26.06.2017

Sprawozdanie z laboratorium sztucznej inteligencji

Łamigłówki (paradygmat CLP(FD)

Temat nr 1: Łamigłówka Rogo (ang. Rogo puzzle)

I. Opis zadania:

Napisać program w języku prolog z użyciem paradygmatu programowania z ograniczeniami (CLP (FD)) rozwiązujący łamigłówki Rogo.
Puzzle Rogo są przedstawiane jako tablica `N x M` pól. Każde z pól może mieć jedną z właściwości:

- pole przechodnie o zerowej wartości,
- pole przechodnie o dodatniej wartości,
- pole nieprzechodnie o ujemnej wartości.

Dane wejściowe niezbędne do utworzenia łamigłówki to:

- ilość kroków – liczba ta określa ilość pól, które znajdą się w rozwiązaniu,
- minimalna oraz maksymalna ilość punktów – liczby te tworzą zakres wartości określających minimalny oraz maksymalny wynik znalezionych rozwiązań,
- wartości pól umieszczone w macierzy o wymiarach `N x M` – przedstawiające pola przechodnie oraz nieprzechodnie.

Aby rozwiązać łamigłówkę należy znaleźć ścieżkę o określonej długości oraz wartości. Wartość ścieżki to suma punktów pól wchodzących w skład ścieżki. Ścieżka nie może być ani krótsza, ani dłuższa niż podano w danych wejściowych. Ścieżka może przebiegać jedynie w poziomie i pionie – polami sąsiednimi nie są pola znajdujące się po skosie.

II. Założenia realizacyjne:

1. Założenia dodatkowe:
   - Projekt zakłada utworzenie interfejsu graficznego umożliwiającego użytkownikowi tworzenie puzzli. Wspomniany interfejs zostanie utworzony przy użyciu technologii WPF oraz języka C#.
   - Ujemna wartość pól nieprzechodnich została wybrana w celu ułatwienia odnajdywania tych pól. Dzięki czemu w procesie odnajdywania pól przechodnich porównania są dokonywane tylko w zbiorze liczb.
2. Metody, strategie oraz algorytmy wykorzystywanie do rozwiązania zadania:

   `Inicjalizuj_liste/2`

   Dane wejściowe:
   - `Dlugosc` – zmienna ta przechowuje długość ścieżki (rozwiązania)

   Wynik:
   - `JakasLista` – lista wyjściowa o podanej długości długości

   Opis:

   Funkcja zapewnia odpowiednią długość listy oraz spełnienie warunku, takiego że pierwszy oraz ostatni element listy będzie taki sam.

`czy_sasiednie/3`

Dane wejściowe:

- `Lista` – lista odzwierciedlająca ścieżkę, która ma być sprawdzona pod kątem czy jej pola są parami sąsiednie
- `Szerokosc` – zmienna przechowująca szerokość planszy do gry
- `Szer` – Zmienna `Szerokosc` pomniejszona o 1

Wynik:

- `True / False`

Opis:

Funkcja sprawdza czy pola w ścieżce znajdują się obok siebie (są sąsiednie).

`sprawdz_sume/3`

Dane wejściowe:

- `SciezkaBezGlowy` – lista indeksów wchodzących w skład znalezionego rozwiązania
- `ListaWartosci` – lista przechowująca wartości wszystkich pól wchodzących w skład puzzli

Wynik:

- `Suma` – wynik sprawdzanego rozwiązania

Opis:

Funkcja oblicza sumę wartości wybranych pól. Wybrane pola są przechowywane w liście `SciezkaBezGlowy` i są reprezentowane w formie indeksów.

`wybierz_niezerowe/3`

Dane wejściowe:

- `ListaWartosci` - lista przechowująca wartości wszystkich pól wchodzących w skład puzzli
- `ZakresIndeksow` - lista przechowująca wartości od 0 do `Szerokosc*Wysokosc`

Wynik:

- `ListaGlow` – lista wyjściowa przechowująca numery indeksów pól, których wartość jest większa od 0

Opis:

Funkcja zwraca listę indeksów pól, których wartość jest dodatnia. Indeksy te są wykorzystywane jako początek wyszukiwanych ścieżek.

`wybierz_nieujemne/3`

Dane wejściowe:

- Dane wejściowe takie jak w funkcji `wybierz_niezerowe/3`

Wynik:

- `ListaNieujemnych` - lista wyjściowa przechowująca numery indeksów pól, których wartość jest większa lub równa 0

Opis:

Funkcja zwraca listę indeksów pól, których wartość jest dodatnia lub wynosi 0. Indeksy te są wykorzystywane jako możliwe pola w wyszukiwanych ścieżkach.

