# Z3-hoz theory solver írás kipróbálása (Python vagy Java API-ból):

# Gráfokkal kapcsolatos constraint propagation

**1. Tervezett munka részletes specifikációja**

**Célkitűzések**

A projekt célja egy olyan Python-alapú eszköz megvalósítása, amely hatékonyan kezeli a gráfok logikai tulajdonságainak vizsgálatát és korlátozások (constraints) propagálását. Az eszköz alapjául a **Z3 Solver** szolgál, amely lehetővé teszi a logikai feltételek formalizálását és érvényesítését. A projekt az alábbi funkciók megvalósítására összpontosít:

1. **Gráfmanipuláció**
   * Csúcsok és élek dinamikus hozzáadása.
   * A gráf struktúra szerkesztése és többféle gráftulajdonság vizsgálata.
2. **Reflexív tranzitív lezárás (RTC) implementációja**
   * A gráf elérhetőségi tulajdonságainak meghatározása tranzitív kapcsolatokkal.
3. **Gráftulajdonságok vizsgálata**
   * Összefüggőség ellenőrzése.
   * Ciklusok azonosítása (ciklusmentesség).
   * Egyszerűség és teljes gráf tulajdonságok vizsgálata.
4. **Constraint Propagation Theory alkalmazása**
   * A gráfokkal kapcsolatos logikai feltételek automatikus propagációja, például:
     + Reflexív tranzitív kapcsolatok propagációja.
     + Transzitivitási konfliktusok detektálása.
5. **Optimalizálás és specifikus propagációk**
   * Reflexív tranzitív kapcsolatok iteratív feldolgozása a skálázódási problémák csökkentésére.
6. **Tesztelés**
   * Automatizált tesztelés és edge-case-ek kezelése a test\_Graph\_Propagator.py segítségével.
   * A gráf tulajdonságainak ellenőrzése, beleértve:
     + Összefüggőség tesztelése (test\_connectivity).
     + Ciklusmentesség tesztelése (test\_acyclicity).
     + Egyszerűség vizsgálata (test\_simple\_graph).
     + Teljesség ellenőrzése (test\_complete).

**Specifikált funkciók**

1. **add\_node(node)**: Csúcsok dinamikus hozzáadása a gráfhoz.
2. **add\_edge(u, v)**: Élek hozzáadása irányított vagy irányítatlan gráfokhoz.
3. **is\_connected()**: A gráf összefüggőségének ellenőrzése.
4. **is\_acyclic()**: A gráf ciklusmentességének vizsgálata.
5. **propagate\_rtc()**: Reflexív tranzitív lezárás kiszámítása.
6. **detect\_transitivity\_conflicts()**: Transzitivitási konfliktusok detektálása a solver segítségével.

**Végső eredmény**

Az elkészült modul egy Python-osztály, amely a Z3 Solver integrálásával képes gráfokat kezelni, azok tulajdonságait vizsgálni, és logikai feltételeket propagálni. A projekt teljes tesztkörnyezetet is biztosít az implementált funkciók validálásához.

**3. Kihívások és nehézségek**

**1. Algoritmikus nehézségek**

* **Reflexív tranzitív lezárás (RTC) implementálása**
  + Az RTC algoritmus, amely az összes tranzitív kapcsolatot meghatározza, jelentős memória- és CPU-kapacitást igényel, főleg nagy gráfok esetén. A tranzitivitás komplexitása az élek számának négyzetével nő, ami skálázódási problémákat okozott.
* **Ciklusok és körök kezelése**
  + A gráfok ciklusmentességének ellenőrzése iteratív DFS algoritmusokkal nehezítette a hiba lokalizálását nagy struktúrákban.

**2. Z3 Solver integrációja**

* A Z3 **User Propagator API**-jának megértése időigényes volt. A dokumentációban szereplő alapvető példákat több esetben át kellett alakítani az általunk kezelt gráfproblémákhoz.
* Az érvényes logikai feltételek felépítése és az eredmények interpretálása több iterációt igényelt.

#### ****3. Teljesítménybeli problémák****

* Nagy gráfok kezelésekor a memóriaigény exponenciálisan nőtt, amit optimalizációs stratégiákkal kellett kezelni. Ez különösen fontos volt a reflexív tranzitív kapcsolatok iteratív feldolgozásánál, hogy a skálázódási problémákat minimalizáljuk.

