



**Eötvös Lóránd Tudományegyetem**

**Informatikai Kar**

**Programozásmélet és Szoftvertechnológiai  
Tanszék**

---

## **Labirintusok automatikus generálása neurális hálók felhasználásával**

*Témavezető:*

Pintér Balázs  
egyetemi adjunktus, Phd

*Szerző:*

Sinkovics Viktória Judit  
programtervező informatikus szak

**Budapest, 2019.**

# Tartalomjegyzék

<b>Bevezetés .....</b>	<b>4</b>
<b>Gráfelmélet .....</b>	<b>5</b>
Irányítatlan és irányított gráf.....	5
Fokszám .....	6
Séta.....	6
Út.....	6
Összefüggőség.....	6
Körmentesség.....	6
Fa.....	6
<b>A labirintus és az útvesztő .....</b>	<b>7</b>
Eredete.....	7
Használata napjainkban.....	8
Labirintusok és útvesztők a gráfelméletben.....	9
Labirintusgenerálás .....	10
Generátor algoritmusok tulajdonságai .....	10
Megoldás, megoldási út .....	11
A labirintusok karakterisztikái .....	11
Dimenzió.....	12
Geometria.....	12
Útvonalak típusa .....	12
Textúra .....	13
A szakdolgozatban algoritmussal generált útvesztők karakterisztikáinak összegzése .....	13
<b>Neurális hálózatok.....</b>	<b>14</b>
Gépi tanulás – Machine learning.....	14
Deep Learning.....	15
Neuron.....	15
Layer .....	16
Teljesen kapcsolt réteg - Dense, fully connected layer.....	16
Konvolúciós réteg – Convolutional layer .....	17
Pooling layer .....	18

Veszteségfüggvény .....	18
Súlyok optimalizálása .....	19
Gradiens Leereszkedés – Gradient descent.....	19
Hiba-visszaterjesztés – Backpropagation.....	21
Adam-optimizer .....	21
<b>Convolutional Variational Autoencoder (CVAE) .....</b>	<b>22</b>
Konvolúciós neurális hálózat (CNN) .....	22
Autoencoder .....	22
CVAE felépítése.....	22
A szakdolgozatban használt CVAE felépítése .....	23
<b>Technológiák.....</b>	<b>24</b>
Keras API.....	24
TensorFlow .....	24
<b>Felhasználói dokumentáció .....</b>	<b>25</b>
Az alkalmazás és a környezet felépítése .....	25
Alkalmazás elindítása.....	26
Az alkalmazás fájlljai .....	29
Alapértelmezett tanítóhalmazok és modellek .....	29
Az alkalmazás által generált fájlok .....	30
A GUI áttekintése.....	31
Neurális hálózatok tanítása a generátor algoritmusok segítségével .....	32
Tanítóhalmazok beállításai .....	32
CVAE beállításai.....	35
Bemutatópéldák generálása.....	37
Generálás algoritmusokkal.....	37
Generálás CVAE modellel.....	39
Az alkalmazás leállítása .....	42
<b>Fejlesztői dokumentáció .....</b>	<b>43</b>
Hardver és szoftver követelmények .....	43
Program szerkezeti felépítése.....	44
Python backend .....	46
Generátoralgoritmusok.....	48

Hunt-and-Kill .....	48
Growing Tree .....	48
Random Walk .....	49
Recursive Backtracker .....	49
Sidewinder .....	50
Ellenőrző algoritmus .....	52
Depth First Search.....	52
Generált labirintusok formátuma .....	52
Gráf .....	52
Bináris .....	53
Tesztelés .....	55
Az alkalmazás szekvenciadiagramjai.....	56
Labirintusgenerálás algoritmussal.....	56
Labirintusgenerálás CVAE modellel .....	56
CVAE modell tanítása, tanítóhalmazok készítése.....	57
Fejlesztési lehetőségek .....	58
<b>Összefoglalás .....</b>	<b>59</b>
Betanított CVAE-k által generált labirintusok .....	59
50*50 méretű labirintusok .....	64
<b>Irodalomjegyzék .....</b>	<b>67</b>
<b>Ábrajegyzék .....</b>	<b>70</b>

## Bevezetés

A labirintusok számos kultúrában az emberi életút, az önismeret vagy istenek jelképeként jelentek meg, de ugyanúgy szerepeltek például a görög mitológiákban is. A pszichológiai vagy neurológiai kísérletek és vizsgálatok segédeszközeként is használatosak, mint például a tájékozódással vagy memóriával kapcsolatos tanulmányok során. Manapság viszont a labirintus szó hallatán általában egy kétdimenziós négyzet vagy kör alakú fejtörő játéokra gondolunk, amelyben egy adott kezdőpontból kell falakkal elválasztott utakon, folyosókon és zsákutcákon keresztül eljutni a célcsúcsba, a kijáráthoz.

A szakdolgozatom célja egy könyvtár készítése, amely labirintus-generáló algoritmusok implementációját tartalmazza, illetve egy neurális hálózat által automatikusan generált labirintusok vizsgálata. A háló tanítópéldáiként a generátor algoritmusaim segítségével előállított útvesztőket használom. Mivel az algoritmikusan generált labirintusok mindegyike azonos alapelvek alapján készült, így arra számítok, hogy a háló kellő tanítás után szintén az alapelveknek megfelelő labirintusokat állít majd elő.

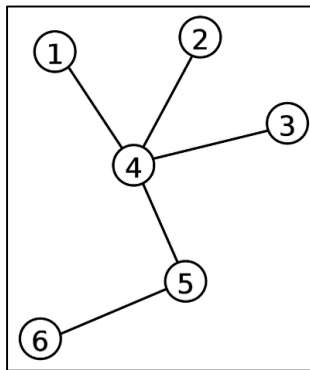
A projekt egy backend részből és egy frontendből áll, amelyeket egy kontroller osztály köt össze. A backend két fő szerepet tölt be. Egyik része, amely Java nyelven készült, tartalmazza néhány algoritmus implementációját, amelyek hasonló tulajdonságú labirintusokat képesek generálni. Valamint ez a komponens felelős a Python scriptek futtatásáért. A másik része egy Pythonban készült neurális hálózat, amelyet a Tensorflow implementációja alapján adaptáltam, hogy a feladatom feltételeinek megfelelően a tanítópéldáimat, amelyek a fent említett algoritmusokkal lettek generálva, értelmezni és használni tudja. A frontendet AWT és Swing csomagok felhasználásával készítettem, amely a kontroller osztályon keresztül kommunikál a backenddel. A kontroller ezen felül gondoskodik a Python scriptek megfelelő paraméterezéséért.

## Gráfelmélet<sup>1</sup>

A gráfelmélet olyan adatszerkezetek – gráfok – tanulmányozásával foglalkozik, amelyekben az adatok vagy objektumok – más néven *csúcsok* – között valamilyen páronkénti kapcsolat áll fent. Ezt a kapcsolatot *élek* nevezzük, amelynek grafikus jelölése a két csúcs között húzott vonal vagy görbe.

### Írányítatlan és irányított gráf

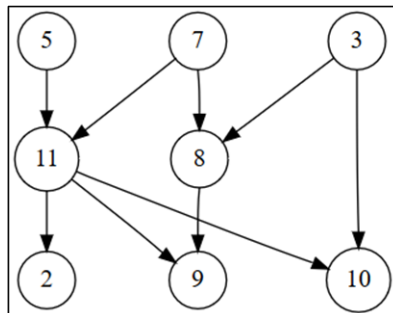
Írányítatlan (egyszerű) gráfnak nevezzük azt a  $G = (V, E)$  rendezett párost, amely egy nem üres csúcshalmazból ( $V$ ) és az ezeket összekötő élek halmazából ( $E \subseteq \{\{x, y\} \mid (x, y) \in V^2 \wedge x \neq y\}$ ) áll, amelyben az egyes éleket az általuk összekötött két csúcs rendezetlen párosával jelöljük.



1. Ábra – Irányítatlan, összefüggő egyszerű gráf. (1)

Írányított (egyszerű) gráfnak nevezzük azt a  $G = (V, A)$  rendezett párost, amely a csúcsok egy nem üres halmazából ( $V$ ), és az ezeket összekötő irányított élek halmazából áll ( $A \subseteq \{(x, y) \mid (x, y) \in V^2 \wedge x \neq y\}$ ), amelyben az irányított éleket az általuk összekötött csúcsok rendezett párosával jelöljük. Tehát különbséget teszünk az a két él között, amelyik  $a$ -ból mutat  $b$ -be ( $a \rightarrow b$ ), és amelyik  $b$ -ből vezet  $a$ -ba ( $b \rightarrow a$ ).

Egyszerű gráf alatt azt értjük, hogy két csúcs legfeljebb egy éllel van összekötve.



2. Ábra – Irányított, összefüggő egyszerű gráf. (2)

<sup>1</sup> A gráfelméleti definíciók az irodalomjegyzék [28] és [29] forrásai alapján készültek.

## Fokszám

Egy  $a$  csúc fokszámát  $d(a)$ -val jelöljük, és azt mondja meg, hogy hány él illeszkedik az adott csúcsra.

## Séta

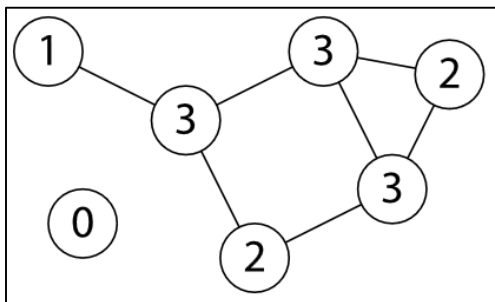
Sétának nevezzük a gráf egy olyan élsorozatát, amelyben minden él végpontja megegyezik a rákövetkező él kezdőpontjával. A séta tartalmazhat csomókat és elágazásokat.

## Út

Egy olyan séta, amelyben egy csúcson legfeljebb egyszer haladunk át, és minden él legfeljebb egyszer szerepel az úton.

## Összefüggőség

Egy (irányított) gráf összefüggő, ha tetszőleges két csúc között létezik (irányított) út, ha viszont létezik legalább két csúc, amelyek közt nincs (irányított) út, akkor a gráf nem összefüggő. Az összefüggő gráfok pontosan egy komponensből állnak, a nem összefüggők pedig legalább kettőből. A komponensszámot  $C(G)$ -vel jelöljük.



3. Ábra – Nem összefüggő gráf. (3)

## Körmentesség

A körmentesség annyit tesz, hogy a gráf tetszőlegesen kiválasztott véges sétájának a kezdő és végpontja különbözik. A vonal egy olyan élsorozat – a gráf éleinek egy rendezett véges halmaza –, ahol egy él végpontja megegyezik a rákövetkező él kezdőpontjával.

A fentebb definiált irányított és irányítatlan gráfok körmentesek, ha viszont engedélyezzük a köröket, akkor mindkét esetben módosítani kell az élhalmazt. Irányítatlan esetben  $E \subseteq \{\{x, y\} \mid (x, y) \in V^2 \wedge x \neq y\}$  helyett  $E \subseteq \{\{x, y\} \mid (x, y) \in V^2\}$  halmazt használunk. Hasonlóan irányított esetben az  $A \subseteq \{(x, y) \mid (x, y) \in V^2 \wedge x \neq y\}$  helyett  $A \subseteq \{(x, y) \mid (x, y) \in V^2\}$ , vagyis mindkét esetben az  $x \neq y$  megszorítást kell eltávolítani.

## Fa

Egy gráf fa, ha összefüggő, és nem tartalmaz kört.

## A labirintus és az útvesztő

### Eredete

A labirintus (ógörögül: *labirinthos*) egy több ezer éves szimbólum, ami mitológiákban és istentiszteleti szimbólumokként jelent meg.

A görög mitológiákban egy bonyolult építményt jelentett, amit Daidalosz, a híres ezermester tervezett és épített Minosz király parancsára Kréta szigetén Knósszosz városában. A labirintus azt a célt szolgálta, hogy a bika fejű, ember testű szörnyet, a Minótauroszt bezárva tartsa. Az építmény állítólag olyan jól sikerült, hogy Daidalosz is alig tudott kijutni. Később Thészeusz, Athén hercege látogatta meg a labirintust, hogy legyőzze a Minótauroszt, miután sikerrel járt, a fonalat követve, ami az Ariadnétól kapott gombolyagból származott, kijutott a labirintusból.

A labirintus a mitológiában egyértelműen egy bonyolult, szerteágazó folyosókból álló építményt jelentett, viszont később új néven hivatkoztak rá: útvesztő. A labirintus szót pedig egyre inkább annak leírására használták, ami egy (tipikusan) kör közepére vezető út, ami elágazásmentes volt, és amibe egy út vezet be, és egy ki. Ez a forma rendszerint a megtisztulás szimbólumaként szerepelt templomok padlóján. A mondás az volt, hogy ha egy erősen elhivatott zárándok, követve a labirintus vonalát, besétált a közepére, majd ugyanazon az úton kijött, akkor a séta közben a megtisztult, és lelkileg újjászületve, megvilágosodva lépett ki a labirintusból. [1]



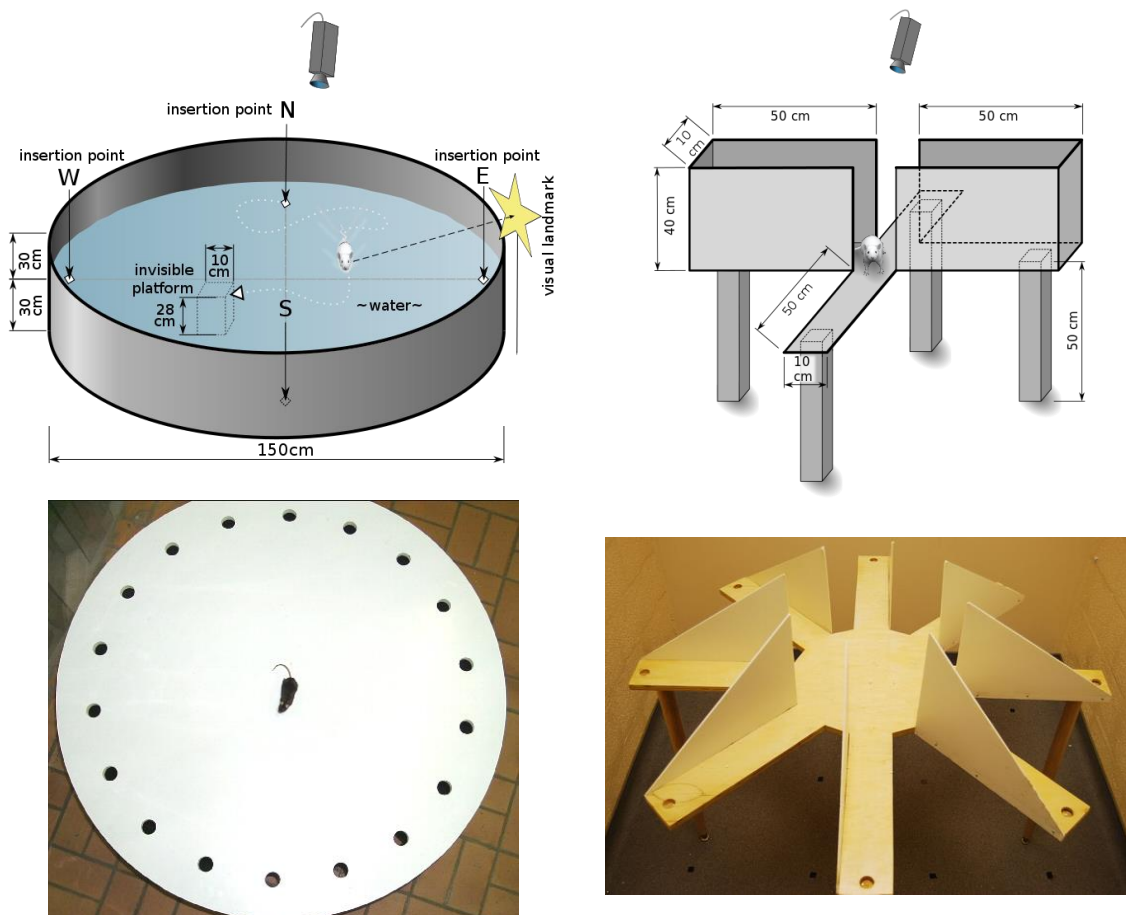
4. Ábra – Labirintus a Chartres-i katedrális padlóján. (4) (5)

Megjegyezném, hogy habár manapság a labirintus és az útvesztő két különböző kinézetű útrendszerre utal, tekintettel a labirintus szó eredeti, mitológiai jelentésére, a dokumentum további részében egymás szinonimájaként fogom őket használni.



## Használata napjainkban

Manapság pszichológiai és neurológiai tanulmányokhoz is, amelyeknek a tesztalanyai főként egerek és laboratóriumi patkányok, használnak különböző speciális formájú útvesztőket, amikkel a térbeli navigációs képességeket, a memóriát vagy akár a tanulási képességet vizsgálják. Több elterjedt labirintusformát is használnak, például a Barnes labirintust: egy kör alakú platform, lyukakkal a szélén, amelyek közül az egyik a „kijárat”; a magasított plusz labirintus, ennek egymással szemben két keskeny, oldalról zárt és két nyitott karja van, ezt a szorongási hajlam vizsgálatára használják. [2] [3] [4]



5. Ábra – Neurológiai és pszichológiai kísérletekhez használt labirintusok. (6) (7) (8) (9)

A T-labirintus például az alábbi módon működik. A kísérlethez egy-egy jutalmat helyeznek – tipikusan ételt vagy egy másik egeret – a felső két kar végébe. Innen két irányba válhat szét a kísérlet: a memória tesztelése, vagy a tanulási képességek felmérése.

- A T két felső karja közül az egyiket lezárják, a másikat nyitva hagyják, és behelyezik az egeret az alsó szárba. Ekkor az egér a nyitott kapuhoz megy, majd megtalálja a jutalmat, ez volt az ismertető kör. Következő körben felnyitják az eddig lezárt kart, majd visszahelyezik a kezdőpontra az egeret, ekkor két lehetősége van, ugyanarra fordul, amerre legutóbb is ment, vagy felfedezi az új irányt.
- A kiinduló helyzet annyiban tér el az előző esettől, hogy egyik kar sincs lezárva, és az egérnek kezdettől fogva meg van a választási lehetősége. Az első döntése után visszahelyezik a kezdőpontra, majd megismétlik a kísérletet.

Legtöbbször egy egészséges egér abba az irányba megy a második körben, amerre nem járt korábban. Viszont egy idegrendszeri rendellenesség a memóriát és a tanulást is befolyásolhatja. Egy 1999-es tanulmányban, amelyben egészséges egerek mellett olyan egereket használtak, amelyeknek egy, az Alzheimer kór tüneteit szimuláló, betegsége volt. Az egészséges egyedek ismételt kísérletek után több mint 80%-ban választották a helyes irányt, viszont a beteg egerek esetében csak a fiatalok, kb. 2 hónaposok, teljesítettek jól, az idősebb, kb. 16 hónapos, egerek viszont már a tanulás alatt is sokkal rosszabbul végezték el a feladatot. Ebből azt a következtetést vonták le, hogy az Alzheimer kórban szenvedők memóriája és tanulási képessége a korral romolhat. [5]

### Labirintusok és útvesztők a gráfelméletben

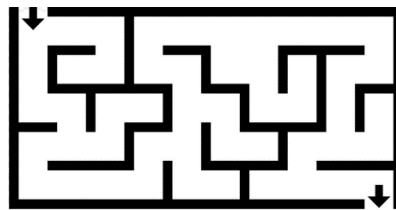
Itt a csúcsok az egyes cellákat jelentik, illetve ha két (szomszédos) csúcs között létezik él, akkor a két cella egy lépésben elérhető egymásból, vagyis egy átjáró van köztük. Amennyiben két csúcs közt nincs közvetlen él, az két dolgot jelenthet: vagy egymás mellett helyezkednek el, de egy fal található köztük, vagyis nincs átjáró; vagy nem egymás melletti cellákról van szó, vagyis a koordinátáik alapján nem szomszédosok. [6]

A labirintus egy összefüggő, körmentes (irányítatlan) gráfot takar, aminek az összes csúcsa egyetlen folytonos úton fekszik, amit neveznek egyébként „unicursal maze”-nek is. Ebben az esetben a gráf minden csúcsának fokszáma pontosan kettő, kivéve a kezdő- és végcsúcsokat, amelyek foka egy. [6]

Az útvesztő egy (irányítatlan) gráf, ami lehet összefüggő vagy nem, tartalmazhat köröket, illetve elágazásokat vagy zsákutcákat, amelyekből a célcsúcs csak úgy érhető el, ha visszamegyünk egy korábbi elágazáshoz. [6]



6. Ábra – Labirintus. (10)



7. Ábra – Útvesztő. (11)

## Labirintusgenerálás

Az a folyamat, amikor eldöntjük, hogy milyen tulajdonságokkal rendelkezzen a labirintus – lehetnek-e benne esetleg körök, majd kivitelezük azt. A megvalósítás történhet manuálisan, például papíron, vagy számítógépen automatizálhatjuk a folyamatot algoritmusok segítségével.

### Generátor algoritmusok tulajdonságai

Két fő kategóriája van a generátoroknak:

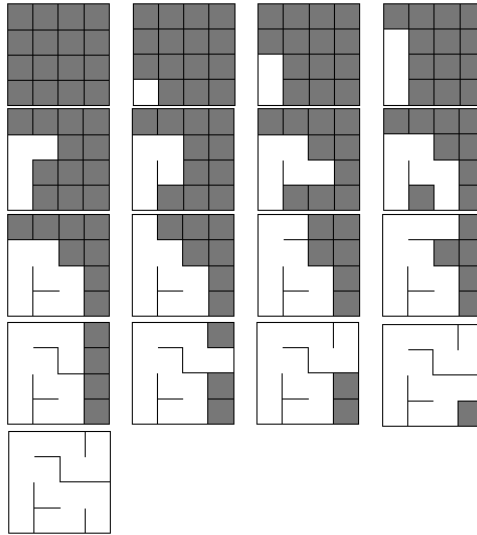
#### *Wall Adder*

Ezt fal építőnek fordíthatnánk. A kiinduló helyzet egy üres felület, cellák falak nélkül, majd minden lépésben egy falat épít az algoritmus egész addig folytatva, amíg egy megfelelő méretű labirintust nem kapunk. Például a *Recursive Division* algoritmus ilyen. [2] [7]

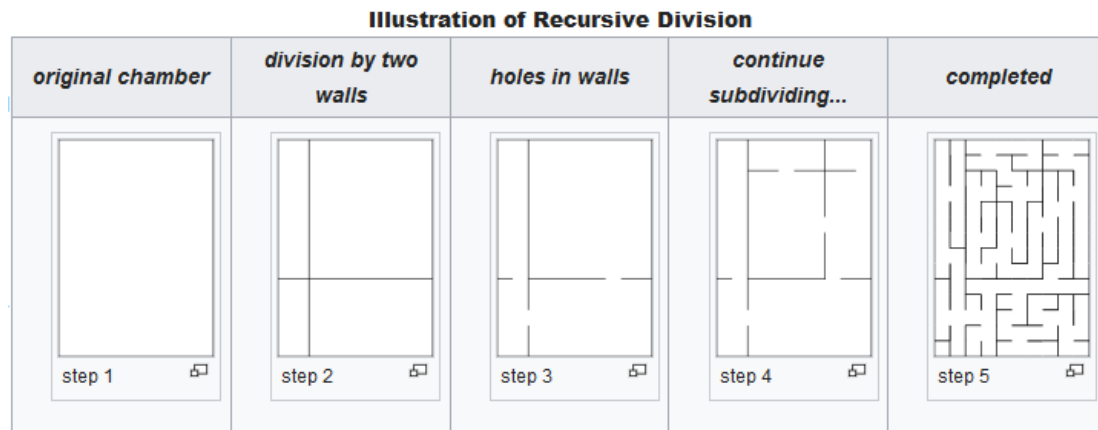
#### *Passage Carver*

Ez alagút, folyosó vágót jelent. A kezdőállapot lehet vagy egy tömör fal, ahol az egyes cellák közepe is be van falazva, vagy üres cellák, amelyeknek minden fala látható. Az algoritmus lépésenként eltüntet egy falat két cella közül, illetve ha szükséges, akkor a cellák közepét is kivágja. Addig fut, amíg az útvesztő megfelelő tulajdonságokkal nem rendelkezik. Például a *Recursive Backtracker* vagy a *Kruskal* algoritmusok. [2] [7]

Ez a két tulajdonság sokkal inkább algoritmusbeli különbség, mint valami, ami a labirintusok vizualitását befolyásolná. A legtöbb típusú útvesztő elkészíthető bármelyik fajta algoritmussal, sőt, néhány algoritmus implementálható *wall adder*-ként, vagy akár *passage carver*-ként is. A *Binary Tree* algoritmus egy jó példa erre. Az algoritmus lényege a két esetben: *passage carver* esetén egyesével minden cellánál eldöntjük, hogy az adott csúcsból felfelé vagy balra vágjunk utat, de nem mind a kettőbe; a *wall adder* változat hasonlóan működik, szintén minden cellánál eldöntjük, hogy a jobb oldalára vagy az aljára építsünk-e falat, de mindenképp csak az egyik oldalra. Párosíthatjuk egyébként a fel-jobb, és a bal-le irányokat is akár, a lényeg, hogy egy horizontális és egy vertikális irányból álljon. [2]



8. Ábra – "Passage Carver" típusú algoritmus vizualizációja. (12)



9. Ábra – "Wall Adder" típusú algoritmus vizualizációja. (13)

## Megoldás, megoldási út

A megoldás a kezdőpontból a célpontba vezető út megkeresését, megtalálását jelenti. Azt az utat nevezzük megoldási útnak, ami legrövidebb módon köti össze a kezdő- és célsúcsokat. Ezen az úton nem szerepelnek zsákutcák, se körök.

## A labirintusok karakterisztikái

Az útvesztők több tulajdonságuk alapján is csoportosíthatók. A következő listában néhány fontosabb karakterisztikát emeltem ki, amelyek mind a labirintusok külsőségeit határozzák meg, illetve generálás szempontjából fontos szerepet játszanak.

Ezekon felül ide tartozhatnak még az ezek megoldását vagy megoldási menetét vizualizáló tulajdonságok, de ezek a generálás folyamatát nem befolyásolják.

