Orbán Dávid bemutatja:

Slitherlink

Tartalom

Bevezető:

A Slitherlinket azért választottam, mivel érdekes kihívásnak tűnt, hogy megírjak egy olyan programot, amely képes egy NP nehéz probléma megoldására tűrhető időn belül.

Technikai adatok:

A projektre a Python-t választottam, mivel annak ellenére, hogy nem volt személyesen még vele sok tapasztalatom, tudtam, hogy a flexibilitása miatt sokkal jobban illik ide, mint a C++, amit leggyakrabban használok.

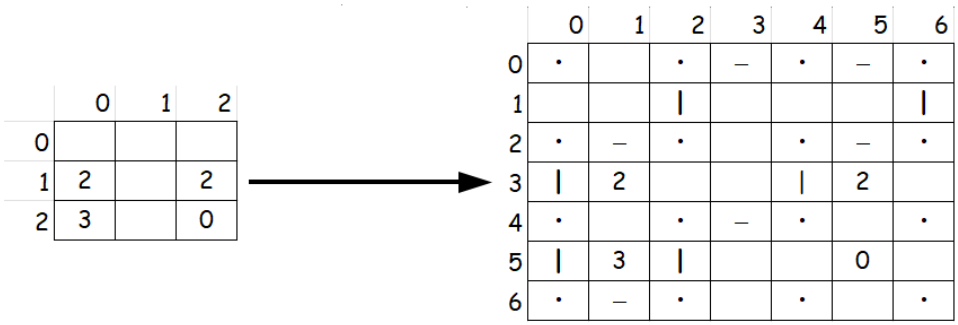
Maga a csatolt mappa két mappából és három fájlból áll: a `pregenboards` előre regenerált játéktáblákat tárol, a `resources` a program által használt betűtípusokat illetve konfigurációs fájlokat tartalmaz, a `main.py` magát a programot, a `requirements.txt` pedig a használt könyvtárakat, illetve verzióikat, a pip által generált formátumban. (A `pip install -r 'requirements.txt'` parancs megkönnyíti az importálásukat.)

(Kérdéses a relevanciája, de azért nem hagyom ki: a saját számítógépemen Visual Studio Code-ban dolgoztam egy Python által létrehozott virtual environment-ben: a Python installációm verziója 3.13.3, a pip-é 25.1.1; továbbá amennyiben valamilyen esetleges probléma merülne fel a megadott fájlokkal, a projekt Github-on is elérhető.)

A keret

Elsősorban essen szó a program keretéről, vagyis minden elemről amely nem a megoldásért, vagy egy új tábla generálásáért felel.