`szukaj/7`

Dane wejściowe:

- `Szerokosc` – jest to wartość odzwierciedlająca ilość pól mieszczącą się na planszy do gry Rogo w szerz;
- `Wysokosc` – jw. tylko w odniesieniu do długości planszy;
- `LiczbaKrokow` – jest to liczba pól, z których składa się lista wynikowa;
- `Dobrze` – minimalna ilość punktów do uzyskania aby rozwiązać puzzle Rogo;
- `Najlepiej` – maksymalna ilość punktów możliwa do uzyskania w tym konkretnym puzzle Rogo;
- `ListaWartosci` – lista przechowująca wartości pól (zaczynając od górnego lewego rogu, w kolejności od lewej do prawej, a następnie wierszami);

Wynik:

- `ListaIndeksow` – wynikowa lista, w której znajdują się indeksy pól wchodzących w skład rozwiązania

Opis:

Jest to główna funkcja wywoływana przez GUI programu, wywołuje ona pomniejsze funkcje oraz przygotowuje dane pomocnicze, o których mowa w punkcie IV podpunkt 2. Jej wynikiem jest znalezione rozwiązanie.

`szukaj_po_glowie/8`

Dane wejściowe:

Dane wejściowe takie same jak w funkcji `szukaj/7` oraz dodatkowo:

- `Glowa` – numer indeksu pola, z którego ma rozpoczynać się rozwiązanie

Wynik:

- Dane wyjściowe identycznej jak w funkcji `szukaj/7`

Opis:

Rozszerzenie funkcji `szukaj_po_glowie/8` o możliwość wyszukiwania rozwiązań zaczynających się w wybranym polu.

3. Języki programowania, narzędzia informatyczne i środowiska używane do implementacji systemu:
   - SWI-Prolog v7.4.1 (wersja 32 bitowa) do obsługi logiki programu,
   - Obiektowy język programowania C# wraz z technologią WPF (Windows Presentation Foundation) do utworzenia interfejsu graficznego,
   - SwiPlCs v1.1.60605.0 (wersja 32 bitowa) – biblioteka umożliwiająca połączenie programu napisanego w języku prolog z programem napisanym w języku C#,
   - DataGrid2DLibrary biblioteka dodająca nową kontrolkę (DataGrid2D), która jest wykorzystywana do przedstawienia planszy Rogo.

III. Podział prac:

Autor	Podzadanie
Mateusz Korolow	Zaprojektowanie interfejsu graficznego programu oraz zabezpieczenie go przed wprowadzaniem błędnych danych
Daniel Hildebrand	Połączenie programu napisanego w języku Prolog z interfejsem graficznym aplikacji, implementacja przykładowych puzzli
Tomasz Braczyński	Implementacja logiki programu, testowanie jej oraz praca nad poprawą wydajności
Cała grupa	Rozwiązywanie napotkanych większych problemów, ustalenie założeń projektu oraz przygotowanie dokumentacji, refaktoryzacja kodu

IV. Opis implementacji:

Implementacja solwera puzzli Rogo w tym przypadku opiera się na graficznym interfejsie napisanym w języku C# wspieranym przez technologię Windows Presentation Foundation. Logika napisana jest w języku Prolog korzystając z paradygmatu programowania z ograniczeniami nad zbiorami skończonymi.

Używane w tym punkcie nazwy zmiennych są nazwami, które zostały użyte podczas implementacji „logiki" aplikacji (cały kod „logiki" przedstawiony jest w ostatnim punkcie tejże dokumentacji). W większości nazwy te są logiczne i wskazują na zawarte w zmiennych dane.

1. Dane wejściowe są generowane poprzez program napisany w języku C# na podstawie wprowadzonych przez użytkownika danych, są to:
    - `Szerokosc` – jest to wartość odzwierciedlająca ilość pól mieszczącą się na planszy do gry Rogo w szerz;
    - `Wysokosc` – jw. tylko w odniesieniu do długości planszy;
    - `LiczbaKrokow` – jest to liczba pól, z których składa się lista wynikowa;
    - `Dobrze` – minimalna ilość punktów do uzyskania aby rozwiązać puzzle Rogo;
    - `Najlepiej` – maksymalna ilość punktów możliwa do uzyskania w tym konkretnym puzzle Rogo;
    - `ListaWartosci` – lista przechowująca wartości pól (zaczynając od górnego lewego rogu, w kolejności od lewej do prawej, a następnie wierszami);
    - `Glowa` – opcjonalny element początkowy, umożliwia wyszukiwanie uproszczone polegające na wyszukiwaniu rozwiązań rozpoczynających, a jednocześnie kończących się we wskazanym polu;
2. Dane pomocnicze - nie są podawane podczas wywoływania funkcji, lecz są niezbędna do działania algorytmu, są one generowane (przez język Prolog) w trakcie wykonywania programu. W ich skład wchodzą m.in.:
    - `IloscPol` – wartość tej zmiennej odzwierciedla liczbę pól znajdujących się na planszy, wynosi ona dokładnie `Szerokosc*Wysokosc`;
    - `SciezkaBezGlowy` – używana do sprawdzenia unikalności pól wchodzących w skład rozwiązania, unikamy błędnego rozpoznania (każda ścieżka jest cykliczna, więc funkcja `all_different` zwróciłaby zawsze negatywną odpowiedź);
    - `ZakresIndeksow` – lista przechowująca wartości od 0 do `Szerokosc*Wysokosc`;
    - `ListaNieujemnych` – w tej liście są zawarte informacje o indeksach, dla których wartość przechowywana w `ListaWartosci` jest większa lub równa zero;
    - `ZestawNieujemnych` – jest to `ListaNieujemnych` przekonwertowana na typ Domain, na jej podstawie są generowane elementy wchodzące w skład rozwiązania;
    - `ListaGlow` – analogiczne jak `ListaNieujemnych`, tyle że wartości indeksów są większe od 0;