## Dimenzió

Dimenzió alatt értjük az útvesztő térbeliségét. A legelterjedtebb a kétdimenziós elhelyezkedés. Ezen felül lehet háromdimenziós, ahol nem csak észak-kelet-dél-nyugat irányba, hanem fel- és lefelé is lehet mozogni. Magasabb dimenzióban is értelmezhetők, de csak 3D-ben lehet őket megjeleníteni. [2] [7]

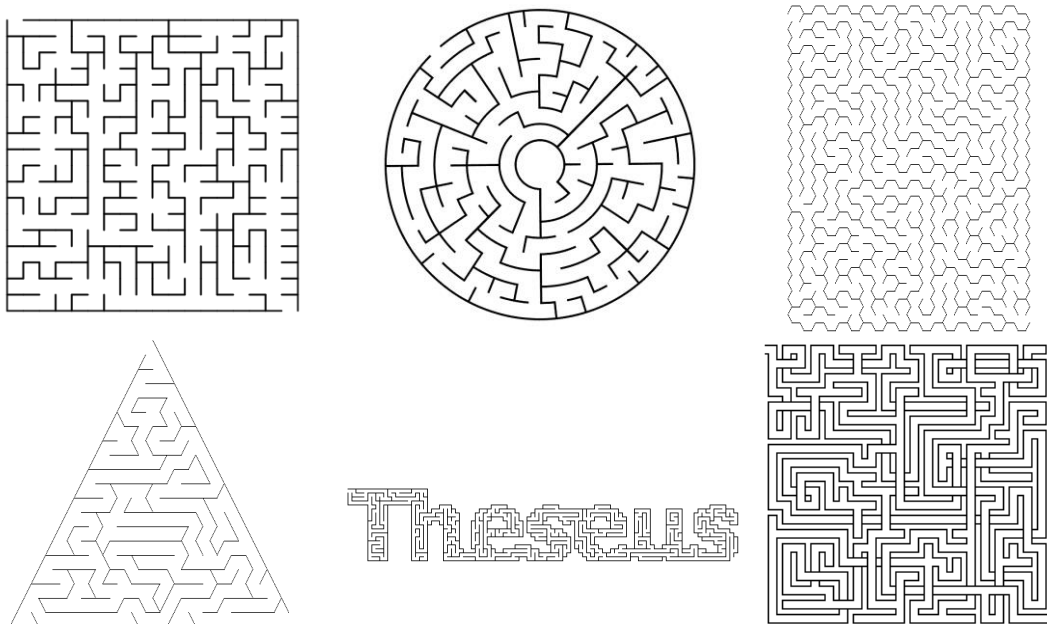
## Geometria

Ez a tulajdonság határozza meg, hogy a labirintus egyes celláinak maximálisan hány ki- és bemeneti pontja lehet. A cellák lehetnek például négyszög alakúak, ekkor mindegyiknek legfeljebb 4 szomszédja lehet, ezt a típust *Orthogonal* labirintusnak nevezzük. Hasonlóan a háromszög (*Delta*) alapúnak cellánként maximum 3 szomszédja lehet, de akár hat- vagy nyolcszög is használhatunk (*Sigma* és *Upsilon*). [2] [7]

## Útvonalak típusa

A labirintusok a bejárhatóságuk alapján is rendezhetők. Ezalatt azt kell érteni, hogy az útvesztőben vannak-e elágazások, zsákutcák, körök, vagy esetleg elérhetetlen cellák. Érdekes a két szélsőséges típust megemlíteni:

- Az úgynevezett tökéletes, vagy „*perfect*” útvesztő, ami körmentes, zsákutcákkal rendelkezik és tetszőleges két cellája között pontosan egy út létezik. Ez a fajta labirintus a gráfelméletbeli fával egyezik meg.
- A „*braid*” labirintusban egyetlen zsákutca sincs, ehelyett rengeteg kör van benne, illetve hosszú kanyargós utcák, amelyek aztán önmagukba kanyarodnak vissza. [2] [7]



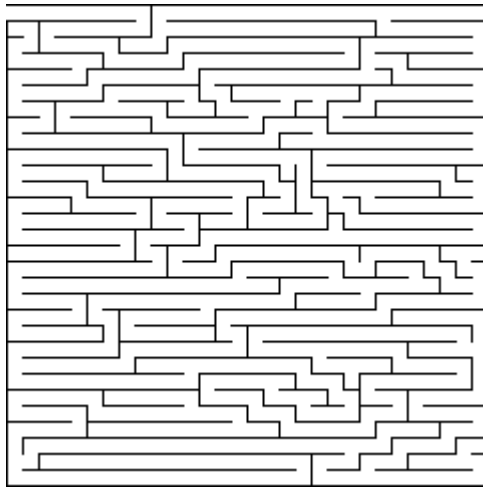
10. Ábra – Példák az útvesztők néhány típusára. (14) (15) (16)

## Textúra

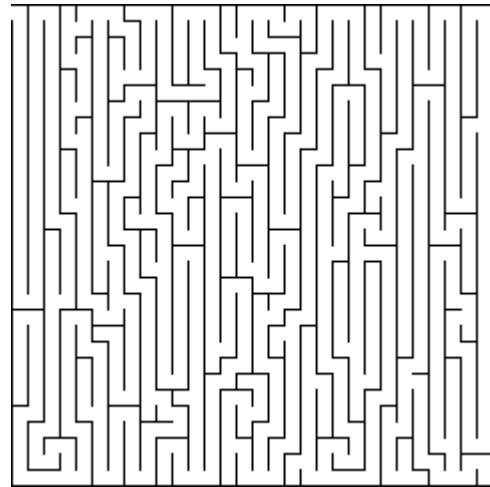
Egy útvesztő textúrája a folyosók átlagos hosszúságát és irányát írja le.

Azt a tulajdonságot, ami a folyosók jellemző irányára vonatkozik, hajlamnak („*bias*”) nevezzük. Egy vertikális hajlamú útvesztőben főként hosszú függőleges szakaszok láthatók, amelyek helyenként rövid – 1-2 cella hosszúságú – vízszintes szakaszokkal vannak összekötve. Hasanlóan működik a horizontális hajlamú labirintus is, ahol hosszú vízszintes szakaszok vannak összekötve rövid vertikális folyosókkal.

Az egyes szakaszok hosszúságát a folyási együttható („*run factor*”) határozza meg, ami annyit tesz, hogy milyen hosszú szakaszon lehet egy úton menni anélkül, hogy el kelljen kanyarodni egy másik irányba – ez nem feltétlenül jelent csak kereszteződést, a labirintus szélső fala vagy egy szomszédos folyosó is kanyarodásra kényszerítheti az utat. A magas folyási faktorú útvesztők erősen hasonlítanak egy vertikális vagy horizontális hajlamúra. Az alacsony folyási együtthatójú labirintusokban jellemzően a leghosszabb egyenes szakaszok 3-4 cellát ölelnek fel. [2] [7]



11. Ábra – Horizontális hajlam. (17)



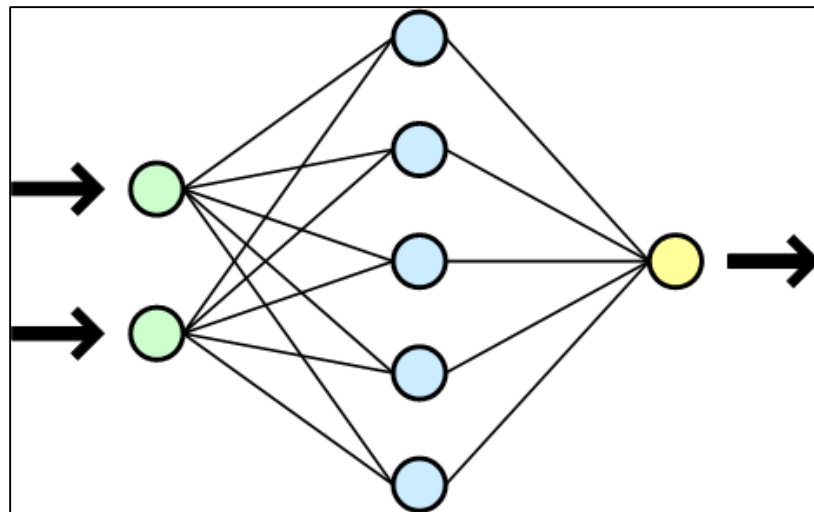
12. Ábra – Vertikális hajlam. (17)

## A szakdolgozatban algoritmussal generált útvesztők karakterisztikáinak összefoglalása

Az implementált algoritmusok mindegyike 2D, négyzet alapú – gráfelméleti szempontból minden csúcs foka legfeljebb 4, azonos sor- és oszlopszámú labirintusokat eredményez, amelyek a *perfect* labirintusok közé tartoznak. Egyik sem alkalmaz olyan tényezőket, amik az eredmény vízszintes vagy függőleges hajlamosságát befolyásolná, illetve ugyanez mondható el a folyási együtthatóról is.

## Neurális hálózatok

A (mesterséges) neurális hálózatok tudománya („*neural computing*”), a biológiai idegrendszerek működésén alapul, ami képes mintákból és példákból nyert tapasztalatok alapján tanulással fejleszteni a feladatmegoldó képességét adott problémák esetén. A neurális hálózatok gyorsaságát és robusztusságát olyan architektúráis tulajdonságok biztosítják, mint a párhuzamos felépítés és működés, a tanulási képesség, valamint a háló egy részének redundanciája. Ezen tulajdonságok mind lehetővé teszik a neurális hálózat környezeti változásokhoz való adaptációját, valamint felépülését, ha egy redundáns háló meghibásodik. [8] [9]



13. Ábra – Egy neurális hálózat egyszerűsített grafikus nézete, ahol balról jobbra láthatók a bemeneti, rejtett és kimeneti rétegek. (18)

## Gépi tanulás – Machine learning

A gépi tanulás olyan rendszerekkel foglalkozik, amelyek képesek bemeneti mintákból következtetési modelleket felállítani, amelyek segítségével általánosított következtetéseket tudnak adni, vagy döntéseket tudnak hozni az új, ismeretlen bemeneti adatok esetén is.

Több típusú tanító algoritmus is létezik, amelyeket különféle rendszerekben használnak adott problémák megoldásának felderítésére, viszont neurális hálózatok esetén főként az alábbi két típus használatos:

### 1. Felügyelt tanulás („*supervised learning*”):

Tanítás során a rendszer bemenetei címkézettek, vagyis a tanítóhalmazban minden példabemenethez párosítva van az elvárt kimenete is. A rendszer feladata megtanulni, hogy milyen típusú bemenethez milyen eredmény tartozik. Amikor a tanulási folyamatnak vége, akkor egy teszhalmaz segítségével, amely kizárólag olyan bemeneti adatokat tartalmaz, amelyet nem használtunk a tanításhoz,

megvizsgáljuk a rendszer teljesítményét, vagyis hogy ismeretlen bemenetek esetén mennyire sikerült jól (közel helyesen) megbecsülnie a kimeneti eredményt.

## 2. Felügyelet nélküli tanulás („unsupervised learning”):

Ebben az esetben a tanítóhalmaz elemei nem címkézettek, vagyis nincs pontosan definiálva, hogy mi az elvárt működés. Tanulás során a rendszernek kell valamilyen mintát felfedeznie a bemenetek között, amelyeket általában hasonlóságaik alapján csoportosít.

Túltanításról („*overfitting*”) beszélünk, amikor a hálózat a tanítópéldákra kapott válaszokra nagyon alacsony a hiba, de bármely más mintára magasabb a válaszok hibája. Ez akkor történhet meg, ha csak a tanítópéldákra kapott válaszok alapján értékelünk. Ennek elkerülése érdekében a teszt vagy kiértékelő halmaz használhatunk, amelyek olyan bemeneti (és kimeneti párokból álló) adatokat tartalmaznak, amelyek a megoldandó feladatból származnak, és amelyeket nem használunk fel tanításra. [8] [9]

A továbbiakban a (gépi) tanulás vagy tanítás a neurális hálók tanulására és tanítására vonatkozik.

### **Deep Learning**

Manapság egyre erősebb hardverek érhetőek el a piacon, aminek hatására egyre többet lehet hallani a „*deep learning*” kifejezést. A *deep learning* egy viszonylag új fogalom a neuronhálók tudományában, amely olyan neurális hálózati modellekkel foglalkozik, amelyek *sok* réteggel rendelkeznek („*deep neural network*”). A hálózatok méretéből adódóan nagyobb adathalmazokat is képesek hatékonyan feldolgozni, illetve jóval nagyobb számítási kapacitást igényelnek, mint a standard neurális hálózatok. [10] [11]

### **Neuron**

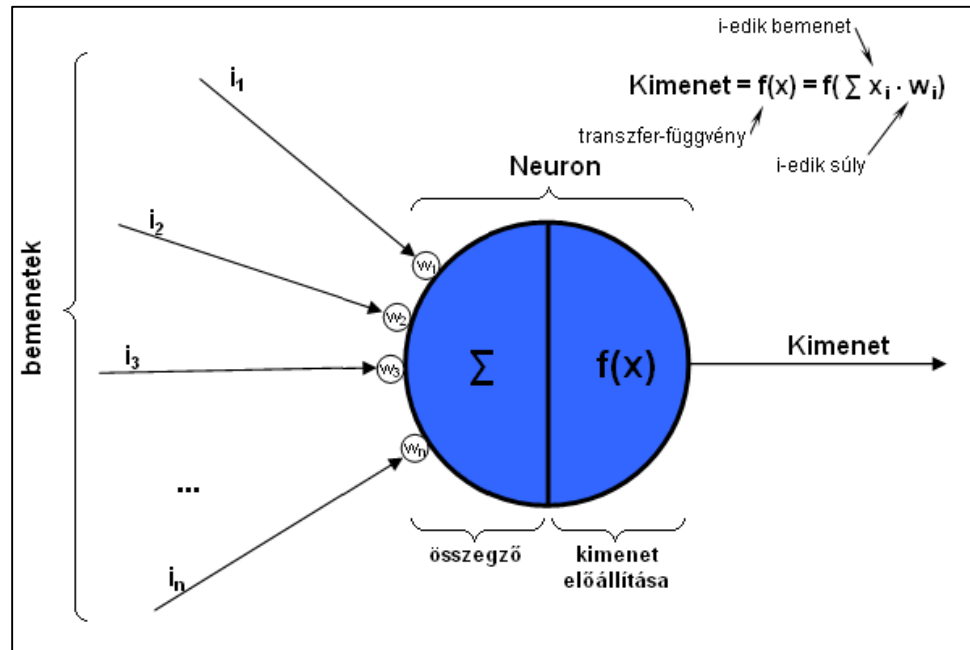
Más néven műveleti vagy processzáló elem egy több-bemenetű és egy vagy több kimenetű eszköz, amely a bemenetek (és esetleg a neuron belső memóriája) és a kimenetek között általában valamilyen nemlineáris leképezést valósít meg. Ezt a nemlineáris leképezést aktivációs függvénynek nevezzük.

A neuronoknak 3 fő fajtája van:

1. Bemeneti („input”): Egy speciális neuron, amelynek a bemenete egyben a hálózat bemenete is. Egy bemenettel és egy kimenettel rendelkezik, illetve az egyetlen feladata a bejövő adatok továbbítása, vagyis más neuronok meghajtása.
2. Rejtett („hidden”): Be- és kimeneteikkel kizárólag csak más neuronokhoz kapcsolódnak.
3. Kimeneti („output”): A kimenete a környezet felé továbbítja a kívánt információt.



A neuronok súlyokkal rendelkeznek, amelyek egyben a hálózat súlyai, a belső működési paraméterei is, valamint tanulás és optimalizálás közben ezeket a súlyokat változtatjuk. [8]



14. Ábra – Egy (mesterséges) neuron ábrázolása. (19)

## Layer

A neuronokat rétegekbe szervezzük. Az egy rétegbe tartozó neuronok hasonló típusúak, illetve jellemzően a kapcsolataik is hasonlóak. Bemeneti rétegről („*input layer*”) beszélünk, ha a réteg bemenete a hálózat bemenetével egyezik meg. A rejtett réteg („*hidden layer*”) bemenetei kizárólag más neuronok kimeneteihez kapcsolódnak, és hasonlóan a kimeneteik egy másik réteg bemenetét képezik. A kimeneti réteg („*output layer*”) kimenetei a teljes hálózat kimeneteit alkotják.

Egy rétegekbe szervezett neurális hálózat legalább két rétegből áll, egy be- és egy kimeneti rétegből, amelyekkel a külvilágtól kapott információt fogadni és a környezet felé (feldolgozva) továbbítani tudja. A két réteg között elméletben tetszőleges számú rejtett réteg helyezkedhet el. [8]

A rejtett rétegek különböző fajtájúak lehetnek, amelyek specifikus működéssel rendelkeznek:

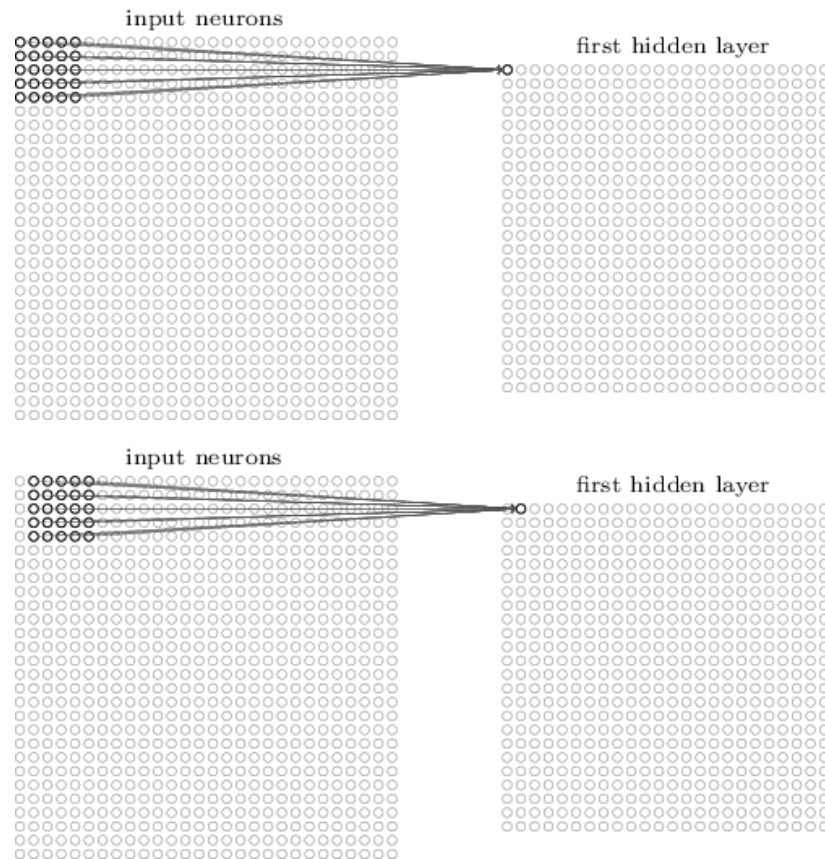
### Teljesen kapcsolt réteg - Dense, fully connected layer

A bemenetek és egy tárolt súlymátrix egy lineáris kombinációját állítja elő.

### Konvolúciós réteg – Convolutional layer

A bemenetek kisebb részeire bontjuk, majd minden részbemeneten egy szűrőt alkalmazunk, amely két mátrix (a részbemenet és a szűrő mátrixa) elemenkénti szorzatának összegét jelenti. Ezeket az összegeket sorban egy *feature map*-be tesszük, amely egyben a bemenet egy leegyszerűsített változata. [12]

A rétegben több konvolúciós szűrőt is alkalmazunk, amelyek együtt adják a réteg kimenetét.

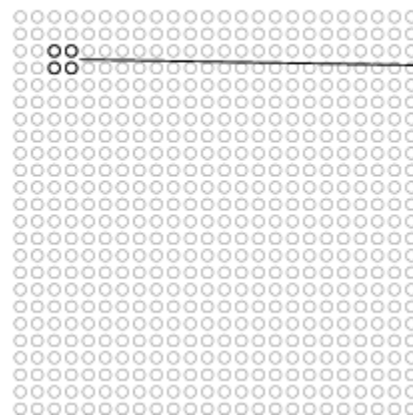


15. Ábra – Konvolúciós rétegben egy filter alkalmazása az input egy részén (bal oldal), majd az érték elhelyezése a feature map-ben (jobb oldal). (20)

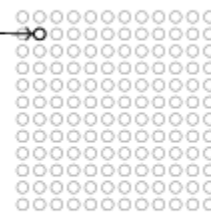
## Pooling layer

A bemeneten dimenziócsökkentést végez – minden dimenziót megváltoztat, kivéve a mélységet, ami a szintérnek felel meg –, aminek köszönhetően kevesebb paraméterre lesz szükség a háló további részében, vagyis a zsugorított méretű adathoz kevesebb neuron szükséges, valamint így a tanulási idő is rövidül. [12]

hidden neurons (output from feature map)



max-pooling units



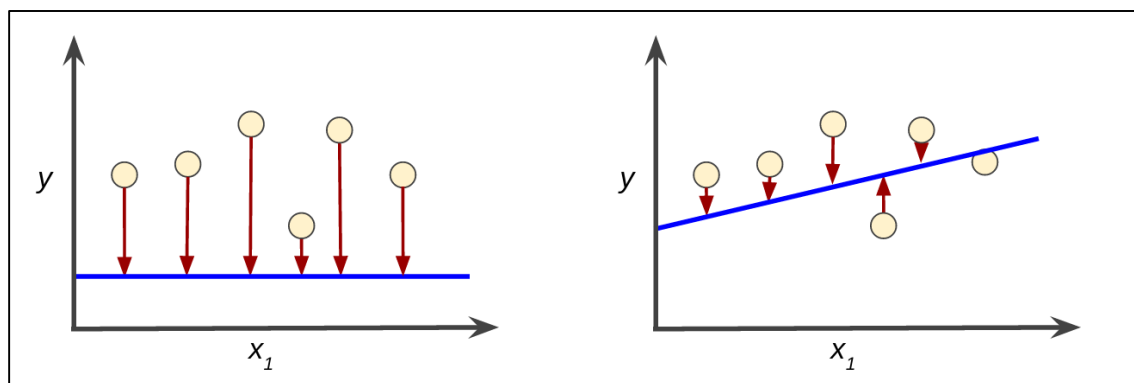
16. Ábra – Max-pooling, itt 2\*2-es területeken (a területek nem fedik egymást) összegezi a bemeneti értékeket egy pontba.

Itt a 24\*24-es bemenetből egy 12\*12-es kimenet keletkezett. (20)

## Veszteségfüggvény

Más néven hiba- vagy költségfüggvény, a kimeneti és az elvárt eredmények közti különbséget – becslési tévedést – reprezentáló függvény. Ha tanítópéldákhoz biztosítva van a hálótól elvárt eredmény, akkor egyértelműen kiszámítható a hiba mértéke és iránya. [8] [9]

Egy modell tanításának célja, hogy átlagosan, minden bemeneti példára alacsony legyen a hiba mérete, vagyis a tanítás egy szélsőérték-kereső eljárás.



17. Ábra – Egy magas hibájú (bal) és egy alacsony hibájú (jobb) modell becslési hibái. A kék vonal a becslést, a piros nyilak a hiba irányát és mértékét jelölik. (21)

## Súlyok optimalizálása

A hibák minimalizálásához a hálózat egyes neuronjainak súlyait kell olyan módon megváltoztatni, hogy ezekkel az értékekkel a veszteségfüggvény a szélsőértékekhez minél közelebbi értékeket vegyen fel. Itt általában a szélsőérték alatt a függvény minimumát értjük, de a keresési módszerek ugyanúgy alkalmazhatók a maximum felderítése esetén is.

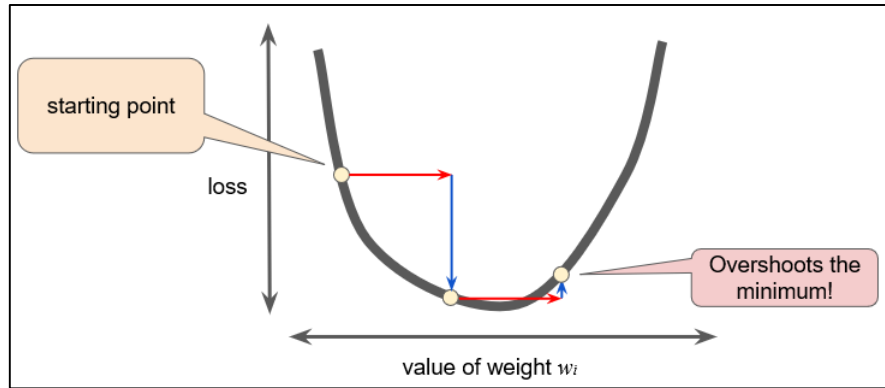
A keresőeljárások során egyrészt használhatunk gradiens alapú eljárásokat a szélsőérték meghatározására, ha a veszteségfüggvény folytonosan deriválható. Másrészt használhatunk valamilyen véletlen vagy determinisztikus algoritmust, amellyel feltérképezzük a lehetséges paramétereket és a hozzájuk tartozó hibák méretét, ekkor a megoldás a hibák megfelelő szélsőértékhez tartozó paraméter lesz. Ezeket a módszereket általában akkor használjuk, ha a gradiens módszer nem alkalmazható, vagy a gradiens analitikus vagy numerikus meghatározása túl bonyolult. [8] [13]

### Gradiens Leereszkedés – Gradient descent

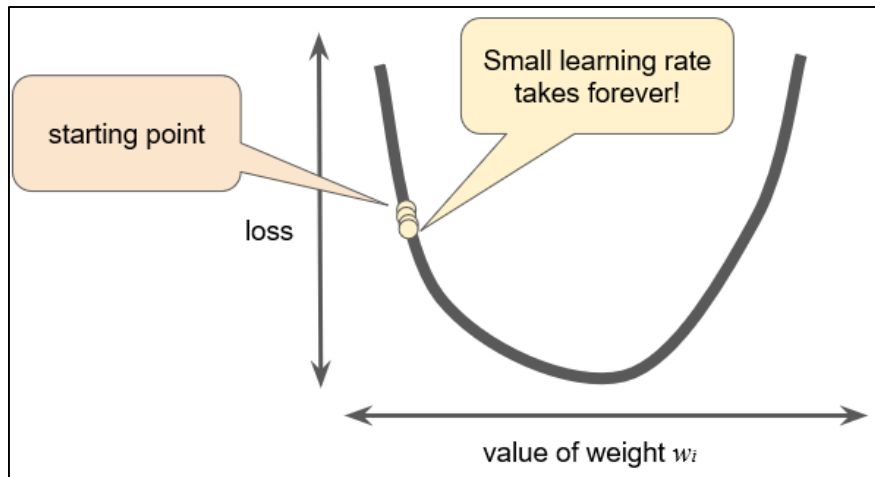
Ez egy iterációs módszer, ahol lépésenként változtatjuk a paramétereket (vagyis a neuronok súlyát) úgy, hogy gradiens irányába lépegetünk, ahol egy lépés mérete a gradiens mérete megszorozva a tanulási rátával, amíg el nem értük a szélsőértéket, vagy elég közel nem vagyunk hozzá. Ez onnan látszik, hogy az átlagos veszteség nem, vagy csak alig változik két lépés közt.

