CS-A1121 Projektityön dokumentti

Henkilötiedot:

Henrik Mäntylä
788429
AUT
05/05/2022

Yleiskuvaus:

Projektityön aiheena on labyrintti. Toteutus on graafiseen ikkunaan piirtyvä weave tyyppinen labyrintti. Ohjelman käynnistyessä käyttäjältä aluksi kysytään, minkä kokoisen labyrintin hän haluaa, kuinka paljon mahdollisia weave tyyppiseen labyrinttiin kuuluvia tunneleita hän haluaa ja haluaako hän aktivoida ansat. Tunneleiden määrään vaikuttavat vaihtoehdot ovat: ei yhtään, muutama, paljon ja niin paljon kuin mahtuu. Nämä valinnat tehtyään ohjelma rakentaa pelaajalle hänen haluamansa labyrintin. Tämän jälkeen pelaaja voi aloittaa labyrintin ratkomisen. Maali on kuvattu pelaajalle vihreänä neliönä, sinne päästyään peli päättyy. Pelaaja voi kesken pelin valita myös luovuttaa pelin, jolloin tietokone liikuttaa pelaajan maaliin. Lopuksi tulostetaan vielä pistelista. Halutessaan pelaaja voi myös tallentaa kyseisen labyrintin tekstimuodossa tiedostoon.

Mielestäni työ täyttää vaativan toteutuksen ehdot. Toteutuneet ehdot selostettu alla.

Helppo:
Kaikkien vaatimusten täyttyminen löytyy edellisestä kappaleesta.

Keskivaikea:

- Graafinen alusta: on toteutettu
- Labyrintin koko on vapaasti annettavissa: on toteutettu
- Hiiren liikuttaminen vaivatonta: on toteutettu, käytetään WASD näppäimiä.
- Yksikkötestit: on toteutettu muutamalle tärkeälle funktiolle.

Vaikea:

- Ratkaisu- ja luontialgoritmien helppo lisääminen: on mahdollista, molemmat löytyvät omista luokistaan, joten ne saa tarpeen vaatiessa helposti vaihdettua uuteen.
- Hiouttu pelimäisyys: Graafinen alusta mahdollistaa selkeän labyrintin piirtämisen. 2D-labyrintti pohja on hyvin selkeä. Tunneleiden lisääminen tekee näkymästä hiukan hankalamman ymmärtää, mutta jos niiden määrä ei mene aivan överiksi asiat pysyvät selkeinä. Pelaaja ja maali erottuvat selvästi ja pelaajan liikuttaminen WASD näppäimistöllä tuntuu mahtavalta. Kokoa, tunneleiden määrä ja ansoja lisäämällä pelistä saa myös oikeasti haastavan, jollei lähes mahdottoman ratkaista, mikä tekee pelistä todella hauskan ja palkitsevan pelata, jos haluaa haastetta.

- Labyrintin luonti- ja ratkaisualgoritmien pitää olla tehokkaita: Tähän kiinnitetty paljon huomiota ja muokattu koodia useasti päästäksemme siihen missä nyt ollaan. Suurin rajoittava tekijä on tällä hetkellä ratkaisualgoritmissä vastaan tuleva pythonin sisäänrakennettu rekursiosyvyys. Se estää yli 200x200 kokoisten labyrinttien rakentamisen. Saamme kuitenkin rakennettua tätä suurempia labyrinttejä helposti kommentoimalla Graphics.py:n rivit 47-49. Tällöin vain ratkaisualgoritmi ja tilastot eivät toimi johtuen samasta ongelmasta eli rekursiosyvyydestä. Tämän jälkeen labyrintin kokoon vaikuttaa vain odotusaika. Kruskal ja labyrintin piirrosta vastaavat algoritmit ovat hyvin tehokkaita, joten 2D-labyrinteissä koko luokan 2000x2000 ympärillä alkavat rakentamisajat olla hiukan liian pitkiä (omien testien perusteella 2000x2000 rakentaminen kestää noin 90 sekuntia). Toinen pullonkaula, mikä tulee vastaan luonti tehokkuudessa on add_crossing funktio, joka vastaa tunneleiden lisäämisestä labyrinttiin. Ongelmia rupeaa ilmentymään, jos halutaan paljon tunneleita. Funktion ongelmana on pythonin sisäänrakennetun remove käskyn käyttäminen, jota joudumme kutsumaan 5 kertaa jokaisessa loopissa. Joten jos halutaan mahdollisimman paljon tunneleita labyrintin kokoa ei kannata suurentaa 350x350 yli (rakentamisaika jossain minuutin paikkeilla). Muistin suhteen ei pitäisi olla mitään ongelmia ohjelma on kuitenkin hyvin pienikokoinen. Funktioiden aikakompleksisuuksien analoimiseen käytin cProfile kirjastoa. Yhteenvetona 200x200 labyrintit ovat maksimi loppupalautuksessa, näissä kaikki toimivat sulavasti. Rakentaminen maksimimäärällä tunneleita kestää pari sekuntia ja ratkaisualgoritmi löytää ratkaisun silmänräpäyksessä. Tätä suurempaa labyrinttiä ei oikeastaan edes tarvita sillä 200x200 ansoilla ja tunneleilla vie ihmiseltä tunteja ratkaista.
- Kokoa ei ole rajoitettu näkyvään alueeseen: Puolittain toteutunut. Labyrintti voi koosta riippuen jatkua myös näkyvän alueen ulkopuolelle, mutta se ei liiku pelaajan mukana WASD näppäimillä, vaan käyttämällä reunoilla olevia palkkeja hiirellä. Ratkaisualgoritmin ollessa käynnissä palkit toimivat hiukan viiveellä, mutta pelaajan seuraaminen ei kuitenkaan ole hankalaa.
- Järkevä pisteytys ja tuloslista: Kun pelaaja pääse maalin, joko omilla voimilla tai tietokoneen avustuksella, hänelle tulostetaan tuloslista, josta löytyy siirtojen määrä, siirtojen määrä kohti oikeaa ratkaisua, lähtöpisteestä maalin paras mahdollinen tulos, ratkaisualgoritmin käyttämien siirtojen määrä sekä aika.

Käyttöohje:

Peli käynnistetään ajamalla main funktio. Ensimmäisenä tulevaan ikkunaan annetaan jokin positiivinen kokonaisluku, joka vastaa labyrintin kokoa (esim. antamalla 10 muodostetaan 10x10 labyrintti), ja painetaan ok. Maksimi koko on 200, kommentoimalla koodista Graphics.py rivit 47-49 saadaan myös suurempia.