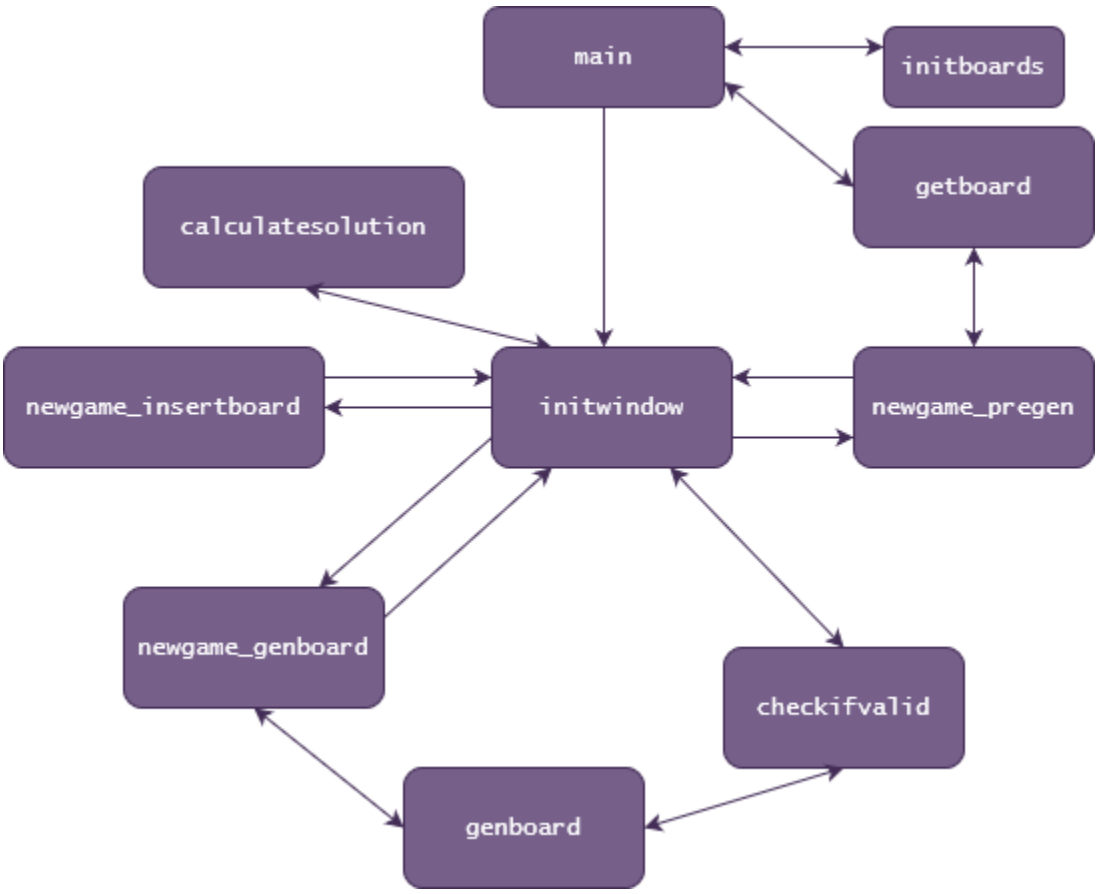
A program négy fő globális változón alapul: az n és m a jelenleg betöltött tábla magasságát, illetve szélességét tárolják el, a v a játékos által is modosítható táblázatot, míg a sol a megoldást tárolja. Ez a két mátrix a következőképpen képezi le a Slitherlink táblát: az $n * m$-es látható táblázatot mint $(2 * n + 1) * (2 * m + 1)$ táblázat tárolja el. A látható értékek a dupla páratlan indexű cellákban vannak, a dupla páros indexű cellákban a pontok (amik nem befolyásolják a táblázatot, de majd később hasznosak lesznek), a többi cellában pedig az élek.



(Egy egyszerűbb feladvány és a vizualizálása annak, hogy a megoldásának melyik elemét a táblázat melyik cellája tárolja.)

A program felépítése

A következő ábra demonstrálja a program felépítését a függvények nézőpontjából:



Ahol a dupla-irányítású nyíl azt jelenti, hogy a függvény futása után a meghívója folytatja a saját feladatát, az egyirányú nyíl pedig azt jelzi, hogy a meghívott függvény meghívása után az adott függvénynek nincsenek további feladatai.

A program a következőféleképpen működik:

- a `main` függvény meghívja az `initboards`-ot, amely detektálja a mappát, ahol a file található, illetve a `getboard`-ot, amely betölti a `v`-be az egyik előre kiválasztott előre generált táblát (egyeseknek közülük a megoldása is előre le van generálva, abban az esetben azt is betölti)
- a `main` függvény ezután meghívja az `initwindow` függvényt, ahol a játék maga fut: a pygame megrajzolja a táblát, érzékeli a klikkeket, et cetera. Az `initwindow` öt függvényt képes meghívni:

- A checkifvalid leellenőrzi, hogy a mostani táblakonfiguráció egy helyes megoldás-e. Ez által a játék manuálisan is játszható.
- A calculatesolution kiszámolja a megoldást, amennyiben az szükséges. (Például akkor nem muszáj, hogyha előre generált tábla aminek meg volt jegyezve a megoldása, vagy ha általunk generált tábla.)
- A három newgame funkció mind egy-egy új játék felállításáért felel:
    - A newgame_pregen kinyitja a pregenboards mappát, ahonnan a felhasználó kiválaszthatja, hogy melyik táblát szeretne betölteni (amely a getboard funkció segítségével történik)
    - A newgame_genboard bekér két értéket, az új-n-et és az új-m-et, és ezeket továbbadja a genboard függvénynek, amelyről majd bővebben a későbbiekben lesz szó.
    - A newgame_insertboard bekér egy Loopy stílusú ID-t, és az alapján betölt egy új táblát.

