ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

ARC MEGOLDÁSA NEM PROCEDURÁLIS PROGRAM INDUKCIÓVAL

Szerző: **Neumann Norbert Károly**
mérnökinformatikus BSc. szak, 4. évf.

Konzulens: **Pintér Ádám**
tanársegéd

Budapest, 2023.

TARTALOM

1 BEVEZETÉS

Napjainkban a mély tanulás számos területen egyre elterjedtebbé válik, ilyen például a kép és a szövegfeldolgozás, az automatikus fordítás, de egyes önvezető autók vezérlése is ezzel a módszerrel történik. A mély tanulás által betanított neurális hálózatoknak, azonban megvannak a limitációi: a módszer csak akkor képes hatékonyan működni, ha a tanítás folyamán a problématér sűrű mintavételezésen esik át, valamint, ha a teszt minták kellőképpen hasonlítanak a tanítási mintákhoz [1]. Tehát bármely variáció hatására a hálózat teljesítménye drasztikusan lecsökken, ha az algoritmus a tanulás során erről a variációról nem kapott elegendő tanító mintát.
Egy 2018-as publikáció [2] szerint a mély tanulás által betanított hálózatok csupán a tanító minták közötti interpolációra képesek, tehát csak a minták körüli kis környezetben nyújtanak helyes kimenetet, de a tanítási téren kívüli extrapolációra nem alkalmasak.
Más terminológiával, de ugyanez a gondolat lett megfogalmazva François Chollet „Deep Learning with Python" című könyvében [1], ahol a módszer fő limitációjaként megjegyzi, hogy a jelenlegi mélytanuló algoritmusok lokális generalizációra képesek csak, de a tanítási téren kívüli extrapolálásra, azaz az extrém generalizációra nem. Ennek megoldására a könyv szerzője 2018-ban publikálta az „Abstraction and Reasoning Corpus" (ARC) problémát [3], amely az extrém generalizáció mérésére szolgál, ezáltal megoldva ezt közelebb juthatunk egy általánosabb tanuló algoritmus felé. A dolgozatom az ARC megoldására tesz kísérletet.

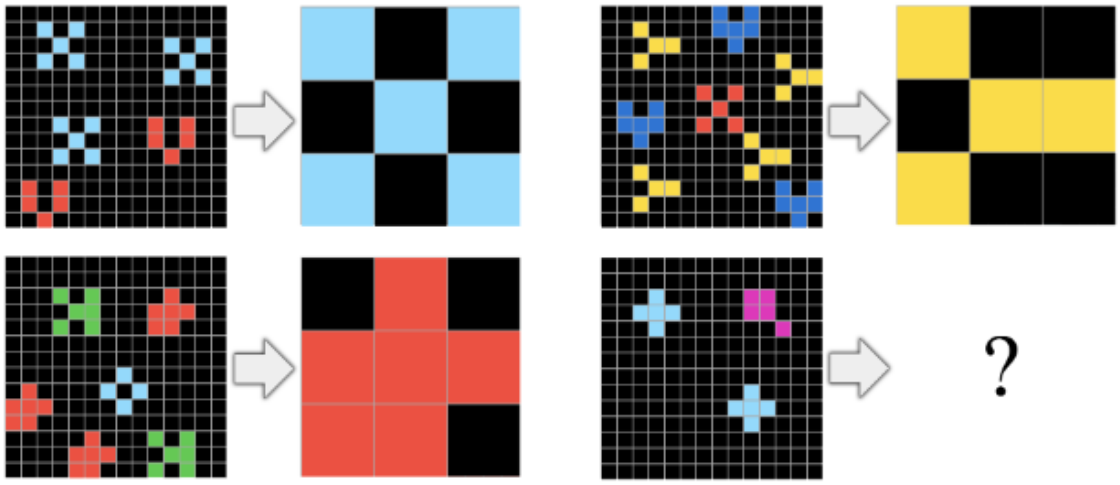
1.1 Probléma bemutatása

Egy feladat során adott egy bizonyos mennyiségű (általában három) demonstrációs példa. Egy példa a bemenetből és a hozzá tartozó kimentből áll, ahol a kimenet előáll a bemenet valamilyen komplex transzformációjából. Mind a bement, mind a kimenet egy 1x1 és 30x30 közötti négyzetrácsból áll, ahol az egyes pozíciókban egy színként vizualizálható szimbólum található. Összesen tíz féle szín lehetséges. A feladatot megoldó algoritmusnak a demonstrációs példákon végzett transzformációt kell felismernie, majd azt végrehajtania az egy (néhány esetben ennél több) teszt példán. Az algoritmus csak a válasz sikerességéről kap visszajelzést, arról, hogy hol hibázott nem. Egy feladat megoldására három próbálkozás lehetséges. Egy tipikus feladat látható a 1.1 ábrán.

A megoldandó feladatok igen változatosak, csoportosításuk nehéz, de a megoldáshoz szükséges előismeretek (a priori-k) a problémát bemutató cikkben részletesen rendelkezésre állnak, ugyanis a feladatok úgy vannak megtervezve, hogy a megoldásukhoz csak a „Core Knowledge" előismeretek legyenek szükségesek, amelyek egy fejlődés pszichológiai elmélet szerint az emberi, és az állati kognitív képességek alapjai [4]. A legfontosabb ilyen előismeretek:

- Objektum a priori: a négyzetrácson található színek objektummá csoportosítása szín, vagy pozíciófolytonosság alapján, az objektumok találkozásának és átfedésének felismerése.
- Szám, és számolás a priori: számos ARC feladat számolást, és a számok összehasonlítását igényli. Ilyenek pl. a legnagyobb/legkisebb objektum felfedezése, a legtöbbet előforduló objektum megtalálása stb.
- Alapvető geometriai és topológiai ismeretek: szimmetriák, forgatások felismerése, azonos alakú, de különböző méretű objektumok megtalálása stb.

A probléma egy 400 feladatot tartalmazó tanítási halmazból, és egy 600 feladatot tartalmazó tesztelési halmazból áll. A tesztelési halmaznak csak a 2/3-a publikus, a 200 privát feladat, amelyhez az algoritmus fejlesztőjének nincs hozzáférése, arra hivatott, hogy a módszer tényleges hatékonyságát mérje, és ne azt, hogy esetlegesen a fejlesztő milyen feladatspecifikus „kiskapukat” épített be a rendszerébe.



1.1. ábra. Egy ARC feladat felépítése [3]. A kimeneten a legtöbbet előforduló objektum szerepel.

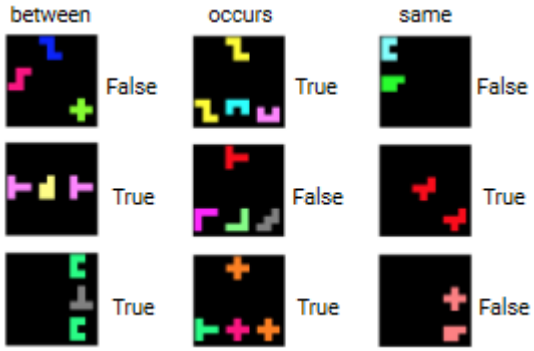
2 IRODALOMKUTATÁS

Az irodalomkutatás során először a hasonló problémákra történt megoldásokat mutatom be, majd az ARC megoldására tett kísérletekről lesz szó.

2.1 Megoldások hasonló problémákra

2.1.1 Absztrakt relációk felfedezése mély tanulással

2019-ben lett publikálva a transzformer alapú PrediNet [5] architektúra, amely ARC-hoz hasonló objektumok, és köztük fennálló relációk felfedezésére lett megalkotva. A tanulmányban felvázolt rendszer bemenete egy egyszerű objektumokat tartalmazó kép, a rendszer kimenete pedig egyetlen érték, amely egy adott reláció jelenlétét indikálja (2.1 ábra). A PrediNet ebben a rendszerben egy komponens, a teljes felépítés ezen kívül egy konvolúciós, és egy előrecsatolt neurális hálózatból áll. A bemeneti képet a konvolúciós háló alakítja feature vektorokká, ez szolgál bemenet ként a PrediNet-ben, amely a relációs struktúrák kinyeréséért felelős. Az előrecsatolt hálózat az előző komponens kimenete, és egy, a keresendő relációt indikáló bemenet alapján tér vissza egyetlen „igaz-hamis” értékkel, amely arra ad választ, hogy az adott reláció jelen van-e a rendszer bemeneteként szolgáló képen. A komponensek mély tanulással vannak betanítva.



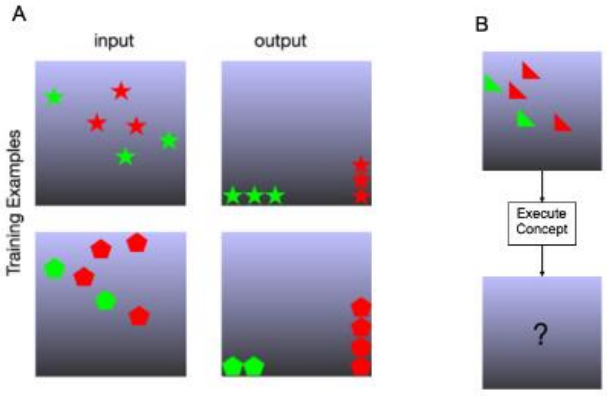
2.1. ábra. Három feladat a „Relations Game” [5] adathalmazból. Az első feladatban (első oszlop) a megoldónak fel kell ismerni, hogy az objektum két másik objektum között helyezkedik-e el, a második feladatban meg kell határozni, hogy a képen belüli felső sorban lévő egyetlen objektum megtalálható-e az alsó sor objektumai között, az utolsó feladatban pedig el kell dönteni, hogy a két képen lévő objektum megegyezik-e. Az összes feladatban egy bináris értéket kell a megoldónak előállítani, ami az adott koncepció jelenlétét indikálja az adott képen. Az egyes képekhez tartozó elvárt bináris érték a képek jobb oldalán találhatóak.

A publikációban közölt architektúra kevés hibával, a legrosszabb esetben is ~95%-os pontossággal oldotta meg a fent említett feladatokat, leelőzve ezzel az összes többi - a szerzők által összehasonlított - módszert. Másik kísérletként a rendszer a 2.1 ábrán látható „between” feladattal előtanítva lett, majd ezután a konvolúciós hálózat és a PrediNet

komponensek be lettek fagyasztva, ami azt jelenti, hogy a tanulás során ezeknek a hálózatoknak a súlytényezői ezentúl nem változnak, csak az előrecsatolt komponens tanúl. Ezzel az vizsgálható, hogy a PrediNet (vagy más, a publikációban a PrediNet-tel összehasonlított módszer) milyen mértékben képes újra használható reprezentációkra szert tenni. Az előtanított PrediNet 85%-os pontossággal működik, még a többi módszer a véletlen találgatással megegyező eredményt ért csak el.

2.1.2 Absztrakt relációk felfedezése program indukcióval

Az 1.1 és a 2.2 ábrát megfigyelve jól látható az ARC és az úgynevezett „Tabletop World" probléma közötti hasonlóság. Ez az adathalmaz szintén bemenet-kimenet párokat tartalmaz, ahol a kimenet előáll a bemeneten elvégzett transzformációból. Egy feladat felépítése is azonos: adott 2-3 demonstrációs példa, és egy teszt bemenet, amin az indukált transzformációt végre kell hajtani. Ezen probléma megoldására törekszik a Vicarious AI nevű mesterséges intelligenciával foglalkozó cég 2018-as publikációja [6]. A felépítés szinte azonos: adott egy bizonyos számú demonstrációs bemenet-kimenet példa, a cél az e közötti transzformáció felfedezése és végrehajtása egy adott teszt bemenetre. A publikáció azt a megközelítést használja, miszerint a kiment előáll egy programból, amit a bemeneten hajtunk végre. Ezáltal a probléma megoldáshoz meg kell találni egy megfelelő (és lehetőleg a legrövidebb) programot, amely az összes demonstráció bemenetére lefuttatva a megfelelő kimenetet adja vissza. Ehhez a szerzők egy feladatspecifikus nyelvet („Domain Specific Language" - DSL) alkottak, amely a tanulandó program utasításkészlete ként funkcionál, valamint az ebből előállt program értelmezésére és végrehajtására szolgáló architektúrát. Az utasításkészletből generálható egy fa, amelynek gyökér eleme egy speciális „program vége" elem. Ennek leszármazottjai az utasításkészletben található összes atomi művelet, továbbá minden atomi művelet leszármazottja ismételten az összes utasításkészletben található elem. Az így előálló végtelen fa magában foglalja az összes lehetséges programot. Egy bizonyos program a fa egy adott eleme, és a gyökérelem közötti útvonal leolvasásával kapható meg.



2.2 ábra. A „Tabletop World" probléma felépítése [7]. (a keresett algoritmus: vörös objektumok kigyűjtése vertikálisan a jobb alsó sarokban, zöld objektumok kigyűjtése horizontálisan a bal alsó sarokban).

A tanulás, mint program indukció megközelítés egyrészt előnyös, hiszen az előállított program minden létező bemenetre a megfelelő kimenetet szolgáltatja, amennyiben a bemenet szintaktikája megfelel az elvártnak, így a tanulás túlmegy a lokális generalizáción. Ugyanakkor a keresendő program hosszával a keresés ideje exponenciálisan nő, ezt a [6] forrásban található cikkben számos technikával próbálták ellensúlyozni, ilyen például a fa azon ágainak a lemetszése, ahol valamilyen szemantikai hiba lép fel. Egy másik eredményes módszer egy neurális hálózat használata, amely az atomi műveletek paraméterének meghatározásáért felelős: ha minden paraméterezési lehetőséget külön instrukciónak számolunk, akkor a forrás utasításkészlete 36 műveletet tartalmaz, ez 24-re csökken amennyiben csak a ténylegesen egyedi műveleteket vesszük, a paraméterek kiválasztását pedig a neurális hálózatra bízzuk. Ez a csökkenés nagyban hozzájárul a keresési sebességhez egy az utasításkészlet méretétől exponenciálisan függő fában. A keresési időt tovább próbálta csökkenteni a szerzők két évvel későbbi munkája [7], amiben a fő hozzájárulás az, hogy az algoritmus sikerességének kiértékelése a képen található objektumok, és nem az egész kép szintjén történik. Ezzel egy olyan program megtalálását követően, ami helyesen előállítja az összes demonstrációban egy bizonyos objektumot (tehát nem az egész kimeneti képet) azt a keresési algoritmus a fa gyökérelemévé választja, azaz a további keresés ekörül a program körül fog történni.

Mindkét megközelítés az 546 feladat közül 534-et, sikeresen teljesített ([6] esetében 535). Példák sikeresen megoldott feladatokra a 2.3 ábrán láthatóak.

