Abstraction and Reasoning

Jakub Brojacz, Weronika Głuszczak, Marta Banel, Rafał Śliwiński, Paweł Szymkiewicz

Wstęp

W swoim projekcie spróbowaliśmy rozwiązać zadanie konkursowe Abstraction and Reasoning Challenge ze strony https://www.kaggle.com/c/abstraction-and-reasoning-challenge/

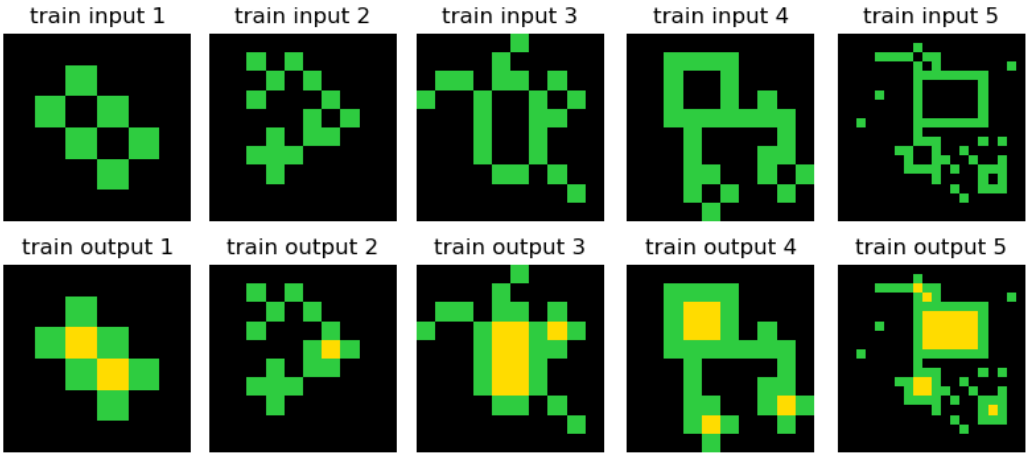
Opis problemu

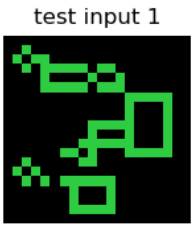
W dzisiejszych czasach większość technik uczenia maszynowego wymaga bardzo dużej ilości danych oraz mocy obliczeniowej. Sztuczna inteligencja potrafi wykrywać jedynie schematy, które już kiedyś widziała. Stosując obecne metody, algorytm może zdobyć nowe umiejętności poprzez wystawienie go na jeszcze większe ilości danych, ale zdolności poznawcze, które mogłyby zasadniczo uogólnić wiele zadań, pozostają niejasne. Dlatego też bardzo trudne jest stworzenie systemów, które poradzą sobie ze zmiennością i nieprzewidywalnością prawdziwego świata, takich jak roboty domowe, czy samochody samobieżne.

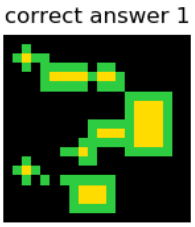
Problem zaproponowany przez twórców konkursu zakłada, że w przyszłości algorytmy uczenia maszynowego będą miały olbrzymią zdolność uogólniania podstawowych schematów na podstawie bardzo małego zbioru danych. Celem konkursu jest napisanie programu, który dla 3-5 par treningowych złożonych z zadania i odpowiedzi będzie potrafił znaleźć poprawną odpowiedź dla zadania testowego. W tym przypadku zadaniem algorytmu jest przekształcenie dwuwymiarowej tablicy liczb (prostego obrazka) w pewien określony sposób. To algorytm ma za zadanie zdecydować, na czym polega oczekiwana transformacja (lub sekwencja transformacji).

Przykładowe zadanie:

- Pięć treningowych zadań i odpowiedzi
- Jedno zadanie testowe wraz z poprawną odpowiedzią







Nasz pomysł

Głównym pomysłem było wyróżnienie pewnych operacji, które można wykonywać na planszy, np. operacja odbicia lustrzanego, operacja przesunięcia elementu. Naszym celem było znajdowanie takich ciągów operacji, które dla par treningowych, po kolejnym nałożeniu na wejściowe plansze, zwracały w efekcie poprawne plansze wyjściowe. Jeśli udało się znaleźć taki ciąg to był on nakładany na testowa planszę wejściową a efekt wykonania tego ciągu był traktowany jako wynik.