A fontosabb függvények:

A keret két legfontosabb függvénye az initwindow illetve a checkifvalid. A következőkben ennek a kettőnek a működéséről lesz szó.

Initwindow

A grafikus felület kezeléséért a Pygame (Community Edition) illetve a Pygame GUI nevű könyvtárak felelnek a legnagyobb mértékben.

Mielőtt elkezdene futni maga a game-loop, a háttérre odarajzolja a pontokat és számokat, illetve kiszámolja az összes él adatait (méret, helyezés). A kezdőgombokat is ekkor rajzolja a képernyő jobb oldalára. Magukat az éleket minden ciklusban újrarajzoljuk, a következőféleképpen: amennyiben egy él nem aktív, rajzolunk egy dezaktivált élet, és azon keresztül detektáljuk, hogy az egér arra a pozicióra klikkelt-e, amelyik esetben átállítjuk aktív éllé.

Párhuzamosan történik a gombok rendezése is: összesen 4 konfiguráció van jelen:

- A kezdő, amikor jelen vannak a New Game, Check if valid és Show Solution (vagy Hide Solution) gombok
- A New Game megnyomása után ezeket lecseréli a három newgame funkciónak megfelelő gomb: a Load board, Generate board, illetve az Insert board
- A második kettő esetében újra megváltozik az interface: amennyiben a Generate board-ra kattintott, a felhasználó két mezőbe írhatja be az új n és m értékeket, míg az Insert board esetében egyetlen mező jelenik meg, az ID számára.

Ami még megemlítendő, az a Show Solution viselkedése: amikor a felhasználó megnyomja a gombot, onnantól a program átvált arra, hogy csak a megoldást mutassa, nem vesz be a táblázat bemenetet ameddig a megoldást a felhasználó el nem rejti (az újonnan megjelent Hide Solution gomb segítségével).

Checkifvalid

A függvény írásának során három kritériumot állítottam fel, amit egy adott Slitherlink táblának teljesíteni kell, hogy helyes megoldásnak nevezhessük:

1. Pont követelmény: minden pont körül vagy 0 vagy 2 aktív él kell legyen
2. Cella követelmény: minden cella körül a saját értékével egyenlő számú aktív él kell legyen
3. Bejárási/árasztási követelmény: Amennyiben indítunk egy bejárást egy aktív élről, és azok mentén haladunk, szükségesen el kell érjük az összes aktív élt
    a. Itt jönnek képbe a pontok: ők is részét képezik az aktív-él rendszernek: azok a pontok, amelyek körül két aktív él van, aktívak önmaguk is, illetve amelyek mellett nincs egy se, azok inaktívak

Ugyanezek a kritériumok képezik az alapját a megoldásért felelős függvénynek is.

A háttérfüggvények

Ezek a függvények nem vesznek részt magának a játéknak a működésében, hanem a fejlesztés közbeni tesztelésre voltak használatosak. Ezek a következők:

- A `valid(i, j)` arról felelt, hogy az adott `(i, j)` kóordináták egy valid poziciót reprezentáltak-e.
- A `printnumbers` kiírja az adott táblázat celláinak értékét, míg a `printtotal` megrajzolja console-ban a teljes táblázatot, élekkel és pontokkal együtt.
- Az `automatedtesting` pedig arra szolgál, hogy a generátor/megoldó gyorsaságát lehessen tesztelni: `testfor` darab `ntest * ntest` méretű táblázat generálásánál/megoldásánál.

A megoldó

A $\mathrm{calculatesolution}$ függvény felel a sol táblázat feltöltéséért, amennyiben az üres.