A paraméterek kezdetben egy véletlenszerű (nem 0, illetve nem egyforma) értékkel rendelkeznek, majd az algoritmus során, figyelembe véve a függvény adott pontjában a meredekséget, lépésenként közelíti a lokális minimumot és finomítja a súlyok értékét egész addig, amíg a veszteségfüggvény lehető legalacsonyabb pontjához nem ér.

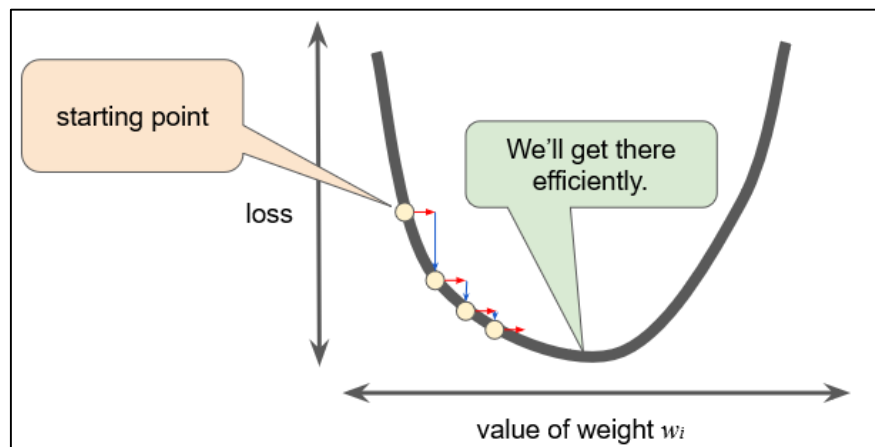
Tanulási rátának („*learning rate*”) nevezzük azt az értéket, amivel befolyásolni tudjuk, hogy a gradiens leereszkedés során az egyes lépések mekkorák legyenek, vagyis hogy milyen gyorsan vagy lassan közelítsük meg az optimális súlyokat. Nagyon fontos, hogy *jó* tanulási rátát válasszunk, mert ha túl nagy, akkor valószínűleg átlépjük a szélsőértéket, ha viszont túl kicsi, akkor nagyon sok ideig is tarthat, amíg megközelítjük a minimumot. [14] [15]



18. Ábra – Túl nagy tanulási ráta. (22)



19. Ábra – Túl kicsi tanulási ráta. (22)



20. Ábra – Jó tanulási ráta. (22)

### Hiba-visszaterjesztés – Backpropagation

Egy olyan algoritmus, amellyel egy neurális hálózat lépésenkénti gradienseit számíthatjuk ki. A célja a hálózat súlyainak optimális beállítása, hogy az elvárt kimenethez minél közelebbi eredményt kapjunk, vagyis minél kisebb legyen a hálózat átlagos hibája.

Az algoritmus lényege, hogy visszaterjesztjük a hibát a kimeneti rétegtől indulva a – normál működés szerinti – legelső rétegig, miközben keressük az optimális súlyokat az egyes rétegekben szereplő neuronokhoz. A szükséges gradiensek kiszámolásához a veszteségfüggvény súlyok szerinti parciális deriváltjait és a láncszabályt használjuk. [8] [9] [16]

### Adam-optimizer

Az Adam-optimizer, amelynek neve az „*adaptive moment estimation*” fogalomból származik, egy iteratív algoritmus, ami kiterjesztése a sztochasztikus gradiens leereszkedésnek.

Abban különbözik az eredeti algoritmustól, hogy nem egy globális tanulási rátája van, hanem a hálózat minden súlyára egy saját tanulási rátát tart számon, amelyeket egyesével adaptív módon számít ki a veszteségfüggvény gradienseinek első és második momentumából. Ehhez két másik algoritmus előnyeit építi magába:

- *Adaptive Gradient Algorithm (AdaGrad)* paraméterszerinti tanulási rátáját, amely növeli az algoritmus teljesítményét ritka gradiensekkel – vagyis olyan gradiensek, amelyek a problémátér kis részén vesznek csak fel nem-nulla értéket – rendelkező problémák esetén. [17]
- *Root Mean Square Propagation (RMSProp)* paraméterszerinti tanulási rátáját, amelyet egy adott súly nemrég kiszámolt gradiensek átlagának mintájára adaptál. Ez olyan feladatok esetén gyorsítja az optimalizációt, ahol valamilyen zavaró tényező van jelen a bemeneti adatok között, például zajos képek.

Az algoritmus egy mozgó exponenciális átlagot számol a sima és négyzetes gradiensekre, valamint  $\beta_1$  és  $\beta_2$  (általában 1 közeli kezdeti értékű) paramétereket használ a fent említett két mozgó átlag bomlási rátájának számontartására. [18] [19]

## Convolutional Variational Autoencoder (CVAE)

### Konvolúciós neurális hálózat (CNN)

Ezeket a hálózatokat főként képelemzési, képosztályozási feladatok megoldására használják, amelyekhez felügyelt tanítási módszert alkalmaznak. Valamint ezen hálózatok által megtanult képek tulajdonságait használják még képaláírások generálására is. Az ilyen típusú hálózatok bemenete egy kép mátrixa, amelyben az egyes értékek a megegyező koordinátán elhelyezkedő pixelek tulajdonságait írják le – például a színét, világosságát.

A hálózat rétegei úgy vannak elhelyezve, hogy az elsők csak kisebb felületeket „látanak” és az egyszerűbb részleteket ismerik fel egy képen (pl. vonalak), a későbbiek pedig már egyre nagyobb részét látják egy képnek, így bonyolult mintákat tudnak azonosítani (pl. tárgyak, arcok). [20]

### Autoencoder

Az autoencoder egy olyan mesterséges neurális hálózat, amely felügyelet nélküli tanulással a bemeneti adatokat tömöríti („*encoding*”), általában zajszűréssel és dimenziócsökkentéssel, illetve utána a tömörített változatból az eredeti adatot rekonstruálja („*decoding*”). Az ilyen típusú neurális hálózati modellek esetén az a rejtett tér („*latent space*”), ahova a bemeneteiket vetítik és ahova azok tömörített változatát helyezi. A rejtett térben klasztereket alkotnak a hasonló tulajdonságokkal rendelkező tömörített adatok, aminek hatására a decoder csak olyan adatokat képes jól rekonstruálni, amelyeket korábban már látott. [9]

### CVAE felépítése

A variational autoencoder-ek (VAE) generatív hálózati modellek, amelyek alapja az autoencoder modell. Egy VAE segítségével a bemenetek (képek) új, módosított változatait lehet generálni. Ebben az esetben az autoencoder rejtett terében kevésbé egyértelműen választhatók szét a klaszterek, lehetővé téve így olyan pontok értékének jobb becslését és dekódolását, amelyekkel a decoder még nem találkozott. A CVAE egy olyan kiterjesztett változata a VAE modellnek, amely által generált kimenetek, képek nem feltétlenül csak véletlenszerű változásokkal rendelkeznek a bemenethez képest, hanem specifikusan bizonyos részei is módosulhatnak. [21] [22]



21. Ábra – Specifikus változtatások balról jobbra: bajusz, majd szemüveg hozzáadása. (23)

### **A szakdolgozatban használt CVAE felépítése**

Két többrétegű hálóból áll össze az itt használt CVAE, egy generatív („*generative net*”) és egy inferencia hálóból („*inference net*”).

Az inferencia háló felel a bemeneti adatok tömörítéséért, vagyis ez a hálózatban az encoder. Neurális hálózatokban az „*inference*” általában azt jelenti, hogy a még soha nem látott adatok rejtett változóinak eloszlását becsüli, amelyhez a korábban vizsgált bemenetekre elállított rejtett változók értékeit használja. [23]

A generatív háló feladata a bemeneti adatok rekonstruálása az inferencia háló által előállított reprezentációk alapján, más szóval ez a hálózatban a decoder. Illetve ha egy bizonyos bemenettípussal még nem találkozott, akkor rejtett térben szereplő becslések alapján kell a kimenetét előállítania.

A két háló egymás felépítését tükrözi, az inferencia háló két konvolúciós és egy teljesen kapcsolt rétegből áll össze, míg a generatív háló egy teljesen kapcsolt, és három transzponált konvolúciós (dekonvolúciós) rétegből áll.

A súlyokat az Adam-optimizer segítségével állítja be, valamint a hiba számításához az „*evidence lower bound*” (ELBO) függvényt használjuk. [24]



## Technológiák

### Keras API

Egy high-level API, amely segítségével építhetünk és taníthatunk neurális hálózatokat. Az API elérhetővé és konfigurálhatóvá teszi a hálózatok építőelemeit, mint például különböző típusú aktivációs függvényeket, rétegeket, veszteségfüggvényeket, optimizereket.

### TensorFlow

Egy, a Google által 2015-ben forgalomba hozott, open-source framework, amely rugalmas eszközöket nyújt neurális hálózatok, gépi tanulási módszerek implementálására, úgy hogy a felhasználók a modelljeik szerkezetére fókuszálhassanak a matematikai részletek helyett.

A TensorFlow *keras* csomagja tartalmazza a Keras API egy implementációját, amely támogatást nyújt a TensorFlow-specifikus működéshez. Például az *eager execution*, ami egy imperatív környezetet biztosít a hálózat tanítása közben végrehajtott műveletek (rész)eredményének könnyebb számontarthatósága érdekében.

A neurális hálózatok gráfként vannak reprezentálva, ahol az egyes csúcsok matematikai műveleteket jelölnek, az élek pedig a csúcsok közt mozgó adattömböket (tenzor) jelölik. A tenzorok, amelyek az egyik fő alkotóelemei az API-nak, alatt értjük a skalárokat, vektorokat és mátrixokat is, amelyeket a TensorFlow-ban  $n$ -dimenziós tömbökkel jelölünk. Minden tenzor rendelkezik egy adattípussal (például `float32` vagy `string`), amely a tenzorban szereplő összes elem típusa, és egy alakkal (`shape`), ami a tömbök dimenzióját – vagy részdimenzióját – adja meg, illetve azt, hogy az egyes dimenziókban hány elem szerepel. [9] [25] [26]

## Felhasználói dokumentáció

A programon keresztül útvesztőket készíthetünk, ezeket vagy az alkalmazásban implementált gráfalgoritmusok segítségével generálhatjuk, vagy olyan neurális hálózatok által, amelyek az előbb említett algoritmusok kimenetét használta/használja a tanulása során. Az alkalmazás grafikus felületén (továbbiakban GUI) különböző beállítások elérhetők, amelyekkel egyrészt a tanítóhalmazok tartalmát és a neurális háló tanítási idejét befolyásolhatjuk, illetve generálhatunk példákat egy-egy algoritmus kiválasztásával, vagy egy CVAE által, amelyet korábban tanítottunk.

### Az alkalmazás és a környezet felépítése

Az alkalmazás futtatásához a Windows 10 operációsrendszer ajánlott. A program az alábbi specifikációknak megfelelő rendszeren lett tesztelve, így megegyező vagy magasabb kapacitású hardverelemek használata ajánlott:

- Processzor, CPU: Intel® Core™ i7-4800MQ; 2,70 GHz
- Memória, RAM: 8GB
- Videókártya: NVIDIA Quadro K1100M. Az alkalmazás csak NVIDIA videokártyával rendelkező számítógépen futtatható.

Másoljuk át az alkalmazást a számítógépre egy tetszőleges helyre, ami a továbbiakban a `${proj_home}` helytartóval lesz jelölve, illetve minden alkalommal ezt a mappaszerkezetet helyettesítsük be a parancsokba is, ahova az imént helyeztük az alkalmazást (pl.: **C:\Felhasználók\felhasználónév\maze\_generator** vagy **C:\maze\_generator**)

Az alkalmazás használatához az alábbi programok telepítése szükséges:

- Java SE Development Kit 8<sup>2</sup>
- Anaconda Distribution, Python 3.x verzióhoz<sup>3</sup>
- CUDA Toolkit 9.0<sup>4</sup>
- cuDNN 7.0.5 for CUDA 9.0<sup>5</sup>
  - A tömörített fájl tartalmát a C meghajtón az alábbi módon helyezzük el:  
`C:\cudnn-9.0-windows10-x64-v7`
  - Adjuk a PATH környezeti változóhoz az alábbi:  
`C:\cudnn-9.0-windows10-x64-v7\cuda\bin`

---

<sup>2</sup> <https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

<sup>3</sup> <https://www.anaconda.com/distribution/>

<sup>4</sup> <https://developer.nvidia.com/cuda-90-download-archive>

<sup>5</sup> <https://developer.nvidia.com/rdp/cudnn-download>

Amennyiben a fenti alkalmazások sikeresen települtek, futtassuk a következő parancsot:

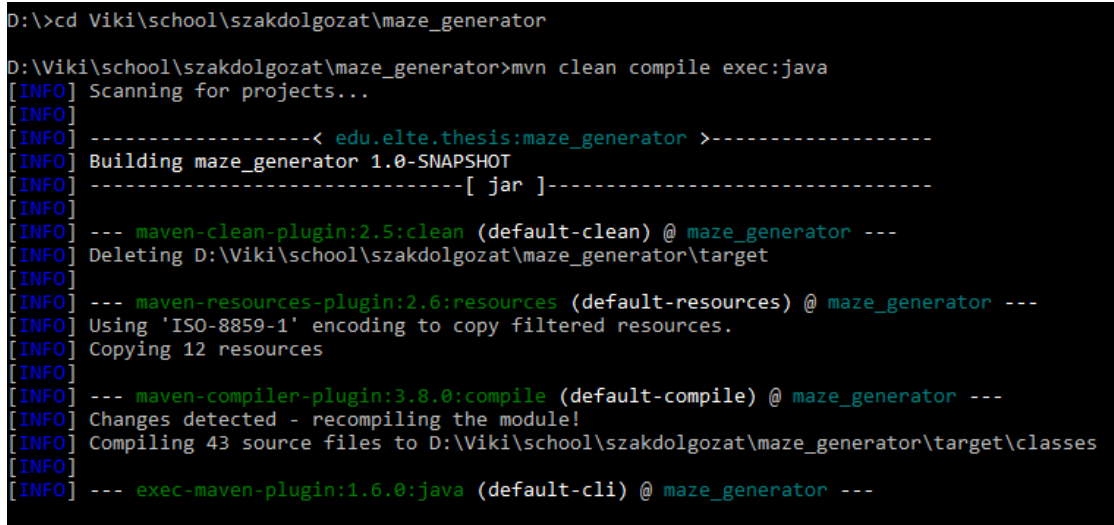
```
conda env create -f  
{proj_home}\src\main\resources\edu\elte\thesis\conda\environment.yaml
```

### Alkalmazás elindítása

Az alkalmazás elindításához futtassuk a parancssorban a következőket:

```
cd ${proj_home}  
mvn clean compile exec:java
```

Ha sikeresen elindult az alkalmazás, akkor alábbi képhez hasonlót látunk.

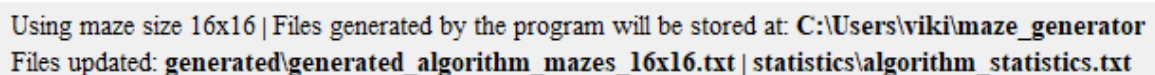


```
D:\>cd Viki\school\szakdolgozat\maze_generator  
  
D:\Viki\school\szakdolgozat\maze_generator>mvn clean compile exec:java  
[INFO] Scanning for projects...  
[INFO]  
[INFO] -----< edu.elte.thesis:maze_generator >-----  
[INFO] Building maze_generator 1.0-SNAPSHOT  
[INFO] -----[ jar ]-----  
[INFO]  
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ maze_generator ---  
[INFO] Deleting D:\Viki\school\szakdolgozat\maze_generator\target  
[INFO]  
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ maze_generator ---  
[INFO] Using 'ISO-8859-1' encoding to copy filtered resources.  
[INFO] Copying 12 resources  
[INFO]  
[INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ maze_generator ---  
[INFO] Changes detected - recompiling the module!  
[INFO] Compiling 43 source files to D:\Viki\school\szakdolgozat\maze_generator\target\classes  
[INFO]  
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ maze_generator ---  
  
Using maze size 16x16 | Files generated by the program will be stored at: C:\Users\wiki\maze_generator  
Files updated: generated\generated_algorithm_mazes_16x16.txt | statistics\algorithm_statistics.txt
```

22. Ábra – Futtatás parancssorból.

A parancssori üzeneteken kívül megjelenik az alkalmazás ablaka is, amellyel különféle módon és méretben készíthetünk labirintusokat. Az ablak három fő részből áll:

1. Egy információs panel, ami az ablak felső részén található. Ez a sáv felel azért, hogy általános információt nyújtson a felhasználónak (pl.: hova menti az újonnan létrehozott fájlokat), illetve értesítse az alkalmazás, vagy az általa használt fájlok állapotváltozásairól (pl.: pontosan melyik mappában és melyik fájl változott meg).



```
Using maze size 16x16 | Files generated by the program will be stored at: C:\Users\wiki\maze_generator  
Files updated: generated\generated_algorithm_mazes_16x16.txt | statistics\algorithm_statistics.txt
```

23. Ábra – Általános információ az útvesztő méretéről és a generált fájlok helyéről (felső sor), változott fájlok listája függőleges vonallal elválasztva (alsó sor)

2. Középen látható az a felület, amelyen a generált útvesztők jelennek meg.

3. Az ablak bal oldalán található panelen két fül közül választhatunk:
- a. **Train:** Egy (létező vagy új) neurális hálózatot taníthatunk labirintusok generálására, amihez vagy egy létező tanítóhalmazt használunk hozzá egy fájlnev megadásával, vagy új tanítópéldákat készíthetünk a felsorolt algoritmusokkal.
  - b. **Generate:** Tetszőlegesen választott algoritmussal, vagy egy – már betanított – neurális hálóval generálhatunk labirintust.

Train

Generate

Training Data Preferences

☒ Load Training Data
 ☐ Generate Training Data

-Load Training Data-

File containing the training data:

-Generate Training Data-

Size of the mazes:

5

Number of the mazes:

10,000

Select one or more algorithms:

☐ growingTree
 ☐ huntAndKill
 ☐ randomWalk
 ☐ recursiveBacktracker
 ☐ sidewinder

VAE Preferences

☒ Train Existing Model
 ☐ Train New Model

Training epochs:

20

☐ Use default model if present

Submit

24. Ábra – A Train fül tartalma.

Train

Generate

Generate with algorithm

Size of the maze:

5

☐ growingTree
 ☐ huntAndKill
 ☐ randomWalk
 ☐ recursiveBacktracker
 ☐ sidewinder

Generate

Generate with CVAE

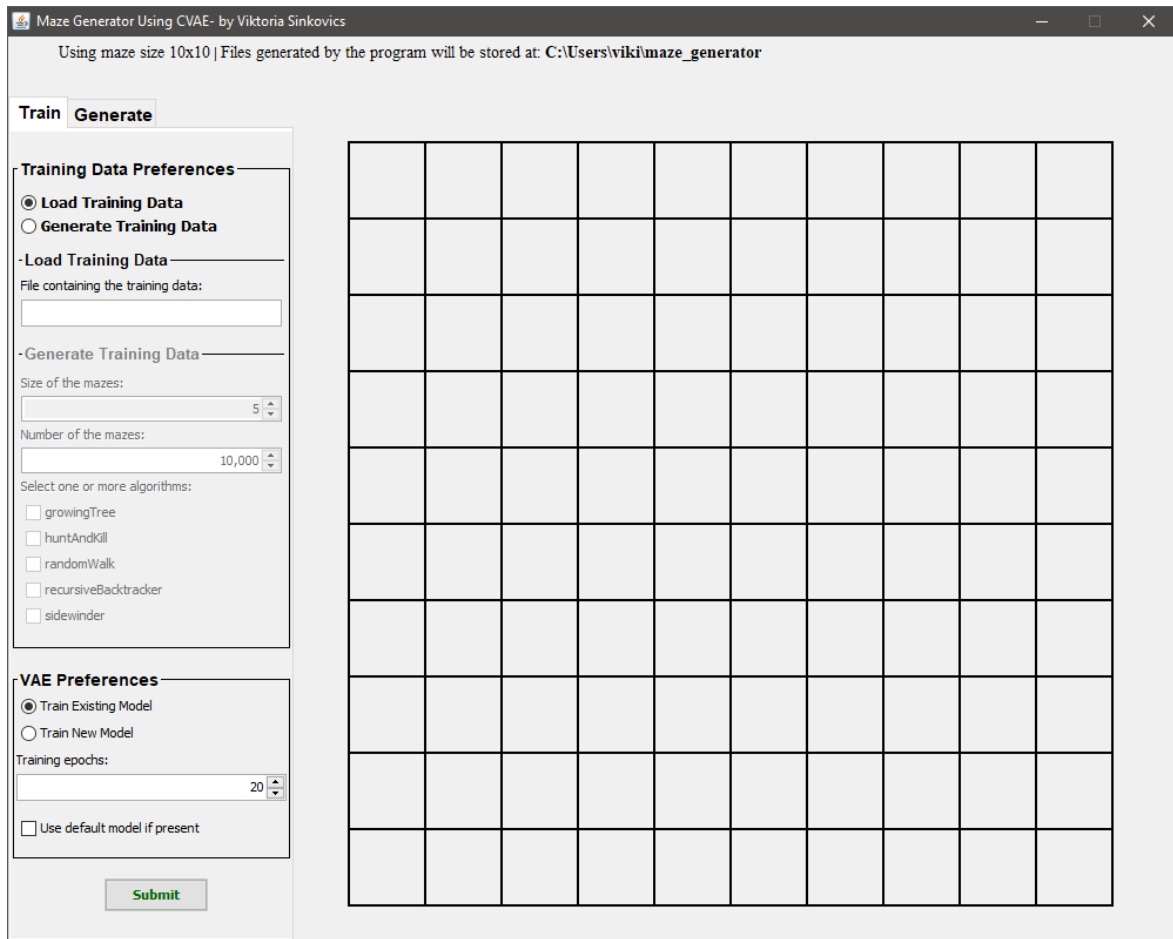
Size of the maze:

5

☐ Generate with default model if present

Generate

25. Ábra – A Generate fül kinézete.



26. Ábra – Az alkalmazás kinézete induláskor.

## Az alkalmazás fájljai

### Alapértelmezett tanítóhalmazok és modellek

Az alkalmazáshoz mellékelve találhatók alapértelmezett tanítóhalmazok és neurális hálózatok modelljei a

`${proj_home}\src\main\resources\edu\elte\thesis\data\` mappában.

A tanítóhalmazok a `training_data` mappában helyezkednek el, amelyben különböző méretű labirintusokat tartalmazó fájlok vannak, amelyek a projektben található algoritmusok által lettek generálva. Például a `training_data_5x5.txt` fájl 30.000 darab 5\*5-ös JSON formátumú labirintust tartalmaz.

A `cvae` mappában találhatók az előre tanított neurális hálózatok, amelyek mindegyike más méretű útvesztők generálására képes. Ezeket a modelleket betölthetjük az alkalmazásba, majd taníthatjuk őket, vagy használhatjuk őket labirintusgenerálásra a jelenlegi állapotukban.

### Az alkalmazás által generált fájlok

Minden esetben, amikor új fájlokat hoz létre az alkalmazás, a felhasználó home könyvtárába (Windows esetén ez a `C:\Felhasználók\felhasználónév` vagy a `C:\Users\felhasználónév`) helyezi el őket egy `maze_generator` nevű mappa alkönyvtárába.

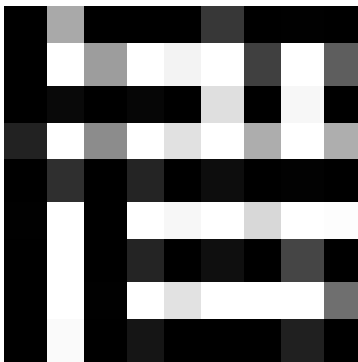
A továbbiakban a `SIZE` változó a labirintusok méretével egyezik meg (pl.: 10x10 formátumban), kivéve ahol külön magyarázat szerepel utána. A `DATE` változó az aktuális dátumot jelöli `YYYY-MM-DD` formátumban (pl.: 2019-04-28). Az `EPOCH` az aktuális epoch sorszámát jelöli (pl.: a 12. epoch a 0012-vel egyezik meg).

- cvae:

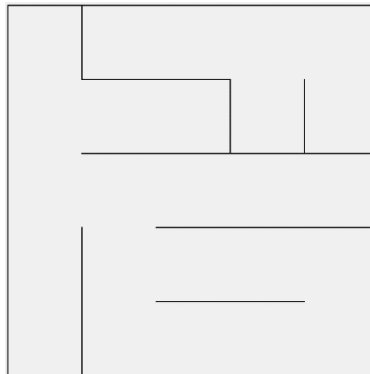
Új modell tanítása esetén az exportált CVAE fájlját menti ide. A fájlok `cvae_model_SIZE_DATE.h5` formátumúak. Tanítás során az alkalmazás minden 10. epoch végén, valamint a tanítás végeztével elmenti a modell állapotát ebbe a fájlba. A modellek által generált labirintusokat kétféleképpen menti le.

Egyrészt JSON formátumban a `generated_cvae_mazes_SIZE.txt` nevű fájlokba – ezek lesznek a GUI-n megjelenítve minden epoch végén.

Másrészt minden epoch-ban a `SIZE_image_at_epoch_EPOCH.png` nevű képfájlokba, ahol a `SIZE` az útvesztők mérete (pl. 005 egy 5\*5-ös labirintus esetén, és 055 egy 55\*55-ös esetén). Ugyanebbe a fájlba menti azokat a labirintusokat is, amelyeket akkor készít, amikor nem tanul, csak a jelenlegi tudása alapján generál egy útvesztőt (GUI `Generate` füle alól betöltve).



27. Ábra – CVAE által generált labirintus képe.



28. Ábra – CVAE által generált JSON-ból kirajzolt labirintus.

```
{ "rows" : [  
  "1111111111",  
  "1111111111",  
  "1010000001",  
  "1011110101",  
  "1000001010",  
  "1011111111",  
  "1000000001",  
  "1010111111",  
  "1010000001",  
  "1010111101",  
  "1010000001",  
  "1111111111"  
]}
```

29. Ábra – CVAE által generált JSON formátumú labirintus

- generated:  
Ebbe a mappába egy `generated_algorithm_mazes_SIZE.txt` nevű fájlba menti a labirintusokat az alkalmazás, amikor a `Generate` fülön algoritmussal generálunk útvesztőket.
- statistics:  
Ha egy – vagy több – algoritmussal generálunk labirintusokat, az alkalmazás elmenti, hogy mennyi idő alatt készült el adott mennyiségű és méretű útvesztő. Ezeket az `algorithm_statistics.txt` fájlba menti el. Ha csak egy darabot generált, akkor a generálás befejezésének időpontját, a tartamát, a használt algoritmus nevét és az útvesztő méretét menti el.  
  