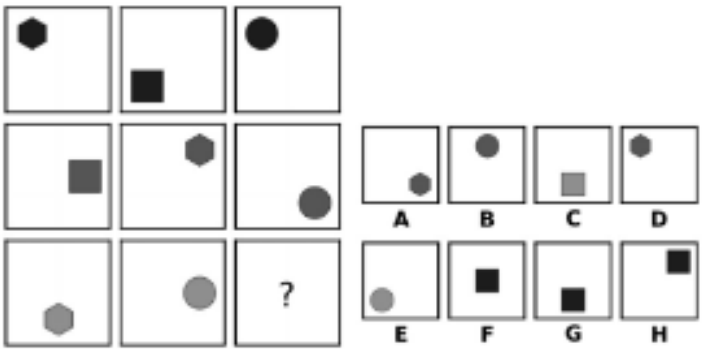
2.3 ábra. Példák sikeresen megoldott feladatokra [6]. Mindegyik feladatban objektumokat kell a kép felső részére mozgatni. Bal felső: összes objektum mozgatása, Bal alsó: csak a vörös objektumok mozgatása, Jobb felső: négyzet alakú objektumok mozgatása, Jobb alsó: vörös objektumok mozgatása, és átszínezése

2.1.3 Absztrakt relációk felfedezése látens vektortérben

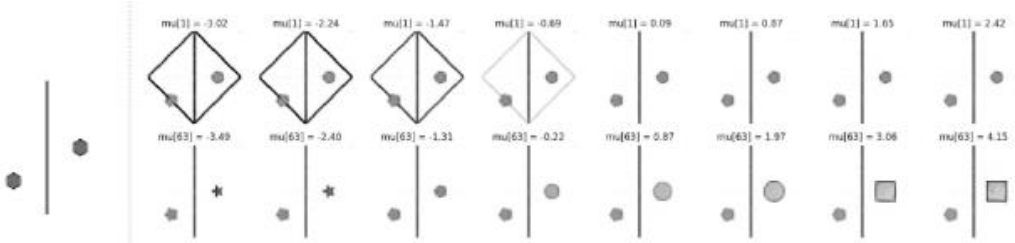
A mély tanulás alapú saját megoldásomban (10.1 melléklet) nagy szerepe van a bemeneti képek vektortérré alakításában oly módon, hogy a képek jellegzetes tulajdonságai megjelenjenek ezen vektortérben. Ez a megközelítés lett felhasználva a szintén absztrakt relációk felfedezéséről szóló Procedurally Generated Matrices (PGM) dataset megoldásához is [8]. Az adathalmaz egyes elemei 3x3 képekből állnak, amelyeken egyszerű geometriai alakzatok találhatóak, valamint az egyes sorok képei között áll fent

valamilyen absztrakt reláció, például objektumok számának, alakjának vagy színének változása. A 9 képből az utolsó hiányzik, adott 8 potenciális válasz, az algoritmus feladata eldönteni, hogy a 8 kép közül melyik kerüljön a hiányzó helyre. Minden esetben csak egy jó megoldás létezik. Egy PGM feladat látható a 2.4 ábrán.

A feladat megoldásához a bemutatni kívánt forrás a már létező Wild Relation Network (WReN) architektúrát [9] vette alapul, ahol a bementi képeket egy kisméretű konvolúciós neurális hálózat alakítja vektorokká. A 8 bemeneti kép vektorreprezentációja mellé hozzáfűződnek az egyes potenciális kimeneti képek vektorreprezentációi, így kapunk 8 darab 9 elemű vektorcsoportot, ahol az egyes csoportok a bemeneti képek vektorjait tartalmazzák, az adott potenciális válaszkép vektorjával megtoldva. Az így kapott vektorcsoportokon végig menve a Relational Network [10] (RN) hálózat feladata, hogy megmondja, az adott potenciális kimeneti kép, mennyire illik hozzá a bemeneti képekhez. Az RN hálózat egy többrétegű előre csatolt neurális hálózat, amelynek kimenete egyetlen sigmoid neuron, amely a potenciális kimenet és a bemeneti kontextus egymáshoz illését indikálja (0-nincs kapcsolat, 1-tökéletes kapcsolat). A 8 vektorcsoporton végig menve az algoritmus azt a kimenetet választja, amelyhez az RN hálózat a legmagasabb értéket tulajdonította. A program 62.6%-os pontossággal választotta a megfelelő kimeneti képet, a tanító minták közül, azonban ez a szám szignifikánsan lecsökkent a tesztelési minták során. Ezt a problémát próbálta megoldani az általam bemutatni kívánt forrás, amely a képek vektortérré alakításáért felelős konvolúciós neurális hálózat helyet egy variációs Autoencoder-t (β-VAE) használt. Ezen architektúra képes kinyerni a bemeneti képek tipikus jellemzőit (mint az alakzat, szín, pozíció) egy látens vektortérbe. Ezen Autoencoder szolgáltatja a vektorcsoportokat az RN hálózat számára. Az így betanított architektúra jobban teljesített az eredetinél, mind a tanító mintákon, mind a tesztelési mintákon, igaz ez sok esetben csak pár százalékos javulásban nyilvánult meg.
A forrás releváns része az Autoencoder által formált látens vektortér, amelyben fellelhetőek a bemeneti képek tulajdonságai (2.5 ábra). Ezt a viselkedést szeretném elérni a saját megoldás során is.



2.4 ábra. A PGM adathalmaz egy eleme (helyes válasz: C) [12]



2.5 *ábra. β-VAE látens vektortere [12]. Az egyes neuronok aktivációját variálva változik a bal oldali kép rekonstrukciója. Az egyes változatok az adathalmazban megtalálható jellemzőket (négyzet, csillag, a hátteret reprezentáló négyszög) tartalmazzák.*

2.1.4 Összefoglalás

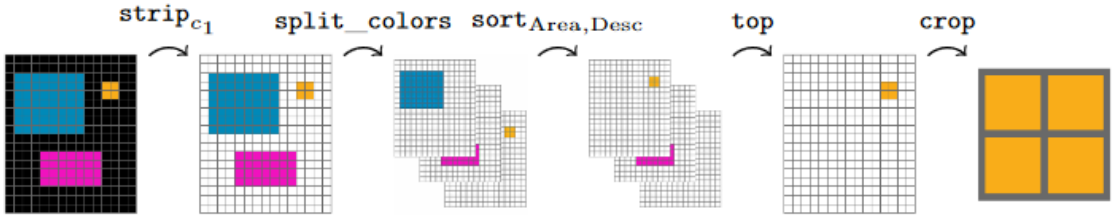
Az ARC-hoz hasonló fajtájú problémák esetén két megközelítés létezik. A 2.1.2 fejezethez hasonló program indukció, amelynek előnye, hogy ha előállítjuk a DSL-ből a célprogramot, az utána minden típusú bemenetre a megfelelő eredményt állít elő, tehát a helyes program megtalálása után bármilyen bemenetre garantált a generalizáció, amíg ez a neurális hálózatoknál nem jelenthető ki. Másik előny, hogy nem kell tanulni, ami a kis számú demonstrációs példáknál igen hasznos. Fő hátránya, hogy nem skálázódik jól komplexebb feladatokra, hiszen a bonyolultabb feladat, általában hosszabb programot, és nagyobb méretű DSL-t eredményez. Mivel a keresési stratégia exponenciálisan nő a megtalálandó program hosszával és a DSL méretével, sokszor nem lehet a gyakorlatban megtalálni a helyes célprogramot mivel a keresési tér túl nagy. Ez a probléma nem jelenik meg a mély tanulás alapú módszereknél, hiszen a kimenet előállítását teljes mértékben a hálózat belső paramétereire bízzuk. Hátránya, hogy ez csak sok tanító minta esetén hatékony, és akkor is csak úgy, ha a teszt példa kellőképpen hasonlít azokra, amikre a rendszer tanítva volt. Az ARC pont a neurális hálózatok hiányosságait kiemelő feladat, így sokszor előfordul, hogy vizuálisan semmi hasonlóság nincs a teszt bemenet és a demonstrációs példák bemenetei közt. Összefoglalva, ha program indukciót szeretnék használni, akkor találni kell egy hatékonyabb keresési stratégiát a lehetséges programok felfedezésére, vagy a DSL-t úgy kell megtervezni, hogy a lehető legkisebb utasításkészlettel rendelkezzen, és az ebből előállított programok a lehető legrövidebbek legyenek. Mély tanulás esetén valahogy úgy kell átfogalmazni a problémát, hogy több tanító minta álljon rendelkezésre, anélkül, hogy ténylegesen plusz adatokat generálnék, vagy előtanítanám a rendszert hasonló problémákon, mivel ez az ARC esetében nem megengedett.

2.2 Létező megoldások

2.2.1 Program indukció evolúciós algoritmussal

Fischer et al. [11] azt a feltételezést követte, miszerint a bemenet és kimenet közötti kapcsolat egyszerű képtranszformációkból áll elő. Ezeket felhasználva, az előzőleg bemutatott munkához hasonlóan egy keresési fa generálható, amiben a szerzők genetikus algoritmus segítségével végeztek keresést. Az előállított képtranszformáció szekvenciát alkalmazva a bemenetre áll elő a helyes megoldás (2.6 ábra).

A 400 tanító minta közül, a megközelítéssel 7.68(±0.61)% lett helyesen megoldva, míg a 100 privát tesztelési mintára az esetek 3%-ban adott az algoritmus helyes megoldást. Az elsőre rossznak tűnő eredmény a 2020-ban megrendezett, 900 résztvevős Kaggle versenyen[1] a legjobb 30 közé tartozott, jól mutatva ezzel a probléma nehézségét (verseny legjobb eredménye 20% volt). A szerzők továbbá letesztelték a keresési fában való véletlenszerű keresést is. Ezzel a tanító minták 6.17(±0.13)% lett sikeresen megoldva, a tesztelési minták közül pedig az algoritmus által nyújtott egyik válasz sem volt megfelelő.



2.6 ábra. Egy ARC feladat megoldásának menete képtranszformációkkal [11]. A képről a legkisebb méretű színfolytonos képrészt kell kivágni. Ezt a feladatot a következő elemi műveletekkel oldotta meg az algoritmus: fekete pixelek eltávolítása, színfolytonos képrészek szétválasztása, az egyes képrészletek méret szerint való rendezése csökkenő sorrendben, a rendezés első elemének kiválasztása, a kimenet ez az elem méretre vágva.

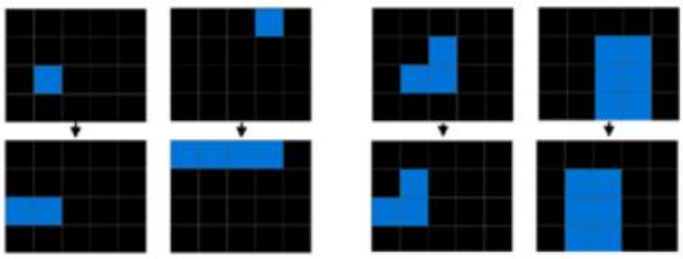
2.2.2 Program indukció Dreamcoder-el

Egy másik ugyancsak program indukció alapú megközelítés a 2020-ban publikált Dreamcoder architektúra [12] segítségével próbálta megoldani az ARC feladatokat. A Dreamcoder bemenet-kimenet között képes program modulokat tanulni. Az így létrejött modulokat a későbbi keresés folyamán újra felhasználhatja, így nem kell azt újból felfedeznie. A Dreamcoder másik tulajdonsága a „tömörítés”, amely során a már sikeresen felfedezett programok közt keres hasonlóságot, és ezekből a hasonló programokból magasabb szintű, absztraktabb modulokat állít elő. Erre az eljárásra látható példa az 2.7 ábrán. Az első feladatban egy egyenest kell balra rajzolnia a megoldónak az adott kezdőpontból, a második feladatban pedig egy adott objektumot kell a négyzetrács bal oldala felé csúsztatnia. A feladatok „draw_line_down”, valamint egy „move_down”

[1] https://www.kaggle.com/c/abstraction-and-reasoning-challenge (utoljára megtekintve: 2023.05.02)

előre betáplált függvény segítségével kerülnek megoldásra, majd a „tömörítés” folyamán az algoritmus az indukált programokat egy magasabb szintű modulba foglalja össze, amely egy tetszőleges műveletet képes a baloldalra végrehajtani.

Banburski et al. [13] 36 előre kiválasztott feladatot próbált megoldani, 5 kezdeti modullal. Minden iteráció során az architektúra mind a 36 feladat megoldására keres helyes programokat, majd a találtakon tömörítést hajt végre, és az ebből kapott modulokkal kiegészítve folytatódik tovább a keresés a következő iterációban. Az architektúra 22 feladat megoldására volt képes 4 iteráció alatt, az ezt követő iterációkban újabb megoldás nem született.



```
(lambda (rotate_cw (draw_line_down (rotate_ccw $0)))) // draws line left
(lambda (rotate_cw (move_down (rotate_ccw $0)))) // moves object left
(lambda (lambda (rotate_cw ($0 (rotate_ccw $1))))) // applies action left
```

2.7 ábra. Tömörítés folyamata [13]. Az első kódsor a jobb oldali feladathoz, a második a bal oldali feladathoz tartozó program. A tömörítés folyamán ezekből egy magasabb szintű művelet alakítható ki, amely egy transzformációt hajt végre az objektum balra forgatása után. rotate_cw: objektum elforgatása jobbra, rotate_ccw: objektum elforgatása balra

Szintén a Dreamcoder architektúrát alkalmazta a Acquaviva et al. [14], amely egy olyan adathalmazt készített, ami az ARC egyes feladataiban megtalálandó program természetes nyelvi leírását tartalmazza. Ezt a nyelvi segítséget a bemenet-kimenet párok mellett a programfa bejárását segítő neurális hálózat szintén megkapta bemenetként. Az architektúra természetes nyelvi leírással a publikus teszthalmazon 22 feladatot oldott meg, anélkül pedig 18-at.

2.2.3 Program indukció brute-force kereséssel

Az eddigi legeredményesebb megoldás [15] a tesztelési feladatok 20% teljesítette sikeresen. A brute-force megközelítés egy kb. 7000 sornyi, kézzel írt képtranszformációk kombinációjával próbált helyes megoldásokat előállítani. A szerző szerint a többi megoldáshoz képest a számottevően több és gyorsabb képtranszformáció okozta a megközelítés sikerességét.