Sitten tulevasta valikosta valitaan tunneleiden määrä.
“None” = perinteinen 2D-labyrintti
“A few” = Weave tyyppinen labyrintti jossa muutama tunneli, paras pienikokoisille labyrinteille.
“Many” = Weave, jossa tunneleiden määrä jotain “A few” ja maksimin väliltä, hyvä vaihtoehto kaikenkokoisille labyrinteille, jos halutaan ansoja mukaan.

“As many as possible” = Niin paljon tunneleita kuin mahtuu, hyvä vaihtoehto kaikenkokoisille labyrinteille, jos ei haluta ansoja.
Seuraavaksi tulee kysymys ansoista. Vastaamalla kyllä labyrinttiin piirtyy tunneleita, jotka eivät toimi, täten labyrinteistä tulee hyvin vaikeita ratkaista. Vastaamalla ei kaikki tunnelit toimivat normaalisti.

Tämän jälkeen rakennetaan labyrintti, jossa pelaaja sijoitetaan keskelle ja maali jonnekin reunalle. Peliä voi pelata WASD näppäimillä. Pelaaja voi myös halutessaan luovuttaa painamalla “Show solution” näppäintä jolloin algoritmi lähtee viemään pelaaja kohti maalia. Labyrintin voi tallentaa tiedostoon nimeltä “CS-A1121_Lab.txt” painamalla “Save labyrinth to a file” näppäintä, jolloin Minimum spanning tree, joka kuvaa labyrinttiä tallentuu. Peli päättyy ja tulostaa tuloslistan, kun pelaaja saavuttaa maalin.

Ulkoiset kirjastot:

Sallittujen kirjastojen lisäksi käytämme:

Kirjastoa random: Useaan otteeseen labyrintin rakentamisvaiheessa, jotta saamme joka kertaa erilaisen labyrintin. Erityisesti kohdissa, kun määritellään tunneleiden paikat, verkon linkkien painoarvot sekä maalin paikka.

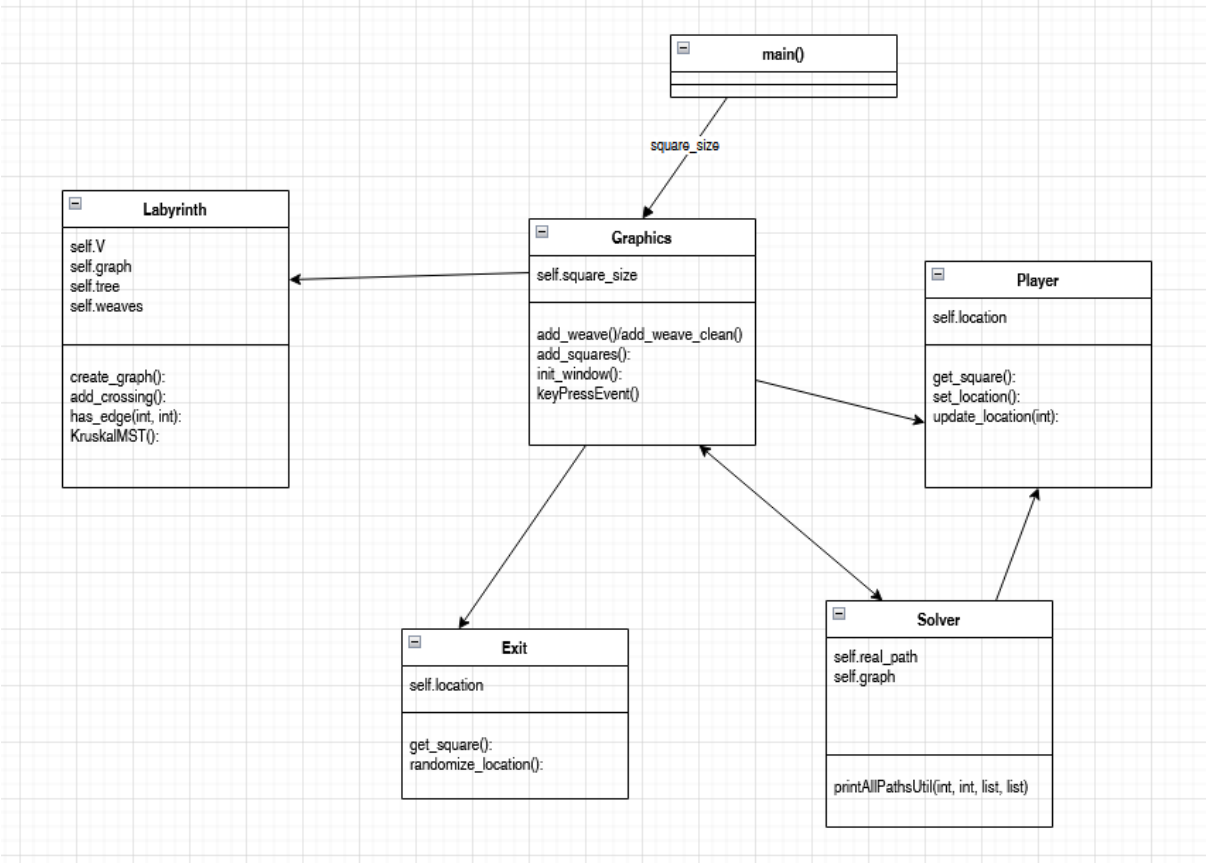
Kirjastosta math funktiota sqrt: Ei oleellinen osa ohjelmaa. Voisimme poistaa, parissa kohtaa kätevä vaihtoehto.

Kirjastoa time: Mittaamaan kokonaisaikaa tuloslistaan, sekä ratkaisualgoritmissä for-looppiin hidasteeksi, jotta pelaajaa on mahdollista seurata kun se etenee kohti ratkaisua.

Kirjastosta collection, defaultdict: Ratkaisualgoritmissä helpottamaan sanakirjan muodostamisessa ja käytössä.

Lisämainintana käytetty kirjastoa cProfile tarkastaessa funktioiden aikakompleksisuuksia, täten helpottaen algoritmien säätämistä. Ei löydy enään koodista.

Ohjelman rakenne:



Kuvasta näkyy selkeästi ohjelman rakenne viisi luokkaa plus main-funktio ja yksikkötestit (ei kuvassa).

Main-funktiosta käynnistetään ja sammutetaan ohjelma. Se kutsuu Graphics luokkaa ilmoittaen sille halutun yhden ruudun koon.

Graphics on taas koko ohjelman sydän. Siellä piirretään pelaaja, maali ja labyrintti graafiselle alustalle, sekä huolehditaan myös kyselyistä, tuloslistasta ja pelaajan liikkeistä. Sen päämetodeja ovat:

- add_weave/add_weave_clean, jotka huolehtivat tunneleiden graafisesta piirrosta. Erona näillä kahdella mikäli halutaan piirtää ansoja vai ei.
- add_square vastaa itse labyrintin piirrosta.
- init_window ja input_windows vastaavat ikkunoiden piirtämisestä.
- keyPressEvent vastaa pelaajan liikkumisesta.