[2019-04-10 16:42:58.753]: 0h 0m 4s - growingTree - 100x100  
Amennyiben többet generált, akkor ugyanazokat az adatokat menti le, mint az előző esetben, illetve hozzáteszi az elkészített labirintusok darabszámát is.  
[2019-04-25 09:04:03.849]: 0h 25m 18s - randomWalk - 50x50 - 5000 pcs.  
  
Ha a neurális hálózat dolgozik, akkor az egyes epoch-okról a `cvae_statistics.txt` fájlba menti az információkat, vagyis az epoch sorszámát, a hiba méretét (ELBO), illetve hogy mennyi ideig tartott az epoch másodpercben.  
  
Epoch: 21, Test set ELBO: -24.004655963897704, elapsed time for current epoch 21.91051173210144  
Epoch: 22, Test set ELBO: -23.996582332611084, elapsed time for current epoch 21.78203010559082
- training data:  
Amikor az alkalmazás `Train` fülén új tanítóhalmazt készítünk a neurális hálózat számára, akkor a halmaz tartalmát ebbe a könyvtárba menti `training_data_SIZE_DATE.txt` fájlnevével, amelyben JSON formátumú útvesztők szerepelnek.

## A GUI áttekintése

Induláskor az alkalmazás `Train` fülén találjuk magunkat, ahol a `Training Data Preferences` dobozban kiválaszthatjuk, hogy betöltünk egy létező tanítóhalmazt (`Load Training Data`), vagy újat készítünk (`Generate Training Data`). Hasonlóan a neurális háló beállításainál (`CVAE Preferences`) választhatunk létező (`Train Existing Model`) és új háló tanítása közt (`Train New Model`). Attól függően, hogy melyiket választjuk, más további beállítások válnak elérhetővé. Alapértelmezett opcióként a `Load Training Data` és a `Train Existing Model` lehetőségek vannak kiválasztva.



A beállítások alatt található egy **Submit** gomb, amelyet megnyomva az alkalmazás megvizsgálja, hogy minden szükséges lehetőséget kitöltöttünk és/vagy kiválasztottunk, illetve hogy ezek helyesek-e (pl.: létezi-e a megadott fájl).

Ha átlépünk a **Generate** fülre, akkor algoritmusokkal vagy modellekkel generálhatunk egy-egy példát. Mindkét esetben egy **Generate** gombbal véglegesíthetjük a paramétereket.

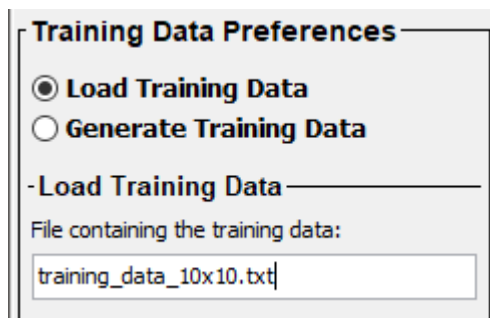
## Neurális hálózatok tanítása a generátor algoritmusok segítségével

### Tanítóhalmazok beállítása

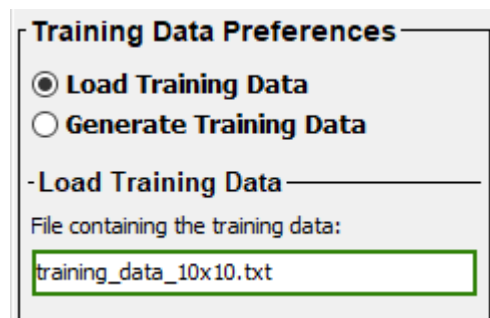
#### Load Training Data

Amennyiben a **Load Training Data** lehetőséget választjuk, egyetlen további beállítás válik elérhetővé, ahol megadhatjuk a fájl nevét, amely tartalmazza a tanítóhalmazt. Mivel az alkalmazás csak két helyen fogja keresni a fájlokat – az alapértelmezett tanítóhalmazokat tartalmazó mappában a projekten belül, illetve a felhasználó **home** könyvtárában, így ebben a mezőben csak a fájl nevét és a kiterjesztését kell megadnunk.

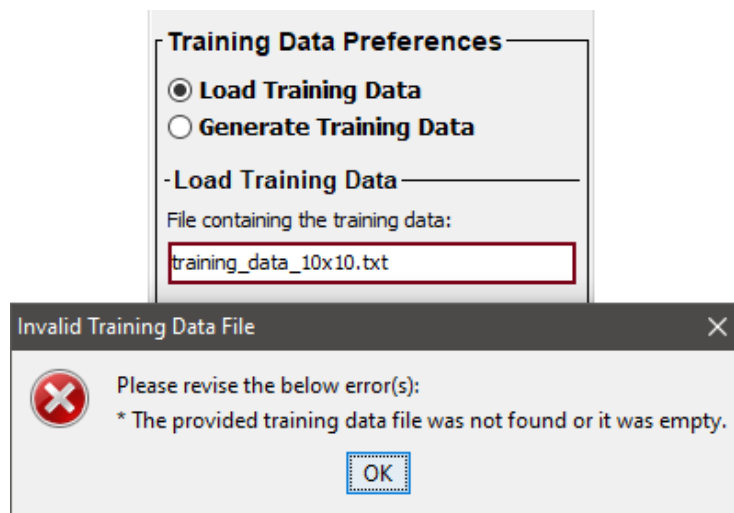
A **Submit** gomb megnyomása után, ha a fájl megtalálható, akkor a mezőt zöld körvonallal jelöli ki, egyébként ha nem létezik, vagy nem tartalmaz adatot, akkor piros lesz a körvonala és egy hibüzenet jelenik meg.



30. Ábra – A tanítóhalmazt tartalmazó fájl nevének megadása.



31. Ábra – A fájl létezik



32. Ábra – A fájl nem található, vagy üres.

### ***Generate Training Data***

Ha ezt a lehetőséget választjuk, akkor három további beállítás válik elérhetővé.

- Méret:

A `Size of the mazes` felirat alatt beállíthatjuk, hogy mekkora labirintusokat szeretnénk generálni – a program által generált útvesztők mindegyike szimmetrikus méret szempontjából. Ide beírhatjuk a választott méretet, vagy a mező jobboldalán található nyilak segítségével növelhetjük, illetve csökkenthetjük az értéket. A labirintusok mérete csak szám formában adható meg, illetve legalább 5 és maximum 100 lehet az értéke.

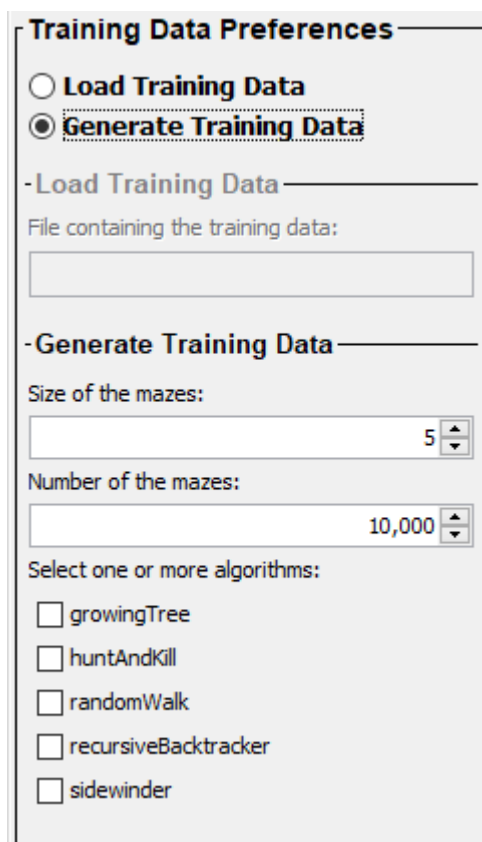
Amennyiben nem szám formátumban adjuk meg, de az érték határon belüli számmal kezdődik, a program a számot megtartja, és a szövegrészt elveti (pl.: 23asd esetén 23 lesz a méret). Egyébként, ha betűvel kezdődik az érték, vagy a határokon kívüli számot (vagy azzal kezdődő szöveget) adunk meg, akkor ezt elveti a program és visszaállítja a mező értékét a legutoljára benne szereplő helyes számra. Például ha a mező értéke 23 volt, majd 101a-t írunk be, az érték visszaáll 23-ra, amint kiléptünk a mezőből.

- Darabszám:

A `Number of the mazes` felirat után adhatjuk meg, hogy hány elemű legyen a tanítóhalmaz, ami minimum 10.000 és maximum 100.000 darab labirintust tartalmazhat. A nyilak segítségével változtathatjuk az értéket, egy lépés mérete 1000. Mivel a mező nem szerkeszthető manuálisan, így ezen nem végez validációt az alkalmazás.

- Generáló algoritmus(ok):

A `Select one or more algorithms` után 5 jelölőnégyzetet látunk a generátor algoritmusok neveivel címkézve: `growingTree`, `huntAndKill`, `randomWalk`, `recursiveBacktracker`, `sidewinder`. Ezek közül választhatjuk



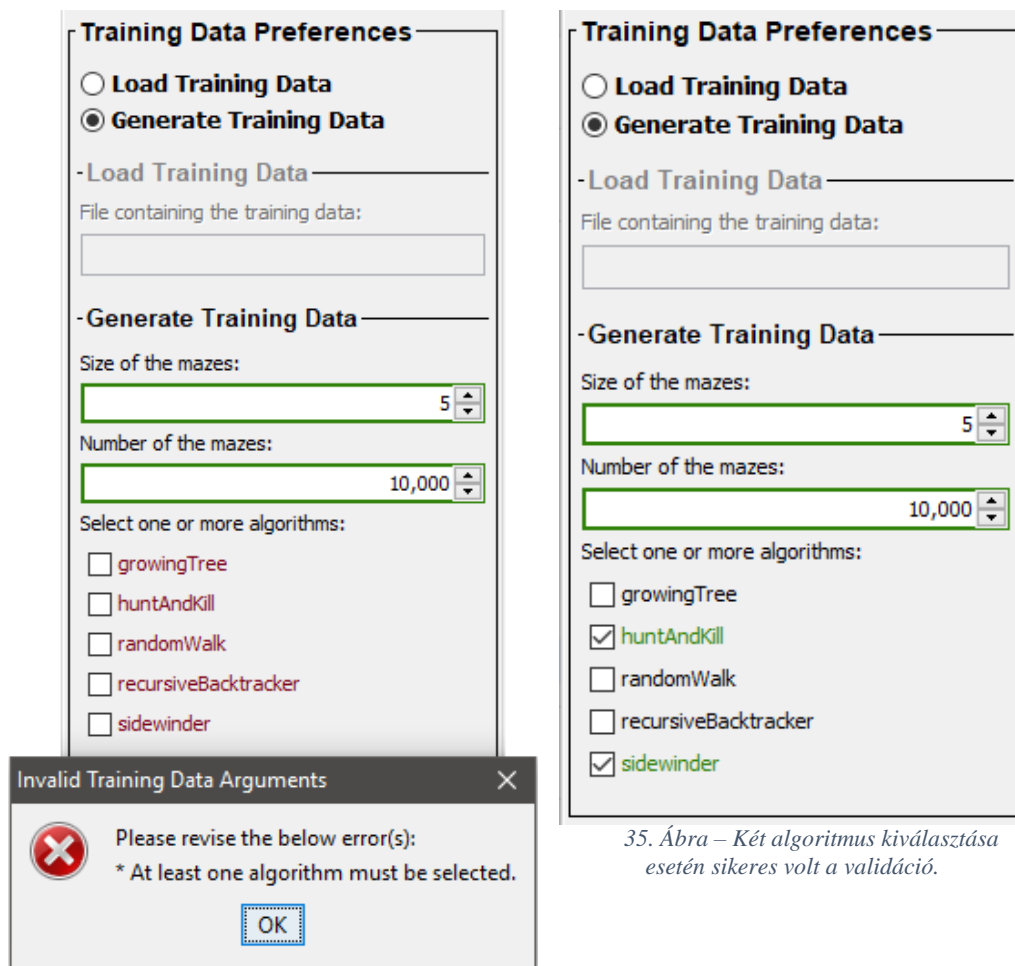
The image shows a window titled "Training Data Preferences". It contains two radio buttons: "Load Training Data" and "Generate Training Data", with the latter selected. Below the radio buttons are two sections. The first section, "Load Training Data", has a label "File containing the training data:" followed by an empty text box. The second section, "Generate Training Data", has two spinners: "Size of the mazes" set to 5 and "Number of the mazes" set to 10,000. Below these is a label "Select one or more algorithms:" followed by five checkboxes, each with a label: "growingTree", "huntAndKill", "randomWalk", "recursiveBacktracker", and "sidewinder".

33. Ábra – Tanítóhalmaz generálásához megadható paraméterek elhelyezkedése a GUI-n.

ki, hogy melyik(ek)et használjuk a tanítóhalmaz elkészítéséhez. Itt a lehetőségek közül legalább egyet ki kell választani.

Amennyiben egy sem lett kijelölve, akkor a `Submit` gomb megnyomása után az algoritmusok piros színre váltanak, és egy hibaüzenet jelenik meg, felszólítva a felhasználót arra, hogy legalább egy algoritmus választása szükséges a generálás elkezdéséhez.

Ha legalább egyet kijelölünk, akkor a halmaz készítése megkezdődhet, és a kijelölt algoritmusok neve zöldre vált.



34. Ábra – Hibaüzenet, amikor nincsenek algoritmusok kijelölve.

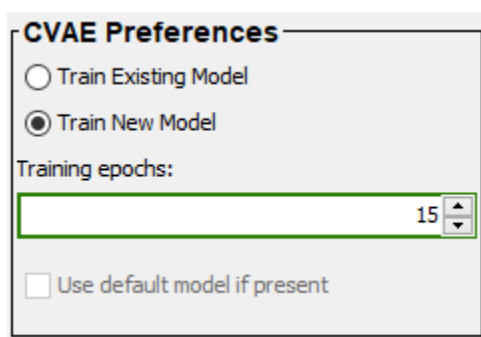
35. Ábra – Két algoritmus kiválasztása esetén sikeres volt a validáció.

Ha a megadott paraméterek vizsgálata sikeres volt, akkor az eredménye késleltetve jelenik meg, illetve a GUI elemeivel ilyenkor nem lehet interakcióba lépni. Ez az állapot egész addig fennáll, amíg az algoritmusok el nem készítették a megadott számú labirintust. Időben tarthat bárhol 1 és 30 perc között is akár, attól függően, hogy hány darab és mekkora labirintusra van szükség, illetve a felhasználó számítógépének teljesítménye is befolyásolhatja.

## CVAE beállításai

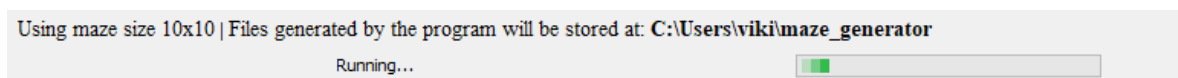
A neurális hálózatoknak a tanulóhalmazokon kívül meg kell adnunk, hogy hány epoch-on keresztül szeretnénk tanítani őket, amit a GUI CVAE Preferences dobozában állíthatunk be a Training epochs címke alatt. Egy hálót legalább 5, legfeljebb 250 epoch-on keresztül taníthatunk, illetve alapértelmezésben a mező értéke 20. Ezen kívül kiválaszthatjuk, hogy létező (Train Existing Modell) vagy új (Train New Modell) CVAE modellt szeretnénk tanítani.

Amennyiben létező modellt használunk, a doboz alján található jelölőnégyzetet bepipálva használhatjuk az alkalmazáshoz mellékelt előre tanított modelleket. Ha nem található olyan modell, amely a megadott labirintusok méretére specializálódott, az alkalmazás ezt egy hibaüzenetben jelzi, illetve a Train Existing Model rádiógomb szövege pirosra vált. Egyébként, ha létezik megfelelő modell (akár az alapértelmezettek, akár a felhasználó által generáltak között), vagy újat készítünk, a hálózat tanítása megkezdődik.

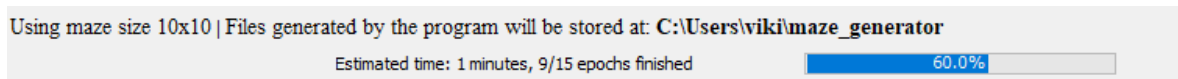


36. Ábra – Új modell tanítása esetén nem számít a tanítóhalmazban szereplő labirintusok mérete, mert az új modell tud hozzájuk igazodni.

Ha sikeresen elindult a modell tanításának folyamata, akkor az információs sávon megjelenik egy határozatlan folyamatjelző sáv. Amint a CVAE végzett az első epoch-al, frissül a folyamatjelző sáv is, amely ezután százalékos formában jelzi a tanulás haladását. Illetve a sáv címkéjében megjelenik, hogy mennyi a becsült hátralévő idő percben és hányadik epoch-nál tart.



37. Ábra – Határozatlan folyamatjelző sáv, amikor a tanítás megkezdődik.



38. Ábra – Határozott folyamatjelzősáv a tanítás közben.

**Train** **Generate**

**Training Data Preferences**

☒ Load Training Data  
☐ Generate Training Data

**-Load Training Data-**

File containing the training data:

**-Generate Training Data-**

Size of the mazes:

Number of the mazes:

Select one or more algorithms:

☐ growingTree  
☐ huntAndKill  
☐ randomWalk  
☐ recursiveBacktracker  
☐ sidewinder

**Invalid Maze Generation Arguments**

Please revise the below error(s):  
 \* Couldn't find any model for the specified size (14x14)

OK

**CVAE Preferences**

☒ Train Existing Model  
☐ Train New Model

Training epochs:

☐ Use default model if present

Submit

39. Ábra – Nem található megfelelő modell, ami illeszkedne a 14\*14-es labirintusokra

**Train** **Generate**

**Training Data Preferences**

☒ Load Training Data  
☐ Generate Training Data

**-Load Training Data-**

File containing the training data:

**-Generate Training Data-**

Size of the mazes:

Number of the mazes:

Select one or more algorithms:

☐ growingTree  
☐ huntAndKill  
☐ randomWalk  
☐ recursiveBacktracker  
☐ sidewinder

**CVAE Preferences**

☒ Train Existing Model  
☐ Train New Model

Training epochs:

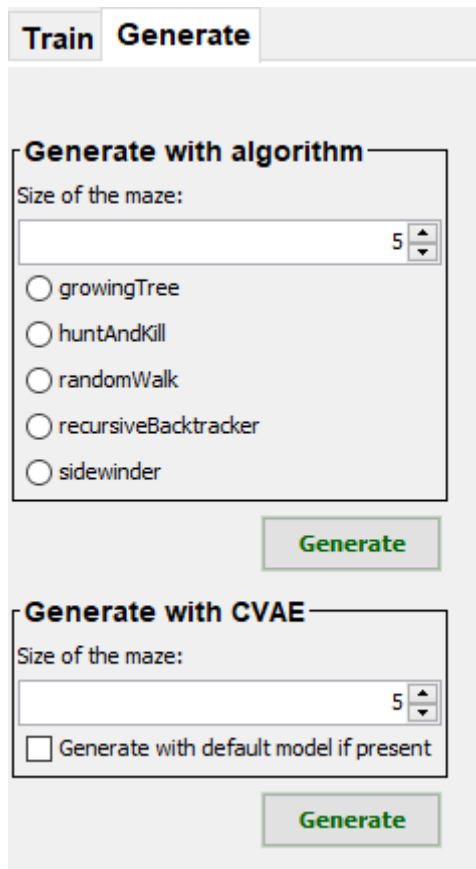
☒ Use default model if present

Submit

40. Ábra – Létezik modell, ami illeszkedik a 10\*10-es labirintusokra

## Bemutatópéldák generálása

A `Generate` fül két dobozzal rendelkezik: az egyikben algoritmusokkal, a másikban betanított neurális hálózatokkal generálhatunk útvesztőket. Mindkét dobozhoz tartozik egy `Generate` gomb, amivel elindíthatjuk a generálás folyamatát.



The screenshot shows the 'Generate' tab of a software interface. It contains two main sections for generating mazes:

- Generate with algorithm:** This section has a 'Size of the maze:' label followed by a text input field containing the number '5'. Below this are five radio button options: `growingTree`, `huntAndKill`, `randomWalk`, `recursiveBacktracker`, and `sidewinder`. A green 'Generate' button is located to the right of these options.
- Generate with CVAE:** This section also has a 'Size of the maze:' label followed by a text input field containing the number '5'. Below this is a checkbox labeled 'Generate with default model if present'. A green 'Generate' button is located to the right of this checkbox.

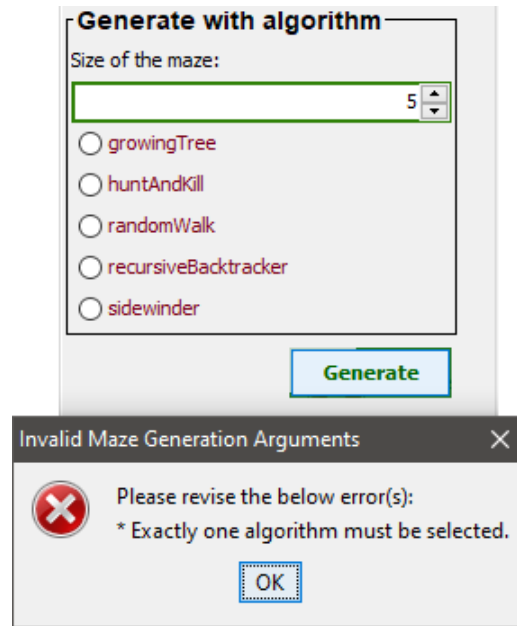
41. Ábra – A `Generate` fül alaphelyzetben.

### Generálás algoritmusokkal

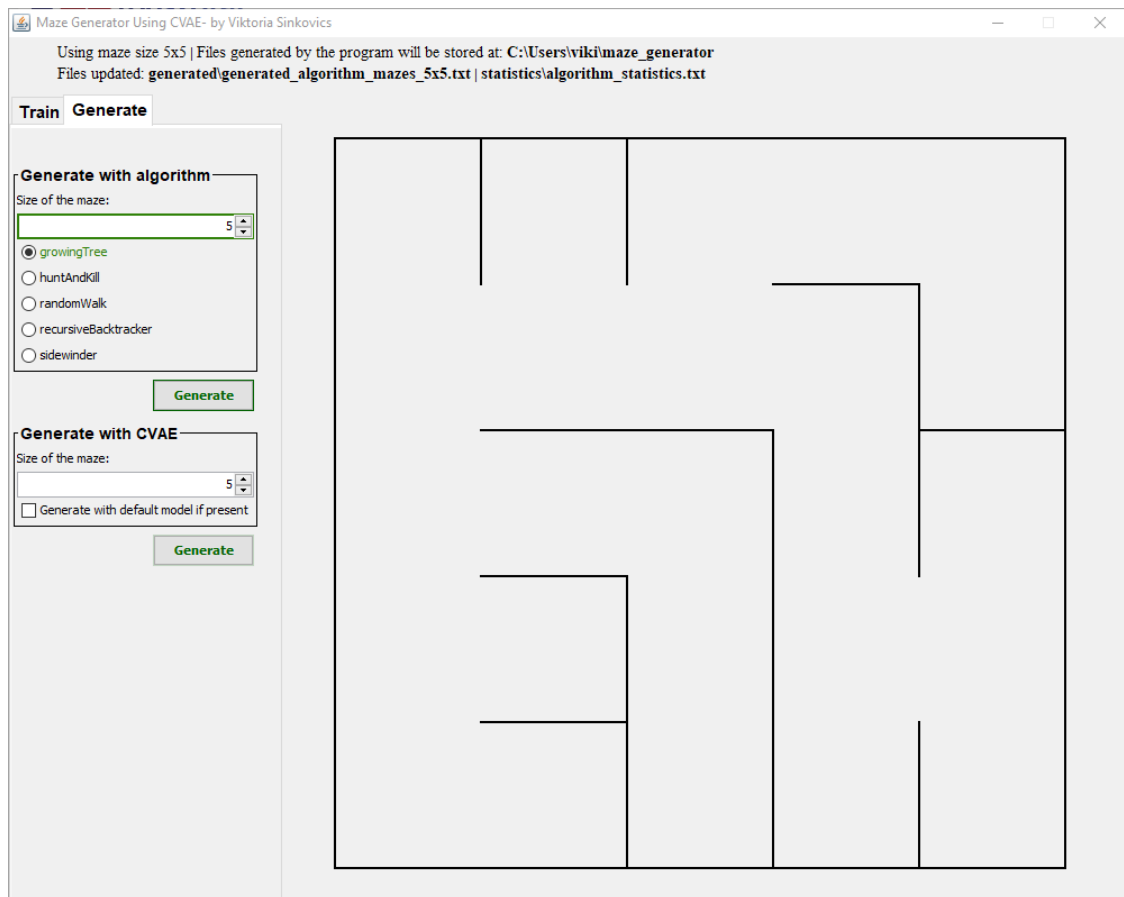
Ha algoritmust szeretnénk használni útvesztők készítésére, akkor meg kell adni a labirintus méretét, amit a `Size of the maze` címke alatti mezőben tehetünk meg. Ez a mező számformátumban fogadja el a méretet, aminek 5 és 100 között kell lennie, illetve az alkalmazás ugyanolyan ellenőrzést végez rajta, mint a tesztalmaz készítésénél.<sup>6</sup>

A felsorolt algoritmusok közül egyet ki kell választani. Ha nem választunk ki egyet se, és megnyomjuk a `Generate` gombot, akkor az alkalmazás egy hibaüzenetet jelenít meg, illetve az algoritmusok nevei átszíneződnek pirosra. Ha kiválasztjuk az egyik lehetőséget, akkor a választott algoritmus neve zöldre vált, és a jobboldali panelen megjelenik a generált útvesztő.

<sup>6</sup> Lásd 31. oldal, „Méret” pont, 2. bekezdés.



42. Ábra – Hibaiúzenet, amikor nincs algoritmus kijelölve.