Egy ehhez hasonló megoldás a VIMRL [16] rendszer, amely szintén brute-force módon járta be a DSL által lefedett programteret. Az előző megoldáshoz képest az itt használt műveletkészlet egy része „magas szintű”, amelyek valamilyen elemi kép vagy

matematikai művelet helyett, egy, az ARC feladatok megoldása szempontjából releváns algoritmust is használ. A keresési tér nagysága itt is problémát okozott, ezért a maximális programhossz megadása mellett egy sztochasztikus keresést is bevezettek. Ehhez előállítottak egy 160 feladatot megoldó programok adathalmazát (részben manuálisan, részben a brute-force módszerrel) és az ebből kinyert gyakoriság alapján történt a programfa bejárása sztochasztikus módon (minél több megoldás született a programfa egy adott ágából, annál nagyobb valószínűséggel választja a keresés azt az ágat). Ez a variáció nem lett sikeresebb, mint az eredeti brute-force megközelítés, amelynek legjobb futása során 104 feladatot oldott meg a tanító halmazon, és 26-ot a publikus teszt halmazon.

2.2.4 Program indukció absztrakt gráf térben

Xu et al. [17] azt a feltételezést követte miszerint az ARC feladatok könnyebben megoldhatóak, ha az egyes képek helyett az ezekből kinyerhető objektumok közötti relációk alapján történik az indukció. A képekből való objektumok kinyerését számos más megoldási kísérlet is használja. Ez az „objektum-centrikus” szemlélet előnyös, mert az egyes pixelek helyett az objektumok által lefedett pixel területek manipulálhatóak, tehát kevesebb művelettel le lehet írni egy transzformációt, a program hosszának csökkenésével pedig a kereső algoritmus is nagyobb eséllyel meg tudja találni az ilyen rövidebb programokat. Az itt bemutatott megoldás a gráffá alakítás után ehhez az adatstruktúrához készített DSL-ben keresi a feladatot megoldó programot, az eddigi megoldásokhoz hasonlóan. A módszer a tanító halmazon 57 feladatot oldott meg.

2.2.5 Program indukció kétirányú kereséssel

Az eddigi megközelítések mindegyike egyirányú, azaz a rendszer a bementből próbálja előállítani az elvárt kimenetet. [11] esetében nincs, [13] esetében pedig csak a keresést segítő neurális hálózatnak van információja a kimenetről. Ez nem hatékony, mivel így olyan programok is ki lesznek próbálva, amelyekre semmi jel nem utal a kimenet alapján. Ezt próbálja javítani a [18] forrás, amelyben a szerzők célja az, hogy a rendszerük az emberi gondolkodáshoz hasonló módon oldja meg az ARC feladatokat. Ez a szerzők szerint úgy történik, hogy a bemeneti és a kimeneti kép együttes vizsgálatával áll elő a helyes transzformáció, több, egymásra épülő lépésben. A cél, hogy a kimeneti kép objektumai „kötésben” legyenek a bemeneti kép valamilyen tulajdonságával, azaz létezzen egy olyan művelet szekvencia, ami a bemenet tulajdonságaiból állítja elő a kimenet aktuálisan vizsgált objektumát. Ez az összeköttetési folyamat kétirányú, ezért az ehhez készített DSL-ben invertálható műveletek találhatóak. Ezt a folyamatot egy megerősítéses tanulás problémaként formalizálták: minden lépésnél az ágens feladata, hogy eldöntse a DSL melyik utasítását, milyen paraméterekre (részeredményekre), és milyen irányban (bemenet-kimenet, vagy kimenet-bemenet) alkalmazza. Az ágens egy neurális hálózat, ami helytelen program előállítása esetén negatív, helyes program esetén

pozitív jutalmat kap. A Dreamcoder-t felhasználó irodalomhoz hasonlóan a megközelítés itt is csak 18 előre kiválasztott szimmetriát használó feladaton lett letesztelve, ebből 14-et sikeresen teljesített.

2.2.6 Létező megoldások összehasonlítása

Az eddigi megoldások eredményeit a 2-1 táblázat tartalmazza. Sok publikáció csak az egyik féle adathalmazon lett kiértékelve, így ezek összehasonlítása nehézkes, de kijelenthető, hogy bármilyen eredményt csak program indukció alapó megközelítéssel lehetett elérni. A keresést egyes források heurisztikával próbálták segíteni, de látható, hogy pont a publikus adatokon a két legjobb eredményt elérő forrás nem használt ilyet. Ez a két megoldás számottevően nagyobb műveletkészlettel rendelkezik, ebből feltételezhető, hogy a teljesítmény fő korlátja nem a keresési stratégia, hanem a DSL kifejező képessége. A 2-1 táblázat alapján kijelenthető, hogy még nem született olyan megoldás, amely meg tudná közelíteni az emberi teljesítményt, ami a [14] tanulmányban mértek alapján átlagosan 80%.

2.2.7 Összefoglalás

A létező megoldások mindegyike program indukció alapú megközelítés, a gyakorlatban mély tanulás nem alkalmazható. A program indukciónál fellépő skálázási problémára két megoldás született: A keresési stratégia javítása, neurális hálózattal, vagy más heurisztikával, valamint a Dreamcoder esetében a szerzők automatikusan összetettebb utasításokat generálnak egyszerűbbekből, tehát a DSL-t úgy változtatják, hogy az több utasítást tartalmazzon, ezáltal növekszik a keresési tér, de ezt ellensúlyozzák azzal, hogy a keresendő program hossza így redukálva van. A keresési fát mélységében csökkentik, szélességében növelik. Bár teljesítmény nem ezt mutatja, mégis az utolsó bemutatott forrást tartom a leg előre mutatóbbnak, a két irányú keresés, a kötés koncepció, és az objektumon alapú program indukció miatt. A két irányú keresés biztosítja, hogy csak olyan programokat vizsgálunk, amelyek a kimenet alapján indokoltak. A kötés koncepció biztosítja, hogy az adott képekből előállított program álltalánosan működni fog, mivel nem a kimenet adott tulajdonságait memorizálja, hanem a kimeneti kép minden tulajdonságát köti a bemenethez valamilyen módon. Így, ha a bemenet változik, garantált a helyes kimenet is. A teljes képes transzformáció helyett használt objektum alapú indukció felosztja az egy nagy, bonyolult problémát, több kisebb, egyszerűbbre.

2-1 táblázat. Létező megoldások összehasonlítása

Algoritmus	Tanító halmaz	Publikus teszt halmaz	Privát teszt halmaz
EA [11]	32	-	3
Kétirányú keresés [15]	14	-	-
Dreamcoder [13]	22	-	-
Brute Force [15]	129	-	21
LARC [14]	-	18	-
MDLP [19]	29	6	0
ARGA [17]	57	-	-
VIMRL [16]	104	26	-

3 SAJÁT MEGOLDÁS

3.1 Mély tanulás alapú megoldás

Az ARC probléma nehézsége, az extrém alacsony tanítóminták száma. Egy körben csupán három demonstrációs példa áll rendelkezésre, amelyek nem elegendőek egy neurális hálózat korrekt betanítására (oly módon, hogy az a teszt mintára is generalizálódjon). Ezen probléma miatt, az eddigi megoldások neurális hálózatok helyett, valamilyen keresőalgoritmus segítségével operáltak. Érdemes lehet a problémát átfogalmazni oly módon, hogy a tanítóminták száma ne csupán három legyen, hanem az összes minta, összes demonstrációs példája egy nagy feladat megoldására nyújtson tanító adatokat. Az eddigiekben az egyes minták demonstrációs példáik között kellett valamilyen transzformációt találni, amelyet a teszt példára alkalmazva előáll az elvárt kimenetet. A megoldandó probléma most legyen a 2.3.1 pontban bemutatott forráshoz hasonlóan a reprezentáció tanulás: adott az összes tanító minta (X), összes demonstrációs példáiban lévő bemenet (x_i^{in}) és kimenet (x_i^{out}) képei. Ezeket a képeket, valamilyen Encoder a sokdimenziós vektortér egy h pontjába helyezi:

$$h_i = Enc(x_i) \tag{1}$$

Az Encoderhez tartozik egy Decoder, amely ugyanezt a folyamatot végzi el, csak fordítva, a vektortér egy pontját egy képpé alakítja:

$$x_i = Dec(h_i) \tag{2}$$

Encoder segítségével az egyes bemeneti és kimeneti képeket reprezentálom a vektortér egy pontja ként. A reprezentáció tanulás akkor sikeres, ha az egy bemenet-kimenet párokon értelmezett v_i vektor mindhárom demonstrációs példában megegyezik:

$$v_i = Enc(x_i^{out}) - Enc\left(x_i^{in}\right) \tag{3}$$

$$v_{task} \approx v_0 \approx v_1 \approx v_2 \tag{4}$$

ahol v_0, v_1, v_2 az egyes bemenet-kimenet párokból előállított vektorok egy demonstrációs példában.

Ha a reprezentáció tanulás sikeres, akkor ezt a vektort hozzáadva a teszt bemenethez tartozó ponthoz, és ezt képpé visszatranszformálva előáll az elvárt kimeneti kép:

$$x_{test}^{out} \approx Dec\left(h_{test}^{in} + v_{task}\right) \tag{5}$$

ahol v_{task} az adott demonstrációs példák közötti transzformációs vektor.

Ez a megközelítés azért hasznos mert a sok, extrém kis tanító mintájú problémát egy, nagyobb problémává alakítottam, ahol már remélhetőleg elegendő minta áll rendelkezésre egy neurális hálózat betanításához. Ha a reprezentáció tanulás sikeres, akkor egy új feladat során további tanulás nem szükséges, csupán a demonstrációs példákat kell vektortérbe ágyazni, majd az így kapott pontokból kell meghatározni a

transzformációs vektort, ezt hozzáadni a tesztpéldához, majd a Decoder segítségével ezt képpé konvertálva előáll az elvárt kimenet. Ezen megközelítés sikerességének kulcsa, hogy a beágyazott vektortérnek a 2.1.3 pontban bemutatott forráshoz hasonló tulajdonságai legyenek.

A probléma átfogalmazása sem nyújtott elegendő adatot a neurális hálózatok megfelelő betanítására. Amíg a tanító halmazon, a rendszer képes volt 25%-os pontosságot elérni, ami jóval a véletlen találgatás értéke felett van (ez az összes lehetséges 1x1 és 30x30 közötti képek számosságából adódóan egy nullához nagyon közeli szám), addig a teszt halmazon egy esetben sem nyújtott megfelelő kimenetet. Továbbá megfigyelhető, hogy a rendszer a teszt halmazon a tanító halmazhoz tartozó kimenetekhez hasonló eredményeket ad, tehát a rendszer csupán bememorizálta a tanító halmazt. A módszer részletei, és az eredmények a 1. mellékletben találhatóak. Az ARC lényege, hogy kevés adatból is lehessen hatékonyan tanulni, így az olyan megoldásokat, amelyek pluszban generáltak további tanító példákat ([20]) az irodalom nem tekinti elfogadottnak. Mivel nem lehet több adatot generálni, és az itt bemutatott betanítási mód sem szolgált elég tanító példával, ezért a továbbiakban a mély tanulás alapú megközelítés helyett program indukcióval próbálom megoldani a problémát.

3.2 Program Indukció alapú megoldás

A létező program indukciós megoldások hátránya, hogy a használt DSL által meghatározott programfa, amelyben a keresés történik, exponenciálisan nő a program hosszával, így a keresési tér túl nagy a megfelelő program megtaláláshoz. Amig az eddigi megoldások a programfában való keresés nehézségét valamilyen heurisztika bevezetésével próbálták orvosolni, a saját megoldásomban a programfában való keresés elhagyását kísérlem meg. Ehhez az szükséges, hogy az indukált program ne procedurális legyen, azaz a program nem utasítások egymásra épülő sorozatából álljon. A saját megoldásomban előállított programok egymástól független, elméletileg akár párhuzamosan is végrehajtható utasításokból épülnek fel. Az ilyen programok indukciója jóval egyszerűbb, mivel az egymásra épülés felbontásával a keresési idő a program hosszával csupán lineárisan nő. Ezen elképzelés szerint tervezem meg az általam használt terület specifikus nyelvet, és az indukciós algoritmust.

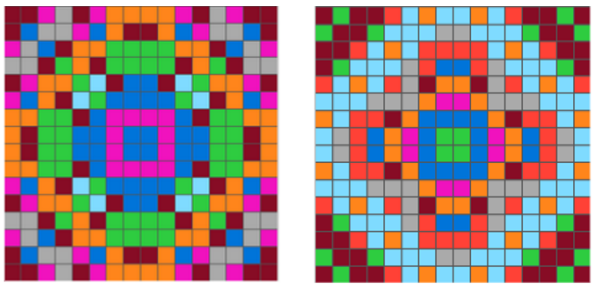
3.2.1 A terület specifikus nyelv felépítése

Számos létező megoldáshoz hasonlóan objektum-centrikus módon próbálom az ARC feladatokat értelmezni. Eszerint az egyes képek egy háttérből, és egy vagy több egymástól független objektumból állnak. A képek ilyen módú értelmezését az előfeldolgozás modul fogja végezni. Így az indukciós algoritmusnak nem az adott kép pixeleivel, hanem az adott képből kinyert objektumaival kell dolgozni. Ehhez a DSL-nek tartalmaznia kell

valamilyen objektumok leképezésére szolgáló leírást. Egy objektumot a DSL az alábbi hét tulajdonsággal reprezentál:

- Shape: az objektum színfüggetlen alakja, a gyakorlatban ez egy kétdimenziós mátrix, ahol a nullánál nagyobb értékek az alak egy régióját jelölik. Ahhoz, hogy az alak vizuálisan megjeleníthető legyen, az adott régiókhoz valamilyen színezést kell hozzárendelni. Ezt a feladatot látja el a C: *régió* → *szín* leképezés. Ennek segítségével az alak színfüggetlen marad, és egy alakból vizuálisan több eltérő objektum állítható elő különböző C színezésekkel.
- X: az objektum függőleges koordinátája.
- Y: az objektum függőleges koordinátája.
- Width: az objektum alakjának szélessége.
- Height: az objektum alakjának magassága.
- C: az objektum színtérképe.
- N: aktuális objektumok zaj objektumai.

Egyes ARC feladatok megkövetelik a végtelen kiterjedésű, valamilyen mintázatot követő alakok létezését is (3.1 ábra). Ezek nem véges kiterjedése miatt van szükség az objektum szélességének és magasságának megadására.