Grafiikka luokka, alustaa Labyrintti luokan sekä kutsuu sen metodeja useaan otteeseen esimerkiksi pelaaja liikutellessa. Labyrintti luokka vastaa itse labyrintin rakentamisesta ja sen ylläpitämisestä. Sen päämetodeja ovat:

- KruskalMST, jossa on suurin osa Kruskalin algoritmistä joka muodostaa verkosta MST:n, joka kuvaa meidän labyrinttiä.
- add_crossing alustaa tunnelit verkkoon ennen Kruskalia
- has_edge kertoo mikäli kahden ruudun välillä on yhteys, tärkeä esimerkiksi pelaajan liikkuessa

- create_graph rakentaa meille halutun kokoisen verkon, josta voimme lähteä rakentamaan labyrinttiä.
- save_to_file tallentaa labyrintin tiedostoon.

Grafiikka luokka alustaa pelaajan ja sijoittaa hänet labyrinttiin. Player luokan päävastuulla on pelaajan lokaation yllä pitäminen. Grafiikka luokka kutsuu pelaaja useasti liikuttaessaan sitä ruudulla. Sen päämetodeja ovat:

- get_square kertoo ruudun numeron, jossa pelaaja on.
- set_location asettaa pelaajan keskelle labyrinttiä
- update_location päivittää pelaajan lokaation haluttuun ruutuun.

Exit luokka on hyvin samankaltainen Player luokkaan verrattuna. Graphics hoitaa sen alustamisen ja sijoittamisen ruudulle. Exit pitää pääsääntöisesti yllä omaa lokaatiotaan. Exitin päämetodeja ovat:

- get_square sama kuin Playerilla kertoo ruudun missä Exit sijaitsee
- randomize_location muuntaa Exitin lokaation jonnekin labyrintin reunalle.

Viimeisenä muttei vähäisimpänä Solver-luokka, jota tarvitaan kun halutaan tietää oikea ratkaisu tai halutaan siirtää pelaaja maaliin. Päämetodeja ovat:

- printAllPathsUtil rekursiivin funktio, joka etsii Depth-first search (DFS) algoritmilla ratkaisua funktioon.
- printAllPaths alustaa yllä olevan metodin sekä huolehtii luovuttaessa pelaajan liikuttamisesta kohti maalia.

Algoritmit:

Koodissa on monia algoritmeja tässä kuvaus oleellisimmista.
Aloitetaan tunnetuista algoritmeistä.

Kruskal:
Muodostaa painotetusta verkostamme pienimmän viritys puun (Minimum Spanning Tree, MST), joka kuvaa labyrinttiä. Aloittaa lisäämään ruutujen välisiä linkkejä puuhun niiden välisten painoarvojen perusteella pienimmästä lähtien. Pitää kirjaa siitä mitä on jo lisätty puuhun, joten looppeja ei muodostu. Lopettaa kun kaikki ruudut löytyvät puusta.

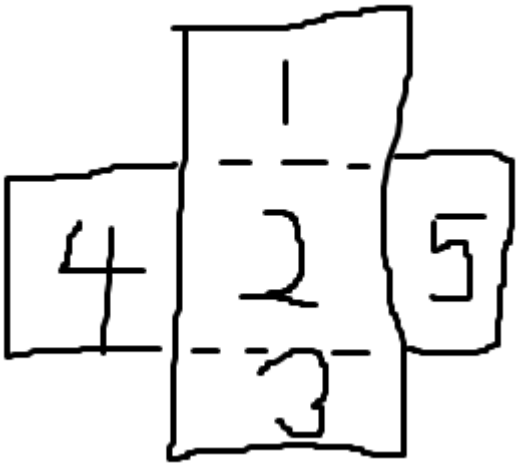
Muita vaihtoehtoja olisi ollut esimerkiksi Primin tai Boruvkan algoritmit, jotka hoitavat saman asian. Aluksi implementoin primin mutta, jälkeenpäin totesin sen aikakompleksisuuden (O(V^2)) olevan liian hidas tarvitsemaamme käyttöön, joten vaihdoin Kruskaliin, jonka aikakompleksisuus on O(E log V), joissa E on linkit ja V nodit.

DFS:
Ei missään nimessä paras vaihtoehto ratkaisualgoritmiksi, mutta meidän tapauksessa riittävä. Ideana on että, lähtee jotain puun haara eteenpäin, kunnes löytää maalin tai ei pysty enään etenemään. Tässä tapauksessa peruuttaa taaksepäin, kunnes löytyy taas uusi haara missä ei olla käyty. Pitää yllä kirjaa vierailluista ruuduista ja heti maalin löydettyään tallentaa siihen johtavan reitin listaan. Parempia vaihtoehtoja olisi ollut vaikka BFS. Syy DFS:n valintaan oli se, että löysin siihen hyvän esimerkin ja sain helposti lisättyä sen ohjelmaan, ennen kuin myöhemmin tajusin ongelman sen käytön kanssa. Toisaalta DFS on meidän

tapauksessa riittävä sillä, ongelmia sen rekursiosyvyyden kanssa alkaa tulemaan vasta tarpeettoman suurilla labyrinteillä.

Sitten kuvaus algoritmeistä, jotka olen itse rakentanut alusta saakka.
Tunneleiden lisäämiseen sain inspiraatiota [2]. Ideana on, että tunnelit lisätään etukäteen labyrinttiin. Omassa koodissani tämä toteutuu puolittain sillä teemme kyllä mahdollisia tunneleita labyrinttiin, mutta annamme Kruskalin päättää mitä niistä lopulta käytetään. Tällä tavalla saamme myös mahdollisia ansa tunneleita labyrinttiin. Labyrintti luokassa oleva metodi add_crossing, joka vastaa tunneleiden tekemisestä toimii seuraavalla tavalla.

add_crossing:
Koska lisäämme tunnelit etukäteen, voimme samalla määritellä suurinpiirtein montako haluamme labyrinttiin. Täten algoritmi määrittelee aluksi looppien määrän käyttäjän toiveen mukaan. Seuraavaksi määrittelemme mitkä ruutujen ali voimme tehdä tunneleita. Näitä ovat kaikki muut paitsi uloimmilla riveillä ja sarakkeilla olevat ruudut, teemme näistä listan, jonka avulla voimme pitää kirjaa mihin ruutuun on mahdollista tehdä tunneli. Teemme myös toisen listan, johon lisäämme kaikki ne linkit, jotka haluamme lopuksi poistaa syntyneiden tunneleiden takia. Sitten lähdemme muodostamaan itse tunneleita. Valitsemme satunnaisesti listasta jonkun ruudun sekä tunnelin suunnan (ylhäältä alas tai vasemmalta oikealle).