Háttér

A projektet azzal kezdtem, hogy utána néztem, melyik lenne a legjobb megközelítés egy teljes megoldásra. Hamar találtam egy olyan Github repository-t, amelynek fejlesztője a Wikipédián is látott szabályokat használta, találgatással kiegészítve. Ám miután tovább keresgéltem, hamar szembementem egy másik megközelítéssel is, pontosabban ezen az oldalon.

Itt találkoztam először a SAT gondolatával, amely a jó iránynak bizonyult, hisz sok más hasonló SAT alapú Slitherlink megoldó programot találtam, de messze a legjobb forrásanyagnak ez a weboldal bizonyult, amely összegezte a máshol is megjelent gondolatokat.

SAT

A Boolean satisfiability problem azt foglalja magába, miszerint egy adott logikai kifejezés minden változójának helyébe behelyettesítve azt, hogy igaz vagy hamis, elérhető-e az, hogy a teljes kifejezés igaz legyen. Ez volt a legelső probléma amelyre bizonyítva volt, hogy NP-teljes. (A Cook-Levin Tétel alapján.)

Mivel igen nagy múltú és fontosságú problémáról van szó, a megoldására számtalan algoritmus volt elkészítve, amelyek, ámbár nem futnak polinomiális időben, mégis elég gyorsak gyakorlati használatra.

A projekten belül a PySAT könyvtár által biztosított Glucose 4.2.1 SAT Solver-t használtam, amely, a legtöbb SAT solverhez hasonlóan, a következőképpen működik:

A felhasználó elkészíti a logikai kifejezést, egy CNF (Conjuctive Normal Form) formájában. Egy CNF diszjunkt állítások konjunkciójából áll, egyszerűbben mondva, olyan kifejezések kezelésére képes, amelyek a következőképpen néznek ki:

$$(\mathrm{A}_1 \text{ VAGY } \mathrm{B}_1 \text{ VAGY } \ldots \mathrm{Z}_1) \text{ ÉS } (\mathrm{A}_2 \text{ VAGY } \mathrm{B}_2 \text{ VAGY } \ldots \mathrm{Z}_2) \text{ ÉS } \ldots . (\mathrm{A}_N \text{ VAGY } \mathrm{B}_N \text{ VAGY } \ldots \mathrm{Z}_N)$$

Ahol minden elem egy szigorúan pozitív egész számot jelöl, illetve akármilyen elem megadható negatív számként is, ami a NOT logikai operátornak felel meg. (Pl. $\text{-P} = \mathrm{NOT\ P}$, vagy egy csinosabb szimbólumot használva, $\neg \mathrm{P}$).

(A továbbiakban az egyszerűség kedvéért a standard logikai szimbólumokat fogom használni: $\neg P = \text{NOT P}$, $A \wedge B = A \text{ ÉS } B$, $A \vee B = A \text{ VAGY } B$)

Láthatjuk, hogy a checkifvalid-ban is megemlített három kondíció közül az első kettő igen egyszerűen felírható CNF formában, csak le kell kezelni a kombinációkat, amelyek ilyen apró számok esetén triviálisak[1].

Illetve ezt mondanám, ha a SAT képes lenne lekezelni konjuktív állítások diszjunkcióit, de persze erre már csak akkor jöttem rá, mikor már ott tartottam, hogy a kódot is meg akartam írni.

Szerencsére van egy beépített függvény, amely, amennyiben a felhasználó megad egy listát változókkal, illettve azt, hogy hány kell belőlük igaz legyen, a háttérben felépíti a szükséges auxiliary változókat. Viszont a Pont követelmény esetében ez nem volt elég, hiszen nem kezelte le azt az esetet, amikor egy vonal se csatlakozik az adott ponthoz hozzájuk.

Tehát az első nagyobb kihívás a megoldás esetében az volt, hogy manuálisan elkészítsem a következő típusú logikai állításokat:

$(A_1 \wedge B_1 \wedge \ldots Z_1) \vee (A_2 \wedge B_2 \wedge \ldots Z_2) \vee \ldots . (A_N \wedge B_N \wedge \ldots Z_N)$,

a CNF keretein belül.