3.1 ábra. Mintázatot követő, periodikus objektumok

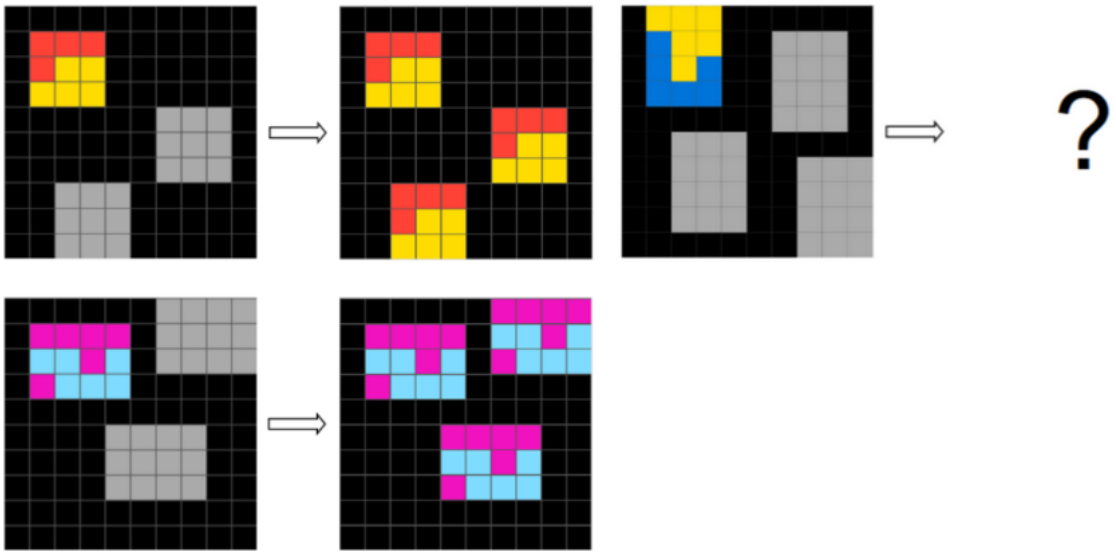
Ideális esetben az első hat tulajdonsággal leírhatóak az egyes objektumok, azonban néhány ARC feladatnál előfordulhat a részleges elfedés, azaz amikor az egyik objektum valamekkora részben kitakarja a másikat. Sok feladat megoldásához szükség van arra az információra, hogy adott objektum mely más objektumokat takar ki részlegesen. N halmaz ezt adja meg. Azzal az esettel, hogy egy adott objektum teljesen kitakar egy másikat nem kell számolni. Amennyiben az adott objektumot nem fedi el semmi, akkor N az üres halmazzal egyenlő.

A DSL műveletei a hét leíró tulajdonság valamelyikét állítják elő. Egy művelet felépítése a következő:

művelet_név [arg0] [arg1],

ahol a második argumentumnak csak néhány műveletben van szerepe. A megoldás során az alábbi műveleteket használom:

- **x_of [*obj*]:** visszatér a paraméterül kapott objektum X koordinátájával,
- **y_of [*obj*]:** visszatér a paraméterül kapott objektum Y koordinátájával,
- **width_of [*obj*]:** visszatér a paraméterül kapott objektum szélességével,
- **height_of [*obj*]:** visszatér a paraméterül kapott objektum magasságával,
- **shape_of [*obj*]:** visszatér a paraméterül kapott objektum alakjával,
- **color_of [*obj*] [*n*]:** visszatér a paraméterül kapott objektum *n*-ik régiójának színével,
- **color_map_of [*obj*]:** visszatér a paraméterül kapott objektum teljes színtérképével,
- **noise_of [*obj*]:** visszatér a paraméterül kapott objektum összes zaj objektumával,
- **nth_noise_of [*obj*] [*n*]:** visszatér a paraméterül kapott objektum *n*-ik zaj objektumával,
- **const_shape [*shape*]:** identitás művelet, visszatér a paraméterül kapott *shape* DSL típussal,
- **const_number [*num*]:** identitás művelet, visszatér a paraméterül kapott egész számmal,
- **const_color [*c*]:** identitás művelet, visszatér a paraméterül kapott színnel,
- **flatten [*shape*]:** „kilapítja” a paraméterül kapott shape adattípust, ennek eredménye szintén egy *shape*, aminek méretei és alakja egyezik a művelet paraméterével, viszont ugyanazt a régiót rendeli minden pozícióhoz,
- **dominant_color [*obj*]:** visszatér azzal a színnel, amely a paraméterül kapott objektum színtérképén a legtöbbször szerepel,
- **find_object [*predicate*]:** visszatér azzal az objektummal, amelyre az adott predikátum igazra értékelődik ki (csak egy ilyen objektum lehet, amennyiben többre igaz a predikátum a művelet hibát dob).



3.2 ábra. e76a88a6 azonosítójú feladat. A szürke objektumok helyére a színes objektum kerül.

A predikátumok egy vagy két paraméteres logikai visszatérésű függvények, felépítésükben egyeznek a műveletekkel. Egy predikátum megadja, hogy egy objektum rendelkezik-e egy bizonyos tulajdonsággal. Ennek a transzformációs program végrehajtásakor lesz szerepe. A DSL teljes predikátumkészlete a 10.2.1-es számú mellékletben található.

Az indukció során előfordulhat, hogy nem tudjuk egy objektum valamilyen tulajdonságát kifejezni a bemeneti kép alapján semmilyen művelettel. Ilyenkor is szükség van valamilyen módszerre, hogy ezek az értékek elérhetőek legyenek. Ehhez vezettem be a konstans const_shape, const_number, const_color műveleteket, amelyek olyan egyparaméteres függvények, amik visszatérési értéke maga a paraméterül kapott adat.

A DSL eddig leírt részeiből felépíthető a transzformációs program, amely a bemeneti képből kinyert objektumokat megkapva paraméterként előállítja az adott feladathoz tartozó elvárt kimenetet. A transzformációs program egy vagy több résztranszformációt tartalmaz. Egy résztranszformáció egy feltétel és egy programból épül fel, ahol a feltétel egy predikátum halmaz, a program pedig egy hét műveletet tartalmazó kifejezés, ahol az egyes műveletek az objektum egyes leíró tulajdonságait állítják elő.

A transzformációs program végrehajtása a következő lépések szerint történik: végig iterálok a paraméterül kapott bemeneti objektumok halmazán. Minden egyes objektumra kiértékelem az egyes résztranszformációk feltétel részét. Ha az adott objektumra az összes predikátum igaz, akkor az adott résztranszformáció program része végrehajtásra kerül. Ennek eredménye egy ARC objektum, amit a kimeneti objektumok halmazához hozzáadok. Ezt megismételve minden bemeneti objektumra elő áll az összes kimenti objektumot tartalmazó halmaz. Ezt a halmazt később az utófeldolgozás alakítja képpé, az előállított kimenetet összehasonlítva az elvárt kimenettel megállapítható, hogy az adott ARC feladatot a rendszer sikeresen megoldotta-e.

```
condition: is_max, is_min
shape:  shape_of [multicolor]
x:  x_of [ANCHOR]
y:  y_of [ANCHOR]
widht:  width_of [ANCHOR]
height: height_of [ANCHOR]
color:  color_map_of [multicolor]
noise:  -
```

3.3 ábra. e76a88a6 feladat transzformációs programja. A program minden objektumra igaz lesz, az előállított objektum minden tulajdonságát megörököli az aktuális bemeneti objektumtól, kivéve az alakot, és a színtérképet, ugyanis ezt az egyetlen színes objektumtól kapja meg

A 3.2 ábrán látható feladathoz indukált transzformációs programja (3.3 ábra) a következő eljárást írja le: a transzformációs program egyetlen résztranszformációból áll. Ennek

feltétele, hogy az objektumra amelyre a program potenciálisan le lesz futtatva az `is_max`, és az `is_min` predikátumok igazak legyenek. `is_max` igaz, ha a vizsgált objektum mérete maximális az adott bemeneti képen, `is_min` pedig ha minimális. Mivel a demonstárciós példák, és a teszt kép bemenetein is ugyanakkora méretű objektumok vannak, ezért ez a két predikátum minden objektum esetén igazra fog kiértékelődni. Ezért együttesen egy „mindig igaz" predikátum ként fogható fel azonos méretű objektumok esetén. A résztranszformáció program része a következő felépítésű: a létrehozandó kimeneti objektum zajmentes, függőleges és vízszintes koordinátái, valamint a szélessége és magassága egyezzen meg annak a bemeneti objektumnak tulajdonságaival, amire éppen a program végrehajtásra kerül. Az alak, és a színtérkép pedig egyezzen meg annak a bementi objektumnak az alakjával és színtérképével, amelyre a `multicolor`, predikátum igaz, azaz több mint egy színből áll. A programokban ilyen módon előforduló predikátumok egyértelműen meghatároznak egy bizonyos objektumot, azaz garantáltan csak egy bemeneti objektumra lesznek igazak.

Látható, hogy a transzformációs programok nem procedurális jellegűek. Az egyes résztranszformációk, sőt a résztranszformációk program részének egyes műveletei is teljesen függetlenek egymástól, ugyanakkor ugyanúgy képesek kifejezni olyan szabályokat és műveleteket amelyekkel az ARC feladatok megoldhatóak.

3.2.2 Ekvivalencia Szaturáció

Egy program futtatásakor a bemenő adatokon végzett műveletek sorozatával áll elő a program kimenete. Program indukció esetén ennek ellenkezője történik: adottak a bemenő adatok, valamint a program kimenete, és ebből kell visszafejteni az elvárt működésű programot. Egy ilyen bement-kimenet pár általában nem definiálja egyértelműen a keresett programot, így a mély tanuláshoz hasonlóan itt is több ilyen tanító minta áll rendelkezésre. A tanító minták ideális mennyisége feladatfüggő, de általánosan kimondható, hogy nagyságrendekkel kevesebb mint a mély tanuláshoz használt adathalmazoknál. Az ARC esetében a 2-3 demonstrációs példa a legtöbb esettben elegendő a keresett program egyértelmű meghatározására.

A legegyszerűbb program indukciós módszer az „inverz-kiértékelés" elvét követi. Adott y kimenet, x bemenet, inverz-kiértékeléskor egy halmazba kigyűjtésre kerül az összes olyan DSL-ből előállítható kifejezés sorozat, amelyet x-re lefuttatva y áll elő. A keresett program megkapható, az összes tanító mintára ilyen módon előállított halmazok metszete ként. A gyakorlatban ez természetesen nem kivitelezhető, mivel, ha valamilyen általános programozási nyelvből szeretnénk indukálni, akkor gyakorlatilag végtelen sok programot kéne kigyűjteni. A DSL-ek jelentősen kisebb utasításkészlete miatt ez a probléma kevésbé jelentős, de a már említett programfában való keresés miatt a hosszabb programok indukálása így is túl sok időt vesz igénybe.

Az ekvivalencia szaturációt („Equivalence Saturation", ES) [21] eredetileg forráskód optimalizálásra fejlesztették ki. Az eljárás bemenetként kap egy tetszőleges nyelvből

előállított programot, és az adott nyelvre igaz átírási szabályokat. Az átírási szabály egy olyan kifejezés vagy kifejezés sorozat, amelyet egy adott helyen kicserélve a programban nem változtatja meg annak működését (bármilyen bemenő adat esetén). Például egy $x = x * 2$ kifejezést a kód bármelyik pontján kicserélhető az $x = x<<1$ kifejezésre, mivel a kettővel való szorzás mindig ekvivalens a bitshift művelettel. Ekkor az $x = x * 2 \rightarrow x = x <<1$ egy átírási szabálynak tekinthető. Az átírási szabályok lehetnek matematikai jellegűek, a bit eltolás példához hasonló programozás jellegűek, vagy magasabb szintű módosítások, például egymásra nem épülő utasítások sorrendjének felcserélése. Forráskód optimalizálásnál az eljárás a kezdeti programot az átírási szabályokkal újabb és újabb ekvivalens programokká alakítja. Az eljárás egyik nagy előnye, hogy az ilyen módon előállított összes program egyetlen, közös, úgynevezett E-PEG struktúrában kerül tárolásra. Ennek tárolási módja miatt a mérete a eltárolt programok függvényében egyre kisebb mértékben nő, ez gyakran egy bizonyos szám után szaturál, azaz mérete konstans marad. A tárolás nem destruktív, azaz az összes előállított program kinyerhető az E-PEG-ből az eredeti állapotában. Forráskód optimalizálásnál a kigyűjtött programok közül valamilyen metrika után kiválasztásra kerül az optimális program.

Program indukciónál az ekvivalencia szaturáció felfogható az inverz-kiértékelés egyfajta memóriaigény szempontjából alacsony erőforrásigényű közelítéseként. Adott (x, y) tanító minta, ennek alapján kerüljön előállításra valamilyen kezdeti program. Ezen egy bizonyos ideig legyen ekvivalencia szaturációt végezve. Minél több ideig fut az eljárás annál pontosabb lesz a közelítés, mivel több idő alatt több átírás történik, azaz több ekvivalens programot fog tartalmazni az E-PEG struktúra.

A saját megoldásomban elsőként az ekvivalencia szaturációhoz hasonló módon állítottam elő a résztranszformációban lévő programokat. A proceduralitás elhagyása miatt azonban a keresési tér annyira lecsökkent, hogy lehetségessé vált, hogy a programok egyes műveleteit ES nélkül, inverz-kiértékeléssel állítsam elő.

A DSL 4 adattípust tartalmaz: alak, szín, szám, és az ezekből felépülő objektum. Az ilyen típusú értékeket előállító kifejezések:

$$ES(shape) = \{expr \in DSL_{SHAPE} \mid eval(expr) = shape\} \quad (6)$$

$$ES(color) = \{expr \in DSL_{COLOR} \mid eval(expr) = color\} \quad (7)$$

$$ES(number) = \{expr \in DSL_{NUMBER} \mid eval(expr) = number\} \quad (8)$$

$$ES(object) = \{expr \in DSL_{OBJECT} \mid eval(expr) = object\} \quad (9)$$

ahol DSL_{SHAPE}, DSL_{COLOR} stb. halmazok a DSL műveletkészletének azon részei, amelyek alak, szín stb. adattípusra értékelődnek ki. Ezek a négy adattípusra a következők:

$$DSL_{SHAPE} = \{group\_of, const\_group, flatten\} \quad (10)$$

$$DSL_{COLOR} = \{color\_of, const\_color, color\_map\_of, dominant\_color\} \quad (11)$$

$$DSL_{number} = \{x\_of, y\_of, width\_of, height\_of, const\_number\} \quad (12)$$

$$DSL_{OBJECT} = \{noise\_of, nth\_noise\_of, find\_object\} \quad (13)$$

3.2.3 Indukciós algoritmus

Az indukció az objektumok leíró reprezentációjával történik, így az egyes bemeneti és kimeneti képekből ki kell szűrni a rajta található objektumokat. Ez az előfeldolgozás feladata, amiről 3.2.4 fejezetben lesz szó. Most feltételezem, hogy adott két halmaz, X, Y, ahol X a bemeneti képen, Y a kimeneti képen található objektumokat tartalmazza.