Seuraavaksi muodostamme itse tunnelin. Kuvassa esimerkki miltä lopputulos, tunneli vasemmalta oikealle näyttää. Eli jos haluamme numeron 2 kohdalle tunnelin vasemmalta oikealle, lisäämme poistettavien listalle verkosta ruudun 2 linkit ruutuihin 4, 5, 1, ja 2. Tämän jälkeen linkitämme ruudut 4 ja 5 toisiinsa. Ruudut 1, 2 sekä 2, 3 linkitämme normaalia pienemmällä painoarvolla jolloin Kruskal valitsee ne ja saadaan ehjiä risteyksiä. Lopuksi poistamme kaikki viisi ruutua listasta, joka pitää yllä mahdollisia tunneleiden paikkoja. Jatkamme tunneleiden tekemistä kunnes labyrinttiin ei mahdu enään yhtään (eli mahdollisten tunneleiden lista on tyhjä) tai olemme saavuttaneet halutun määrän looppeja. Ihan viimeiseksi poistamme verkosta kaikki poistolistalla olevat linkit. Suurin ongelma algoritmissa on se, että se kutsuu python remove funktiota jokaisella loopilla 5 kertaa, vaikka kaikki ruudut eivät enään olisi listassa jossa ovat mahdolliset tunneleiden lokaatiot. Jos tietorakenteemme tietäisi montako kertaa remove kutsua täytyisi käyttää aina uuden ruudun kohdalla, paljon turhia kutsuja jäisi käymättä ja algoritmista tulisi huomattavasti nopeampi.

add_weave:
Toinen Graphics-luokassa olevista tunneleiden piirto algoritmeista. Tämä piirtää myös ansoiksi tarkoitetut tunnelit. Algoritmi käy läpi add_crossing luokassa tehtyä listaa, johon on tallennettuna yhteen muuttujaan ruutu, jossa tunneli on sekä sen tyyppi (vas-oik, yl-al). Tässä listassa on kaikki etukäteen verkkoon tehdyt risteykset, josta Kruskal valitsee osan joista tulee tunneleita labyrinttiin. Täten meille syntyy ansoja. Listassa olevan ruudun numeron perusteella algoritmi laskee sen lokaation labyrintissä kaavalla, joka on useaan otteeseen käytössä ohjelmassamme.

x = num % rivin koko
y = num // rivin koko

Pythonissa operaattori // antaa meille alempaan kokonaislukuun pyöristetyn numeron, jonka avulla saamme tietää millä rivillä numero löytyy. Operaattori % vastaavasti laskee jakolaskusta jäävä jakojäännöksen, jonka avulla voimme sanoa monesko alkio ruutu on kyseisellä rivillä vasemmalta lähtien. Näiden tietojen avulla saamme algoritmissa piirrettyä tunnelin suu- ja peräaukon, joita kuvaa neljä mustasta valkoiseen haalenevaa viivaa.

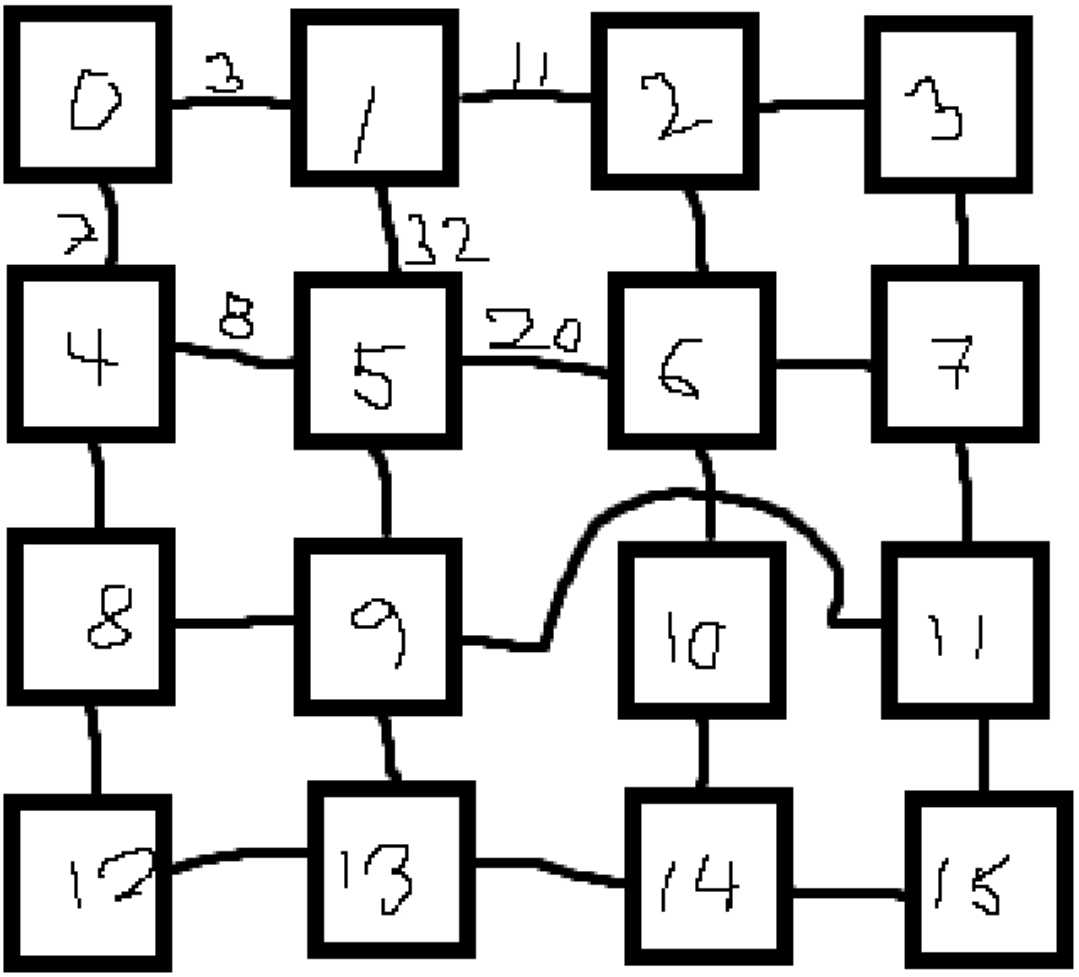
add_weave_clean:
Vastaa myös tunneleiden piirtämisestä, mutta sitä kutsutaan vain jos ei haluta ansoja. Idea on muuten sama, mutta emme katso tunneleiden lokaatiota ja tyyppiä valmiista listasta vaan päättelemme ne MST:n linkkien perusteella. Käymme MST:n läpi ja jokainen linkki joka liittää yhteen ruutuja joiden etäisyys numeroissa on joko 2 tai 2 kertaa yhden rivin koko, on tunneli. Piirrämme näiden kohdalle viivat ja saamme nyt labyrintin, jossa ei ole yhtään ansa tunneleita.

add_squares:
Vastaa itse 2D-labyrintin piirtämisestä. Aluksi se piirtää neliön jokaisen ruudun kohdalle. Sitten piirrämme mustan reunaviivan päälle valkoisen viivan, jos ruudut ovat linkattuina toisiinsa. Käyttää samaa ideaa kuin add_weave_clean eli käy MST:n läpi ja jos linkkien ruutujen välinen etäisyys on 1 tai rivien koko, piirretään valkoinen viiva, joko vasempaan tai alareunaan. Täten saamme hyvin selkeän näköisen 2D-labyrintin.