Az általános módszert illusztrálom egy kisebb példán keresztül; legyen az általunk várt logikai kifejezés $(A \wedge B) \vee (C \wedge D)$. Észrevehetjük, hogy amennyiben az $A \wedge B$-t, illetve a $C \wedge D$-t el tudjuk tárolni, mint két másik változó, például X és Y, akkor az $X \vee Y$ feltétel könnyedén hozzáadható a CNF-hez.

Ahhoz, hogy felállítsuk az ilyen jellegű logikai ekvivalenciákat ($A \wedge B = X$, $C \wedge D = Y$), a következőképpen kell eljárjunk: fel kell állítani egy logikai bikondicionalitási összefüggést az elemek között, vagyis:

$A \wedge B \leftrightarrow X$

Ezt két részre bontjuk:

[1] Minden pontra (maximimum) 7 esetünk van: a C_4^2 amelyben behúzunk két élet, illetve egy, ahol egyet se. Elnevezve a körülötte levő éleket, a példa kedvéért ABCD-vel, egy természetes megközelítés így nézne ki: $(A \wedge B \wedge \neg C \wedge \neg D) \vee (A \wedge \neg B \wedge C \wedge \neg D) \vee \ldots \vee (\neg A \wedge \neg B \wedge C \wedge D)$. Hasonló megközelítés érvényes a cellá követelményekre is.

Először is, amennyiben X igaz, akkor A és B is igaz kell legyen: ez standard logikai formában $X \to A$, illetve $X \to B$, a CNF számára viszont a következőképpen szükséges átdolgozni: $\neg X \vee A$, illetve $\neg X \vee B$

A második része az állításnak az a $A \wedge B \to X$, amely a CNF által elvárt struktúra szerint a következő: $\neg A \vee \neg B \vee X$.

Mindezeket a kifejezéseket hozzá kell adnunk a CNF-hez, illetve hasonló módon szükséges felállítani az ekvivalenciát az Y esetében is, legvégül pedig az eredetileg is kívánt összefüggést pedig hozzáadhatjuk a CNF-hez mint $X \vee Y$.

Ezzel egy valid megoldás kétharmada le van fedve, és marad a nagy kérdés, hogy hogy fejezzük ki az összefüggőségi feltételt ilyen CNF formában. A válasz, amelyre az előbb említett modell készítői is jutottak, hogy nem muszáj. A jelenlegi algoritmus tehát, az első két szabály alapján generált megoldást aláveti a harmadik követelménynek: minden megtalált hurkot egyenként hozzáad, mint új „negatív” követelmény (vagyis, $(\neg A \vee \neg B \vee \neg C \vee \ldots \neg Z)$, ahol a számok az adott hurok éleinek indexeit jelölik, ez azt eredményezi, hogy nem lehet az összes újra egyszerre igaz, tehát nem jöhet létre ugyanaz a hurok): amennyiben csak egy hurok van, a generált megoldást visszaküldi, mint helyes megoldás, amennyiben pedig több, akkor az újonnan hozzáadott feltételeket felhasználva újra futtatja a Glucose-t.

Technikai részletek

A függvény minden élhez hozzárendel egy természetes számot mint index, amely folyamatért az IDPool nevű függvény felel, biztosítva, hogy egy bizonyos index nincs több objektumra felhasználva. A hurkok megkeresése, hasonlóan mint a checkifvalid esetében, egy BFS algoritmus segítségével történik.

Futási idő

Fontos kontextus: az én számítógépem, amelyen ezeket a teszteket futattam, tartalmaz egy Ryzen 7 5800X3D CPU-t, egy RX 6600 GPU-t és 32 GB 2400 MHz-es RAM-ot. Ennek függvényében, két szett tesztet futtatam le: az elsőben, a Loopy által generált Hard táblákat manuálisan beraktam ID szerint, és így körül-belül tíz táblát oldott meg méretenként. Mivel ez elég kicsi sample size, ezért a következő tesztet automatizáltam, és 1000 táblát generáltam a saját generátorommal, amelyen aztán teszteltem a solver-t.