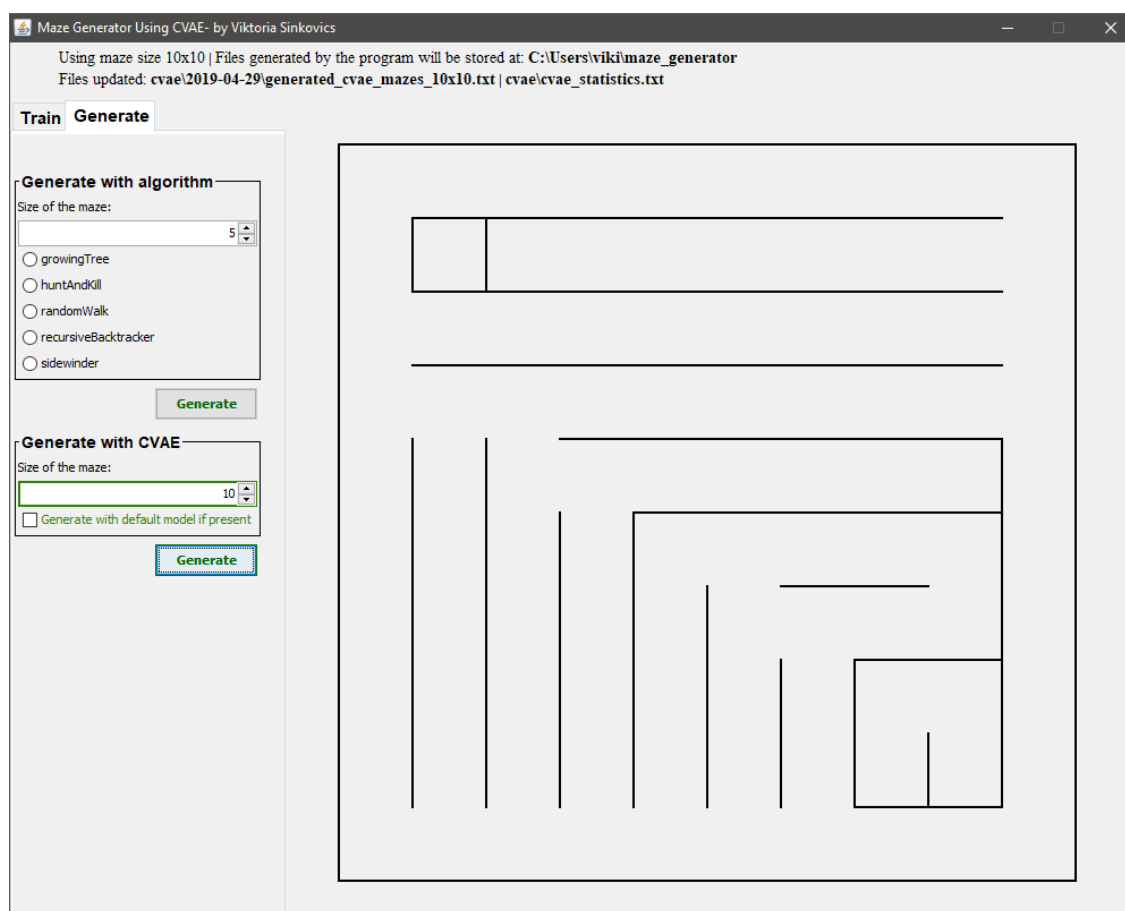
43. Ábra - Algoritmussal generált 5\*5-ös labirintus.

## Generálás CVAE modellel

Neurális hálóval csak akkor tudunk generálni új labirintust, ha a megadott méretre létezik mentett modell.

A generáláshoz meg kell adni a labirintusok méretét a `Size of the maze` címke alatt található mezőben, valamint az érték ellenőrzését ugyanolyan módon végzi az alkalmazás, mint a korábban említett mezőket, amelyek az útvesztők méretéért felelősek.<sup>7</sup> A mezőbe 5 és 100 közötti értéket kell megadni, amit vagy manuálisan írhatunk be, vagy lépkedhetünk a mező jobboldalán található nyilak segítségével. Ezen kívül a doboz alján található jelölőnégyzetet kiválasztva használhatjuk az alapértelmezett modelleket is, amennyiben léteznek.

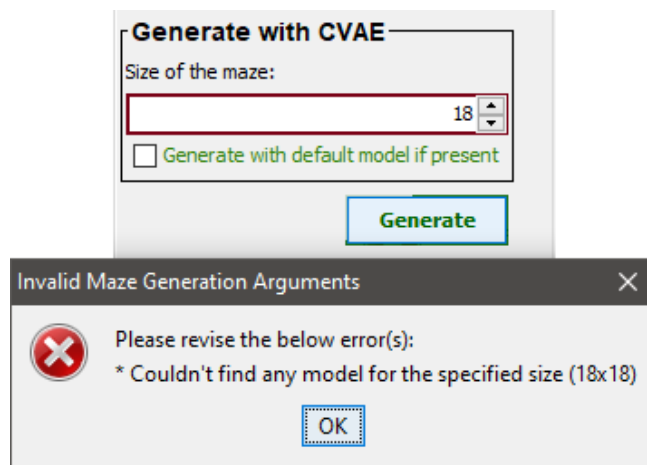
Ha a megadott mérethez nem található modell, akkor az alkalmazás a méret mező piros bekeretezésével, valamint egy hibaüzenettel válaszol. Ha létezik modell, akkor zöldre színezi a doboz tartalmát, majd jobboldali panelen megjelenik az általa generált labirintus.



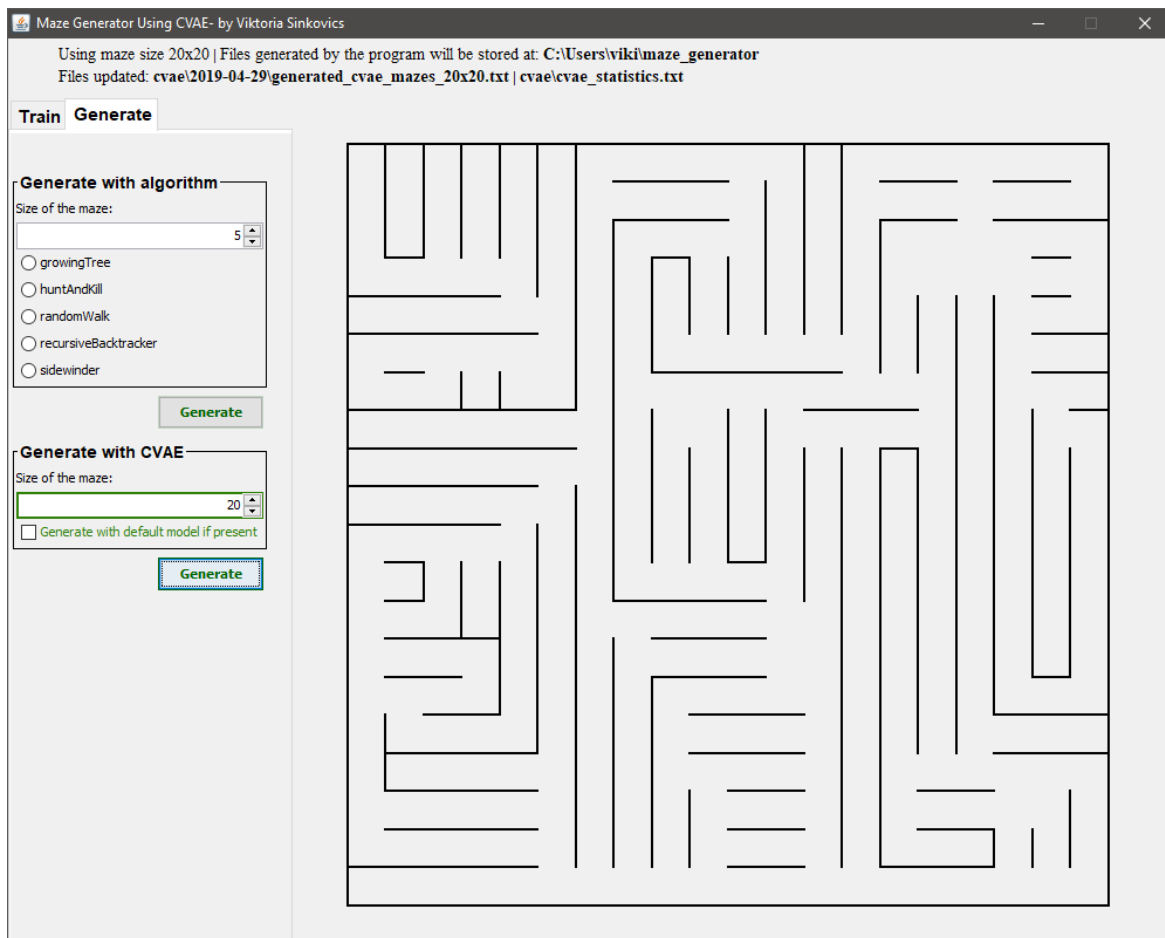
44. Ábra – Létező modell fájl esetén új labirintus jelenik meg jobboldalt.

<sup>7</sup> Lásd 31. oldal, „Méret” pont, 2. bekezdés.





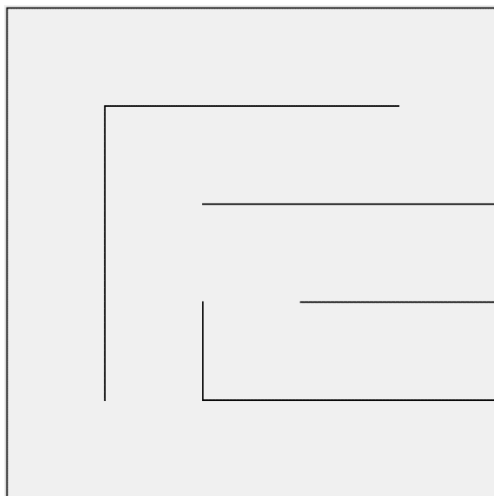
45. Ábra – Ha nem talál megfelelő modellt, akkor hibaüzenettel jelzi az alkalmazás.



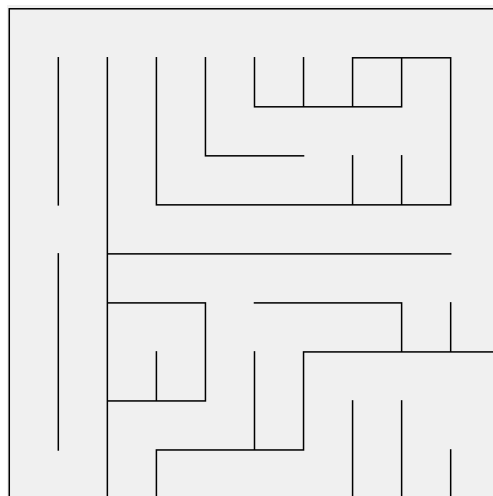
46. Ábra - CVAE által generált 20\*20-as példa labirintus.

A 46. ábrán látható labirintust egy olyan modell generálta, amely összesen 80 epoch-ot tanult, illetve a tanítóhalmaza a growingTree és sidewinder algoritmusok által generált útvesztőkből állt.

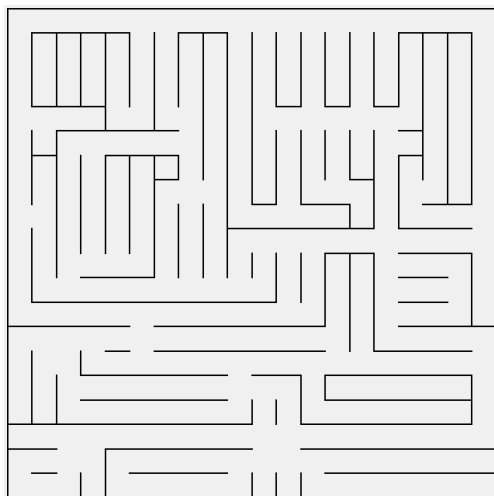
Az alapértelmezett CVAE modellek 5\*5, 10\*10, 20\*20 és 50\*50 méretű útvesztők generálására képesek. Mindegyike 200 epoch-ot tanult, illetve mindegyik egy 10.000 elemű tanítóhalmazzal dolgozott, amely growingTree és sidewinder algoritmusok által generált labirintusokat tartalmazott fele-fele arányban. A mellékelt képeken jól látszik a sidewinder algoritmus befolyása, vagyis az, hogy az útvesztők első sorában nincs fal – kivéve az 50\*50-es példa, ahol inkább hosszabb utak jellemzik az első sort.



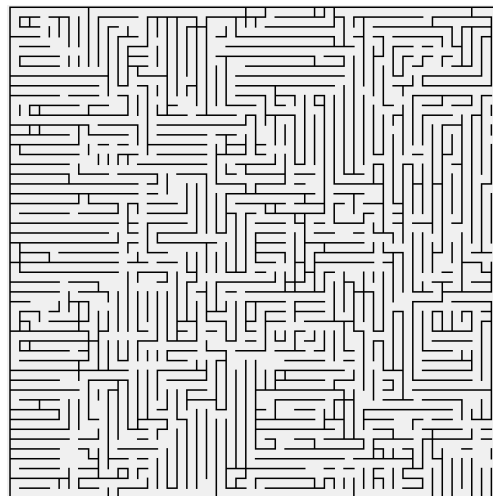
47. Ábra – CVAE által generált 5\*5-ös labirintus.



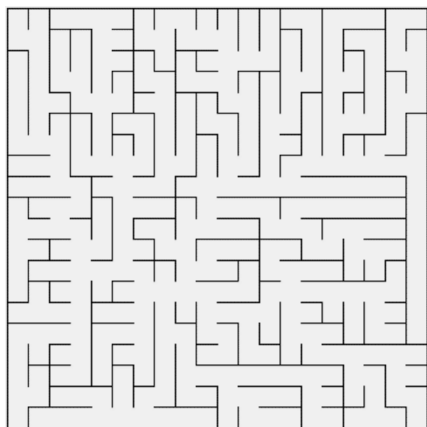
48. Ábra – CVAE által generált 10\*10-es labirintus.



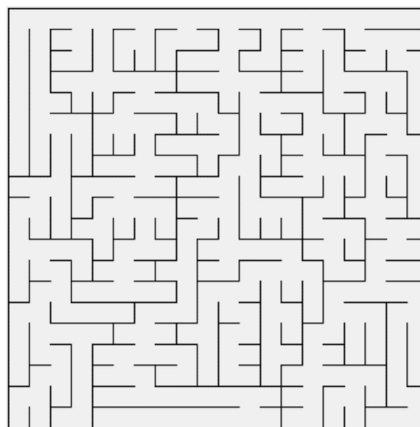
49. Ábra – CVAE által generált 20\*20-as labirintus.



50. Ábra – CVAE által generált 50\*50-es labirintus.



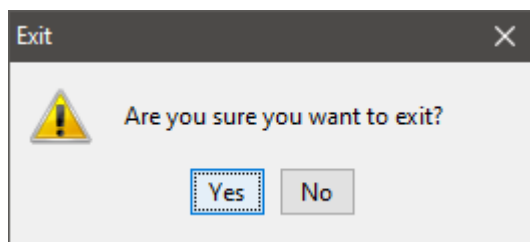
51. Ábra – GrowingTree algoritmus által generált  
20\*20-as labirintus.



52. Ábra – Sidewinder algoritmus által generált  
20\*20-as útvesztő.

### Az alkalmazás leállítása

A bezárás gomb segítségével léphetünk ki az alkalmazásból, amely megnyomásakor megjelenik egy párbeszédpanel, amelyen a Yes gomb megnyomásával megerősíthetjük a kilépési szándékunkat, ekkor az alkalmazás leáll. A No megnyomása esetén folytathatjuk az alkalmazás használatát ott, ahol abbahagytuk.



53. Ábra – Kilépéskor megjelenő párbeszédpanel.

## Fejlesztői dokumentáció

A programon keresztül útvesztőket készíthetünk. Ezt megtehetjük az implementált gráfalgoritmusok segítségével, amelyek körmentes összefüggő útvesztőket eredményeznek. Vagy használhatunk olyan neurális hálózatokat, amelyek az előbb említett algoritmusok kimenetét használta/használja a tanulása során, illetve ebben az esetben a keletkezett labirintusok tartalmazhatnak köröket és lehetnek nem összefüggők is. Az alkalmazás grafikus felületén (továbbiakban GUI) különböző beállítások elérhetők, amelyekkel egyrészt a tanítóhalmazok tartalmát és a neurális háló tanítási idejét befolyásolhatjuk, illetve generálhatunk példákat egy-egy algoritmus kiválasztásával, vagy egy CVAE által, amelyet korábban tanítottunk.

## Hardver és szoftver követelmények

Az alkalmazás Windows 10 operációsrendszeren készült, az alábbi specifikációknak megfelelő hardveren, így megegyező vagy magasabb kapacitású hardverelemek használata ajánlott:

- Processzor, CPU: Intel® Core™ i7-4800MQ; 2,70 GHz
- Memória, RAM: 8GB
- Videókártya: NVIDIA Quadro K1100M. Az alkalmazás csak NVIDIA videokártyával rendelkező számítógépen használható.

A program futtatásához és fejlesztéshez az alábbi programok telepítése szükséges:

- Java SE Development Kit 8<sup>8</sup>
- Anaconda Distribution, Python 3.x verzióhoz<sup>9</sup>
- CUDA Toolkit 9.0<sup>10</sup>
- cuDNN 7.0.5 for CUDA 9.0<sup>11</sup>
  - A tömörített fájl tartalmát a C meghajtón az alábbi módon helyezzük el:  
C:\cudnn-9.0-windows10-x64-v7
  - Adjuk a PATH környezeti változóhoz az alábbi:  
C:\cudnn-9.0-windows10-x64-v7\cuda\bin

A fenti alkalmazások sikeres telepítése után, az alábbi parancs futtatásával építhetjük fel azt a környezetet, amely a neurális hálózathoz szükséges Python könyvtárakat tartalmazza:

```
conda env create -f  
{proj_home}\src\main\resources\edu\elte\thesis\conda\environment.yaml
```

---

<sup>8</sup> <https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

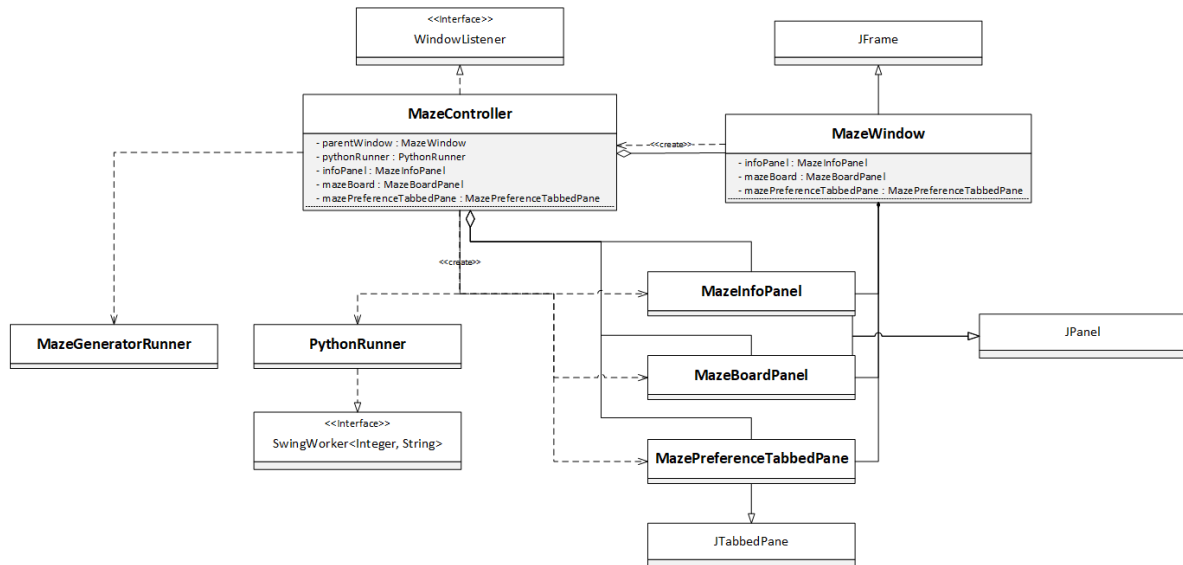
<sup>9</sup> <https://www.anaconda.com/distribution/>

<sup>10</sup> <https://developer.nvidia.com/cuda-90-download-archive>

<sup>11</sup> <https://developer.nvidia.com/rdp/cudnn-download>

## Program szerkezeti felépítése

A projekt egy backend és egy frontend részből áll, amelyeket egy kontroller osztály köt össze. A kontroller hivatkozásokat tart számon a GUI ablakáról (`MazeWindow`), és annak fő paneljeiről, mivel ő felel azok létrehozásáért és frissítéséért. Ezek a panelek az információért (`MazeInfoPanel`), a labirintusok megjelenítésért (`MazeBoardPanel`) és a beállításokért (`MazePreferenceTabbedPane`) felelős részei a GUI-nak. A kontroller szükség szerint meghívja a backend oldali `MazeGeneratorRunner` osztály metódusait, amelyek paraméterezéstől függően hívják a megfelelő generátor algoritmusok osztályait. Ezen felül a kontroller felel azon backend folyamatok létrehozásáért, amelyek a CVAE modelleket létrehozzák és tanítják. Ehhez a `PythonRunner` osztályt példányosítja, amely az `execute()` metódusának hívásakor egy új szálát indít, aminek segítségével a háttérben végrehajtja a megadott parancsokat.



54. Ábra – A `MazeController` kapcsolata a front- és backend komponenseivel.

A GUI-n szereplő `Submit` és `Generate` gombok akcióin keresztül jutnak el a felhasználó kérései a backendhez. A gombok megnyomásakor azok `actionPerformed(event)` metódusa hívódik meg, amely a panelen szereplő mezők állapotát és értékeit összegyűjti, ellenőrzi, majd a kontroller megfelelő metódusának átadja ezeket.

A Submit gomb esetén végrehajtott akció (SubmitButtonAction) a következő metódusokat hívhatja a kontrollerből:

```
public File handleGenerateTrainingData(int mazeSize, int  
mazeCount, List<MazeGeneratorAlgorithm> algorithmNames):
```

Bemeneti paraméterként kapja meg a labirintusok méretét, darabszámát, és hogy melyik algoritmus(ok) által legyenek generálva. A metódus létrehozza az új fájlt, vagy megnyitja, ha már létezik, amibe az újonnan generált labirintusokat menti, illetve megnyitja vagy létrehozza az `algorithm_statistics.txt` fájlt. Ezután meghívja a `MazeGeneratorRunner` osztály `generate()` metódusát, amely elkészíti a kért labirintusokat. Ha a generálás befejeződött, a metódus visszatér az útvesztőket tartalmazó fájlal.

```
public Optional<File> handleLoadTrainingData(String filename):
```

Egy fájlt keres, amelynek neve megegyezik a paraméterként kapott `String`-gel. Először megvizsgálja, hogy egy útvonalat kapott-e, ami egy nem üres fájlra mutat, ha igen, akkor visszatér a fájlal. Egyébként, ha a nem üres alapértelmezett tanulóhalmazokat tartalmazó fájlok egyikével egyezik meg a fájlnev, akkor azzal tér vissza. Különben ha a felhasználó által generált fájlok valamelyikének nevével egyezik meg a paraméter, és az a fájl nem üres, akkor ezt adja vissza. Minden egyéb esetben a megadott paraméter helytelen (üres, vagy nemlétező fájlra utal), ekkor a visszatérési érték `Optional.empty()`.

```
public boolean handleTrainModel(boolean loadModel, boolean  
defaultModel, int mazeSize, File trainingData, int epochs):
```

Ha a `load_model` értéke `false`, akkor új modell fájlt hoz létre. Ha `loadModel` paraméter értéke `true`, akkor megkeresi a modellt tartalmazó fájlt – az alapértelmezett helyen (ha a `default_model` értéke `true`) vagy a felhasználó home könyvtárában. Ha a modell fájl nem található, akkor `false` értékkel tér vissza a metódus, egyébként példányosítja a `PythonRunner` osztályt, majd elindítja a folyamatot a paraméterekből készített paranccsal és a metódus visszatér `true` értékkel.

A Generate gombok esetén végrehajtott akciók (`GenerateWithVaeButtonAction`, `GenerateWithAlgorithmButtonAction`) a következő kontrollerbéli metódusok közül egyet hívnak (CVAE használat esetén az elsőt, algoritmusok esetén a másodikat):

```
public boolean handleGenerateExample(int mazeSize, boolean  
defaultModel):
```

Először a modellt tartalmazó fájlt keresi meg, ha a `defaultModel` paraméter értéke `true`, akkor az alapértelmezett modellek közt is keresi – ilyenkor, ha ezek között talál megfelelő modellt, akkor a felhasználó által generáltak között már nem fog keresni. Különben csak a felhasználó által készített modellek közül próbálja kiválasztani. Ha a

felhasználó home könyvtárában talált megfelelő nevű fájlt vagy fájlokat, akkor azt fogja használni, amely módosítási dátuma a legfrissebb. Ha nem talált a fájlnevével megegyező nevű modellfájlt, akkor a metódus visszatér `false` értékkel.

Ha sikerült megtalálni a modell fájlt, akkor példányosítja a `PythonRunner` osztályt, majd elindítja a folyamatot a paraméterek alapján összeállított paranccsal, valamint a `generate_only` és `load_model` kapcsolókkal, majd a metódus visszatért `true` értékkel.

```
public void handleGenerateExample(int mazeSize,  
MazeGeneratorAlgorithm algorithm)
```

Meghívja a `MazeGeneratorRunner` osztály `generate()` metódusát az útvesztő méretével (`mazeSize`), az algoritmus nevével (`algorithm`), valamint a `generated_algorithm_mazes_SIZE.txt` és `algorithm_statistics.txt` fájlokkal paraméterezve. Amikor a `generate()` visszatér az új labirintussal, a kontroller frissíti a `mazeBoardPanel`-t és az `infoPanel`-t.

### Python backend

Az itt használt CVAE a [27] adaptált változata. A Python backend fájljai az alábbi helyen találhatók:

```
${proj_home}\src\main\resources\edu\elte\thesis\python
```

A `maze_generator_cvae.py` segítségével indíthatjuk el a modellek tanítását, vagy tölthetjük be őket és generálhatunk új labirintusokat, valamint ez a script menti el a modell állapotát a tanítás befejeztével. Az útvesztőket JSON és PNG formátumban menti el a felhasználó home könyvtárának `cvae` mappájába. Az alábbi argumentumokat kell, illetve lehet megadni neki:

#### Kötelező argumentumok:

- **--dimension DIMENSION**  
A `DIMENSION` helyére behelyettesített szám az útvesztők méretét adja meg. Például egy 12\*12-es labirintus esetén a 12-t kell behelyettesíteni.
- **--model\_file MODEL\_FILE**  
Annak a fájlnak az elérési útvonalát és nevét adhatjuk meg, amelybe a modellt menthetjük, vagy ahonnan a modellt betölthetjük.

#### Opcionális argumentumok:

- **--load\_model**  
Ez a kapcsoló jelzi, hogy a `--model_file` paramétereként megadott fájlból szeretnénk betölteni a modellt. Ha a kapcsoló nincs a megadott argumentumok között, akkor alapértelmezésben nem tölti be a modellt.