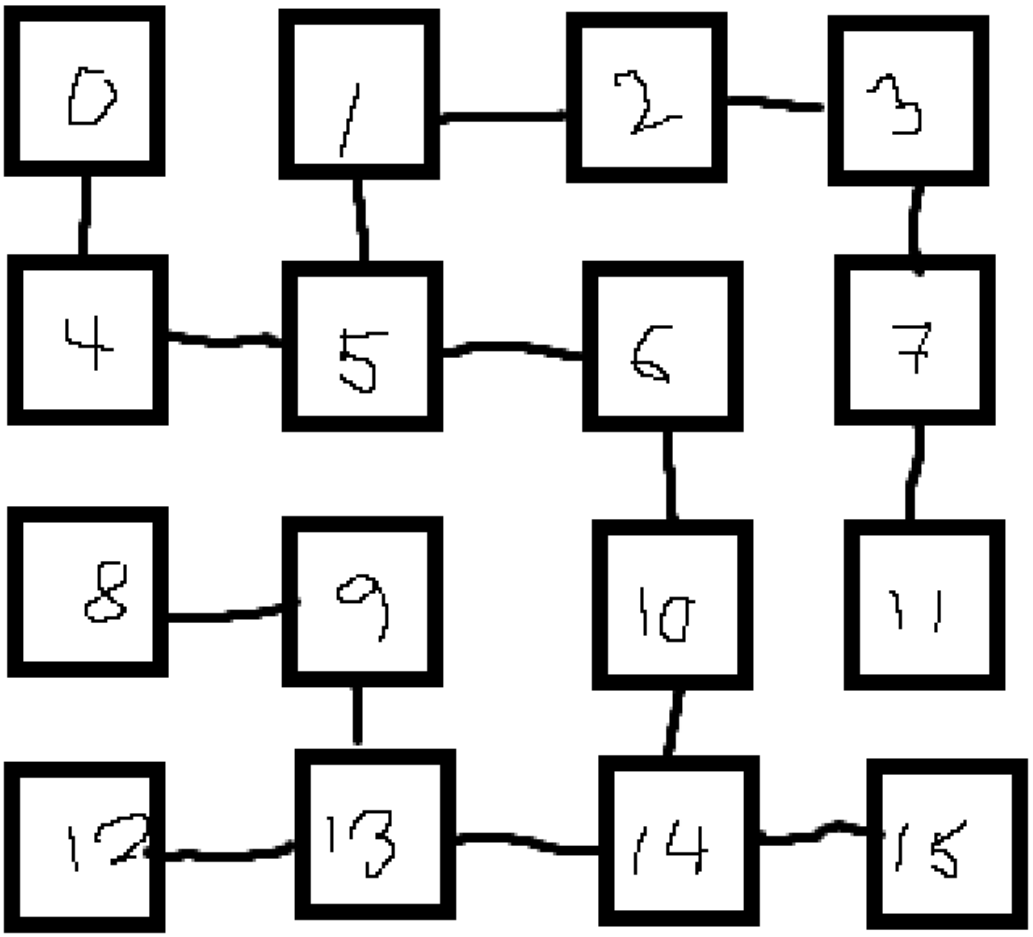
Tietorakenteet:

Tärkeimmät käyttämäni tietorakenteet olivat verkko ja pienin virityspuu (MST). Listoja tuli käytettyä paljon ja ratkaisualgoritmissa python omaa sanakirjaa defauldict. Verkko ja sen läpikäynti algoritmit olivat selkeä ratkaisua labyrinttiä tehdessä. Listat olivat nopeita ja helppoja ratkaisuja moneen kohtaan, ja niistä tulikin parissa kohtaa pullonkauloja algoritmien tehokkuudelle, täten esimerkiksi sanakirjakin oli vaihtoehto. Kaikki tietorakenteet olivat mielestäni muuttuvatilaisia. Esittelen verkon ja MST:n koska ne olivat keskeisessä osassa.

Verkko:



Yllä esimerkki miltä 4x4 verkko näyttäisi. Ruudun 10 kohdalle lisätty verkkoon tunneli ruudusta 9 ruutuun 11. Numerot ruutujen välissä kuvaavat linkin painoarvoa, jonka perusteella Kruskal muodostaa virityspuun. Kaikkiin linkkeihin en piirtänyt esimerkissä numeroa, mutta oikeassa verkossa kaikista sellainen löytyy. Pythonnissa verkko näyttäisi seuraavanlaiselta:
[[0, 1, 3], [0, 4, 7], [1, 2, 11], [1, 5, 32], …, [4, 5, 8], …, [5, 6, 20], …], jossa yhden alkion vasen luku on pienempi yhdistettävä ruutu, keskimmäinen suurempi yhdistettävä luku ja oikean puoleinen linkin painoarvo. Tästä verkosta rakennettu labyrintti voisi näyttää seuraavanlaiselta.



Huomattavaa tässä on se, että koska tunneli lisättiin etukäteen verkkoon Kruskal ei välttämättä valitsekaan sitä. Tämän voisi estää laittamalla tunneleille aina pienimmän painoarvon, mutta emme toimineet tällä tavalla. MST näyttää lähes samalta kuin verkko pythonissa [[0, 4], [1, 2], [1, 5], …, [4, 5], [5, 6], [6, 10] …], sieltä on vain poistunut ylimääräiset linkit ja painoarvot.

Tiedostot:

Ohjelma ei tarvitse tiedostoja koodin lisäksi toimiakseen, paitsi jos halutaan tallentaa labyrintti tiedostoon. Tällöin ohjelma tallentaa MST:n, joka kuvaa labyrinttiä, tekstimuodossa tiedostoon “CS-A1121_Lab.txt”. Tiedostoista ei voida tuoda myöskään mitään dataa.