**4. Tesztelési nehézségek**

* Az edge-case-ek, mint az üres gráfok vagy nagy redundanciával rendelkező élek kezelése, pontosan paraméterezett tesztkörnyezetet igényelt.

**4. Használt eszközök elemzése**

**Z3 Solver**

* **Előnyök:**
  + Kiemelkedő teljesítőképesség komplex logikai problémák megoldására.
  + Reflexív tranzitív lezárások és logikai tulajdonságok érvényesítésére optimalizált.
  + Python-kompatibilitás a z3 modul segítségével.
* **Hátrányok:**
  + Dokumentáció hiányos, különösen a User Propagator API mélyebb funkcióit illetően.

**Python**

* **Előnyök:**
  + Magas szintű nyelv, amely gyors prototípus-készítést tesz lehetővé.
  + Széles körű könyvtári támogatás gráfok manipulálásához.
* **Hátrányok:**
  + Nagy gráfok kezelésénél a Python dinamikus jellege miatt a teljesítmény limitált lehet.

**Pytest**

* **Előnyök:**
  + Egyszerű használat, kiváló integráció a Python tesztkörnyezetébe.
  + Paraméterezhető tesztek, amelyek gyors és automatizált validálást tesznek lehetővé.
* **Hátrányok:**
  + Komplex gráfstruktúrákhoz kapcsolódó hibák lokalizálása időigényes.

**5. Személyes vélemény és tanulságok**

**Szakmai vélemény**

A projekt során megtanultam a Z3 Solver logikai problémák kezelésében rejlő erejét, különösen a tranzitív lezárások és a gráfok logikai tulajdonságai terén. A Python egyszerűsége és gyors prototípus-készítési képessége lehetővé tette a funkciók gyors iterációját.

**Tanulságok**

1. A komplex gráfproblémák formalizálása alapvetően meghatározza a megoldás hatékonyságát.
2. A Z3 Solver támogatásával gyorsan implementálhatóak az előre definiált logikai operációk, de további optimalizációkra van szükség nagyobb adathalmazok kezeléséhez.
3. Az automatizált tesztelés kritikus szerepet játszik a modul élettartama során fellépő hibák minimalizásában.

### ****Graph\_Propagator.py részletes dokumentáció****

#### ****Áttekintés****

A **Graph\_Propagator.py** modul célja a gráfstruktúrák kezelésére alkalmas osztály biztosítása, amely támogatja a gráfok tulajdonságainak vizsgálatát és logikai feltételek (constraint-ek) propagációját. Az implementáció alapja a **Z3 Solver**, amely lehetővé teszi a gráf logikai feltételeinek megadását és automatikus ellenőrzését.

### ****Fő Osztály:**** GraphConstraintPropagator

Ez az osztály tartalmazza a gráfok kezelésére szolgáló funkciókat.

#### ****Attribútumok****

1. **solver**: A Z3 Solver példánya, amely a logikai feltételek kezelését végzi.
2. **edges**: Az élek listája.
3. **nodes**: Az egyedi csúcsok halmaza.
4. **graph**: Szótár a csúcsok és szomszédos csúcsok kapcsolatának tárolására.
5. **directed**: Logikai érték, amely meghatározza, hogy a gráf irányított-e.

### ****Függvények: Alapvető funkciók****

#### ****1. Gráfmanipuláció****

* **add\_node(node)**:
  + Hozzáad egy csúcsot a gráfhoz.
  + **Jövőbeni felhasználás**: Dinamikus gráfbővítésekhez használható.
* **add\_edge(u, v)**:
  + Élet ad a gráfhoz (irányított vagy irányítatlan).
  + **Jövőbeni felhasználás**: Gráfok közötti kapcsolat modellezéséhez.

#### ****2. Gráftulajdonságok ellenőrzése****

* **is\_connected()**:
  + Ellenőrzi, hogy a gráf összefüggő-e.
  + **Jövőbeni felhasználás**: Nagy hálózatok elemzése során.
* **is\_acyclic()**:
  + Vizsgálja, hogy a gráf ciklusmentes-e.
  + **Jövőbeni felhasználás**: Fa-szerkezetek modellezésekor.
* **is\_simple\_graph()**:
  + Ellenőrzi, hogy a gráf egyszerű-e (nincs önhivatkozás, nincs párhuzamos él).
  + **Jövőbeni felhasználás**: Tisztán strukturált gráfok validálásakor.
* **is\_complete()**:
  + Ellenőrzi, hogy a gráf teljes-e.
  + **Jövőbeni felhasználás**: Hálózatok maximális összefüggésének modellezése.