Minden y ∈ Y kimeneti objektumhoz tartozik egy „testvér" objektum. y testvér objektuma, az az x ∈ X bementi objektum, aminek a 7 leíró jellemzői közül a legtöbb egyezik y jellemzőivel. A testvér objektumoknak az indukció későbbi részében lesz szerepük. Egy tetszőleges y testvér objektumát az $anchor(y)$ függvény adja vissza.

Az első lépésben minden x ∈ X bementi objektumhoz hozzárendelek két halmazt: $P(x)$ halmaz tartalmazza azokat a predikátumokat, amelyeket kiértékelve az adott x objektumra igazzal térnek vissza:

$$P(x) = \{\, p \in U_{predicates} \mid p(x) = 1 \,\} \quad (14)$$

$P_{unique}(x)$ halmaz tartalmazza azokat a predikátumokat, amelyek csak az adott x objektumra értékelődnek ki igazra az X objektumok közül:

$$P_{unique}(x) = \{\, p \in P(x) \mid \nexists\, x_i \in X - x \; ahol \; p(x_i) = 1 \,\} \quad (15)$$

A következő lépésben minden y ∈ Y kimeneti objektum minden leíró jellemzőire ekvivalencia szaturációt végzek. Ennek eredménye egy adott jellemzőre lefuttatva egy halmaz, amely minden olyan DSL kifejezést tartalmaz, amit kiértékelve az adott jellemző értéke állna elő. Mivel az objektum reprezentációja 7 ilyen jellemzőből áll, így ennek eredménye egy kimeneti objektumra 7 db halmaz:

$$ES(y) = (ES^S(y), ES^x(y), ES^y(y), ES^w(y), ES^h(y), ES^c(y), ES^N(y)) \quad (16)$$

Az ES által előállított kifejezések a DSL műveleteiből és objektum referenciákból épülnek fel. A következő lépésben kicserélem az objektum referenciákat az adott objektum egyedi predikátumaira. Például, ha adott egy $(rotate\ x\ 180)$ kifejezés, és $P_{unique}(x) = \{P_1, P_2, P_3\}$ akkor a behelyettesítés eredménye:

$$\{\, (rotate\ P_1\ 180), \quad (rotate\ P_2\ 180), \quad (rotate\ P_3\ 180) \,\}$$

Ezt elvégzem az összes kifejezés halmaz elemeire. Az így kapott 7 db halmazt (amelynek elemei már nem tartalmaznak objektum referenciákat) $program_{pseudo}(y)$ ként jelölöm.

Az eddigi lépéseket megismétlem az adott ARC feladat összes bemenet-kimenet párjainak összes objektumára. Az így előállított pszeudo programokat egy halmazba gyűjtöm:

$$U_{programs} = \bigcup_{\forall Y_i \in T} \{\, program_{pseudo}(y) \mid \forall y \in Y_i \,\} \tag{17}$$

A pszeudo programokon egy „metszet” és egy „üresség” művelet értelmezett. Két pszeudo program metszete, az egyes leíró tulajdonságokhoz tartozó halmazok metszete. Egy pszeudo program üres, ha bármely leíró tulajdonságához tartozó halmaz üres.

Ezek egyelőre nem ténylegesen végrehajtható programok, mivel minden egyes leíró tulajdonsághoz egy kifejezés helyett egy kifejezés halmaz adott. A következő lépésben kiválasztásra kerül minden egyes jellemzőre az az egy kifejezés, amely a tényleges program része lesz.

Ehhez felhasználom azt, hogy egy adott feladatban több demonstrációs példa van megadva, ahol különböző bemenetekre ugyanaz a keresett célprogram van végrehajtva. Ezért feltételezhető, hogy azok a kifejezések, amelyek gyakran fordulnak elő a pszeudo program egy adott leíró tulajdonságában nagy valószínűséggel a célprogram részei lesznek. Ennek megtalálásához minden kifejezéshez hozzárendelésre kerül egy számláló. Ha szeretném megtalálni y kimenti objektum programját, akkor végig kell iterálni az összes pszedo programon, és meg kell nézni, hogy y pszedo programja, és az aktuális pszeudo programnak van-e közös metszete. Ha van, inkrementálásra kerülnek azon kifejezések számláló, amelyek megtalálhatóak mindkét halmazban. Így egy y objektum $expr$ kifejezéshez tartozó számlálója felírható a következő módon:

$$count(y, expr) = |\{\, p \;\forall p \in U_{programs} \mid expr \in program_{pseudo}(p) \\ és\; program_{pseudo}(y) \cap p \neq \emptyset \,\}| \tag{18}$$

Y végleges programja azok a kifejezések, amelyeknek számlálója az adott leíró jellemzőben maximális:

$$program(y)^I = \underset{\forall expr \in program^I_{pseudo}(y)}{argmax} count(y, expr) \tag{19}$$

ahol az I a felső indexben a 7 elemű objektum leírás egyik tulajdonságát jelzi. Ha több ilyen kifejezés van, akkor véletlenszerűen kiválasztásra kerül egy. Ezt elvégzem minden kimeneti objektumra, majd csoportokba rendezem ezeket a kimeneti objektumokat az előállított program alapján.

Az indukció utolsó lépése, azon predikátumok megtalálása, amelyek eldöntik, hogy egy adott objektum melyik csoportba tartozzon. Egy adott c osztály csoportosító predikátumai úgy kapható meg, hogy veszem az összes c osztályba tartozó kimeneti objektumok testvér

objektumait, és megnézem melyek azok a predikátumok, amelyek az összes így kapott objektumokra igazak, de az összes többi bemeneti objektumokra hamisak:

$$predicates(c) = \{\, p \in U_{predicates} \mid \forall y^{+} \in c - re: p\big(anchor(y^{+})\big) = 1 \text{ és } \forall y^{-} \in C \backslash c - re: p\big(anchor(y^{-})\big) = 0 \,\} \quad (20)$$

Ezeket felhasználva, egy adott feladat megoldása a 3.1 algoritmus alapján történik.

3.1 Algoritmus Transzformáció előállítása adott ARC feladatra

Bemenet: T – ARC feladat, *i* - interpretáció
Kimenet: R – transzformáció

1. **ciklus** I_x , $I_y \in$ T
2. X bemeneti objektumok halmazának előállítása I_x képből;
3. Y kimeneti objektumok halmazának előállítása I_y képből;
4. **ciklus** x ∈ X
5. $P(x)$ meghatározása (15) alapján;
6. $P_{unique}(x)$ meghatározása (16) alapján;
7. **ciklus vége**
8. **ciklus** $y \in Y$
9. y testvér objektumainak meghatározása;
10. $program_{pseudo}(y)$ meghatározása;
11. **ciklus vége**
12. **ciklus vége**

13. **ciklus** $program_{pseudo}(y) \in Y$
14. $program(y)$ meghatározása (20) alapján;
15. **ha** ∃ c ∈ C ahol $c.program = program(y)$ akkor
16. $c.objects \leftarrow c.objects \cup \{ y \}$;
17. **különben**
18. c ← $\big(\{ y \}, program(y)\big)$
19. $C \leftarrow C \cup \{ c \}$
20. **elágazás vége**
21. **ciklus vége**

22. **ciklus** c ∈ C
23. $predicates(c)$ meghatározása (21) alapján;
24. **ciklus vége**
25. **vissza** $\big(c.program,\ predicates(c)\big)\ \forall\ c \in C$

3.2.4 Előfeldolgozás

Az előfeldolgozás során meghatározásra kerülnek az egyes képeken szereplő objektumok. Az ARC nehézsége, hogy nincs semmilyen stratégia, amivel konzisztens módon meg lehet mondani, hogy mely képrészlet tekinthető egy objektumnak. Egyes esetekben objektumok az egyszínű, folytonos képrészletek, más esetekben csak a pozíció folytonosság elegendő önmagában. Gyakran valamilyen periodikus mintázatot kell objektum ként kezelni, amit zajok fedhetnek el részlegesen. Ha a többi megoldáshoz hasonlóan csak egy módszert használnék, akkor a feladatok nagy részét már eleve olyan módon reprezentálnám, amire nem lehet helyes programot indukálni. Ezért Xu et al. [17] -hoz hasonlóan az egyes képeket több algoritmussal próbálom interpretálni. Ezeknek alapja a fenti három értelmezési szempont. Így az előfeldolgozás egy képre több objektum felbontást ad vissza. A gyakorlatban minden egyes interpretációs algoritmust lefuttatom a képekre. Ha az így kapott X, Y halmazokon az indukciós algoritmus le tud futni, akkor az adott interpretációs algoritmus fog végrehajtódni a teszt bemenet képeire is. Ha az X, Y halmazokból nem lehet értelmes programot előállítani, akkor az indukciós algoritmus „megakad” az osztályozó predikátumok keresésénél, ugyanis ekkor valamelyik, vagy az összes csoport osztályozó predikátumai az üres halmazzal egyenlőek. Ekkor a megoldó program a 3.2 algoritmus alapján állítja elő az X, Y halmazt. Amennyiben egyetlen értelmezés sem helyes, akkor az adott feladat megoldását a program átugorja.

3.2 Algoritmus ARC feladat megoldása

Bemenet: T – ARC feladat

1. **ciklus** $i \in INT$
2. R meghatározása i interpretációval az (3.1) algoritmus alapján;
3. **ha** $\nexists r \in R$ ahol $r.predicates = \emptyset$ akkor
4. teszt objektum halmazok előállítása i interpretációval;
5. R alkalmazása minden előállított halmazra a (3.3) algoritmus alapján;
6. az így kapott kimeneti objektum halmazok visszaalakítása képpé;
7. **ha** az összes előállított kép egyezik a hozzá tartozó cél képpel

a. **vissza** igaz

8. **elágazás vége**
9. **elágazás vége**
10. **ciklus vége**
11. **vissza** hamis

3.3 Algoritmus Kimeneti objektumok előállítása

Bemenet: X – bemeneti objektumok halmaza, R – indukált transzformáció
Kimenet: Y – kimeneti objektumok halmaza

1. $Y \leftarrow \emptyset$
2. **ciklus** x ∈ X
3. **ha** $\exists\, r \in R$ ahol $\forall p \in U_{predicates}$-re $p(x) = 1$ akkor
4. $Y \leftarrow Y \cup \{ r.program(x) \}$
5. **elágazás vége**
6. **ciklus vége**
7. **vissza** Y

3.2.5 Utófeldolgozás

Az indukált programot végrehajtva a teszt bemeneten azokat az objektumokat állnak elő, amelyek a kimeneti képen szerepelni fognak. Azonban ezt az objektum halmazt még képpé is kell alakítani. A kép tartalma triviális: egyszerűen az összes objektum formáját ki kell rajzolni az adott függőleges, és vízszintes koordinátákra, az adott színekkel, és ha van az adott zaj objektumokkal együtt. Ezek mellett a megoldó algoritmus feladata, hogy a teljes kép szélességét és hosszúságát is meghatározza. A kiértékelés során csak akkor számít egy kép helyesnek, ha annak a tartalma és a mérete is azonos az elvárt cél képpel. A kép méretének meghatározásához három lehetséges esetet vizsgálok:

1. Konstans méret: ha az összes demonstrációs példa kimeneti képeinek azonos a szélessége, és hosszúsága. Ekkor ezt a két konstans értéket letárolom, és az előállított képek méretét is erre állítom.

2. Konstans relatív méret: ha az összes demonstrációs példa kimeneti képeinek a mérete megkapható a hozzá tartozó bemeneti kép méretének konstans szorosából. Ekkor ebből a két értékből, és a teszt bemenet méretéből számolom ki az előállított kép méretét.

3. Legkisebb teljes kép: amennyiben az előző két eset nem teljesül, akkor a kimeneti kép méretét úgy állítom be, hogy az pont megegyezzen a legszélső objektumok pozícióival, így az összes objektum rajta lesz a képen, de üres sorok, vagy oszlopok a kép szélein nem lesznek.

4 IMPLEMENTÁCIÓ

Az implementáció (10.3 melléklet) funkcionalitást nyújtó statikus osztályokat (4.1 ábra), és az ezeket összekötő adatosztályokat (4.2 ábra) tartalmaz. Az adatosztályok valamilyen entitás adatait tárolják, esetlegesen az azokon végezhető minimális műveletekkel kiegészülve. A tényleges függvények, amelyek a megoldás egyes lépésit végzik, ezeken az adatosztályokon dolgoznak.

Az `ARCObject` a már bemutatott objektum leírást valósítja meg. Ezeken az adatokon értelmezett két művelet: a `Size()` megadja, hány pixelt tartalmaz az adott objektum, ezt azok a predikátumok használják, amelyek az objektum méretére vonatkoznak. A `Visual()` „kirajzolja” az adott objektumot egy kétdimenziós tömbként. Ezt a teszt kimeneti képek előállításánál használom.

Az `ARCProgram` az indukciós algoritmusban „program” ként hivatkozott entitást implementálja, így ez az adatosztály a 7 leíró tulajdonsághoz tartalmaz egy-egy kifejezést. A kifejezések szintén adatosztályok, ezekről később lesz szó.

A saját megoldásnál bemutatott módon, az ekvivalencia szaturáció pszeudo-programokat ad vissza, ahol egy leíró jellemzőhöz nem egyetlen, hanem az összes létező kifejezés fel van sorolva, amely az adott jellemzőt elő tudja állítani. A `SuperimposedARCProgram` reprezentálja ezeket a pszeudo-programokat, amely csupán annyiban tér el az `ARCProgram`-tól, hogy a 7 kifejezés helyett 7 db kifejezés listát tartalmaz, az előbb említett ok miatt. Továbbá az osztály tartalmazza az indukciós algoritmusnál említett metszet, és az ürességet ellenőrző műveletet.

Az indukció utolsó lépésében az objektumokat a hozzájuk indukált programok szerint csoportosítjuk. Ezeket a csoportokat az `EquivalenceClass` adatosztály implementálja, amely csupán egy programot, és egy objektum listát tartalmaz, műveletet nem végez.

A `PredicateEngine` osztály tartalmazza a predikátumokat. Ezek `bool` visszatérésű függvények, amelyek paraméter ként biztosan kapnak legalább egy `ARCObject`-et, amelyre a predikátum ki lesz értékelve. A további paraméterekben azonban az egyes predikátumok különbözhetnek. Ezen különbségek elrejtésére a Command tervezési mintához hasonlóan egy `Predicate` osztályt is használok, amely tartalmaz minden információt az adott predikátum kiértékelésével kapcsolatban, így a kód többi részében minden predikátum egységesen kezelhető. Mivel a predikátumokat a `PredicateEngine` tartalmazza, így azok kiértékelésére is itt kerül sor. Ezeket az `Execute` függvény végzi, amely egy vagy több predikátumot kap, és visszatér azzal az objektummal, amelyre az összes paraméterül kapott predikátum igaz.