Testaus:

Ohjelmaa testattiin pääsääntöisesti trial and error tyyppisesti sekä debuggerin avulla. Ohjelmasta löytyy myös kaksi yksikkötestiä, jotka testaavat, että pelaaja ja exit ovat asetettuina oikeisiin ruutuihin. Periaatteessa yksikkötestit testaavat myös, että labyrintti on rakennettu oikein, sillä muuten pelaaja ja exit eivät menisi oikeisiin ruutuihin. Kaikki yksikkötestit menevät läpi.

Suunnitelmaan nähden ohjelmaan toteutettiin paljon vähemmän yksikkötestejä kuin oli tarkoituksena. Tähän syynä se, että aloitin tekemällä toimivan graafiikan ja heti jos jotain oli pielessä muualla koodissa näin sen heti grafiikkaikkunasta tai peliä pelatessa. Tämän jälkeen debuggerilla ongelma löytyi yleensä kohtuullisen pian. Lisäyksikkötesteistä olisi varmasti ollut hyötyä joissakin kohdissa.

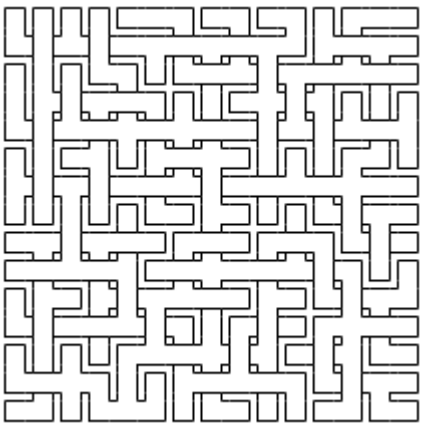
Ohjelman tunnetut puutteet ja viat:

Puutteita ja vikoja on tullut paljon mainittua pitkin tekstiä, mutta niistä oleellisin on selvästikin ratkaisualgoritmin rekursiosyvyys ongelma, kun labyrintin koko ylittää 200x200. Tämän saisi korjattua jos toteuttaisi toisenlaisen ratkaisualgoritmin esimerkkinä BFS (Breadth-first search). Tähän liittyen labyrintin kokoa kysyttäessä en laittanut arvoa 200 maksimiksi vaikka sen ylittävät arvot kaatuvat erroriin. Syynä tähän se, että jos haluaa testata pelkästään labyrinttien rakentamista, koodista tarvitsee vain kommentoida rivit 47-49 Graphics luokasta, jolloin ratkaisualgoritmia ei kysytä luontivaiheessa.

Grafiikan kanssa on myös pari mielenkiintoista vikaa, jotka liittyvät siihen kun ruutua liikutellaan. Ensimmäinen on jos esimerkiksi liikuttelee ruutua ja lähtee liikkeelle edelliselle paikalle jää ns. haamu pelaajasta, joka katoaa jos liikkuu takaisin ja lähtee uudelleen liikkeelle. Toinen on jos yrittää liikuttaa näyttöä kun ratkaisualgoritmi on käytössä se ei liiku kauhean hyvin.

3 parasta ja 3 heikointa kohtaa:

Heikoimmat kohdat on helpointa listata ensin. Ne ovat mielestäni ratkaisualgoritmi, yksikkötestit ja weave-labyrintin ulkonäkö. Ratkaisualgoritmista on oikeastaan kaikki mikä on vikana kerrottu muualla tekstissä, joten ainoa mikä siinä on hyvää on se, että se juuri ja juuri toimii meidän käyttötarkoitukseen, muuten siinä on lähes kaikki heikkoa verrattuna muihin ratkaisuihin, joita olisi voinut toteuttaa. Yksikkötestit ovat käytännössä olemattomassa osassa ohjelmassa, joten niitä ei voi mainita kuin heikkona lenkkinä. Viimeinen heikko kohta on weave-labyrintin ulkonäkö se ei ainakaan helpota labyrinttien ratkaisemista. Ideaali olisi ollut jotain tämän näköistä. (Kuva otettu lähteestä [1].)



Hyvinä kohtina pidän kuitenkin aikakompleksisuutta (niin pitkälle kuin se toimii), pelaajan liikuttamista labyrintissä sekä labyrintin luontia. Alle 200x200 kokoiset labyrintit tulevat luodaan ja piirretään parissa sekunnissa. Tämä ei ollut näin alkuperäisessä toteutuksessa 100x100 labyrintissäkin kesti minuutteja. Sain kuitenkin tiputettua luonti aikaa huomattavasti

vaihtamalla Primin algoritmista pois ja optimoimalla muutamaa muuta funktiota ohjelmassa. Pelaajan liikuttaminen WASD näppäimistöllä toimii kuin unelma ja tekee pelin pelaamisesta huomattavasti sulavampaa. Alunperin saatu idea, että labyrintti kuvataan pienimpänä viristyspuuna joka kootaan verkosta oli loistava ja onnistuin sen toteutuksessa hyvin.

Poikkeamat suunnitelmasta:

Jonkin verran tuli eroa suunnitelmaan. Primi vaihtui Kruskaliin. Verkon esitystapa muuttui. Weave labyrintin rakentaminen muuttui. Valikot muuttuivat. Yksikkötestaus pieneni. Perusidea pysyi kuitenkin samana verrattuna suunnitelmaan.

Ajankäyttöarvio osui aikalailla oikeaan. Tein yhteensä suunnitellun 80h verran työtä, jokseenkin se jakaantui hiukan eri tavalla, mitä suunnittelin sillä keskivaiheilla tuli muutama kiireinen viikko, jolloin en ehtinyt työtä tehdä. Toteutusjärjestys osui myöskin aika nappiin.

Toteutunut työjärjestys ja aikataulu:

Suunnitelmademon jälkeen ennen ensimmäistä välipalautusta, eli maaliskuun ajan, tein töitä yhtenä päivän viikossa noin 6-8h. Järjestys oli sama kuin suunnitelmassa eli tein aluksi toimivan graafisen ikkunan ja labyrintin luonnin, sitten labyrintin piirto, pelaajan lisääminen ja exitin lisääminen. Joten sain toimivan pelin. Juuri välipalautusta ennen poikkesin suunnitelmasta ja yritin weave labyrintin sijasta aloittaa tekemään 3D-labyrinttia. Tähän syynä se etten saanut weavea toimimaan. Heti välipalautuksen jälkeen huomasin kuitenkin, että jos haluan toteuttaa 3D-labyrintin täytyy minun muuttaa paljon jo toteutettua toimivaa koodia, joten yritin weavea uudestaan ja sain sen toimimaan halutulla tavalla. Samalla tuli keksittyä mitä tekisin ansoiksi, kun pelasin yhtä ei kunnolla toimivaa weave toteutusta ja totesin sen mukavan haastavaksi, kun kaikki tunnelit eivät toimineetkaan. Huhtikuun alussa ennen välipalautusta muissa kursseissa oli paljon kiireitä, enkä ehtinyt tehdä labyrinttiä juuri ollenkaan. Toteutin pari toimivaa yksikkötestiä juuri ennen toista välipalautusta. Seuraavat pari viikkoa tein taas kerran viikossa yhden päivän töitä, jolloin rakensin puuttuvat osat ratkaisualgoritmin, tiedostoon tallentamisen ja käyttäjäkyselyt. Optimoin myös paljon koodia. Viimeisellä viikolla tein noin 15h työtä, jolloin tein loppusilaukset ja kirjoittelin suunnitelman.