Méret	Loopy Hard táblák átlagmegoldásideje (másodpercben) (~10 tábla átlaga)	Saját generált táblák átlagmegoldásideje (másodpercben) (1000 tábla átlaga) (kihagyva a legjobb és legrosszabb 5%-ot)
5x5	0.004138649	0.002630752
7x7	0.007655412	0.004975691
10x10	0.016155039	0.011292147
15x15	0.037890752	0.029501139
20x20	0.081676865	0.060441767
25x25	0.164252162	0.117149645
30x30	0.249380418	0.201967567
35x35	0.378236771	0.338929758
40x40	0.615737319	0.569826421

Vannak viszont olyan táblák, amelyek az átlagszámolásból ki lettek hagyva, amik messze meghaladják ezeket az átlagértékeket. A pregenboards-ba elmentettem egy párat, viszont ezt csak a tesztelés végén vezettem be, ezért nincs olyan sok. A legrosszabb azok közül amelyeket elmenttettem az a sol_lagger_40x40_568s.txt nevű fájlban található: ahogy a nevében is megjelenik, 568 másodpercbe telt ameddig megoldotta a solver.

A generátor

Háttér

Ebben a témában kezdetben csak egy különálló forrásanyagot találtam: ezen blogposztot. Az alapgondolatot átvettem onnan, ha először a többit nem is: az általános gondolat egy probléma generálására, az egy színezős hozzáállás, amely alapján random módon beszínezünk egy összefüggő teret, és a színezett és színezetlen területlen közötti vonal az, amely a Slitherlink probléma alapját fogja képezni.

Az első próbálkozásom egy egyszerű flood fill volt, amely egy random meghatározott belső pozicióról indult, és minden cellánál egy egyszerű random esély alapján döntötte el, hogy melyik szomszédaiba fog tovább terjedni. Ez az algoritmus fölötté unalmas formákat generált, ezért megpróbáltam közelebb követni a blogposztban leírt megközelítest: annak a szerzője azt a megközelítést alkalmazta, hogy először megrajzolt egy nagyon egyszerű formát, és a színezett és beszínezetlen rész közötti vonalat próbálta minél kacsgaringósabbá tegye. Számomra ez a megközelítés nem vezetett sikerre, ezért visszatértem inkább az eredeti ötletemre, csak fordítva.

Algoritmus

A végső megközelítés így szólt: elképzeljük, hogy a tábla kezdetben teljesen be van színezve. A formáját úgy igyekezzük érdekesebbé tenni, hogy „kivágunk” belőle darabokat.

Maga az algoritmus nem olyan komplex, de sok apró részlete van, ezeket megjelenésük sorrendjében említem:

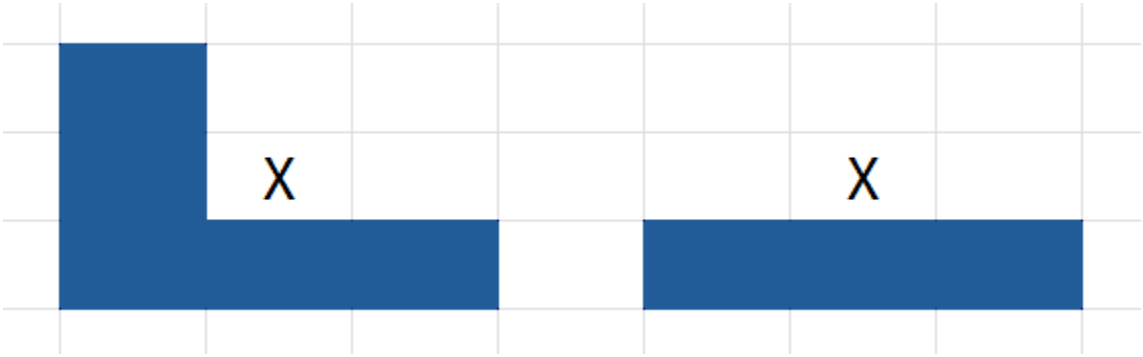
- Lerögzítjük az elárasztandó cellák számát: $\mathrm{toflood} = \mathrm{random}(\lceil (n * m)/2 \rceil, \lceil (n * m) * 3/4 \rceil)$, ahol a $\mathrm{random}(\mathrm{a}, \mathrm{b})$ visszaad egy x számot ($\mathrm{a} <= \mathrm{x} <= \mathrm{b}, \mathrm{x} \in \mathbb{Z}$)
- Ezután kiszámoljuk a szélen levő cellák számát, $\mathrm{edgecells} = 2 * n + 2 * (m - 2)$, és ezt felhasználva meghatározzuk a kivágások/betüremkedések számát: $\mathrm{intrusions} = \lfloor 0.66 * random(\lfloor edgecells/5 \rfloor, \lfloor edgecells/2.5 \rfloor) \rfloor$
- Ezt követve egy Dirichlet disztribúció segítségével generálunk minden intrusion-re egy egyedi értéket, amelyeknek összege pontosan $\mathrm{toflood}$
- A következő lépésben legeneráljuk a kezdőpoziciókat: minden $\mathrm{intrusion}$-re egyet. Az algoritmus random sorrekbe rakja a négy oldalt, és mindegyikre

generál egy insertionpoint-ot: amennyiben mind a négyre generált egyet, újra sorrendbe rendezi a őket, és folytatja tovább. Ezt addig teszi, amíg legenerálja a kért számú insertionpoint-ot.

- Ezután felállít egy pár változót, amelyek a fő lerakás-loop folytatását vezérlik: floodedcells, totaltries, totalfailures, maxfuilres. Ezekre alapulva, a fő loop a következő logikai kifejezés függvényében folytatja a próbálkozásait:
  toflood – floodedcells $>$ 0 and totaltries $<=$ intrusions * 5 and totalfailures $<$ maxfailures * 2
- A loop-on belül a következő folyamat történik: amennyiben még van előregenerált insertionpoint, azt felhasználja, másképp továbbmegy a táblázat széle mentén, és a cellákat amelyek színezettek, és mindkét oldalsó szomszédjuk színezett, berakja egy listába, ahonnan random kiválasztja az egyik elemet. Az ilyen módon stabilizált kezdőpontot felhasználva indít el egy bejárást.
- A bejárás mérete előre meg van határozva: amennyiben az egyik eredeti insertionpoint, akkor az egyik előregenerált betüremkedésméretet használja, másképp generál egy újat: fsize = random($\lceil rem/2 \rceil$, $\lceil rem * 1.25 \rceil$), ahol rem = toflood – floodedcells
- Illetve meghatározza a kezdőszélet is, amely függvényében inicializálja az „ideális” koordinátákat: Ezek a középtől 10%-ra vannak, általában. Az elméleti céljuk az, hogy miután egy betüremkedés elér egy minimum „mélységet” azután széltében kezdjen inkább terjeszkedni
- Ezt követve kezdi el magát az árasztást: egy szortírozott listát használva számon-tartja a lehetséges pozíciókat, és mindig azokat árasztja el, amelyeknek a legnagyobb a súlyuk
- A szomszédos cellák súlyának meghatározásánál három faktor van jelen:
  - Az orthoweight az alapján befolyásolja, hogy az adott cella hányadik a látogatási sorrendben: minden árasztásnál, attól függően hogy melyik széltől indul. Például ha a legfelső széltől indul, akkor legelőször az alatta levő cellát nézi, aztán random módon a jobbra-balra levő cellákat, és csak legutolsónak a felfele levőt.
  - A distweight azt határozza meg, hogy az adott pozició milyen közel van az „ideális” koordinátákhoz
  - A randomweight pedig az, aminek hangzik
- Ezeket egy posexp nevű listában tárolja el a program, és csak akkor addja hozzá a látogatandó cellák listájához, amennyiben a mostani cella

átmegy a teszteken amelyek biztosítják, hogy kivágásával a beszínezett rész nem szakad több komponensre.