Poszukiwanie ciągu operacji dla pojedynczej pary planszy wejściowej i wyjściowej przebiegało zgodnie z następującymi etapami:

1. Podział planszy wejściowej na elementy.
   Niektóre operacje działają dla całej planszy, a niektóre potrzebują wyróżnionych elementów, np. żeby je przesunąć lub obrócić. Sprawdzano 6 sposobów podziałów na elementy, w których to, czy dane punkty (pojedyncze pola planszy) należą do tego samego elementu, czy nie, było sprawdzane zgodnie z regułami:
   a. Pola nie są w kolorze tła (czyli traktowanie wszystkich pól w innym kolorze niż kolor tła jako jeden element).
   b. Pola są w tym samym kolorze.
   c. Pola mają wspólny bok.
   d. Pola mają wspólny bok i są w tym samym kolorze.
   e. Pola mają co najmniej jeden wspólny wierzchołek.
   f. Pola mają co najmniej jeden wspólny wierzchołek i są w tym samym kolorze.

2. Wyróżnienie grupy elementów.
   Żeby nie wykonywać operacji dla wszystkich możliwych kombinacji elementów, sprawdzano tylko określone grupy:
   a. Wszystkie elementy na planszy.
   b. Elementy stykające się z którąś krawędzią planszy.
   c. Elementy największe pod względem pola najmniejszego prostokąta pokrywającego dany element.
   d. Element zawierający unikatowy kolor – kolor, który jako jedyny występuje tylko w jednym elemencie.
   e. Elementy będące pojedynczym punktem.
   f. Elementy niebędące pojedynczym punktem.
3. Wybór operacji.
   Sprawdzano następujące operacje:
   a. Powiększenie planszy.
   b. Zmiana koloru elementu.
   c. Łączenie par pól, będących pojedynczymi punktami, linią lub traktowanie ich jako punktów przekątnej rysowanego prostokąta.
   d. Usunięcie elementu.
   e. Powiększenie elementu na całą planszę.
   f. Wypełnienie elementu kolorem.
   g. Dodanie ramki do planszy.
   h. Wyróżnienie części wspólnej/sumy/różnicy/różnicy symetrycznej dwóch połówek planszy.
   i. Pozostawienie jedynie wyróżniających się elementów.
   j. Odbicie lustrzane planszy.
   k. Przesunięcie elementu.
   l. Obrócenie planszy.
   m. Odtworzenie elementów po powiększeniu planszy.
   n. Odwrócenie symetryczne elementu.
4. Wybór szczegółów działania operacji.
   Niektóre z operacji potrzebują dodatkowych argumentów, np. w którą stronę przesunąć element lub według której osi wykonać odbicie symetryczne. Argumenty były generowane dla każdej operacji osobno i sprawdzano wszystkie możliwe opcje.
5. Sprawdzenie, czy dana operacja z danymi argumentami może być wykonana dla danej planszy z daną grupą elementów, np. łączenie elementów było wykonywane tylko gdy wszystkie wyróżnione elementy były pojedynczymi punktami.
6. Wykonanie operacji.
7. Wybór kolejnej operacji z nowymi argumentami i jej wykonanie.
8. Podjęcie decyzji, czy dany ciąg operacji traktujemy jako poprawny.
   Zaimplementowano dwie strategie:
   a. Sprawdzenie czy ciąg przeprowadza dana pierwszą planszę wejściową w wyjściową (strategia szybsza, ale potencjalnie działająca z większymi błędami niż strategia kolejna).
   b. Dodatkowo sprawdzenie czy ciąg przeprowadza pozostałe plansze wejściowe w odpowiadające im plansze wyjściowe (strategia użyta w końcowych testach projektu).
9. Podjęcie decyzji, czy poszukiwanie kolejnych ciągów będzie kontynuowane, czyli czy limit liczby zwracanych plansz nie został osiągnięty.

