salvestada sõlmed loendisse nendele sattumise järjekorras ja iga pöördumise ajal vaadata kõik sõlmed üle. See on lihtne, kuid väga aeglane suuremate kaartide korral. Võib ka hoida loendit F järgi järjestatuna ja valida tööt lemiseks esimene sõlm loendist.Tõsised A* programmeerijad kasutavad nn "kahendkuhja" (binary heap). Seda on kasutatud ka lisatud näites. See on keskmiselt ca 2-3 korda tavalistes situatsioonides ja üle 10 korra kiirem suurte kaartide korral. Lähemalt on kahendkuhja kirjeldatud artiklis Using Binary Heaps in A* Pathfinding[10] .Teine võimalik pudelikael on andmestruktuuride haldamine ja uuestiarvutamine teeleidmise vahepeal. Massiivide kasutamine on reeglina märksa kiirem kui dünaamiliste struktuuride. Jõudlusprobleemide korral tuleks hoiduda objekt-orienteeritud lahendustest. Üks võimalus massiivide kasutamiseks: Kahemõõtmeline massiiv Loend(x,y), mis määrab iga sõlme avatud või suletud loendis olevaks. Peale teeledimisprotsessi lõppu pole vaja aega raisata massiivi nullimiseks, vaid piisab kasutatavate väärtuste ümberdefineerimisest, näiteks suurendades avatud loendis olemise või suletud

[7] http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html
[8] http://www.policyalmanac.org/games/twoTiered.htm
[9] http://www.policyalmanac.org/games/twoTiered.htm
[10] http://www.policyalmanac.org/games/binaryHeaps.htm

loendis olemise koodi +5 võrra. Hindade F, G ja H väärtusi pole vaja siis samuti nullida, need lihtsalt kirjutatakse t*öö*tluse käigus *ü*le ja vanade väärtuste kasutamise välistab loendis olemise tunnuse uue väärtuse kasutamine. Ainus miinus massiivide kasutamisel on suurema hulga mälu kasutamine.

3.8 8. Dijkstra algoritm

Tee leidmise jaoks on kasutusel veel ka Dijkstra algoritm. See sarnab A*-le, kuid erinevuse seisneb heuristika puudumisest- ehk väärtust H ei leita. Seega laieneb otsimine võrdse kiirusega igasse suunda, muutes otsiala märksa suuremaks kui tee leidmiseks vaja ning seetõttu on ka reegline aeglasem A*-st. See algoritm osutub kasulikuks siis, kui me ei tea, kus sihtmärk asub. Kui eesmärgiks on mitme sihtmärgi seast lähimani jõudmine, siis osutub Dijkstra algoritm paremaks. A* korral tuleks korrata tee otsimist iga sihtmärgi jaoks ja valida tulemustest l*ü*him, Dijkstra algoritmi kasutades saab lähimani esimese korraga.

4 Täiendavat kirjandust

- Näidiskood: A* Pathfinder (2D) Version 1.9[11]

Blitz Basicu demo on kättesaadav Blitz Basic[12] -u veebilehelt.

- Amiti A* leht[13] ja Amiti arvamused[14] .
- Smart Moves: Intelligent Path Finding[15] : Delfis kirjutatud näide A* ja mõnede alternatiivide kasutamisest. Lugemiseks vajalik registreetumine.
- Terrain Analysis[16] : Veidi keerukam artikkel maastiku anal*üü*simisest ja *ü*ldisemalt AI ninge teeleidmise kontseptsioonist. (Selle konkreetse dokumendi saab miskipärast ka registreerimata käätte...)

Veebisaidid, kus veel mitmesuguseid asjassepuutuvaid materjale saadaval:

- aiGuru: Pathfinding[17]
- Game AI Resource: Pathfinding[18]
- GameDev.net: Pathfinding[19]

[11] http://www.policyalmanac.org/games/AStar.zip
[12] http://www.blitzbasic.com/
[13] http://www-cs-students.stanford.edu/∼amitp/gameprog.html#Paths
[14] http://theory.stanford.edu/∼amitp/GameProgramming/
[15] http://www.gamasutra.com/features/19970801/pathfinding.htm
[16] http://www.gamasutra.com/features/gdcarchive/2000/pottinger.doc
[17] http://www.aiguru.com/pathfinding.htm
[18] http://www.gameai.com/pathfinding.html
[19] http://www.gamedev.net/reference/list.asp?categoryid=18#94