Arvio lopputuloksesta:

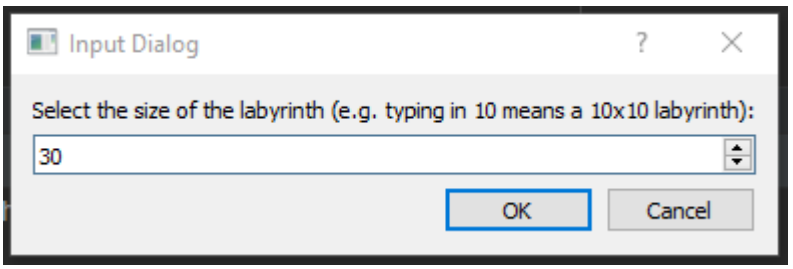
Hyvin lyhyt yhteenveto sillä monet asiat selitetty tarkemmin yllä.
Ohjelma mielestäni täyttää vaikea vaikeustason vaatimukset. Se on yleisellä käyttötasolla hyvin toimiva ja hauska pelata. Hyviä puolia aikakompleksisuus, pelaajan liikuttaminen ja labyrintin luonti. Heikkoja ratkaisualgoritmi, yksikkötestit ja weave-labyrintin ulkonäkö. Ei oleellisia puutteita. Tulevaisuudessa ensimmäisenä parempi ratkaisualgoritmi, jolloin saamme suurempia labyrinttejä ja selvemmän näköinen weave. Tietorakenteissa listoja olisi voinut vaihtaa johonkin muihin rakenteisiin rajoittavat tietyissä kohdissa aikakompleksisuutta. Ohjelma soveltuu hyvin tietyn tyyppisiin muutoksiin, ratkaisu- ja luontialgoritmejä on helppo vaihtaa. Kaikki missä pitää muuttaa graafista esitystä vie enemmän aikaa, esim kuusikulmainen tai 3D-labyrintti.

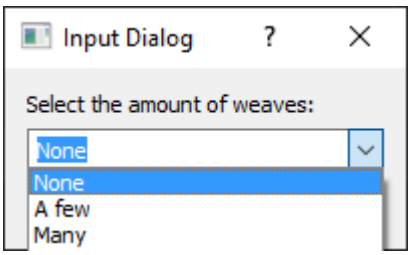
Viitteet:

1. IDG TECHtalk: Using "cProfile" to analyze Python code performance, 11.5.2020. Katsottu 25.04.2022
2. Jamis, Buck: Maze Generation: More weave mazes (https://weblog.jamisbuck.org/2011/3/17/maze-generation-more-weave-mazes.html) 17.3.2011. Katsottu 4.4.2022
3. Think Labyrinth: Maze Algorithms (http://www.astrolog.org/labyrnth/algrithm.htm). Katsottu 25.2.2022
4. OpenDSA: Data Structures and Algorithms Y (https://tarjotin.cs.aalto.fi/cs-a1141/OpenDSA/Books/CS-A1141/html/index.html), 03.10.2019. Katsottu 25.2.2022.
5. Jamis, Buck: Maze Generation: Growing Tree algorithm (https://weblog.jamisbuck.org/2011/1/27/maze-generation-growing-tree-algorithm) 27.1.2011. Katsottu 25.2.2022
6. Jamis, Buck: Maze Generation: Weave mazes (https://weblog.jamisbuck.org/2011/3/4/maze-generation-weave-mazes.html) 4.3.2011. Katsottu 25.2.2022
7. GeeksforGeeks: Kruskal's Minimum Spanning Tree Algorithm (https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/), 13.04.2022. Katsottu 5.5.2022
8. GeeksforGeeks: Print all paths from a given source to a destination (https://www.geeksforgeeks.org/find-paths-given-source-destination/), 19.1.2022. Katsottu 5.5.2022
9. CS-A1121: Tehtävä RobotWorld (https://plus.cs.aalto.fi/y2/2022/materials_k06/robots_pyqt_robots_pyqt/). Katsottu 1.3.2022
10. W3Schools: Python Operators (https://www.w3schools.com/python/python_operators.asp). Katsottu 25.3.2022

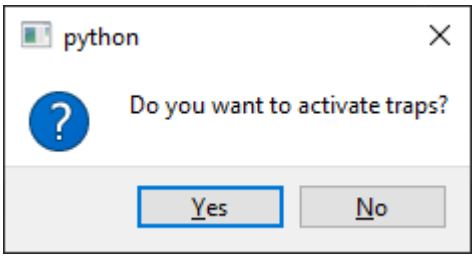
Liitteet:

Kuvat esimerkkiajosta:
Koonvalinta

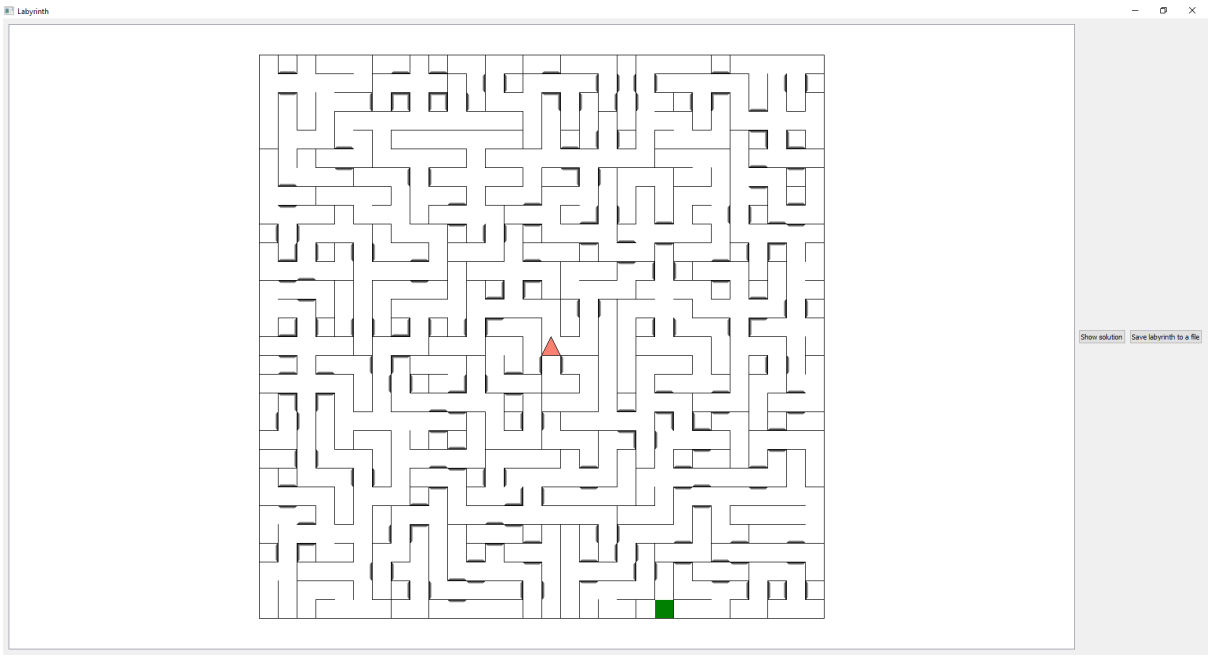




Weave valinta, viimeinen kohta “As many as possible” ei näy.