Pseudokod

Poniższa funkcja dla obiektu reprezentującego zbiór plansz treningowych i testowych zwraca rozwiązania do wszystkich zadań testowych. Jest to uogólniony kod naszego podejścia do problemu.

```
# task - aktualnie rozwiązywane zadanie, zawiera plansze treningowe i testowe
# max_results - maksymalna dopuszczana liczba rozwiązań dla zadania
process_task(task, max_results)
{
   num_test = length(task.test);
   remaining_result_boards = max_results;
   result_boards = [[] for i in range(num_test)];

   # sposoby podziału planszy na elementy (punkt 1)
   for (split_type in SPLITTING_TYPES)
   {
      # dzieli zadanie na elementy zależnie od sposobu
      set_split_type(task, split_type);

      # wyróżnianie grup elementów (punkt 2)
      for (element_group_type in ELEMENT_GROUPS)
      {
         # wyróżnia grupy zależnie od sposobu
         set_element_group_type(task, element_group_type);

         # ciąg trenginowych par (zadanie, wynik)
         sets = [(t.input, t.output) for t in task.train];

         # lista ścieżek przeprowadzających zadanie w wynik
         # scieżka - lista par (operacja, argumenty operacji)
         paths = [];

         results_to_return = remaining_results_board;

         # wybór pierwszej operacji (punkt 3)
         for (op1 in OPERATIONS)
         {
            # iteracja po wszystkich dozwolonych argumentach danej operacji (punkt 4, punkt 5)
            # wykorzystujemy do tego celu pierwsze zadanie (sets[0][0])
            for (args in op1.generate_arguments(sets[0][0], board.element_group))
            {
               # wykonanie operacji (punkt 6) na pierwszej parze treningowej
               operation_result, arguments = op1.run(sets[0][0], args);
               # sprawdzenie, czy operacja przeprowadza pierwsze zadanie w wynik (punkt 8)
               if (operation_result.equals(sets[0][1]))
               {
                  path = [(op1, args1)];
                  matched = true;
                  # iteracja po pozostałych parach zadanie - odpowiedź
                  for (i in range(1, length(sets)))
                  {
                     processed_board = sets[i][0].copy();
```

```
                for path_operation, path_args in path
                {
                  # wykonanie operacji (punkt 6) na pozostałych parach treningowych
                  processed_board = path_operation.run(processed_board, path_args);
                }
                # sprawdzenie, czy operacja przeprowadza pozostałe zadania w wyniki (punkt 8)
                if (! processed_board.equals(sets[i][1]))
                {
                  matched = false;
                  break;
                }
              }
              if (matched)
              {
                paths.append(path);
                results_to_return -= 1;
              }
              if (results_to_return == 0)
              {
                # wyjście z pętli generujących operacje
                break 2;
              }
            }
            for (op2 in OPERATIONS: # wybór drugiej operacji (punkt 7))
            {
              # działanie analogiczne jak pętla 'for op1 in OPERATIONS'
              # nakłada nową operację na wynik działania operacji op1
              …
            }
          }
        }

        # iterujemy po wszystkich zadaniach testowych
        for (i in range(num_test))
        {
          # na każde zadanie testowe nakładamy ścieżkę
          for (path in paths)
          {
            output_board = task.test[i].input.copy();
            for (operation, args in path)
              output_board = operation.run(output_board, args);
            result_boards[i].append(output_board);
          }
        }
        remaining_result_boards -= length(paths);

        # jeżeli osiągnęliśmy limit dopuszczalnych odpowiedzi, zwracamy je
        if (remaining_result_boards <= 0)
          return result_boards;
      }
    }
    return result_boards;
```

```
}
```

Działanie programu

Sposób uruchamiania programu:

W katalogu głównym projektu uruchomić komendę

python .\src\main.py

Program był testowany dla pythona w wersji 3.7

Poniżej przedstawiony jest wynik uruchomienia programu dla zbioru treningowego złożonego z 4 przykładów:

```
PS C:\Users\Ja\source\repos\Abstraction-and-Reasoning> python .\src\main.py
data\training_sample\00d62c1b.json
Task 1: 1/1 tests passed with 3/3 correct boards
data\training_sample\1b2d62fb.json
Task 2: 1/1 tests passed with 3/3 correct boards
data\training_sample\1cf80156.json
Task 3: 1/1 tests passed with 3/3 correct boards
data\training_sample\1f876c06.json
Task 4: 1/1 tests passed with 3/3 correct boards
Results:
Passed tests: 4
All tests: 4
```

Program można uruchomić z następującymi parametrami:

- --help, -h Wyświetla wiadomość pomocniczą i kończy program
- --filter, -f Filtruje tylko zadania z czarnym tłem
- --visualize, -v Wizualizuje zadania
- --time, -t Mierzy czas wykonania

Dodatkowe parametry programu można modyfikować w pliku src/config.py. Są to:

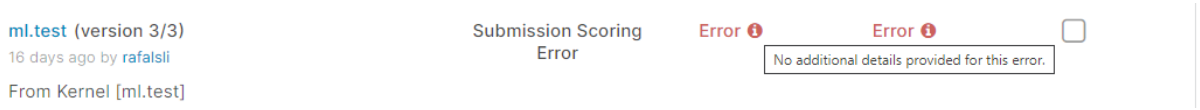
- Ścieżki do folderów z zadaniami ewaluacyjnymi i treningowymi
- Wybrana strategia (ONE_BY_ONE – sprawdzanie ciągu operacji na wszystkich dostępnych parach plansz wejściowej i wyjściowej lub FIRST_ONLY - tylko dla pierwszej)
- Maksymalna liczba zwracanych planszy do pojedynczego zadania
- Maksymalna powierzchnia rozważanych planszy
- Minimalna liczba kolorów rozważanych planszy
- Maksymalna liczba kolorów rozważanych planszy
- Warunek, czy rozważane plansze muszą zawierać kolor czarny (jako kolor czarny przyjęliśmy kolor o numerze 0)
- Kolor przyjęty jako kolor tła
- Kolor przyjęty jako transparentny (potrzebny przy nakładaniu elementów, aby rysować tylko faktyczne punkty a nie cały prostokąt zawierający element)
- Maksymalna liczba kolorów na wszystkich planszach
- Maksymalna długość boku planszy

Zbiory danych

Za zbiory danych posłużyły nam zbiory udostępnione w konkursie. Było to 400 zadań treningowych i 400 ewaluacyjnych, w których każde składało się z kilku przykładów treningowych i z reguły jednego testowego. Na podstawie ich analizy opracowaliśmy schemat działania programu oraz testowaliśmy nasze rozwiązanie.

Wyniki

Na serwisie Kaggle nie udało nam się przetestować naszego rozwiązania. Program uruchomiony na nieznanym zbiorze testowym, na podstawie którego przyznawana była punktacja zwracał nieznany błąd:



Błędem z naszej strony była próba uruchomienia programu na nieznanym zbiorze dopiero pod koniec naszej pracy. Nie udało nam się zidentyfikować błędu powodującego powyższy komunikat. Przez cały czas pracowaliśmy na GitHubie, a nie na Kaggle. Praca zespołowa na Kagglu przy złożonych projektach z dużą ilością kodu jest utrudniona, ponieważ dostarcza on możliwość edycji wszystkiego de facto w jednym pliku, a aby dołączyć kolejne trzeba niepotrzebnie kombinować.

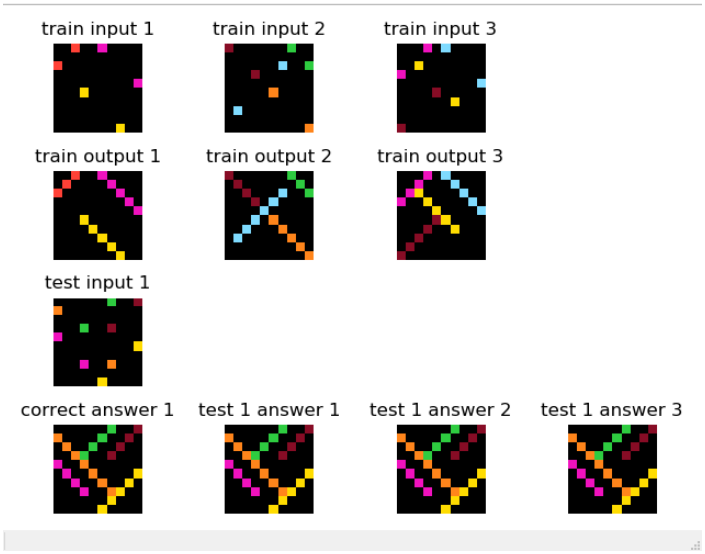
Udało nam się natomiast uruchomić program na zadaniach treningowych i ewaluacyjnych. Wyniki zamieszczone są poniżej:

- Zbiór treningowy
  - Poprawne rozwiązania: 44/416 (10,6%)
  - Czas obliczeń: 52 minuty 37 sekund
- Zbiór ewaluacyjny
  - Poprawne rozwiązania: 16/419 (3,8%)
  - Czas obliczeń: 1 godzina 19 minut 40 sekund

Z uwagi na naturę zadania ciężko jest głębiej analizować wyniki, gdyż niemalże wszystkie błędne odpowiedzi wynikają z tego, że albo nasz program nie znał wymaganej operacji, albo potrzebna ilość nałożonych na siebie operacji była większa od dwóch.