- **--generate\_only**

Ez a kapcsoló jelzi, hogy a modellt nem tanítjuk, csak a jelenlegi állapotában egy labirintust generálunk vele. Ha a kapcsoló nem szerepel a megadott argumentumok között, akkor a modellt tanítani fogjuk.

- **--training\_data TRAINING\_DATA**

Amennyiben tanítani szeretnénk a modellt, annak a fájlnek az elérési útvonalát és nevét adhatjuk meg ezzel, amelyben tanítópéldák találhatók. A `DIMENSION` értékével megegyező méretű labirintusokat kell tartalmaznia.

- **--epochs EPOCHS**

Amennyiben tanítani szeretnénk a modellt, megadhatjuk az epoch-ok számát. Az `EPOCHS` változó alapértelmezett értéke 20.

A `vae_utils.py` olyan funkciókat tartalmaz, amelyeken keresztül betölthetjük a tanítóhalmazokat, illetve betölthetjük, elmenthetjük vagy taníthatjuk a modelleket.

A `cvae.py` tartalmazza azt az osztályt, amely a CVAE hálózati modell architektúráját implementálja.

A `generate_mazes.py` által tudunk betanított modellek használatával nagyobb mennyiségű labirintust készíteni, illetve ez a script a projekt fő folyamatában nincs használva. A script az alábbi kötelező argumentumokat várja:

- **--dimension DIMENSION**

A `DIMENSION` helyére behelyettesített szám az útvesztők méretét adja meg. Például egy 12\*12-es labirintus esetén a 12-t kell behelyettesíteni.

- **--count COUNT**

A generálni kívánt labirintusok mennyiségét adhatjuk meg vele.

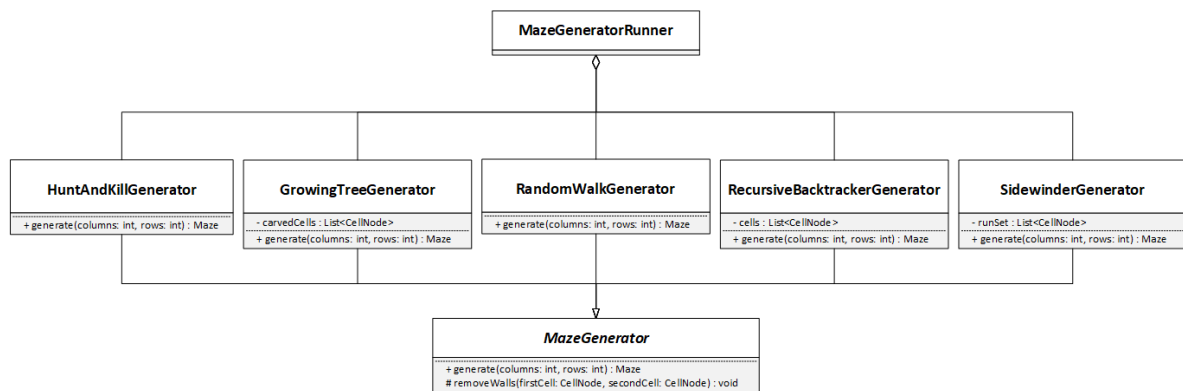
- **--model\_file MODEL\_FILE**

Annak a fájlnek az elérési útvonalát és nevét adhatjuk meg, ahonnan a modellt betölthetjük.

A `create_histogram.py` script – amely szintén nincs a projekt normál folyamatán belül használva – segítségével olyan hisztogramokat készíthetünk, amelyek több azonos méretű útvesztő specifikus tulajdonságait jelenítik meg grafikusan. A script a bemeneti labirintusok leghosszabb sétáit tudja összegezni egy oszlopdiagram segítségével, illetve szintén oszlopdiagram formájában tudja megjeleníteni azt, hogy az egyes útvesztők hány részből állnak (összefüggő gráfok esetén ez a szám 1). Az egyetlen elvárt argumentuma a **--filename FILENAME**, amiben vesszővel elválasztva adhatjuk meg a séták hosszát, illetve a nem összefüggő részgráfok számát. A hisztogramok készítéséhez szükséges fájlokat a Java backend `MazeDataExtractor` osztályával állíthatjuk elő.



## Generátoralgoritmusok



55. Ábra – Generátor algoritmusok kapcsolata a MazeGeneratorRunner osztállyal.

### Hunt-and-Kill

Az algoritmus lépései:

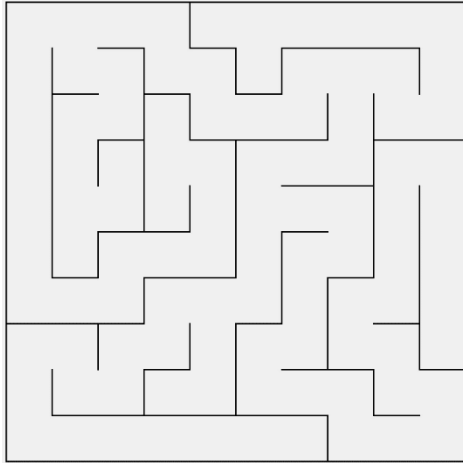
1. Véletlenszerűen válasszunk egy kezdőpontot, jelöljük meg látogatottként. Illetve legyen ez az *aktuális* cellánk.
2. Ismételjük addig a következő lépéseket, amíg a labirintus minden celláját meg nem jelöltük látogatottként:
  - a. Az *aktuális* cella meg nem látogatott szomszédjai közül véletlenszerűen válasszunk egyet.
  - b. Ha van ilyen cella, akkor legyen ez a *következő* cella.
  - c. Különben, ha minden szomszédját már látogattuk, akkor keressünk egy olyan cellát, amelyet még nem látogattunk meg, de legalább egy szomszédját már igen (ez a *hunt* lépés). Legyen ez a *következő* cella.
  - d. Töröljük ki az *aktuális* és a *következő* cella közül a falat, jelöljük meg a *következőt* látogatottként, majd legyen ez az *aktuális* cellánk.

### Growing Tree

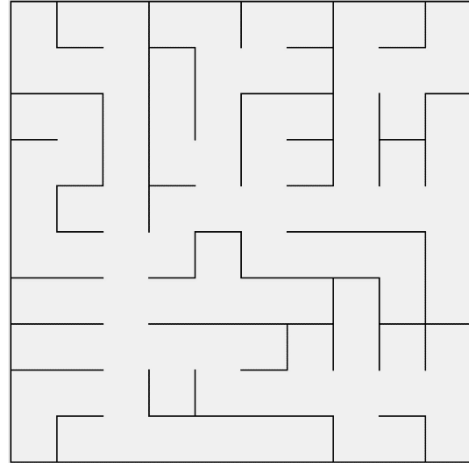
Az algoritmus lépései:

1. Véletlenszerűen válasszunk egy kezdőpontot, jelöljük meg látogatottként, majd helyezzük el egy listában, amiben azokat a cellákat tarjuk, amelyeket részben feldolgoztunk.
2. Ismételjük az alábbi lépéseket, amíg a lista ki nem ürül:
  - a. Válasszunk egy véletlenszerű elemet a listából (de ne töröljük ki belőle).

- b. Ha van olyan szomszédja, amelyet még nem látogattunk meg, akkor válasszunk ezek közül véletlenszerűen egyet, és töröljük ki a falat közülük.
- c. Különben, ha minden szomszédja meg van jelölve látogatottként, akkor töröljük ki a listából.



56. Ábra – Hunt-and-Kill algoritmus által generált 10\*10-es labirintus.



57. Ábra – Growing Tree algoritmus által generált 10\*10-es labirintus.

### Random Walk

Az algoritmus lépései:

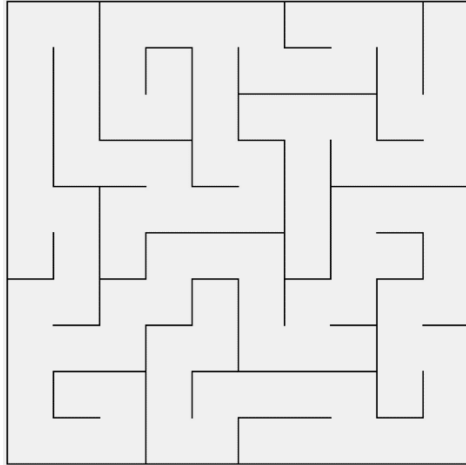
1. Véletlenszerűen válasszunk egy kezdőpontot, jelöljük meg látogatottként.
2. Az alábbi lépéseket ismételjük, amíg vissza nem érünk a kezdőpontba, illetve annak minden szomszédját meg nem látogattuk:
  - a. Az *aktuális* cella szomszédjai közül válasszunk egyet, amelyet még nem látogattunk meg.
  - b. Ha van ilyen szomszédja, akkor töröljük ki ez és az *aktuális* cella közül a falat. Legyen ez az *aktuális* cella, és jelöljük meg látogatottként.
  - c. Ha nincs ilyen szomszédja, akkor lépünk vissza az egyel korábban látogatott cellára, vagyis a szülőjére, és legyen ez az *aktuális* cella.

### Recursive Backtracker

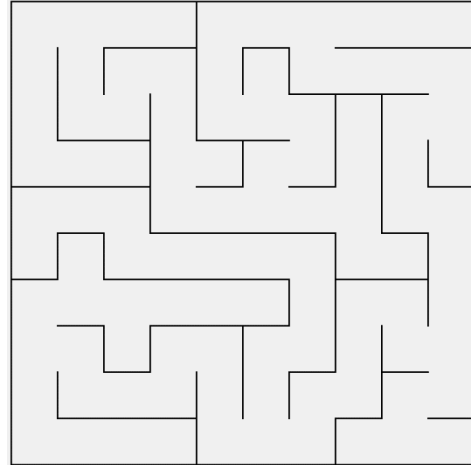
Az algoritmus lépései:

1. Véletlenszerűen válasszunk egy kezdőpontot, legyen ez az *aktuális* cellánk, majd tegyük egy verembe.
2. Ismételjük a következő lépéseket, amíg a verem ki nem ürül:
  - a. Jelöljük meg az *aktuális* cellát látogatottként.
  - b. Az *aktuális* cella szomszédjai között keressünk egyet, amelyet még nem látogattunk.

- c. Ha van ilyen szomszéd, akkor töröljük ki a falat ez és az *aktuális* közül. Legyen ez az *aktuális* cella és tegyük a verem tetejére.
- d. Ha nincs ilyen szomszédja, akkor vegyünk le egy cellát a verem tetejéről, és legyen ez az *aktuális* (ez a rekurzív lépés).



58. Ábra – Random Walk algoritmus által generált 10\*10-es labirintus.



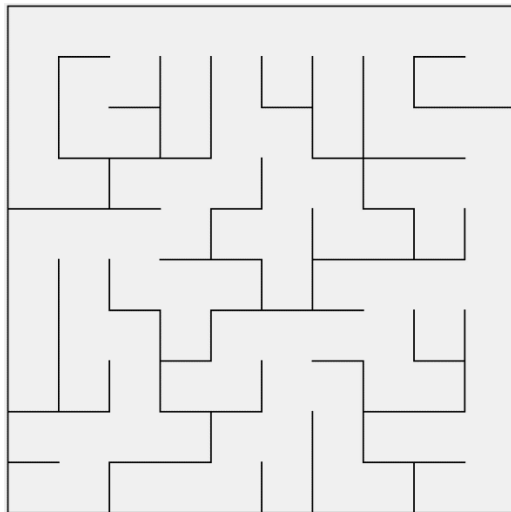
59. Ábra – Recursive Backtracker algoritmus által generált 10\*10-es labirintus.

### Sidewinder

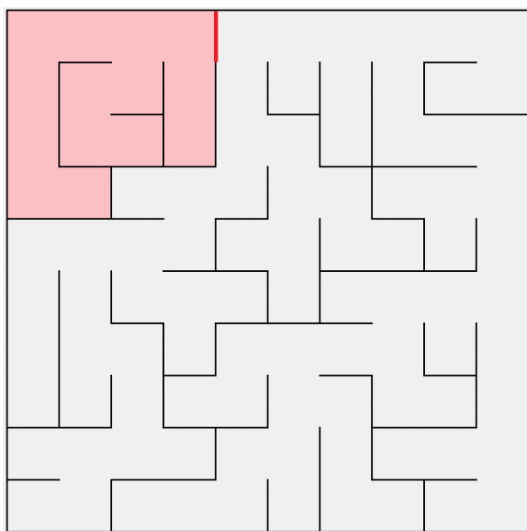
Az algoritmus lépései:

1. A bal felső sarokból indulva, balról jobbra és sorról-sorra haladva ismétljük addig a következő lépéseket, amíg a labirintus minden celláját meg nem jelöltük látogatottként:
  - a. Az *aktuális* cellát jelöljük meg látogatottként, majd adjuk hozzá futási halmazhoz.
  - b. Egy `boolean` változó értékét állítsuk be véletlenszerűen, ez mondja meg, hogy vájjunk-e átjárót jobbra, vagy ne.
  - c. Ha nem az utolsó oszlopban vagyunk, és vagy az első sorban vagyunk, vagy a `boolean` változó értéke `true`, akkor töröljük ki a falat az *aktuális* cella és annak jobboldali szomszédja közül.
  - d. Különben, ha nem az első sorban vagyunk és vagy az utolsó oszlopban vagyunk, vagy a `boolean` változó értéke `false`, akkor a futási halmaz elemeiből válasszunk egyet véletlenszerűen, amelyből felfelé vájunk átjárót. Majd ürítsük ki a futási halmazt.

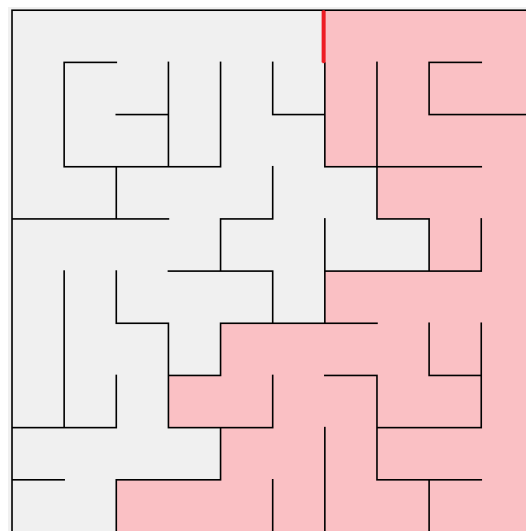
Az itt használt algoritmusok közül egyedül a Sidewinder algoritmus által generált labirintusoknak van láthatóan megkülönböztető jellegzetessége, vagyis hogy a legelső sorában nincs fal a cellák közt. Ezzel biztosítja az algoritmus, hogy a későbbi sorokban hozott döntések ne okozzanak olyan részeket a gráfban, amelyek nem érhetőek el.



60. Ábra – Sidewinder algoritmussal generált 10\*10-es labirintus.



61. Ábra – A piros fal elhelyezése miatt egy kisebb és egy nagyobb részre bomlott a labirintus.



62. Ábra – Az első sorban elhelyezett piros fal kettészakította a labirintust.

## Ellenőrző algoritmus

Azt, hogy az egyes labirintusok összefüggők-e, illetve van-e bennük kör, könnyen ellenőrizhető más gráfalgoritmusok segítségével, amennyiben egy útvesztő rendelkezésünkre áll gráf formában.

### Depth First Search

A DFS algoritmus lényege, hogy a gráf gyökeréből kiindulva meglátogat minden gyerekcsúcsot, amihez számon tartja egy listában vagy tömbben a meglátogatott cellákat, illetve egy verem segítségével tartja számon, hogy melyik cella szomszédjait kell még esetlegesen megvizsgálni.

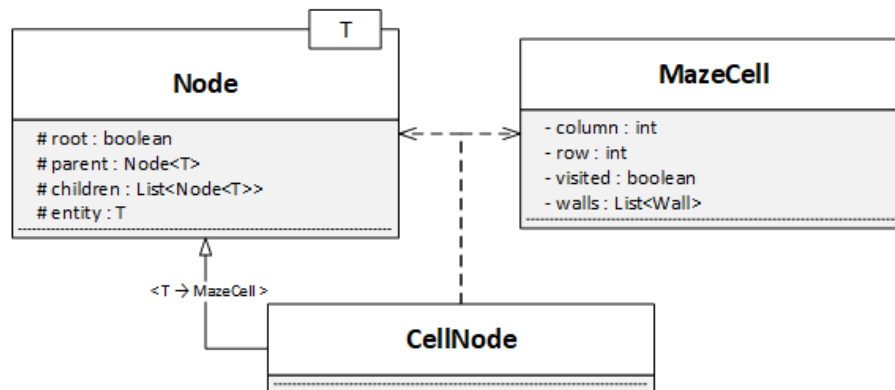
Az algoritmus lépései:

1. Keressük meg a gyökér csúcsot, és tegyük a verem tetejére.
2. Ismételjük az alábbi lépéseket amíg a verem ki nem ürül:
  - a. Vegyünk ki a verem legfelső elemét, legyen ez az *aktuális* cellánk.
  - b. Ha az *aktuális* cella már benne van a látogatott listában, akkor a gráfban **van kör**. (Itt megszakíthatjuk az algoritmust és vissza is térhet ezzel az információval.)
  - c. Különben, ha az *aktuális* cella nincs a látogatott listában, akkor tegyük bele.
  - d. Az *aktuális* elem gyerekeit (, ha van neki) tegyük egyesével a verem tetejére.
3. Ha van a gráfnak olyan csúcsa a verem kiürülése után, amelyik nincs benne a látogatott listában, akkor a gráf **nem összefüggő**.
4. Egyébként, ha a gráf minden csúcsa a látogatott listában van, és a verem kiürült, akkor a vizsgált gráf **összefüggő és körmentes**.

## Generált labirintusok formátuma

### Gráf

Az algoritmusok által generált labirintusokat egy irányított gráfként tárolja a program, amelyet a `Node<MazeCell>` osztály reprezentál. Ebben az esetben minden gráfnak egy gyökércsúcsa van, illetve minden csúcs ismeri a szülőcsúcsát, és a saját gyerekeit. A CVAE által generált útvesztők esetében nincs garantálva, hogy a gráfok összefüggők vagy körmentesek lesznek, így amikor azokat konvertáljuk gráffá, nem töltjük ki a szülő értékét. A `MazeCell` osztály segítségével menti el egy cella koordinátáját, a falainak állapotát és azt, hogy a cella meg volt-e látogatva (elsősorban generálás közben használatos ez a változó).



63. Ábra – A cellák ábrázolásához használt osztályok kapcsolata.

## Bináris

A `BinarizedMaze` osztály segítségével lehet bináris formátumba konvertálni a gráf formátumú labirintusokat (`binarizeMaze(Maze maze)` metódus). Ez azt jelenti, hogy a falak 1-ekkel, a szabad járatok 0-ákkal vannak jelölve. A konvertálás során az egyes celláknak csak a jobb és alsó falának láthatóságát veszi figyelembe, mivel ezek megegyeznek a szomszédos cellák felső és bal oldali falaival. Illetve ugyanezen osztály segítségével konvertálhatunk egy bináris labirintust gráf formátumba, ekkor a cellák szülőit nem mentjük el, csak a cella gyerekei felé mutató kapcsolatokat (`createGraphFromBinarizedMaze()` metódus).

### Gráfból bináris labirintus

Egy cella egy  $2 \times 2$ -es négyzettel reprezentálható, ahol az első sor „0x”, a második „x1”, ahol az „x” mindkét esetben lehet 0 vagy 1. Miután az egyes cellákat konvertálta, a labirintus felső és bal oldalára egy sor és egy oszlop 1-est helyez, mivel ezek a szélső falak nem voltak figyelembe véve, amíg a cellákat konvertáltuk. Így az elkészült bináris labirintus mérete  $\text{BIN\_SIZE} := \text{SIZE} * 2 + 1$  lesz, ahol a `SIZE` az útvesztő eredeti mérete.

Ezután 1-ekből álló sorokat és oszlopokat csatol a bináris útvesztőhöz, amennyiben a dimenzió nem osztható 4-el, mivel a CVAE két konvolúciós rétege a labirintus méretét a negyedére csökkenti – amelynek egész számnak kell lennie. Az alábbi módon számolja ki a program, hogy amennyiben szükséges új oszlopot vagy sort hozzáadni a labirintushoz, hova és hány darabot helyezzen el:

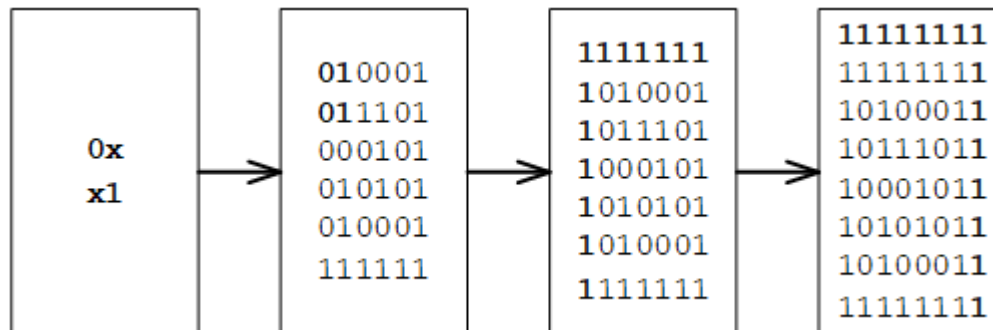
Ha a  $\text{BIN\_SIZE} \% 4$  maradékos osztás eredménye 3, akkor egy-egy új sort és oszlopot kell hozzátenni, amelyeket az útvesztő felső és jobb oldalán helyez el.

Különben ha az eredmény 2, akkor két új sorra és két új oszlopra van szükség, amelyeket egyesével az útvesztő négy oldalára rak.

Különben ha az eredmény 1, akkor 3 sor és 3 oszlop szükséges. Két sort helyez fentre, illetve egy sort alulra, majd két oszlopot jobbra és egyet balra.

Egyébként, ha maradéktalanul osztható 4-el, akkor nem kell semmit változtatni a bináris labirintuson.

Az alábbi ábrán egy 3\*3-as labirintus binárisra alakításának folyamata látható az egyes cellák általános kinézetétől a ráadás sorok hozzáadásáig.



64. Ábra – 1.: Egy cella általánosságban, ahol az  $x$  lehet 1 vagy 0.  
 2.: A labirintus minden cellája konvertálva lett, egyelőre csak a jobb és alsó falak láthatóságát ábrázolja.  
 3.: A felső és bal oldali falak is hozzá lettek csatolva.  
 4. A bináris labirintus mérete (7\*7) nem osztható 4-el, így egy sort és oszlopot helyezünk fentre és jobbra.

### Bináris labirintusból gráf

A bináris formátumú labirintusokat vissza lehet konvertálni gráf alakba, viszont ekkor az egyes csúcsok nem mentik el a szülőcsúcsukat, mivel a gráf körmentessége nincs biztosítva és előfordulhat, hogy az algoritmus több szülőt szeretne egy csúcshoz rendelni.

Az algoritmus először eltünteti azokat a sorokat és oszlopokat, amelyek a 4-el oszthatóság miatt kerültek rá a bináris labirintusra, ez a binárisra alakítás fordítottja. Ehhez megvizsgálja, hogy van-e több egyforma sor fent és/vagy lent:

Ha a bináris labirintus első három sora megegyezik, akkor az eredeti bináris útvesztőhöz képest három plusz sor és oszlop lett hozzácsatolva, vagyis kettő sort kell levenni fentről, egyet lentől, illetve kettő oszlopot jobbról és egyet balról.

Ha az első kettő és az utolsó kettő sor egyezik meg, akkor minden oldalról egy sort és oszlopot kell letörölni.

Ha csak az első két sor egyforma, akkor fentről kell egy sort és jobbról egy oszlopot levenni a bináris labirintusról.

Ha sikerült minden oldalról levenni a „felesleges” sorokat és oszlopokat, akkor ki lehet számolni a labirintus eredeti méretét, ami a  $SIZE = (BIN\_SIZE - 1) / 2$  értékkel egyezik meg. Ezután megkezdjük a gráf felépítését a bal felső sarokban lévő cellából indítva.

1. A kezdő cellát behelyezzük a befejezetlen celláknak fenntartott *listába*.
2. Ismételjük az alábbi lépéseket, amíg a *lista* ki nem ürül:
  - a. Vegyük ki a legkorábban behelyezett elemet a *listából*, majd jelöljük meg látogatottként. Legyen ez az *aktuális* cella.

- b. Ha *aktuális* cella jobboldali fala hiányzik, és a gyerekei között nem szerepel az *aktuális* csúcs, akkor adjuk az *aktuális* cella gyerekei közé a jobb szomszédját, illetve tegyük a jobb szomszédot a *lista*-ba.
  - c. Ismételjük meg a b. pontot az *aktuális* cella alsó, bal oldali és felső szomszédjaival.
3. Ha a *lista* üres, de van még nem látogatott cella, akkor rendezzük őket a koordinátaik szerint növekvő sorba (oszlop szerint, majd sor szerint), majd az első elemmel ismételjük meg az algoritmust az 1. lépéstől.
  4. Ha nincs több olyan cella, amelyet nem látogattunk meg, akkor a gráfot felépítettük.

## Tesztelés

A projekt unit tesztekkel ellenőrzi a program osztályainak helyes működését, összesen 55 tesztet van.

Az esetek közt szerepelnek olyanok, amelyek vizsgálják az labirintus generátor algoritmusok helyes működését rossz paraméterezés esetén (például 0 vagy negatív méretű oszlop vagy sor megadása), illetve az előállított labirintusok körmentességét és összefüggőségét ellenőrzik a DFS algoritmus felhasználásával.

A DFS algoritmus működését tesztelik szintén helytelen bemenet esetén (bementi `null` labirintus), valamint vizsgálják a körmentes és körrel rendelkező illetve a (nem) összefüggő gráfokra adott válaszábanak helyességét.

Valamint a tesztesetek közt szerepelnek olyanok, amelyek az útvesztők és az egyes cellák műveleteinek helyes működésére irányulnak. Ilyen például egy cella megtalálása koordináták alapján, vagy egy adott oldali fal eltüntetése, illetve a cellák (nem látogatott) szomszédjainak megkeresése.