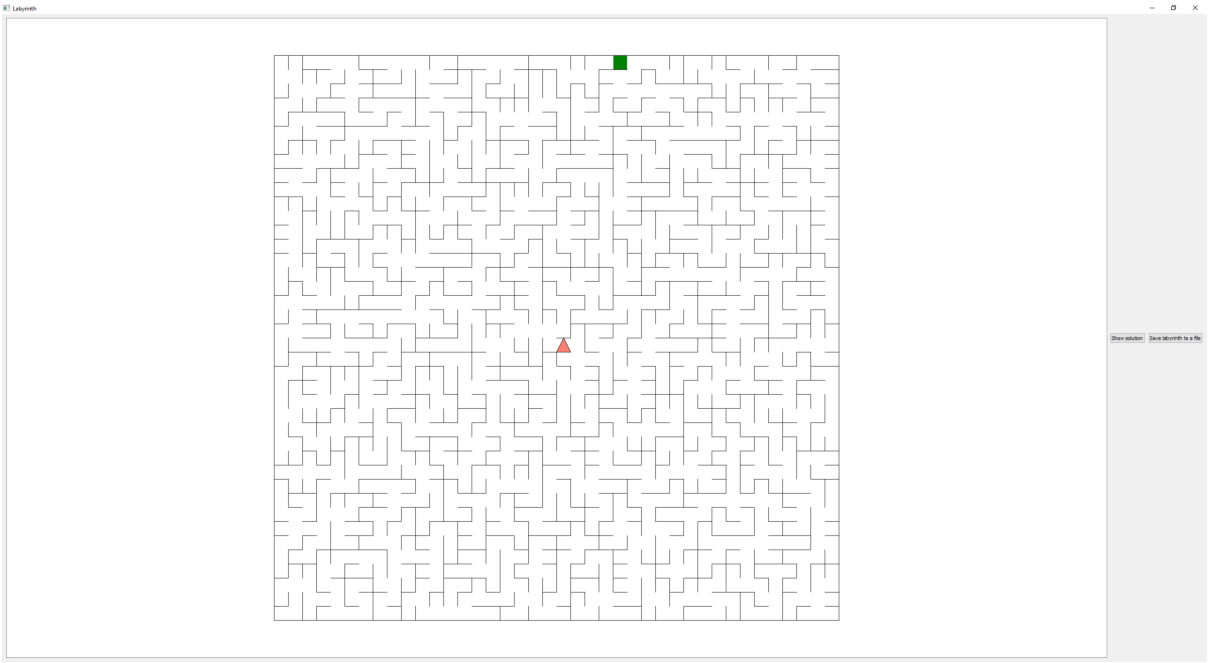
Kysely ansoja laittamisesta päälle.



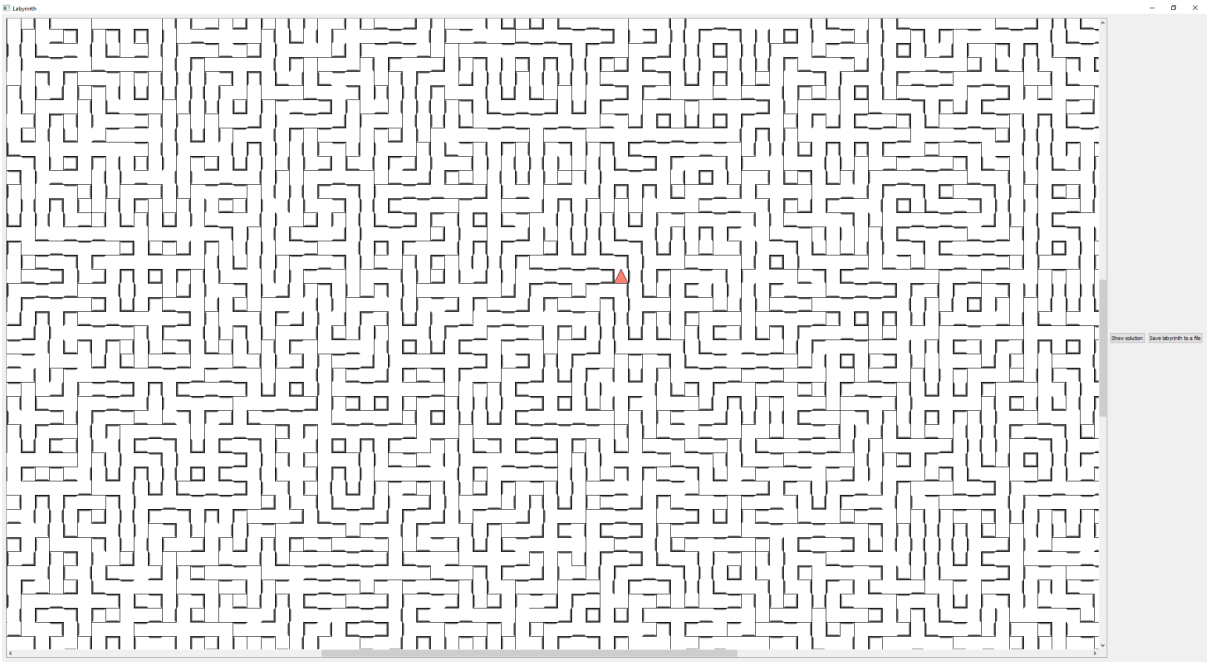
Weave-labyrintti, jonka koko 30x30, “Many” ja ei ansoja.

```
Scoreboard:

Number of moves:
  Player:
    Total: 80
    Correct: -4
  Minimum amount required: 73
  Autosolver: 77

Time: 137.23s
```

Tulostaulukko



40x40 2D-labyrintti



200x200 weave-labyrintti, ansoilla

Kuva tiedostoon tallennetusta labyrintistä