#### ****3. Logikai kényszerek propagációja****

* **propagate\_rtc()**:
  + Reflexív tranzitív lezárást számít ki.
  + **Jövőbeni felhasználás**: Elérési kapcsolatok elemzése.
* **detect\_transitivity\_conflicts()**:
  + Transzitivitási konfliktusokat detektál.
  + **Jövőbeni felhasználás**: Logikai modellek validálására használható.

### ****Haladó funkciók és jövőbeni felhasználási lehetőségek****

#### ****1. Adatszerkezetek vizsgálata****

* **compute\_treedepth()**:
  + A gráf fa-mélységét számítja ki.
  + **Jövőbeni felhasználás**: Hálózatok hierarchikus szerkezetének elemzésére.
* **propagate\_dependency\_graph()**:
  + Adatfüggőségi gráfokat kezel.
  + **Jövőbeni felhasználás**: Függőségi kapcsolatok modellezése nagy rendszerekben.

#### ****2. Optimalizáció****

* **optimize\_sparsity()**:
  + Gráf sparsity vizsgálata és optimalizálása.
  + **Jövőbeni felhasználás**: Ritka gráfstruktúrák kezeléséhez.

#### ****3. Kényszerek terjesztése****

* **propagate\_k\_hop\_dominance(k)**:
  + K-hop távolságon belüli dominancia propagáció.
  + **Jövőbeni felhasználás**: Korlátozott elérhetőségű hálózatok modellezésére.
* **enforce\_cycle\_constraints()**:
  + Ciklusok korlátozását biztosítja.
  + **Jövőbeni felhasználás**: Körök kizárása vagy engedélyezése adott rendszerekben.

### ****Debugging és Teszteléshez kapcsolódó funkciók****

* **log\_event(event)**:
  + Események naplózása, segíti a hibakeresést.
  + **Jövőbeni felhasználás**: Rendszerek futásának figyelése.
* **explore\_model()**:
  + Solver-modell feltárása.
  + **Jövőbeni felhasználás**: Megoldási stratégiák ellenőrzése és debugolás.
* **add\_assertions(\*constraints)**:
  + Logikai feltételek tömeges hozzáadása.
  + **Jövőbeni felhasználás**: Nagy modellek egyszerűsített validálása.

#### ****2. test\_Graph\_Propagator.py – Tesztek és Automatizáció****

Ez a fájl az **Graph\_Propagator.py** funkcióinak automatizált tesztelésére szolgál, a pytest keretrendszert használva.

##### **Tesztelt funkciók**

1. **test\_connectivity()**:
   * Ellenőrzi a gráf összefüggőségét különböző bemeneti élek alapján.
   * Példa: [(1, 2), (2, 3)] → True (összefüggő gráf), [(1, 2), (3, 4)] → False (nem összefüggő gráf).
2. **test\_acyclicity()**:
   * Vizsgálja, hogy a gráf tartalmaz-e ciklust.
   * Példa: [(A, B), (B, C)] → True, [(A, B), (B, C), (C, A)] → False.
3. **test\_simple\_graph()**:
   * Ellenőrzi, hogy a gráf egyszerű-e.
   * Példa: [(A, B), (B, C)] → True, [(A, B), (B, A)] → False.
4. **test\_complete()**:
   * Ellenőrzi, hogy a gráf teljes-e.
   * Példa: Egy 4 csúcsos gráfban minden csúcs össze van kötve minden más csúccsal.

##### **Tesztelési eredmények**

* A tesztelési forgatókönyvek teljes mértékben lefedik a gráfok tulajdonságainak ellenőrzésére vonatkozó funkciókat.
* Az edge-case-ek, például üres gráfok és redundáns élek is tesztelve vannak.

#### ****Összegzés****

* A **Graph\_Propagator.py** kód lehetővé teszi a gráfstruktúrák dinamikus manipulációját és tulajdonságainak ellenőrzését a Z3 Solver segítségével.
* A **test\_Graph\_Propagator.py** kód automatizált módon ellenőrzi a fő funkciók helyes működését, paraméterezett tesztesetekkel lefedve a tipikus és szélsőséges forgatókönyveket.
* Mindkét fájl szorosan összefügg és biztosítja a projekt funkcionális és logikai helyességét.