A predikátum kezeléshez hasonlóan jártam el a kifejezések implementálásánál is. Az `ExpressionEngine` tartalmazza a függvényeket, amely a tényleges kifejezés műveletet elvégzik, de az ezeket között fellépő különbségek elrejtésére, és az egységes kifejezés kezelésre az `Expression` osztályt használom.

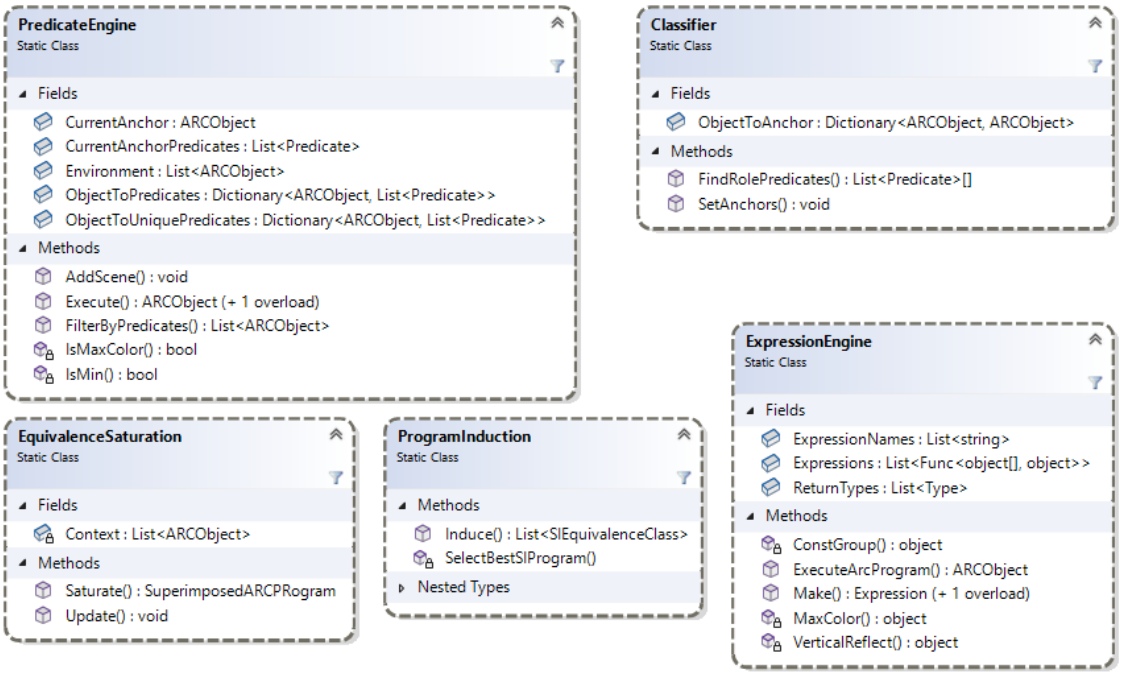
A `Preprocessing` implementálja az előfeldolgozást, azaz ez végzi a képek objektum halmazokká alakítását, és itt vannak definiálva az interpretációs metódusok is. A képek feldolgozásához sok elemi képművelet kell (elárasztásos kitöltés, kivágás stb.), ezeket kiszerveztem az `ImageProcessing` osztályba, így a `Preprocessing` csak az interpretációs módokat és az azokhoz közvetlenül kapcsolódó függvényeket tartalmazza, alacsony szintű képfeldolgozással kapcsolatos műveleteket nem.

A `ProgramInduction` feladata a pszeudo programokból a végleges programok kiválasztása. Ezt az `Induce()` metódus végzi az 3.1 algoritmus alapján. Az objektumokat az indukált program szerint csoportosítja, és ezekkel a csoportokkal tér vissza. Tehát a metódus visszatérése `EquivalenceClass` példányok lesznek.

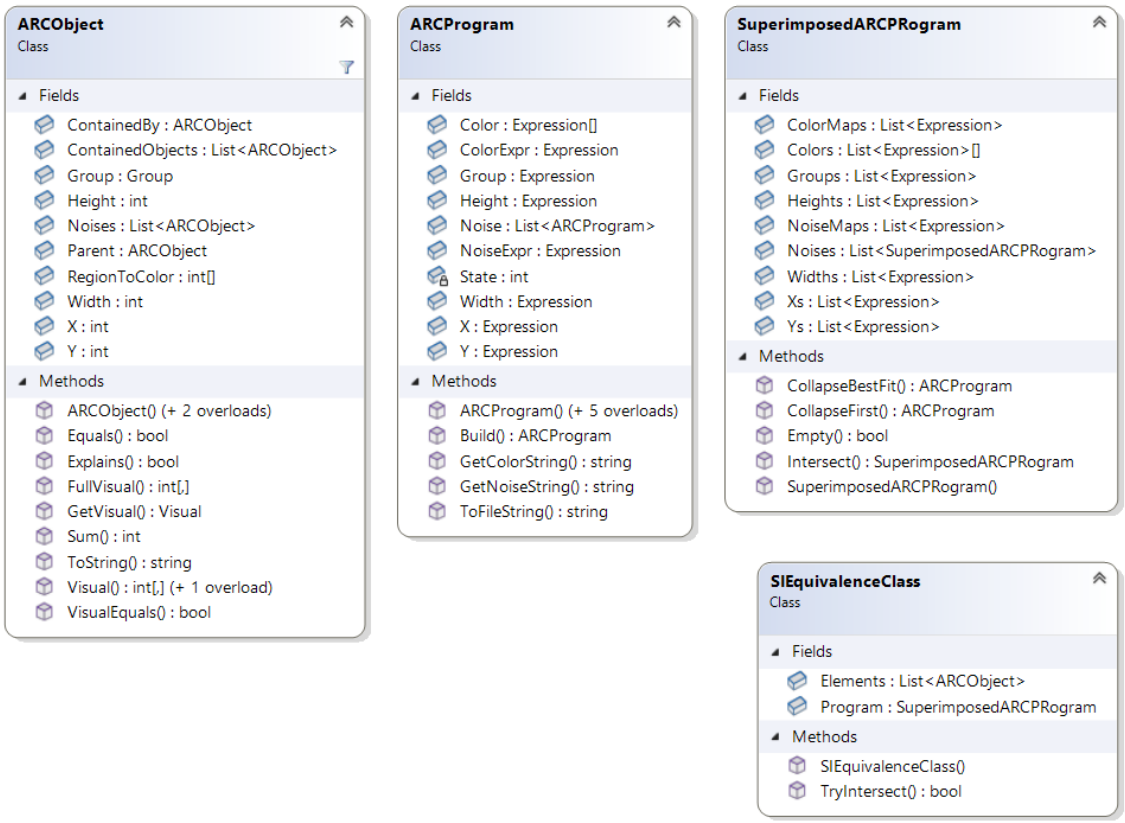
Az osztályozó predikátumokkal kapcsolatos számításokat a `Classifier` osztály végzi. Ehhez szükséges az indukciós algoritmusban említett testvér objektumok meghatározása. Ezt `a SetAchors()` végzi, és az `ObjectToAnchor` struktúrában tárolja el. Ez kívülről is elérhető az ekvivalencia szaturációs végző osztály számára. Magát az osztályozó predikátumok megtalálását a `FindRolePredicates()` metódus végzi.

Az utófeldolgozást implementáló `Postprocessing` osztály feladata a kimeneti objektum halmaz képpé alakítása, és ennek méretének meghatározása. Ez a 3.2.5 fejezetben leírtak alapján történik, a `Render` függvény pedig egy `ARCObject` listából állítja elő a teljes képet (az egyes objektumok `Visual()` függvényeinek meghívásával), majd a meghatározott vágási módszerrel a végleges méretre vágja azt.

Az eddigi osztályokat felhasználva egy adott ARC feladat megoldását a `Solver` osztály végzi 3.2 algoritmus alapján.



4.1 ábra. Számítást végző osztályok diagramja



4.2 ábra. Adatosztályok osztály diagramja

5 EREDMÉNYEK

A rendszer eredményei az 5-1 táblázatban láthatóak. A rendszer a publikus tanitóhalmaz 400 feladatai közül 41-et, a publikus teszt halmazból pedig 10 feladatot oldott meg sikeresen. Ez a többi irodalommal összehasonlítva a negyedik legjobb eredmény (5.1 ábra). Futási idő szempontjából az átlagos feladat megoldási idő 0.48, és 1.67 másodperc, az egyes adathalmazokon. Számos más megoldásnál is fel lett tüntetve ez az idő, viszont a nem egységes futási környezet miatt ezek összehasonlítása nem ad pontos képet az egyes algoritmusok tényleges sebességéről. Ennek ellenére megfigyelhető, hogy a bemutatott módszer átlagos futási ideje nagyságrendekkel kisebb a többinél (5.2 ábra).

5-1 táblázat: Az algoritmus eredménye és futási ideje

	Tanító halmaz	Publikus teszt halmaz	Privát teszt halmaz
Megoldott feladatok	41	10	0
Átlagos megoldás idő (mp)	0.48	1.67	-

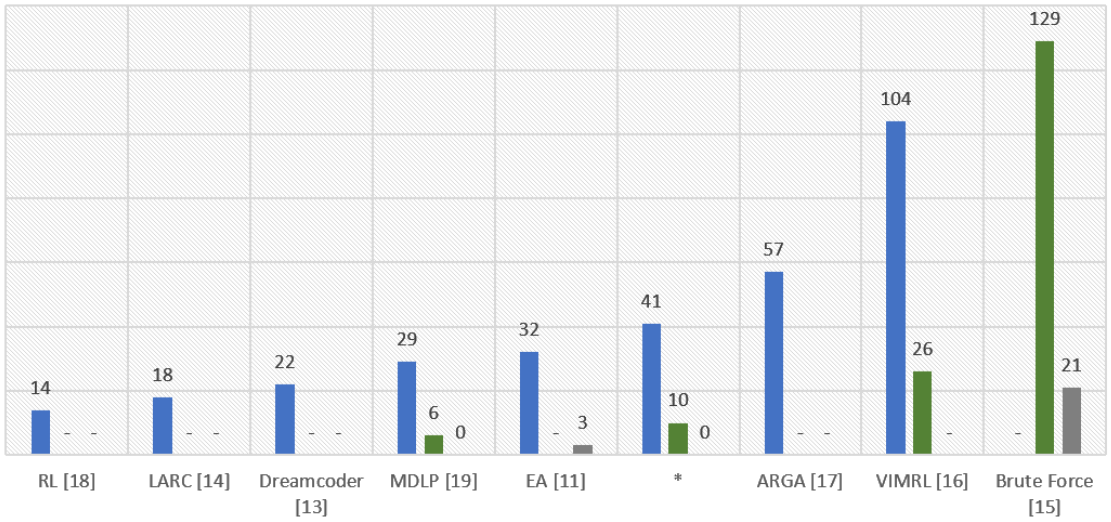
Az előfeldolgozás mindig egy adott interpretációval alakítja a képet objektum halmazzá. Az egyes ilyen interpretációk részesedése a megoldott feladatok tekintetében az 5-2 táblázatban találhatóak.

5-2 táblázat: Megoldások eloszlása az interpretációs módok között

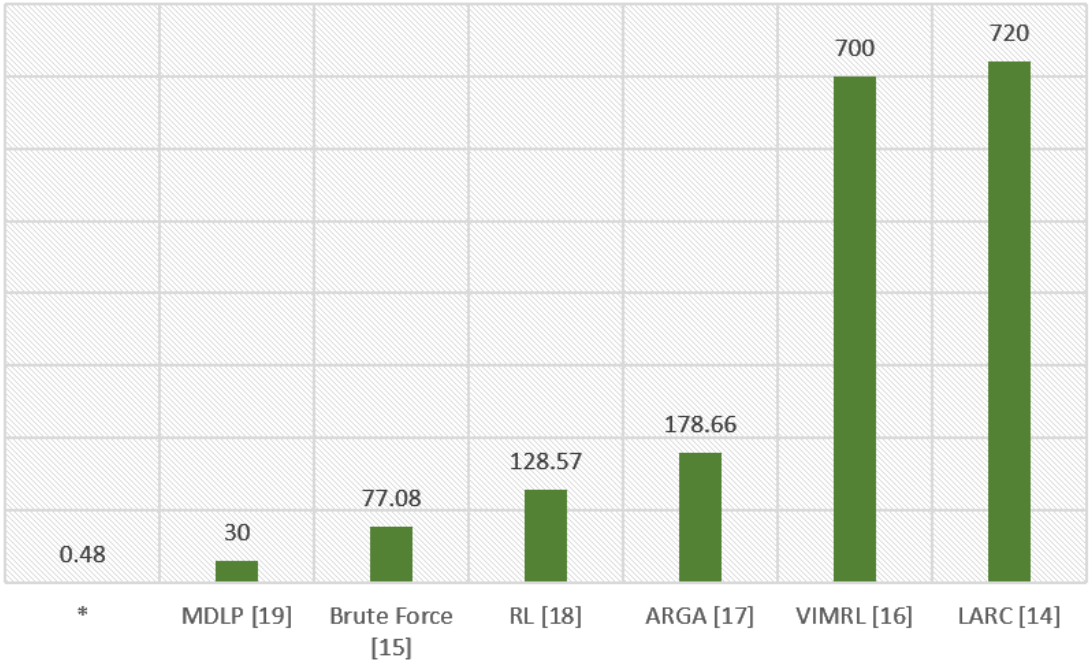
	Szín folytonos	Pozíció folytonos	Minta
Publikus tanító h.	25	11	5
Publikus teszt h.	4	5	1

A tanító halmaz 351 feladatára nem született sikeres megoldás. Ebből 52 olyan feladat volt, amelyet az előfeldolgozás nem tudott objektumokká alakítani, 118 olyan feladat volt, ahol sikeresen indukált programot az algoritmus, de azt nem tudta végrehajtani a teszt bemenetre (általában a predikátumok egynél több objektumra, vagy egyik objektumra sem voltak igazak). A maradék 189 feladatnál pedig az előállított kép nem egyezett meg az elvárttal. Ezekben az esetekben a leggyakoribb ok a „memorizálás” volt, azaz amikor a transzformációs program egy általános algoritmus helyett egyszerűen feljegyezte az összes bemenet-kimenet programokat anélkül, hogy azokat általánosította volna. Ez akkor történik, ha a DSL nem tartalmazza azt a műveletet, amelyet az adott feladat általános megoldásához kellene. Ekkor az inverz kiértékelés csak más műveletekkel tudja megmagyarázni a kimeneti objektumokat, ezek azonban nem egyeznek a demonstrációs példák között, így az indukciós algoritmus kénytelen ezeket a kevésbé általános programokat kiválasztani. A publikus teszt halmazon 78 feladat akadt

meg az előfeldolgozás során, 149 esetben a sikeresen indukált programot nem lehetett végrehajtani a teszt képre, és 163 feladatnál az előállított kép nem egyezett az elvárttal.



