Orbán Dávid bemutatja:

**Slitherlink**

Tartalom

[Bevezető: 2](#_Toc197432662)

[Technikai adatok: 2](#_Toc197432663)

[A keret 3](#_Toc197432664)

[A program felépítése 4](#_Toc197432665)

[A fontosabb függvények: 6](#_Toc197432666)

[Initwindow 6](#_Toc197432667)

[Checkifvalid 7](#_Toc197432668)

[A megoldó 8](#_Toc197432669)

[Háttér 8](#_Toc197432670)

[SAT 8](#_Toc197432671)

# 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 két 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](https://github.com/odbv/Slitherlink-Termtud).)

# 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.

A black arrow pointing to a grid

AI-generated content may be incorrect.

(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:

A diagram of a software system

AI-generated content may be incorrect.

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. (Nem muszáj pont azt a struktúrát tartani: igazából az mxn és a kettőspont között akármi lehet, mert azt figyelmen kívül hagyja, így egyes más implementációkkal is kompatibilis.)

## 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-téglalap adatait. 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. Amennyiben indítunk egy bejárást egy aktív élről, és azok mentén haladunk, szükségesen el kell érjük az összes aktív élt
   1. 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 megoldó

A 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](https://www.dougandjean.com/slither/index.html).

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](https://esolangs.org/wiki/User:Hakerh400/How_to_solve_Slitherlink_using_SAT_solver) bizonyult, amely összegezte a máshol is megjelent gondolatokat.

## SAT

A [Boolean satisfiability problem](https://en.wikipedia.org/wiki/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:

(A1 VAGY B1 VAGY . . . Z1) ÉS (A2 VAGY B2 VAGY . . . Z2) ÉS . . . . (AN VAGY BN VAGY . . . ZN)

Ahol minden elem egy szigorúan pozitív egész számot jelöl, illetve akármilyen elem megadható negatív számként is, ami a NOT logikai operátornak felel meg. (Pl. -P -> NOT P, vagy egy csinosabb szimbólumot használva, ¬P).

(A továbbiakban az egyszerűség kedvéért inkább a standard logikai szimbólumokat fogom használni.)

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.

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:

(A1 ∧ B1 ∧ . . . Z1) ∨ (A2 ∧ B2 ∧ . . . Z2) ∨ . . . . (AN ∧ BN ∧ . . . ZN),

a CNF keretein belül.

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

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

A ∧ B ↔ X

Ezt két részre bontjuk:

Először is, amennyiben X igaz, akkor A és B is igaz kell legyen: ez standard logikai formában X → A, illetve X → B, a CNF számára viszont a következőképpen szükséges átdolgozni: ¬X ∨ A, illetve ¬X ∨ B

A második része az állításnak az a A ∧ B → X, amely a CNF által elvárt struktúra szerint a következő: ¬A ∨ ¬B ∨ 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 ∨ 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,