Poniżej przedstawiono kilka otrzymanych rozwiązań do wspomnianych zadań:

1) 1f876c06.json



Warto zwrócić uwagę, że algorytm zwrócił trzy plansze (wszystkie poprawne), co spowodowane jest tym, że znalazł trzy konfiguracje prawidłowo rozwiązujące zadania ze zbioru treningowego. Jedna z tych konfiguracji to:

1) Rodzaj podziału: kolor bez znaczenia, obszar 4-spójny. Każda kropka jest więc osobnym elementem.

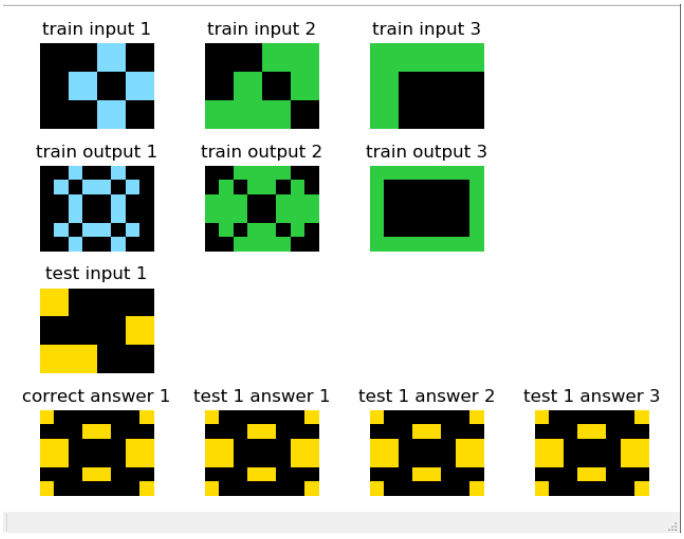
2) Grupa elementów: wszystkie elementy są w grupie.

3) Operacja: Łączenie kropek

Argument operacji: Łączenie po przekątnej.

Do pozostałych dwóch ścieżek weszły dodatkowo operacje, które nie zmieniły już dalej wyniku.

2) 3af2c5a8.json



Znalezione konfiguracje zawierały:

Konfiguracja 1:

1.1) Rodzaj podziału: wszystko jest jednym elementem (oprócz tła).

1.2) Grupa elementów: wszystkie elementy są w grupie.

1.3) Operacje: odbij planszę w prawo, odbij planszę w dół.

Konfiguracja 2:

2.1) Rodzaj podziału: wszystko jest jednym elementem (oprócz tła).

2.2) Grupa elementów: wszystkie elementy są w grupie.

2.3) Operacje: odbij planszę w dół, odbij planszę w prawo.

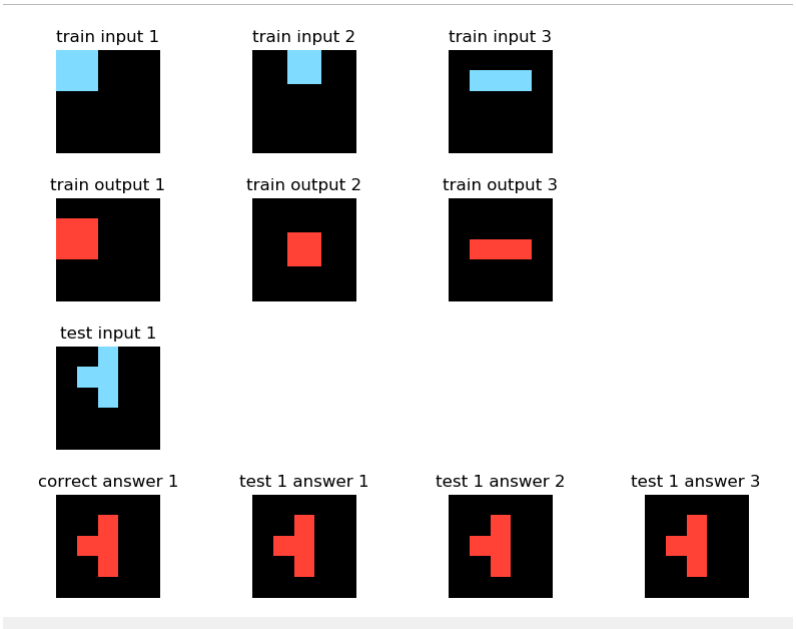
Konfiguracja 3:

3.1) Rodzaj podziału: wszystko jest jednym elementem (oprócz tła).