5.1 ábra. Egyes megoldások teljesítménye a tanító (kék), publikus teszt (zöld), és privát teszt (szürke) halmazokon. A saját megoldásom eredményeit a „” oszlop jelöli*



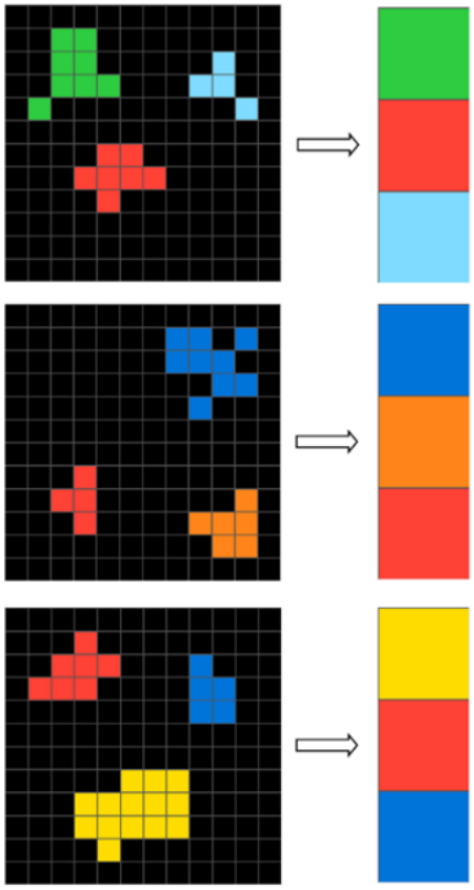
5.2 ábra. Az egyes megoldások átlagos feladatmegoldási ideje másodpercben. A saját megoldásomhoz tartozó időt a „” oszlop jelöli*

A DSL felépítéséből következően a sikeresen megoldott feladatok többnyire objektum-centrikus felépítésűek, azonban egyes esetekben fellelhetőek olyan feladatok, amelyek ilyen módon nem megoldhatóak (5.3 ábra). Ezekben az esetekben az indukciós algoritmus a DSL műveletein keresztül valamilyen „kiskapu” használatával állítja elő az elvárt kimeneti képet.

Ilyen például az f8ff0b80 feladathoz tartozó transzformációs program (5.3 ábra). A feladat méret szerint való rendezést igényel, az indukált megoldás ilyen művelet hiányában az `is_min`, `is_max` predikátumokkal oldotta meg a feladatot: a transzformációs programból az olvasható le, hogy a kimenet előáll három 1x1-es objektumból, amelyeknek függőleges koordinátájuk nulla, egy és kettő. A nulla koordinátájú színe legyen a bemenetei képen a maximális méretű objektum, a kettő koordinátájú a minimális, a kettő közötti pedig ami se nem maximális, se nem minimális.

Sok más algoritmushoz hasonlóan a bemutatott rendszer egyetlen feladatot sem tudott megoldani a privát teszt halmazon. Ennek pontos okát csak a feladatok ismeretében lehetne meghatározni, de a létező megoldások eredményeiből arra következtetek, hogy a privát feladatok nem, vagy csak valamilyen bonyolult módon oldhatóak meg objektum-centrikus megközelítéssel. Az 5.1 ábrán látható, hogy az összes ilyen szemléletet alkalmazó algoritmus teljesítménye nullára esett. A két algoritmus, ami viszont ennél jobb eredményt ért el, az pont az a két módszer, amely teljes képes műveleteket használ, igaz ezek teljesítménye is körülbelül a tizedére csökkent a publikus adathalmazhoz képest.

A



B

```
condition: is_max
shape:  const_shape ['0]
x:  const [0]
y:  const [0]
width:  const [1]
height: const [1]
color:  color_map_of [ANCHOR]
noise:  -

condition: !is_max, !is_min
shape:  const_shape ['0]
x:  const [1]
y:  const [0]
width:  const [1]
height: const [1]
color:  color_map_of [ANCHOR]
noise:  -

condition: is_min
shape:  const_shape ['0]
x:  const [2]
y:  const [0]
width:  const [1]
height: const [1]
color:  color_map_of [ANCHOR]
noise:  -
```

5.3 ábra. f8ff0b80 azonosítójú feladat (A), és az erre indukált program (B).

6 ÖSSZEFOGLALÁS

A dolgozat célja egy olyan DSL és indukciós algoritmus bemutatása volt, amely a proceduralitás elhagyásával kiküszöböli a bejárhatatlanul nagy keresési teret, ezáltal a létező algoritmusokhoz képest azonos, vagy jobb eredményt érjen el az ARC problémán. Ez részben sikerült, a bemutatott megoldás nagy mértékben redukálta a keresési teret olyan mértékben, hogy az indukciós algoritmusban a lehetséges összes megoldást számba lehetett venni, miközben számos más módszernél jobb eredményt ért el gyengébb számítási körülmények mellett. Említésre méltó eredmény elérése a privát teszt halmazon azonban sikertelen volt. Ennek legfőbb oka a feltételezés, miszerint az eddigi megoldások nem kielégítő eredményének egyetlen oka a túl nagy keresési tér. Bár ez a probléma kétségtelenül jelentős, mégis az én megoldásom annak ellenére, hogy ezt sikeresen kiküszöbölte nem tudott egyetlen feladatot sem megoldani a privát adathalmazon. Ennek egyik oka lehet az, hogy a DSL közel sem elégséges az összes feladat megoldására, hiszen a tanító halmazon is csak a példák kisebb részét oldotta meg. Ugyanakkor arra számítottam, hogy ha a publikus teszt halmazon nullánál jobb eredményt ér el a rendszer, akkor a privát teszt halmazon is várható valamilyen nem nulla eredmény (ami már jó eredménynek számít ezen az adathalmazon). A valószínűbb ok a rossz teljesítményre, hogy a privát feladatok nem oldhatóak meg objektum-centrikus műveletekkel, az előző fejezetben leírt okok miatt. Ezt a két okot figyelembe véve a rendszer továbbfejlesztésében az elsődleges feladat a művelet- és predikátumkészlet bővítése. Ha ez megtörtént, de az eredmény a privát teszt halmazra változatlan, akkor el kell gondolkodni azon, hogy lehetséges-e olyan indukciós algoritmust alkotni, amely képes a [11, 15] megoldásokhoz hasonló teljes képes műveletekből az ebben a dolgozatban bemutatott módon nem procedurális programokat indukálni.

A mély tanulás alapú megoldásomban a túl kevés tanító minta problémáját próbáltam kiküszöbölni a tanulási cél megváltoztatásával, oly módon, hogy a hálózat paramétereire elsődleges módon a reprezentáció tanulás legyen hatással és nem az aktuális elvárt kimenet előállítása. Ez a próbálkozás sikertelen volt, a rendszer számára ez sem biztosított elegendő tanító példát. A megoldás során csak előrecsatolt és konvolúciós neurális hálózatot használtam, így ennél komplexebb architektúrák és kifinomultabb hiperparaméter keresés irányában lehet érdemes elindulni, de úgy gondolom, hogy plusz tanító minták generálása nélkül ez sem fog sokkal jobb teljesítményt eredményezni.

A dolgozatban bemutatott megközelítés nem alkalmazható általános program indukciós problémákra, ugyanis vannak olyan esetek, amelyekben a proceduralitás nem kikerülhető. Az ARC-hoz hasonló absztrakció és analógia keresést igénylő feladatoknál [7, 8] azonban ez nem áll fent, így a bemutatott módszer eredményesen működhet az ilyen problémákra is.

7 ÁBRAJEGYZÉK

8 TÁBLÁZATJEGYZÉK

9 IRODALOMJEGYZÉK

[1] Chollet, Francois. *Deep learning with Python*. Vol. 361. New York: Manning, 2018.

[2] Marcus, Gary. "Deep learning: A critical appraisal." *arXiv preprint arXiv:1801.00631* (2018).

[3] Chollet, François. "On the measure of intelligence." *arXiv preprint arXiv:1911.01547* (2019).

[4] Spelke, Elizabeth S., and Katherine D. Kinzler. "Core knowledge." *Developmental science* 10.1 (2007): 89-96.

[5] Shanahan, Murray, et al. "An explicitly relational neural network architecture." *International Conference on Machine Learning*. PMLR, 2020.

[6] Lázaro-Gredilla, Miguel, et al. "Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs." *Science Robotics* 4.26 (2019).

[7] Sawyer, Daniel P., Miguel Lázaro-Gredilla, and Dileep George. "A Model of Fast Concept Inference with Object-Factorized Cognitive Programs." *arXiv preprint arXiv:2002.04021* (2020).

[8] Steenbrugge, X., Leroux, S., Verbelen, T., & Dhoedt, B. (2018). Improving generalization for abstract reasoning tasks using disentangled feature representations. *arXiv preprint arXiv:1811.04784*.

[9] Barrett, D., Hill, F., Santoro, A., Morcos, A., & Lillicrap, T. (2018, July). Measuring abstract reasoning in neural networks. In *International conference on machine learning* (pp. 511-520). PMLR.

[10] Santoro, Adam, et al. "A simple neural network module for relational reasoning." *Advances in neural information processing systems* 30 (2017).

[11] Fischer, Raphael, et al. "Solving Abstract Reasoning Tasks with Grammatical Evolution." (2020).

[12] Ellis, Kevin, et al. "Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning." *arXiv preprint arXiv:2006.08381* (2020).

[13] Banburski, Andrzej, et al. *Dreaming with ARC*. Center for Brains, Minds and Machines (CBMM), 2020.

[14] Acquaviva, S., Pu, Y., Nye, M., Wong, C., Tessler, M. H., & Tenenbaum, J. (2021). LARC: Language annotated Abstraction and Reasoning Corpus. In *Proceedings of the Annual Meeting of the Cognitive Science Society* (Vol. 43, No. 43).

[15] top-quarks/ARC-solution, url: https://github.com/top-quarks/ARC-solution utoljára megtekintve: 2022.12.11

[16] Ainooson, James, et al. "An Approach for Solving Tasks on the Abstract Reasoning Corpus." *arXiv preprint arXiv:2302.09425* (2023).

[17] Xu, Yudong, Elias B. Khalil, and Scott Sanner. "Graphs, Constraints, and Search for the Abstraction and Reasoning Corpus." *arXiv preprint arXiv:2210.09880* (2022).

[18] Alford, Simon, et al. "Neural-Guided, Bidirectional Program Search for Abstraction and Reasoning." *International Conference on Complex Networks and Their Applications*. Springer, Cham, 2021.

[19] Ferré, Sébastien. "First Steps of an Approach to the ARC Challenge based on Descriptive Grid Models and the Minimum Description Length Principle." *arXiv preprint arXiv:2112.00848* (2021).

[20] Kolev, Victor, Bogdan Georgiev, and Svetlin Penkov. "Neural Abstract Reasoner." *arXiv preprint arXiv:2011.09860* (2020)

[21] Tate, Ross, et al. "Equality saturation: a new approach to optimization." *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2009.

[22] Mikolov, Tomas, et al. "Efficient estimation of word representations in vector space." *arXiv preprint arXiv:1301.3781* (2013).

10 MELLÉKLETEK

10.1 1. számú melléklet: látens vektortér alapú megközelítés

10.1.1 Rendszer felépítése

A rendszer működésének első lépése a demonstrációs példa képeinek vektortérbe ágyazása, az encoder segítésével. A megfelelő bemenet-kimenet párokat kivonva kapunk három vektort, amelyek a képek közötti keresett absztrakt relációt reprezentálják. Ezeket a vektorokat átlagolva kapjuk meg a „feladat vektort", amelyet így a teszt bemenet kódolt pontjához hozzáadva megkapjuk a kívánt kimenetet a látens térben. Ezt a dekóder fogja kép formátumba visszaalakítani.

A látens vektorteret két módon lehet kialakítani (direkt és indirekt), az alapján, hogy a neurális hálózat milyen bemeneti adatot és milyen célváltozót kap. Mindkét megoldás végeredménye egy vektortér, amit ez után az eddig leírtaknak megfelelően használok.

Indirekt megközelítésben a tanítás az egyes képeken történik. Adott x_i kép át transzformálódik a vektortérbe, majd ezt visszakonvertálva előáll a rekonstrukció. A tanítás során a rekonstrukciós hibát kell minimalizálni, amely azt adja meg, milyen mértékben sikerült egy tetszőleges bemeneti képet a látens térből visszatranszformálni az eredeti képpé:

$$L_{rec} = \frac{1}{N}\sum_{i=1}^{N}(x_i - Dec(h_i))^2 \tag{21}$$

ahol *N* az adathalmaz mérete, x_i az adathalmaz egy képe, h_i az encoder által látens térbe ágyazott x_i.

A rendszer csak a vektortérbe ágyazást végzi, ezért teszteléskor a feladat vektort, és az így kapott kimenetet egy különálló kód végzi el. Indirekt módszer, mivel a hálózatnak nem mondjuk meg konkrétan, hogy a kialakított vektorteret mire fogjuk használni, csak azt kapja meg feladatául, hogy rekonstruálható módon legyen képes az egyes képeket reprezentálni, és ettől reméljük, hogy a Word2Vec [22]-hez hasonlóan a vektortér szemantikai jellemzőkkel bír, amik alapján a teszt példákat is meg tudjuk oldani.

Direkt megközelítésben a tanítás az egyes feladatokon történik. Adott feladat összes bemenet-kimenet párjait konvertálom a látens térbe, majd ezeket kivonva és átlagolva áll elő a „feladat vektor" -t. Ezt végül hozzáadom a teszt bemenet látens változatához, majd ezt visszakonvertálva előáll a predikció. Így a kimeneti hibát kell minimalizálni, amely megadja milyen mértékben különbözik a rendszer által előállított kép a cél kimenettől:

$$L_{out} = \frac{1}{N}\sum_{i=1}^{N}\left(y_i^{out} - Dec(h_i^{in} + v_{task})\right)^2 \tag{22}$$

ahol, *N* az adathalmaz teszt elemeinek száma, y_i^{out} az adathalmaz i-ik cél kimenete, h_i^{in} az adathalmaz i-ik teszt mintához tartozó bemenet látens térbeli reprezentációja, v_{task} pedig az absztrakt relációt reprezentáló „feladat vektor”.