▼ TestGrowingTreeGenerator	803 ms	▼ TestSidewinderGenerator	10 ms	▼ TestMazeCell	5 ms
✓ testGenerate	802 ms	✓ testGenerate	10 ms	✓ testRemoveWall_nullPosition	1 ms
✓ testGenerate_negativeColumn	0 ms	✓ testGenerate_negativeColumn	0 ms	✓ testMarkAsVisited	0 ms
✓ testGenerate_negativeRow	0 ms	✓ testGenerate_negativeRow	0 ms	✓ testRemoveWall	4 ms
✓ testGenerate_zeroColumn	1 ms	✓ testGenerate_zeroColumn	0 ms	▼ TestWall	0 ms
✓ testGenerate_zeroRow	0 ms	✓ testGenerate_zeroRow	0 ms	✓ testSetInvisible_alreadyInvisible	0 ms
▼ TestHuntAndKillGenerator	19 ms	▼ TestBinarizedMaze	18 ms	✓ testSetInvisible	0 ms
✓ testGenerate	18 ms	✓ testCreateGraphFromBinarizedMaze_disconnec	14 ms	▼ TestCellNode	0 ms
✓ testGenerate_negativeColumn	0 ms	✓ testCreateGraphFromBinarizedMaze_acyclicMaz	2 ms	✓ testIsLowerNeighbourOfItself	0 ms
✓ testGenerate_negativeRow	0 ms	✓ testCreateGraphFromBinarizedMaze_cyclicMaze	2 ms	✓ testIsLeftNeighbourOfItself	0 ms
✓ testGenerate_zeroColumn	1 ms	▼ TestDepthFirstSearchRunner	5 ms	✓ testMarkAsVisited	0 ms
✓ testGenerate_zeroRow	0 ms	✓ testRun_cyclic	0 ms	✓ testIsNeighbourOf	0 ms
▼ TestRandomWalkGenerator	9 ms	✓ testRun_acyclic	1 ms	✓ testIsLeftNeighbourOf	0 ms
✓ testGenerate	9 ms	✓ testRun_nullMaze	0 ms	✓ testIsRightNeighbourOf	0 ms
✓ testGenerate_negativeColumn	0 ms	✓ testRun_acyclic_disconnected_singleRoot	1 ms	✓ testIsUpperNeighbourOfItself	0 ms
✓ testGenerate_negativeRow	0 ms	✓ testRun_acyclic_disconnected_multipleRoots	3 ms	✓ testIsRightNeighbourOfItself	0 ms
✓ testGenerate_zeroColumn	0 ms	▼ TestMaze	1 ms	✓ testIsUpperNeighbourOf	0 ms
✓ testGenerate_zeroRow	0 ms	✓ testGetCellByCoordinates_biggerColumnPositio	0 ms	✓ testIsLowerNeighbourOf	0 ms
▼ TestRecursiveBacktrackerGenerator	4 ms	✓ testGetCellByCoordinates_biggerRowPositionTh	0 ms		
✓ testGenerate	3 ms	✓ testGetCellByCoordinates	1 ms		
✓ testGenerate_negativeColumn	1 ms	✓ testSelectStartPoint_random	0 ms		
✓ testGenerate_negativeRow	0 ms	✓ testSelectStartPoint_notRandom	0 ms		
✓ testGenerate_zeroColumn	0 ms	✓ testGetCellByCoordinates_negativeColumn	0 ms		
✓ testGenerate_zeroRow	0 ms	✓ testGetCellByCoordinates_negativeRow	0 ms		

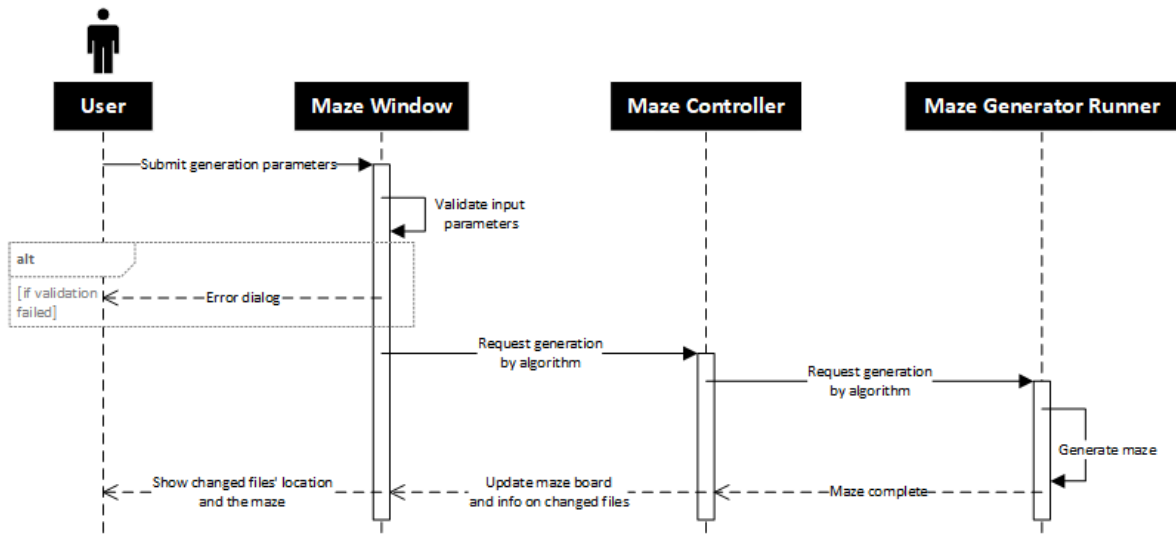
Tests run: 55, Failures: 0, Errors: 0, Skipped: 0

65. Ábra – A projektben szereplő tesztesetek futtatási eredménye.



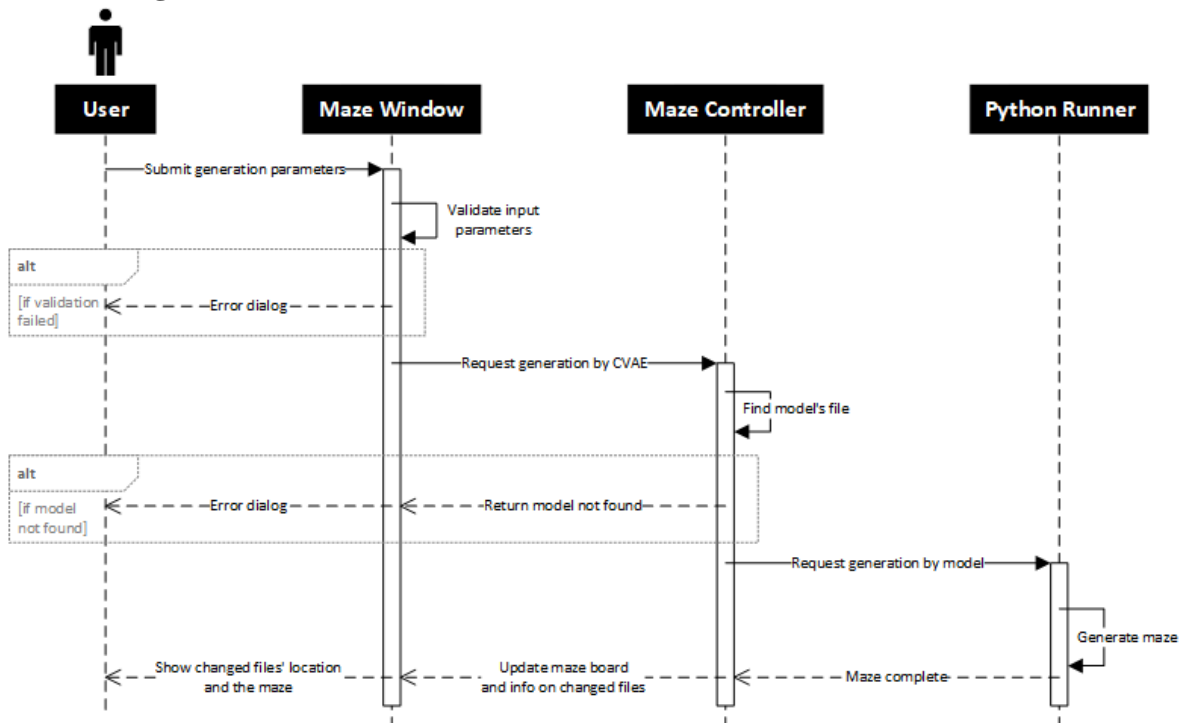
## Az alkalmazás szekvenciadiagramjai

### Labirintusgenerálás algoritmussal



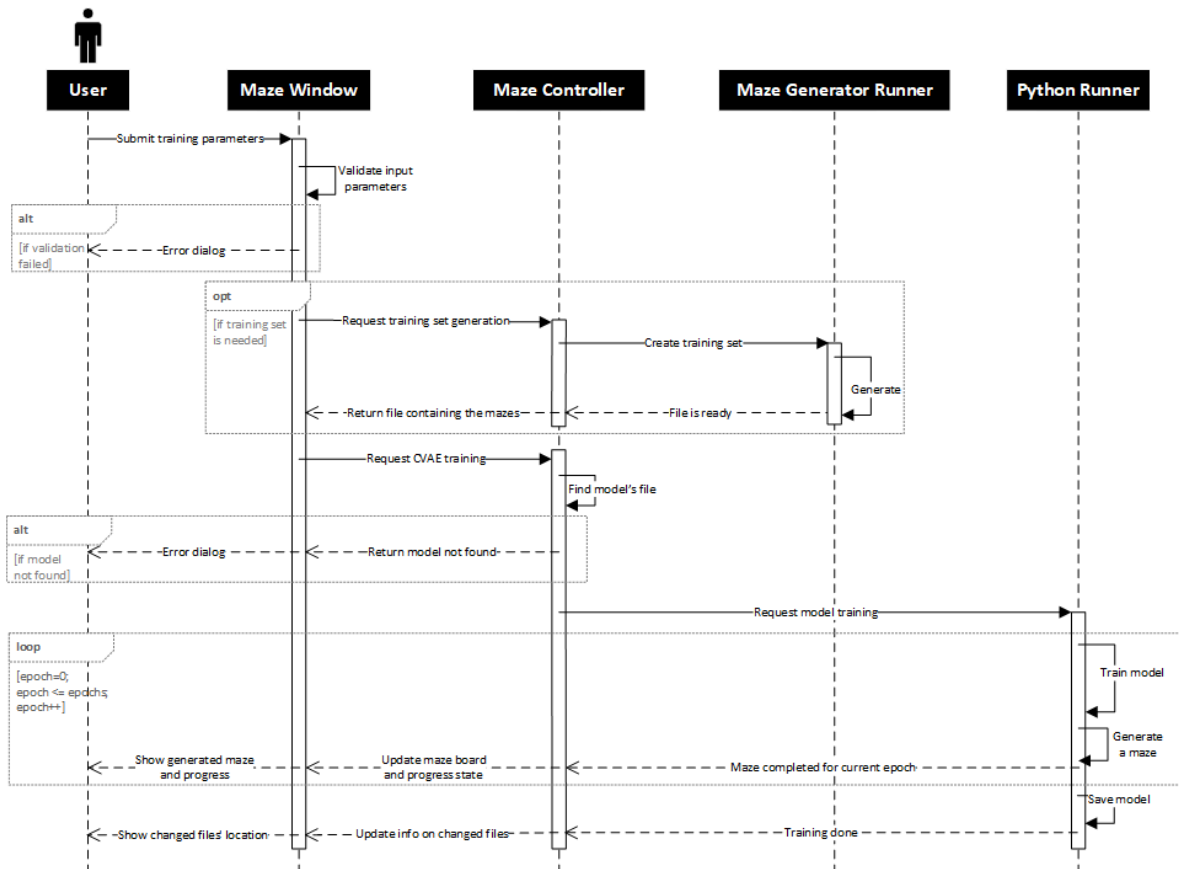
66. Ábra – Algoritmussal történő labirintusgenerálás folyamata.

### Labirintusgenerálás CVAE modellel



67. Ábra – Útvesszők generálásának folyamata CVAE modellel.

## CVAE modell tanítása, tanítóhalmazok készítése



68. Ábra – Modellek tanítási folyamata.

## Fejlesztési lehetőségek

Jelenleg 5 féle generátor algoritmus implementációját tartalmazza a projekt, de érdemes lehet kipróbálni új algoritmusokat is, amelyek nem feltétlenül csak tökéletes labirintusokat képesek generálni, vagy olyanokat, amelyek például vertikális vagy horizontális hajlammal rendelkező útvesztőket tudnak előállítani. Akár az útvesztők geometriájára irányuló változtatások is érdekes irányokba terelhetik a hálózatok által generált labirintusokat.

Akár más neurális hálózati modellek felhasználása, vagy más módszerek alkalmazása, amelyek a hálózati paramétereket optimalizálják, is mutathatnak például jobb eredményt a hálózat tanulási idejében.

Neurális hálózatok használata közben (tanítás, generálás), az alkalmazás felületét szabadon használhatjuk, vagyis új háttérbeli szálakat indíthatunk, ha a `Submit` vagy `Generate` gombok egyikét megnyomjuk. Ezek potenciálisan megterhelhetik a számítógépet a megnövekedett memória és CPU használattal, amelyek nem kívánt eseményeket idézhetnek elő (például lelassulhat az operációsrendszer). Ez a probléma megoldható azzal, hogy amíg egy neurális hálózatot tanítunk, vagy használunk, addig letiltjuk a `Submit` és `Generate` gombokat, ezzel megakadályozva az akciójuk életbe lépését és újabb szálak létrehozásának kezdeményezését. Majd amikor végzett a háttérben futó folyamat, feloldjuk róluk a tiltást.

## Összefoglalás

A neurális hálózatok bizonyos tanítási idő után generált labirintusai közelítik a tanulóhalmazokban szereplő útvesztők alapelveit, illetve a tanulási folyamat vége felé pedig már érdekes labirintusokat is kapunk, amelyekben zsákutcák, kanyargós utak és körök találhatóak – habár a nagyobb méretű útvesztők esetén ez még nem annyira hangsúlyos. Az alkalmazáshoz mellékelt CVAE modellek mindegyike 200 epoch-ot tanult és a tanítóhalmazuk growing tree és sidewinder algoritmus által generált útvesztőket tartalmazott.

A neurális hálók időigényét nehéz pontosan meghatározni, mivel erősen függ attól, hogy a rendszer milyen hardverekkel rendelkezik. A felhasználói és fejlesztői dokumentációban specifikált hardverekkel például egy 50\*50-es labirintusok generálására képes CVAE epoch-onként 163-199 másodpercet igényel, míg egy GTX 1070 videokártyával rendelkező számítógépen egy epoch 23-25 másodpercig tart.

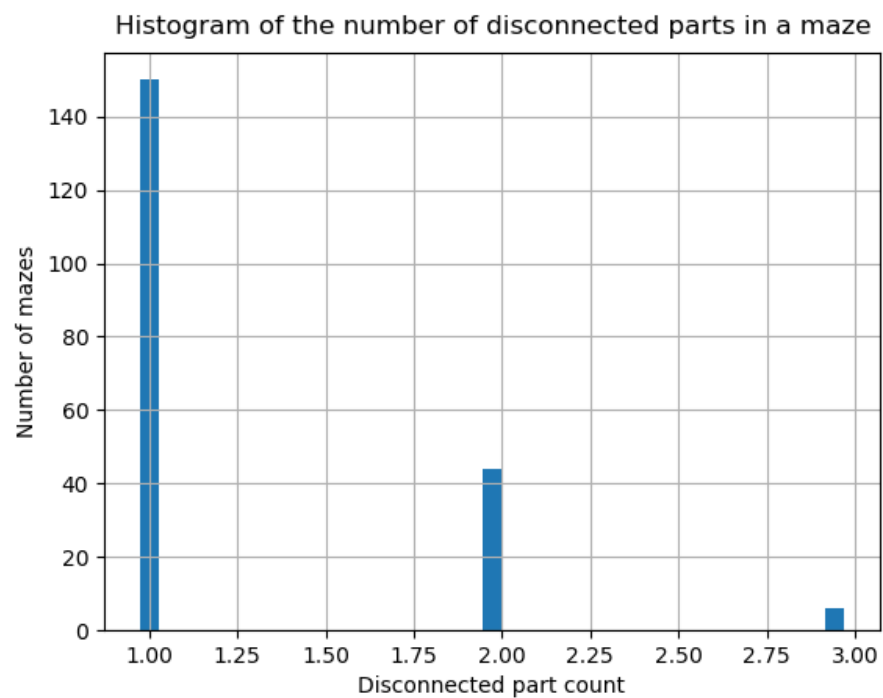
A szakdolgozatban implementált generátoralgoritmusok időigényigény szempontjából körülbelül ugyanolyan jól teljesítenek, kivéve a growing tree algoritmus, amelyik több mint kétszer annyi időt vesz igénybe, mint a többi. Például 5.000 darab 20\*20-as labirintus generálásához a következőképpen teljesítettek:

0h 3m 43s	growingTree	20x20	5000 pcs.
0h 1m 31s	huntAndKill	20x20	5000 pcs.
0h 1m 12s	randomWalk	20x20	5000 pcs.
0h 1m 8s	sidewinder	20x20	5000 pcs.
0h 1m 3s	recursiveBacktracker	20x20	5000 pcs.

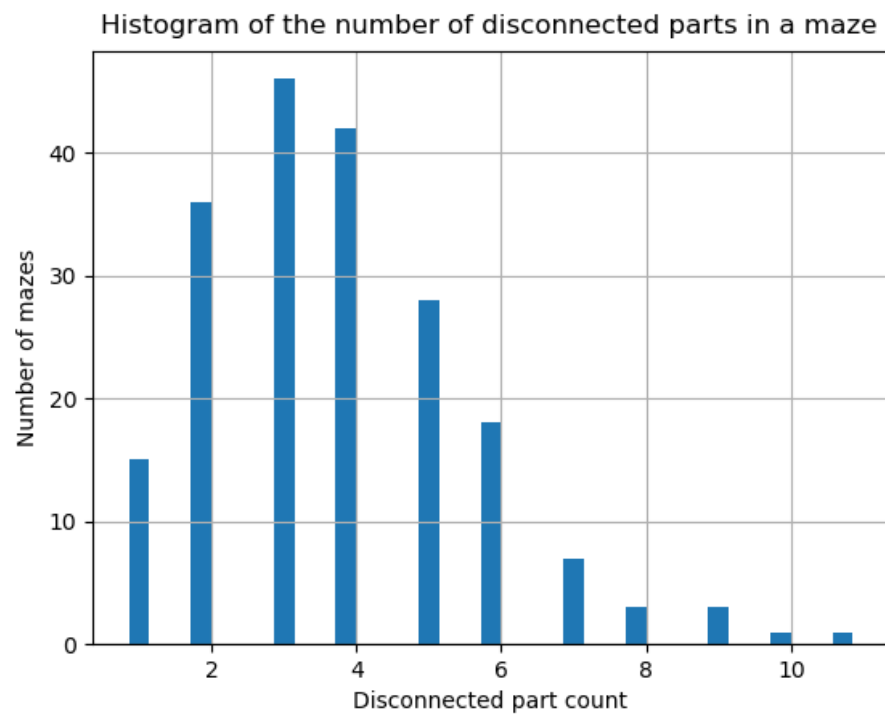
### Betanított CVAE-k által generált labirintusok

A hálózatok által generált kisméretű (5\*5) útvesztők nagyrészt összefüggő gráfokat alkotnak, de a legtöbbjük köröket is tartalmazott. 10\*10 és 20\*20 méretű labirintusok esetében a 200 epoch kicsit kevésnek bizonyult, mivel több olyan komponens is keletkezett bennük, amelyek elérhetetlenek a gráf többi részéből, viszont a legtöbbször volt egy centrális része, amelyik egy nagyobb sétát írt le. Az 50\*50-es gráfok esetén pedig lényegesen több epoch-ra lett volna szükség, mivel a labirintusok sok kisebb különálló komponensből álltak, és csak nagyon kevés tartalmazott egy hosszú sétát.

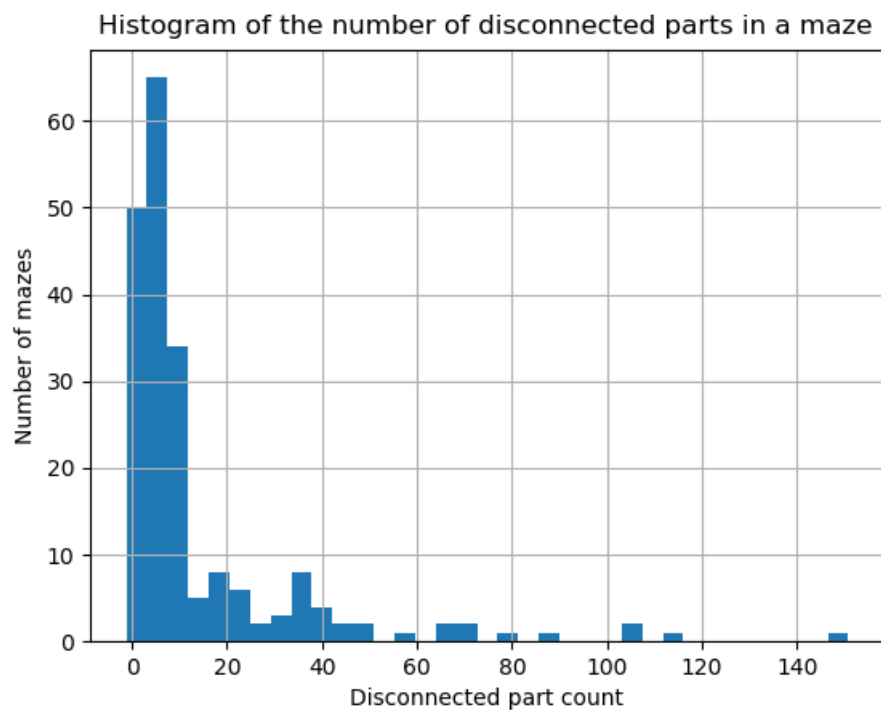
Az alábbi ábrák azt mutatják, hogy a modellek által készített 200 darab labirintus közül hány volt összefüggő – vagyis 1 komponensből állt, illetve ha nem összefüggő a gráf, akkor hány komponensből áll az egész labirintus. A hisztogramok vízszintes tengelyén a komponensek száma látható, a függőlegesen pedig az, hogy a 200-ból hány labirintus áll annyi részből.



69. Ábra – Részgráfok száma 5\*5-ös labirintusok esetén.

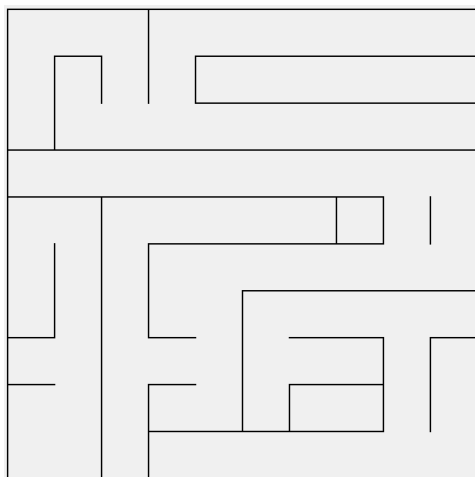


70. Ábra – Részgráfok száma 10\*10-es labirintusok esetén.

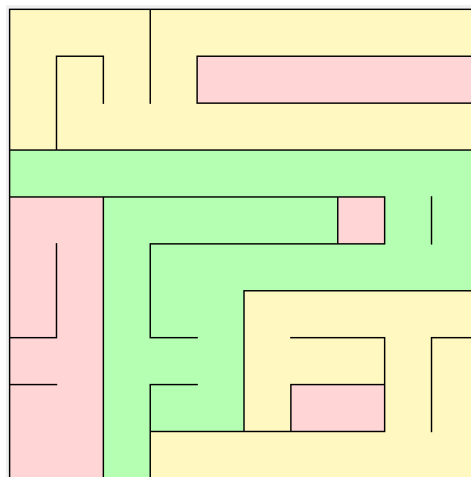


71. Ábra – Részgráfok száma 20\*20-as labirintusok esetén.

A komponensek az alábbi módon különíthetők el a labirintusokban:



72. Ábra – Eredeti 10\*10-es labirintus.

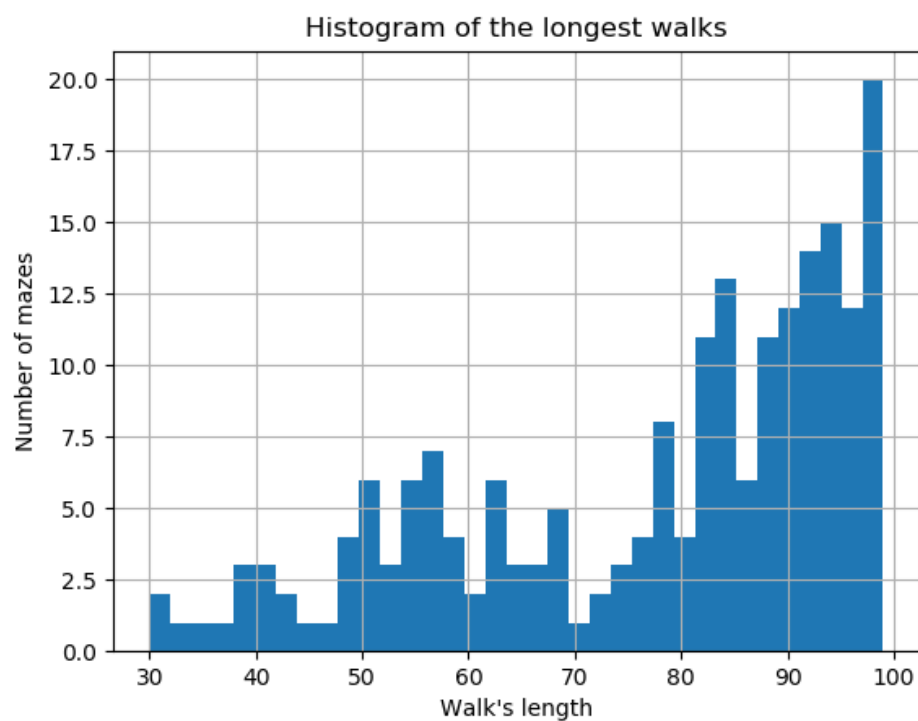


73. Ábra – Az egyes komponensek kiemelve.