- A kapott táblázatot még két módosításnak vettjük alá: az elsőben az olyan egyedüli cellákat keressük, amelyek be vannak színezve, viszont az összes szomszédjuk nincs. Ezeket dezaktiváljuk.
- Továbbá, visszatérve a blogposzthoz, párszor átfutunk a táblázaton és próbáljuk növelni a kacskaringóságát. Egy cella megváltoztatása a következő két esetben (és az összes fordítással kapott változatukban) eredményezi a kacskaringósság növekedését:



Ahol az X-el jelölt cella az, amelyiket épp nézünk. Azt, hogy hányszor futja végig a táblázatot, a `maxwiggleincrease` határozza meg. Ez jelenleg `3`-ra van állítva, mivel gyakran ahhoz vezet, hogy a táblázat helytelenné válik.

- Az utolsó előtti lépés a táblázat ellenőrzése: felépítjük magát a Slitherlink táblázatot, a megoldással együtt, és beküldjük a `checkifvalid`-ba: amennyiben nem helyes, a folyamatot elölről kezdjük.
- Az utolsó lépés az, hogy a generált táblából kitöröljünk egyes értékeket, és betöltsük a `v`-be. Itt pedig úgy döntöttem, hogy figyelmen kívül hagyom az egyik követelményét a játéknak, és nem próbálom ellenőrizni, hogy biztosan csak egyetlen lehetséges megoldás van-e minden változtatás után. A számok kitörlését tehát egy egyszerű random esélyszámítás vezérli: a legnagyobb eséllyel a nullákat törli ki, majd az egyeseket és hármasokat, és a legkevesebb eséllyel a ketteseket: a szerint voltak sorolva, hogy melyik cellaérték biztosítja a legtöbb információt.
- Még egy utolsó lépésben pedig megint végigfut a kész táblázaton, és ott, ahol egy cella körül egyetlen látható érték sincs, ott visszarajzolja, és ahol sok látható érték van összetömörülve, ott kitöröl párat.

Szubjektív megítélés

Kicsi méretekre (10x10 alatt) eléggé sablonos/unalmas formájú problémákat generál, de viszont 15x15-től felfele már eléggé érdekes formákat tud kihozni, tehát össz-vissz elégedett vagyok vele.

Futási idő

Amint feljebb is meg volt említve, ezek az adatok az én számítógépemen való futtatásból származnak, amely tartalmaz egy Ryzen 7 5800X3D CPU-t, egy RX 6600 GPU-t és 32 GB 2400 MHz-es RAM-ot. Ennek függvényében, a bizonyos táblaméretek átlag generálási ideje, mindegyiknél 1000 random generált tábla generálási idejének átlagát véve:

Méret	Átlaggenerálásidő (másodpercben) (Átlagszámoláskor az értékek közül ki volt hagyva a legjobb és legrosszabb 5%)
5x5	0.000904294
7x7	0.001734423
10x10	0.004087591
15x15	0.013633817
20x20	0.038971167
25x25	0.110207774
30x30	0.308101618
35x35	0.857622278
40x40	2.486280336

Esetleges kérdések

Q: Volt AI használva a program készítésére?
A: A ChatGPT-hez nem egyszer fordultam egy-egy kérdéssel, de nem nagyon használtam direkt az általa írt kódot.

Q: Milyen rendszereken volt tesztelve?
A: Windows 10 illetve CachyOS (Arch alapú Linux disztribúció)

Források

- A Slitherlink megoldása klasszikusan, szabályok segítségével
  - pinkston3/Slitherlink
- Boolean satisfiability problem
  - Wikipédia
  - Glucose SAT Solver
  - PySAT
- A Slitherlink megoldása SAT Solver segítségével
  - Slitherlink, Minisat and Emscripten
  - How to solve Slitherlink using SAT solver
- Problémák generálása
  - Slitherlink Comps Project
  - Liam Appelbe: How to generate Slither Link puzzles