3.2) Grupa elementów: element z krawędzi. Ponieważ wszystkie komórki różne od tła należą do jednego elementu to do grupy wszedł cały duży element. Efekt jest więc taki sam jak dla konfiguracji 1 i 2.

3.3) Operacje: odbij planszę w prawo, odbij planszę w dół.

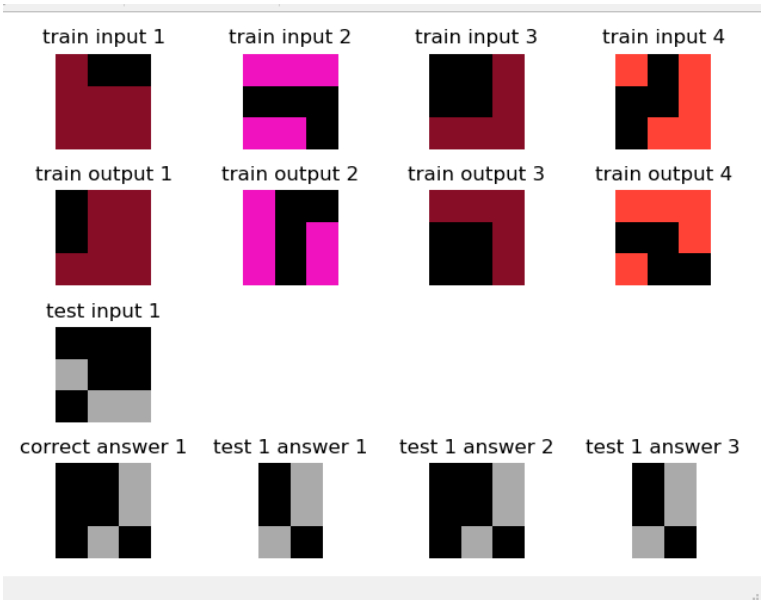
3) a79310a0.json



Ponieważ na planszy jest jeden element niezależnie od sposobu podziału w konfiguracjach pojawiły się różne podziały i różne grupy. Niezależnie od tych parametrów do rozwiązania zadania potrzebne było wykonanie dwóch operacji: zmień kolor (na zawsze ten sam) i przesuń o jeden w dół.

4) ed36ccf7.json

Przykład ten pokazuje, że znalezienie odpowiedniej konfiguracji dla wszystkich zadań treningowych nie oznacza, że zadanie testowe zostanie rozwiązane poprawnie:



W każdej ścieżce znalazła się operacja obrotu planszy. W dwóch jednak dodatkowo znalazła się operacja wyodrębnienia elementu. Ponieważ wszystko było traktowane jako jeden duży element, operacja ta nie wpłynęła na zadania treningowe. Zmieniła jednak rozwiązanie testowego zadania.

Wnioski

Ciężko jednoznacznie podsumować osiągnięte rezultaty. Nie udało się wziąć udziału w zawodach na Kagglu. Z drugiej strony mieliśmy dostępne kilkaset zadań i na tej podstawie jesteśmy w stanie ocenić skuteczność algorytmu. Chociaż skuteczność wynosi kilka procent to i tak jest to zadawalający wynik. Podchodząc do zawodów na Kagglu najlepszy wynik wynosił 3% i wiedzieliśmy, że osiągnięcie takiego rezultatu byłoby sukcesem. Ostatecznie ciężko zweryfikować, czy nasze rozwiązanie jest na tyle uniwersalne, żeby osiągnąć taki wynik, jednak czytając udostępnione rozwiązania najlepszych prac na forum Kaggle stwierdzamy, że przyjęta idea była słuszna.

Ogólnie, ostatecznie na Kaggle zwyciężyło rozwiązanie, które osiągnęło 21% skuteczności.
W poniższym linku zamieszczono przedstawienie tego rozwiązania:

https://www.kaggle.com/c/abstraction-and-reasoning-challenge/discussion/154597

Nasze rozwiązanie nie jest rozwiązaniem, które osiągnęło już maksimum swoich możliwości. Dodawanie kolejnych operacji oraz grup elementów powinno jedynie zwiększać skuteczność algorytmu. Przy dalszym rozbudowywaniu programu należałoby jednak zoptymalizować budowanie ciągu operacji.