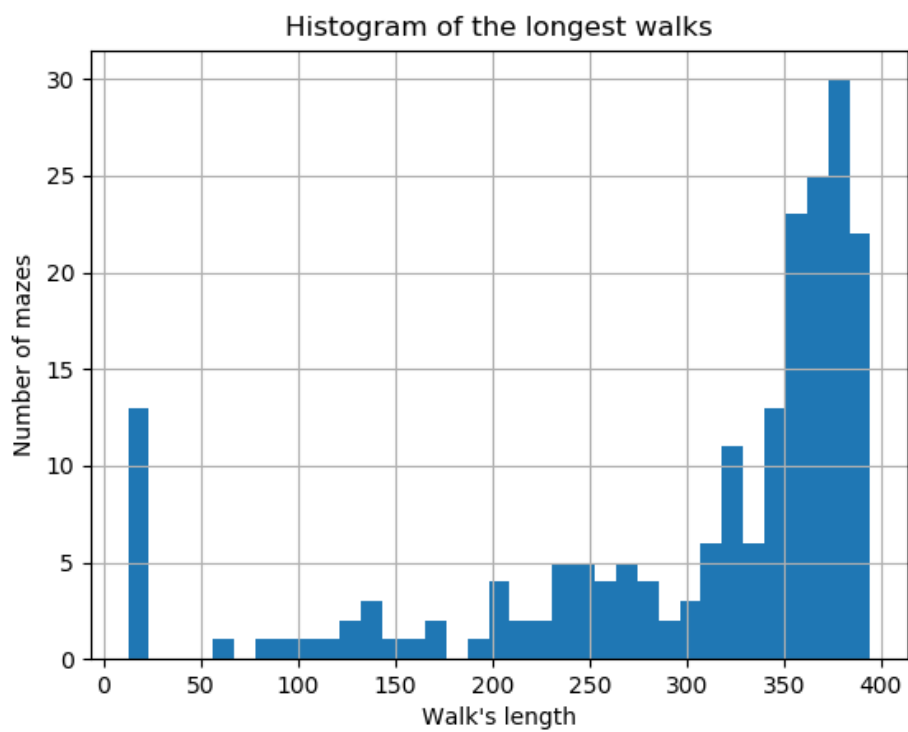
Amit a CVAE által generált labirintusokban még érdemes megvizsgálni, az a séták hossza, vagyis a legnagyobb komponens mérete. A komponensek darabszáma megmondja, hogy hány olyan részből áll egy útvesztő, amelyekből nem tudunk átmenni egy másik részébe, viszont az nem derül ki belőle, hogy több körülbelül egyforma részre szakadt a labirintus, vagy van egy központi összefüggő komponense egy-két cellányi elzárt részekkel. Az alábbi hisztogramokon az egyes labirintusok leghosszabb sétáit láthatjuk. A séta hossza a vízszintes tengelyen látható, illetve a függőlegesen pedig az, hogy a 200-ból hány darab útvesztőnek volt akkora a leghosszabb sétája. Ezeken a hisztogramokon ugyanazon labirintusokra vonatkozó adatok láthatók, mint a fentebb említett részgráfok esetében.



74. Ábra – 5\*5-ös labirintusok esetén a leghosszabb séták mérete.



75. Ábra – Leghosszabb séták mérete 10\*10-es labirintusok esetén.



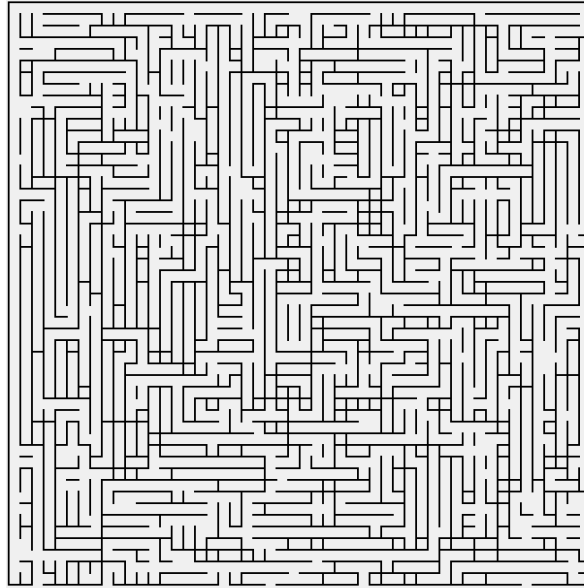
76. Ábra – Leghosszabb séták mérete 20\*20-as labirintusok esetén.



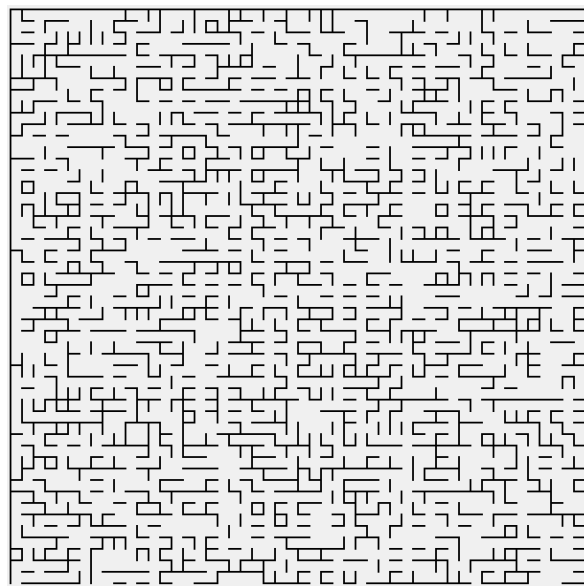
### 50\*50 méretű labirintusok

Az 50\*50-es labirintusok esetén jól látható, hogy a 200 epoch kevés volt, mivel az útvesztők sok részgráfból álltak (79. ábra), amelyek mind nagyon rövid sétákat írtak le, a 80. ábrán, amely a leghosszabb séták méretét ábrázolja, a legelső – és egyben legtöbb labirintusra vonatkozó – oszlop az 51 hosszú sétákat jelöli.

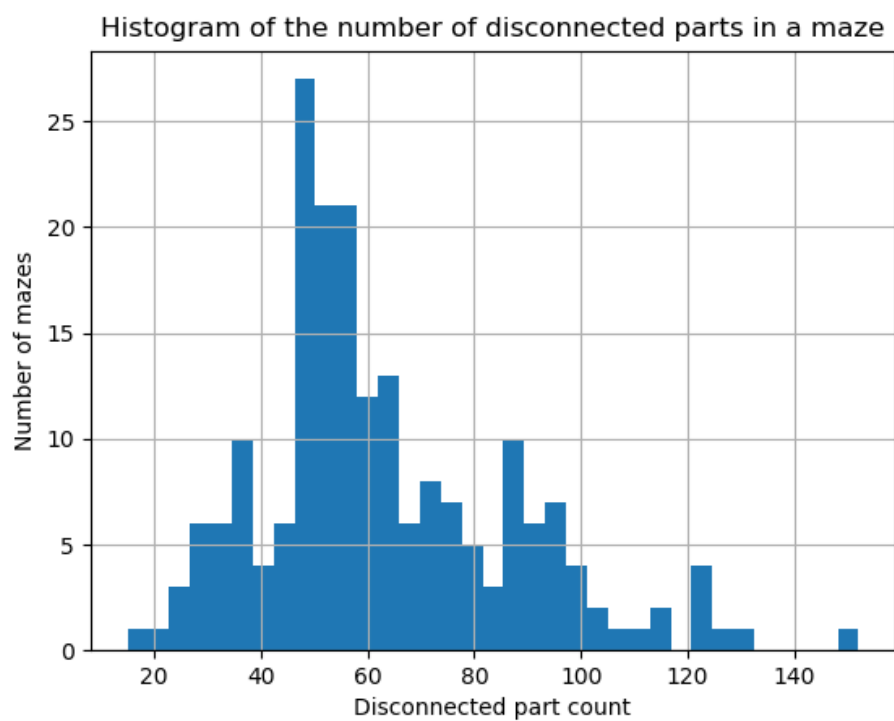
Azok az 50\*50 méretű útvesztők, amelyeket egy 1200 epoch-ot tanult modell generált, még mindig viszonylag sok komponensből állnak (81. ábra), de a legtöbb már rendelkezik egy központi résszel, amely hosszabb sétát ír le, mint a többi (82. ábra).



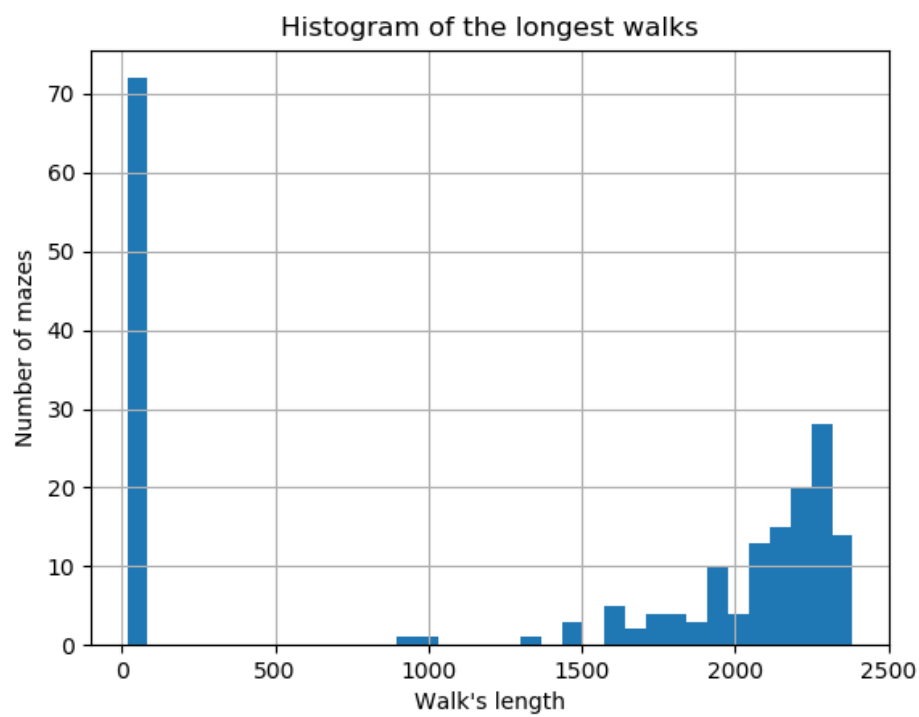
77. Ábra – Példa CVAE labirintus 200 epoch tanulás után.



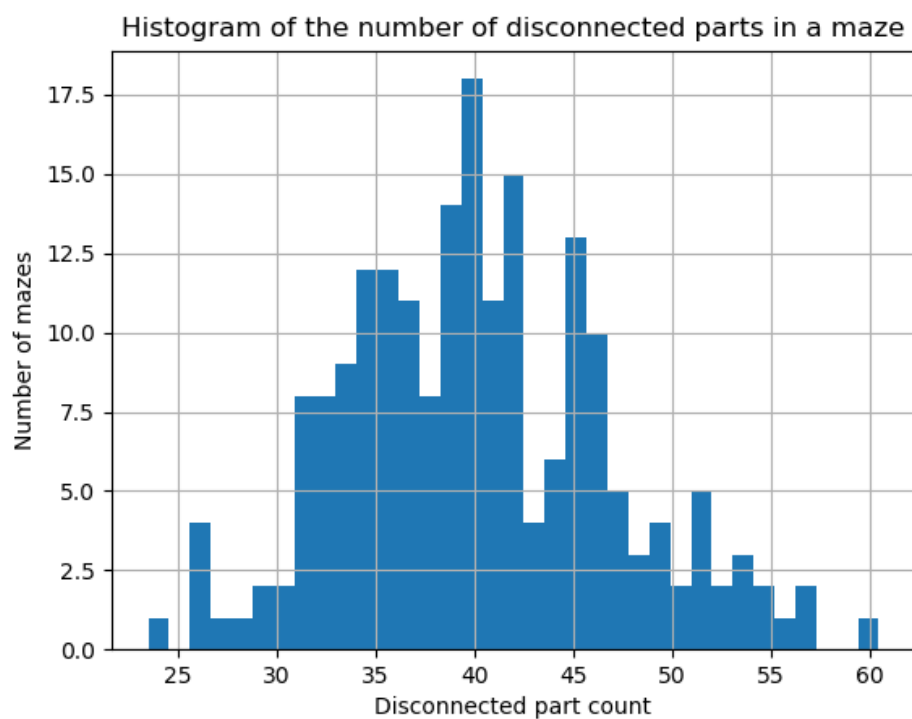
78. Ábra – Példa CVAE labirintus 1200 epoch tanulás után.



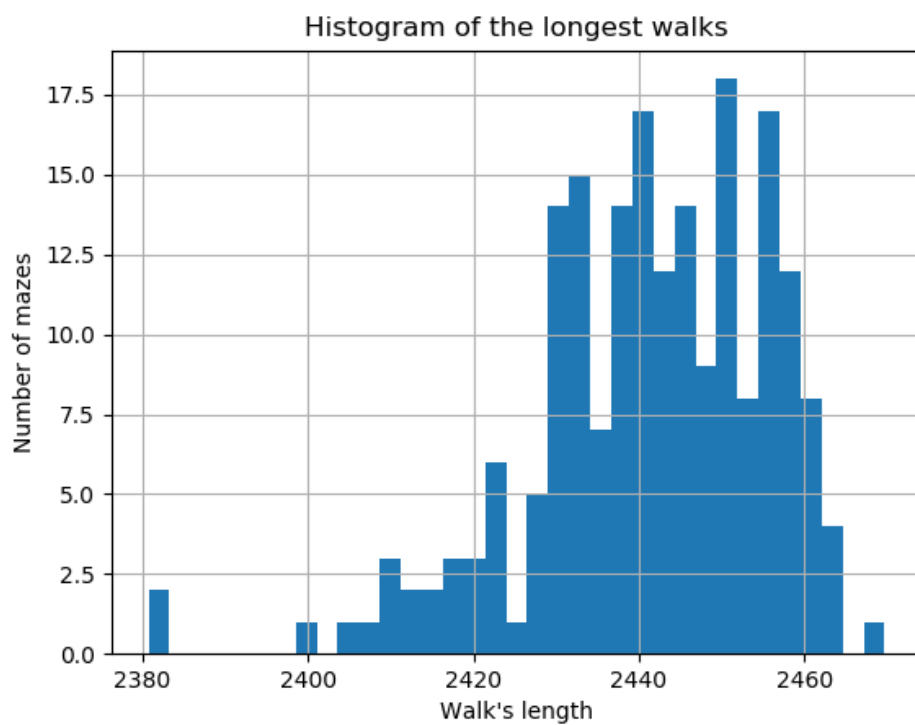
79. Ábra – Részgráfok száma 50\*50-es labirintusok esetén (200 epoch tanulás után).



80. Ábra – Leghosszabb séták mérete 50\*50-es labirintusok esetén (200 epoch tanulás után).



81. Ábra – Komponensek száma 50\*50-es labirintusok esetén (1200 epoch tanulás után).



82. Ábra – Leghosszabb séták mérete 50\*50-es labirintusok esetén (1200 epoch tanulás után).

## Irodalomjegyzék

- [1] Crystalinks, „Labyrinths,” [Online]. Available: <https://www.crystalinks.com/labyrinths.html>. [Hozzáférés dátuma: 16. április 2019.].
- [2] J. Buck, *Mazes for Programmers*, Pragmatic Bookshelf, 2015.
- [3] „Barnes Maze,” MazeEngineers, [Online]. Available: <https://mazeengineers.com/portfolio/barnes-maze/#documentation>. [Hozzáférés dátuma: 15. április 2019.].
- [4] „Elevated Plus Maze,” MazeEngineers, [Online]. Available: <https://mazeengineers.com/portfolio/elevated-plus-maze/#documentation>. [Hozzáférés dátuma: 15. április 2019.].
- [5] S. He, „Maze Basics: T Maze,” MazeEngineers, [Online]. Available: <https://mazeengineers.com/maze-basics-t-maze-test/>. [Hozzáférés dátuma: 15. április 2019.].
- [6] J. Xu és C. S. Kaplan, „Image-Guided Maze Construction,” *ACM Transactions on Graphics*, 26. kötet, pp. 29-38, 2007.
- [7] D. P. Walter, „Think Labyrinth: Maze Algorithms,” [Online]. Available: <http://www.astrolog.org/labyrnth/algrithm.htm>. [Hozzáférés dátuma: 16. április 2019.].
- [8] M. Altrichter, G. Horváth, B. Pataki, G. Strausz, G. Takács és J. Valyon, *Neurális hálózatok*, Budapest: Hungarian Edition Panem Könyvkiadó Kft., 2006.
- [9] G. Zaccane, M. R. Karim és A. Menshawy, *Deep Learning with TensorFlow*, Packt Publishing Ltd., 2017.
- [10] J. Schmidhuber, „Deep learning in neural networks: An overview,” *Neural Networks*, 2014.
- [11] S. Mahapatra, „Why Deep Learning over Traditional Machine Learning?,” Towards Data Science Inc., 2018.. [Online]. Available: <https://towardsdatascience.com/why-deep-learning-is-needed-over-traditional-machine-learning-1b6a99177063>. [Hozzáférés dátuma: 27. április 2019.].

- [12] A. Dertat, „Applied Deep Learning - Part 4: Convolutional Neural Networks,” Towards Data Science Inc., 2017. [Online]. Available: <https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2>. [Hozzáférés dátuma: 22. április 2019.].
- [13] „Machine Learning Crash Course,” Google, [Online]. Available: <https://developers.google.com/machine-learning/crash-course/>. [Hozzáférés dátuma: 22. április 2019.].
- [14] „Reducing Loss: Learning Rate,” Google, [Online]. Available: <https://developers.google.com/machine-learning/crash-course/reducing-loss/learning-rate>. [Hozzáférés dátuma: 26. április 2019.].
- [15] N. Donges, „Gradient Descent in a Nutshell,” Towards Data Science Inc., 2018. [Online]. Available: <https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0>. [Hozzáférés dátuma: 22. április 2019.].
- [16] D. E. Rumelhart, G. E. Ginton és R. J. Williams, „Learning Representations by Back-Propagating Errors,” in *Cognitive Modeling*, MIT Press, 1988, p. 213.
- [17] T. Schaul és Y. LeCun, „Adaptive learning rates and parallelization for stochastic, sparse, non-smooth gradients,” in *ICLR*, 2013..
- [18] D. P. Kingma és J. Ba, „Adam: A method for stochastic optimization,” in *ICLR*, 2014.
- [19] J. Brownlee, „Gentle Introduction to the Adam Optimization Algorithm for Deep Learning,” Machine Learning Mastery, [Online]. Available: <https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/>. [Hozzáférés dátuma: 21. április 2019.].
- [20] M. Mishra, „Convolutional Neural Networks, Explained,” Oracle Corporation, [Online]. Available: <https://www.datascience.com/blog/convolutional-neural-network>. [Hozzáférés dátuma: 22. április 2019.].
- [21] K. Sohn, H. Lee és X. Yan, „Learning structured output representation using deep conditional generative models,” *Advances in neural information processing systems*, 28. kötet, pp. 3483-3491, 2015.
- [22] I. Shafkat, „Intuitively Understanding Variational Autoencoders,” Towards Data Science Inc., 2018. [Online]. Available: <https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf>. [Hozzáférés dátuma: 23. április 2019.].

- [23] J. Altosaar, „Tutorial - What is a variational autoencoder?,” [Online]. Available: <https://jaan.io/what-is-variational-autoencoder-vae-tutorial/>. [Hozzáférés dátuma: 23. április 2019.].
- [24] „Convolutional Variational Autoencoder,” TensorFlow, [Online]. Available: <https://www.tensorflow.org/alpha/tutorials/generative/cvae>. [Hozzáférés dátuma: 26. április 2019.].
- [25] F. Chollet és others, „Keras,” 2015. [Online]. Available: <https://keras.io>. [Hozzáférés dátuma: 23. április 2019.].
- [26] „TensorFlow,” Google, [Online]. Available: <https://www.tensorflow.org/>. [Hozzáférés dátuma: 23. április 2019.].
- [27] „tensorflow/cvae.ipynb at r1.13,” TensorFlow, [Online]. Available: [https://github.com/tensorflow/tensorflow/blob/r1.13/tensorflow/contrib/eager/python/examples/generative\\_examples/cvae.ipynb](https://github.com/tensorflow/tensorflow/blob/r1.13/tensorflow/contrib/eager/python/examples/generative_examples/cvae.ipynb). [Hozzáférés dátuma: 3. május 2019.].
- [28] „Graph theory,” Wikipedia, [Online]. Available: [https://en.wikipedia.org/wiki/Graph\\_theory](https://en.wikipedia.org/wiki/Graph_theory). [Hozzáférés dátuma: 16. április 2019.].
- [29] E. A. Bender és S. G. Williamson, Lists, Decisions and Graphs, S. Gill Williamson, 2010, pp. 148-171.
- [30] „LatentConstraints.ipynb,” Google, [Online]. Available: [https://colab.research.google.com/notebooks/latent\\_constraints/latentconstraints.ipynb](https://colab.research.google.com/notebooks/latent_constraints/latentconstraints.ipynb). [Hozzáférés dátuma: 26. április 2019.].
- [31] „Overview - Matplotlib 3.0.3 documentation,” [Online]. Available: <https://matplotlib.org/contents.html>. [Hozzáférés dátuma: 6. május 2019.].

## Ábrajegyzék

- (1) „File:Tree\_graph.svg,” Wikipedia, [Online]. Available: [https://en.wikipedia.org/wiki/File:Tree\\_graph.svg](https://en.wikipedia.org/wiki/File:Tree_graph.svg). [Hozzáférés dátuma: 22. április 2019.].
- (2) „File:Directed\_acyclic\_graph\_2.svg,” Wikipedia, [Online]. Available: [https://en.wikipedia.org/wiki/File:Directed\\_acyclic\\_graph\\_2.svg](https://en.wikipedia.org/wiki/File:Directed_acyclic_graph_2.svg). [Hozzáférés dátuma: 22. április 2019.].
- (3) „File:UndirectedDegrees.svg,” Wikipedia, [Online]. Available: <https://en.wikipedia.org/wiki/File:UndirectedDegrees.svg>. [Hozzáférés dátuma: 22. április 2019.].
- (4) „File:Labyrinth at Chartres Cathedral.JPG,” Wikipedia, [Online]. Available: [https://en.wikipedia.org/wiki/File:Labyrinth\\_at\\_Chartres\\_Cathedral.JPG](https://en.wikipedia.org/wiki/File:Labyrinth_at_Chartres_Cathedral.JPG). [Hozzáférés dátuma: 22. április 2019.].
- (5) Ssolbergj, „File:Labyrinthus.svg,” [Online]. Available: <https://commons.wikimedia.org/w/index.php?curid=23252646>. [Hozzáférés dátuma: 22. április 2019.].
- (6) Samueljohn.de, „File:MorrisWaterMaze.svg,” Wikipedia, [Online]. Available: <https://en.wikipedia.org/wiki/File:MorrisWaterMaze.svg>. [Hozzáférés dátuma: 22. április 2019.].
- (7) Samueljohn.de, „File:ElevatedPlusMaze.svg,” Wikipedia, [Online]. Available: <https://en.wikipedia.org/wiki/File:ElevatedPlusMaze.svg>. [Hozzáférés dátuma: 22. április 2019.].
- (8) Bd008, „File:Barnes maze.jpg,” Wikipedia, [Online]. Available: [https://en.wikipedia.org/wiki/File:Barnes\\_maze.jpg](https://en.wikipedia.org/wiki/File:Barnes_maze.jpg). [Hozzáférés dátuma: 22. április 2019.].
- (9) Mcole13, „File:Simple Radial Maze.JPG,” Wikipedia, [Online]. Available: [https://en.wikipedia.org/wiki/File:Simple\\_Radial\\_Maze.JPG](https://en.wikipedia.org/wiki/File:Simple_Radial_Maze.JPG). [Hozzáférés dátuma: 22. április 2019.].
- (10) OpenClipart-Vectors, „Labyrinth Maze Meander,” Pixabay, [Online]. Available: <https://pixabay.com/vectors/labyrinth-maze-meander-concentrical-153958/>. [Hozzáférés dátuma: 22. április 2019.].

- (11) V. Nexoth, „File:Simple Maze.svg,” Wikipedia, [Online]. Available: [https://en.wikipedia.org/wiki/File:Simple\\_Maze.svg](https://en.wikipedia.org/wiki/File:Simple_Maze.svg). [Hozzáférés dátuma: 22. április 2019.].
- (12) J. Buck, „Buckblog: Maze Generation: Hunt-and-Kill Algoritm,” [Online]. Available: <http://weblog.jamisbuck.org/2011/1/24/maze-generation-hunt-and-kill-algorithm>. [Hozzáférés dátuma: 22. április 2019.].
- (13) Zieben007, „Maze Generation Algorithm,” Wikipedia, [Online]. Available: [https://en.wikipedia.org/wiki/Maze\\_generation\\_algorithm#Recursive\\_division\\_method](https://en.wikipedia.org/wiki/Maze_generation_algorithm#Recursive_division_method). [Hozzáférés dátuma: 22. április 2019.].
- (14) „Category:Mazes,” Wikimedia Commons, [Online]. Available: <https://commons.wikimedia.org/wiki/Category:Mazes>. [Hozzáférés dátuma: 22. április 2019.].
- (15) J. Buck, „Bucklog: Theseus 1.0,” [Online]. Available: <http://weblog.jamisbuck.org/2010/12/20/theseus-1-0.html>. [Hozzáférés dátuma: 22. április 2019.].
- (16) J. Buck, „Bucklog: Maze Generation: Weave mazes,” [Online]. Available: <http://weblog.jamisbuck.org/2011/3/4/maze-generation-weave-mazes.html>. [Hozzáférés dátuma: 22. április 2019.].
- (17) C. Goulding, „mazes,” MazeGen, [Online]. Available: <https://mazegen.wordpress.com/tag/mazes/>. [Hozzáférés dátuma: 22. április 2019.].
- (18) Dake és Mysid, „File:Neural network.svg,” Wikimedia Commons, [Online]. Available: [https://commons.wikimedia.org/wiki/File:Neural\\_network.svg](https://commons.wikimedia.org/wiki/File:Neural_network.svg). [Hozzáférés dátuma: 24. április 2019.].
- (19) ROzzy, „Fájl:Mesterséges neuron vázlatos rajza.png,” Wikimedia Commons, [Online]. Available: [https://hu.wikipedia.org/wiki/F%C3%A1jl:Mesters%C3%A9ges\\_neuron\\_v%C3%A1zlatos\\_rajza.png](https://hu.wikipedia.org/wiki/F%C3%A1jl:Mesters%C3%A9ges_neuron_v%C3%A1zlatos_rajza.png). [Hozzáférés dátuma: 24. április 2019.].
- (20) M. A. Nielsen, Neural Networks and Deep Learning, Determination Press, 2015.
- (21) „Descending into ML: Training and Loss,” Google, [Online]. Available: <https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss>. [Hozzáférés dátuma: 24. április 2019.].



- (22) „Reducing Loss: Learning Rate,” Google, [Online]. Available: <https://developers.google.com/machine-learning/crash-course/reducing-loss/learning-rate>. [Hozzáférés dátuma: 26. április 2019.].
- (23) „LatentConstraints.ipynb,” Google, [Online]. Available: [https://colab.research.google.com/notebooks/latent\\_constraints/latentconstraints.ipynb](https://colab.research.google.com/notebooks/latent_constraints/latentconstraints.ipynb). [Hozzáférés dátuma: 26. április 2019.].