A rendszer a vektortérbe ágyazáson kívül a feladatvektort és a prediktált kimenetet is előállítja, teszteléskor erre nem kell külön kód. Tehát ez a módszer direkt módon lett betanítva, mivel a rendszer tudja, hogy úgy alakítsa ki a vektorteret, hogy azt a teszt kimenet prediktálására lesz használva.

Több neurális hálózat felépítést teszteltem, a részletes architektúrák és hiperparaméterek a 10-1 táblázatban láthatóak

10.1.2 Előfeldolgozás

Az adatok egy kétdimenziós mátrixban vannak reprezentálva. Mivel a képeken csak 10 szín szerepel, ezért pixelértékek helyett tíz egész szám indikálja, mely szín van az adott pozícióban. A képek maximális mérete 30x30-as. Az első megoldandó probléma a képek dinamikus méretéből adódik, mivel ezek 1x1 és 30x30 között bárhol lehetnek, viszont a neurális hálózat csak fix méretű kimenetet tud előállítani. Ennek megoldására, a tíz alapszín mellé egy kiegészítő „határszín” kerül, amely azt indikálja, hogy az adott pozíció a képhez tartozik-e. Ezekkel a határszínekkel kiegészítve, így minden kép egy 30x30-as mátrixként reprezentálható. Bár a neurális hálózatok tetszőleges valós számokkal operálnak, a tanítás gyorsítása céljából érdemes 0 és 1 közé konvertálni a bemenet értékeit.

Egy ilyen módszer lehet a Word2Vec-nél is használt „one-hot” reprezentáció, ahol egy színt az ARC esetében egy 11 méretű (10 alapszín + 1 határszín) vektor reprezentál, ahol minden érték nulla, kivéve azt az egy helyet, ahol a tényleges szín jelen van, ekkor az ott lévő érték egy.

Egy másik módszer, amikor a kép előfeldolgozását is egy neurális hálózat végzi. Betanítható egy neurális hálózat, hogy a kép adott pozíciójában található régiót egy sokdimenziós vektorrá alakítsa, szintén a Word2Vec-hez hasonló módon, azaz a kép környezetéből prediktálja az adott régiót. Ebből a neurális hálózatból kinyert súlytényezők szolgálhatnak az egyes színek reprezentálására, ezek elméleti előnye, hogy valamilyen szemantikai jelentéssel bírnak ezek a reprezentációk, így a tanulás potenciálisan felgyorsítható. Az utolsó módszer az, amikor egyszerűen mindegyik 11 színhez egy-egy sokdimenziós vektort rendelünk, ahol a vektor elemei 0 és 1 közötti véletlen értékkel rendelkeznek. Mivel egyik módszer sem eredményezett jelentősebb gyorsulást a tanításban, így one-hot vektor reprezentációt választottam, amely kevésbé volt memóriaigényes.

10.1.3 Tanítás

Tanítás során további előfeldolgozás nélkül, a rendszer hamar túltanul a minták leggyakoribb statisztikai jellemzőére a 10x10 fekete háttérre (10.1 ábra első képe), ugyanis a bemeneti adatok jelentős része ilyen 10x10-es fekete hátterű kép. Ennek elkerülésére különböző optimalizálók, és különböző tanulási rátákkal próbálkoztam, amelyek csak minimálisan javították a hálózat rekonstrukciós teljesítményét. Tanítás után bármely bemenetre az túltanult kimeneti képet adta vissza.

A probléma megoldásának első lépése a bemeneti adathalmaz megkeverése volt. Alapesetben a képek az egyes feladatokból vannak kinyerve, amelyek esetenként nagyban hasonlíthatnak egymásra amennyiben a képek közötti absztrakt reláció csak kis mértékben változtatja meg a bemenetet. Ezeknek a hasonló képsorozatoknak a felbontása nagyban hozzájárult a probléma megoldásához. A teljes megoldást a tanítási ráta feladathoz illesztése adta, 10^{-5} – 10^{-6} közti értékeket használt.

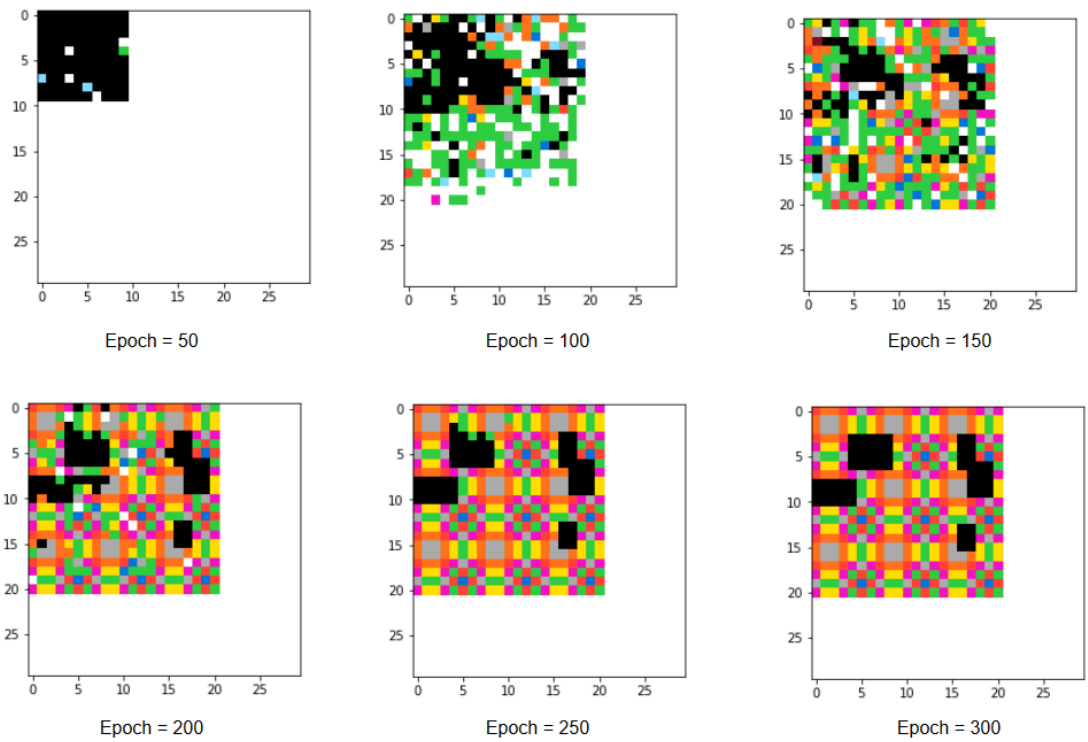
Az 10.1 ábrán látható a hálózat rekonstrukciós teljesítménye 300 epoch-on keresztül. Megfigyelhető, hogy az adathalmaz megkeverésével és a tanítási ráta csökkentésével nem a lokális minimumba jutást kerülte el a hálózat, hanem az abból való kijutást tette lehetővé.

10.1.4 Értékelés

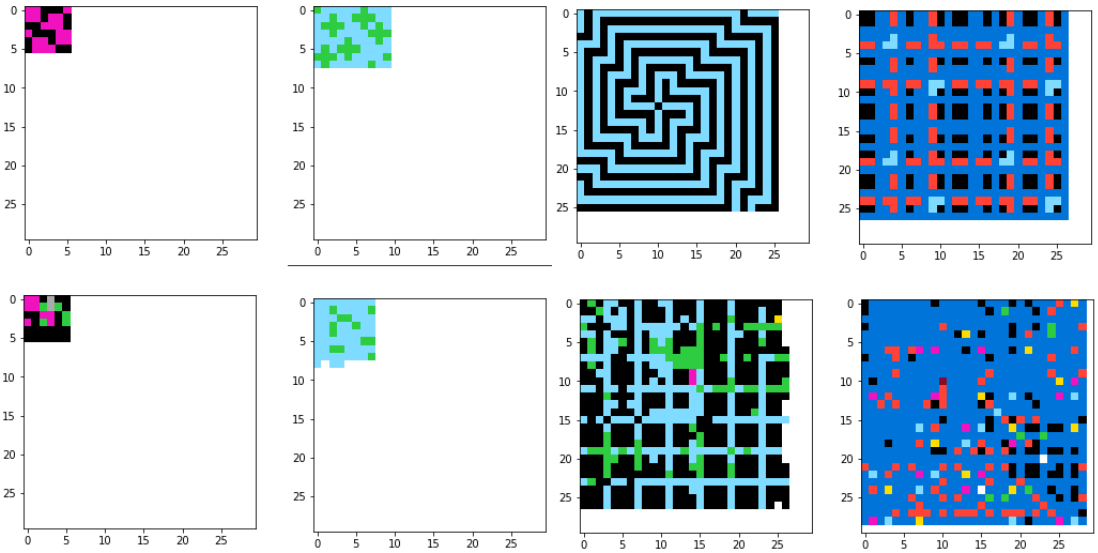
Az bemutatott rendszer betanítás után közel hiba nélkül képes az adatok rekonstruálására. A 300 feladat közül a rendszer egyetlen esetben sem tudta pontosan előállítani az elvárt kimenetet, ugyanakkor bizonyos esetekben a rendszer által megjósolt kép felületesen, de tartalmazza az elvárt kimenet részleteit (10.2 ábra).

Megfigyelhető, hogy a hálózat a teszt halmazon a tanító halmazhoz tartozó kimenetekhez hasonló eredményeket ad, tehát a rendszer csupán bememorizálta a tanító halmazt. Ez gyakori jelenség olyan esetekben amikor túl kevés adat áll rendelkezésre.

A hálózat méreteinek növelése nem eredményezett javulást, azonban további adatok generálása feltehetően javítana a teljesítményen, hiszen az Autoencoder körülbelül 1200 példán lett csak betanítva, azonban az ARC feladat lényege, hogy kevés adatból is lehessen hatékonyan tanulni, így az olyan megoldásokat, amelyek pluszban generáltak további tanító példákat ([20]) az irodalom nem tekinti elfogadott megoldásnak.



10.1 ábra. Rekonstrukciós teljesítmény betanítás alatt. A rendszer az 50. epoch után kilép a lokális minimumból



10.2 ábra. A hálózat által előállított kimenet (alsó sor), és az elvárt kimenet (felső sor) összehasonlítása

10-1 táblázat. Egyes architektúrák eredményei (MSE – Mean Squared Error)

Architektúra	Paraméter szám	Mód	Epoch	MSE tanító	MSE teszt	Pontosság tanító
AE	2.57M	Direkt	2000	0.66	12.52	9%
AE + Dropout	2.57M	Direkt	2500	1.81	11.36	1%
AE	10.4M	Direkt	1700	0.28	11.91	27%
AE + Dropout	10.4M	Direkt	2500	0.52	11.01	12%
β-VAE	1.3M	Direkt	500	-	-	0%
Conv-AE	0.14M	Indirekt	2000	9.52	14.99	0%
Conv-AE	0.26M	Indirekt	1500	12.89	17.51	0%

10.2 2 számú melléklet: a DSL teljes predikátumkészlete

10.2.1 Predikátumkészlet

Szín csoport: két vagy több objektum c színcsoportot alkot, ha az összes ilyen objektum színtérképén megtalálható a c szín.

Vizuális csoport: két objektum vizuálisan egyezik, amennyiben a két objektum alak tulajdonsága, szélessége, és hosszúsága egyezik. Ezen reláció szerint lehet vizuális csoportokat alkotni, a színcsoportokhoz hasonló módon.

belongs_to_color_group [*obj*]: igaz, ha a paraméterül kapott objektum legalább egy színcsoportba tartozik, hamis egyébként

unique_color [*obj*]: igaz, ha a paraméterül kapott objektum egyszínű, és nem tartozik semmilyen színcsoportba, hamis egyébként

belongs_to_specific_color_group [*obj*] [c]: igaz, ha a paraméterül kapott objektum a *c* színcsoportba tartozik (tehát a csoport színe c), hamis egyébként

belongs_to_visual_group [*obj*]: igaz, ha a paraméterül kapott objektum vizuális csoportba tartozik, hamis egyébként

unique_visual [*obj*]: igaz, ha a paraméterül kapott objektum nem tartozik vizuális csoportba, hamis egyébként

belongs_to_specific_visual_group [*obj*] [*V*]: igaz, ha a paraméterül kapott objektum *V* vizuális csoportba tartozik, hamis egyébként

const_visual [*obj*] [*V*]: igaz, ha a paraméterül kapott objektum vizuálisan egyezik *V*-vel, hamis egyébként

has_color [*obj*] [*c*]: igaz, ha a paraméterül kapott objektumszíntérképe tartalmazza *c* színt, hamis egyébként

container [*obj*]: igaz, ha a paraméterül kapott objektum magába foglal egy másik objektumot, hamis egyébként

contained [*obj*]: igaz, ha a paraméterül kapott objektumot magába foglalja egy másik objektum, hamis egyébként

parent [*obj*]: igaz, ha a paraméterül kapott objektumnak vannak zaj objektumai, hamis egyébként

child [*obj*]: igaz, ha a paraméterül kapott objektum zaj objektuma egy másiknak, hamis egyébként

unicolor [*obj*]: igaz, ha a paraméterül kapott objektum egyszínű, hamis egyébként

multicolor [*obj*]: igaz, ha a paraméterül kapott objektum többszínű, hamis egyébként

is_max [*obj*]: igaz, ha a paraméterül kapott objektum mérete az összes képből kiszűrt objektum közül a legnagyobb, hamis egyébként

is_min [*obj*]: igaz, ha a paraméterül kapott objektum mérete az összes képből kiszűrt objektum közül a legkisebb, hamis egyébként

is_max_color [*obj*] [*c*]: igaz, ha a paraméterül kapott objektum tartalmazza a *c* színt, és ez a szín ebben az objektumban található meg a legtöbbször

contained_by_const_visual [*obj*] [*V*]: igaz, ha a paraméterül kapott objektumot magába foglalja egy másik objektum, amely vizuálisan egyezik *V*-vel, hamis egyébként

belongs_to_visual_group_contained_by_const_visual [*obj*] [*V*]: igaz, ha a paraméterül kapott objektum vizuális csoportba tartozik, és a csoportnak van legalább egy eleme, amelyet magába foglal egy olyan objektum, amely vizuálisan egyezik *V*-vel, hamis egyébként

10.3 3 számú melléklet: az implementáció kódja

Az implementáció kódja elérhető az alábbi linken: https://github.com/norbert-neumann/Szakdolgozat-ZN8VJ5