- `ZestawGlow` – jest to `ListaGlow` przekonwertowana na typ Domain, znacznie ogranicza przestrzeń poszukiwań, dzięki tej zmiennej każde rozwiązanie zaczyna się od pola, w którym jest dodatnia wartość;
- `Suma` – jest to suma wartości pól wchodzących w skład rozwiązania.
- `LK` – to długość listy wynikowej, gdzie element pierwszy jest dodany jako ostatni
- `Szer` - jest używane w sprawdzaniu sąsiedniości elementów, wynosi Szerokosc-1

3. Dane wyjściowe składają się z następujących po sobie N list, gdzie N jest liczbą odnalezionych rozwiązań. Pojedyncze rozwiązanie jest reprezentowane przez:
   - `ListaIndeksow` - jest ostatnim argumentem podawanym podczas wywoływania funkcji `szukaj` lub `szukaj_po_glowie` (odpowiadającej za wyszukiwanie uproszczone).. `ListaIndeksow` jest reprezentowana przez listę, której pierwszy i ostatni element jest taki sam. To właśnie te dane reprezentują znalezione rozwiązanie, które ostatecznie wyświetlane jest przez interfejs graficzny użytkownika.
4. Programowanie z ograniczeniami jest paradygmatem programowania, w którym relacje między zmiennymi podaje się w formie ograniczeń. Ograniczenia określają właściwości znalezionego rozwiązania. Logika programu korzysta z następujących ograniczeń:
   - Zapewnienie odpowiedniej długości ścieżki równej ilości kroków+1 – funkcja `inicjalizuj_liste(LK, ListaIndeksow)`.
   - Zapewnienie, że ostatni oraz pierwszy element rozwiązania (tj. znalezionej ścieżki) jest taki sam, co powoduje, że prawidłowe rozwiązanie jest w formie ścieżki-cyklu – funkcja `inicjalizuj_liste(LK, ListaIndeksow)`.
   - Wszystkie elementy (indeksy) rozwiązania muszą mieścić się w zakresie <0; szerokość*wysokość-1>.
   - Wybranie indeksów pól niezerowych w celu przyspieszenia znalezienia rozwiązania, dzięki temu unikamy sprawdzania czy pola ścieżki nie należą do pól nieprzechodnich – funkcja `ListaIndeksow ins ZestawNieujemnych`.
   - Wybranie indeksów pól o wartościach dodatnich w celu przyspieszenia znalezienia rozwiązania oraz zapewnienie, że element pierwszy i ostatni znajduje się w tym zbiorze, pozwala to na znaczne ograniczenie liczby rozwiązań symetrycznych – funkcja `wybierz_niezerowe(ListaWartosci, ZakresIndeksow, ListaGlow)`.
   - Wszystkie pola w znalezionym rozwiązaniu muszą być różne – funkcja `all_different(SciezkaBezGlowy)`.
   - Zapewnienie, że suma wartości pól wchodzących w skład rozwiązania mieści się w zakresie `<dobry_wynik; najlepszy_wynik>` – funkcja `Suma#>=Dobrze, Suma#=<Najlepiej`.
   - Sprawdzenie czy wszystkie pola są sąsiednie poprzez wykonanie obliczeń: numer_indeksu modulo szerokość = X, jeśli X:
     - jest równe 0 to takie pole znajduje się na początku wiersza, więc sąsiednimi polami są pola po prawej, powyżej/poniżej,
     - jest równe szerokość – 1 to takie pole znajduje się na końcu wiersza, więc sąsiednimi polami są pola po lewej, powyżej/poniżej,
     - jest różne od powyższych wartości to sąsiednimi polami są pola na górze/dole oraz po lewej/prawej.

     Pola są sąsiednie jeśli ich indeksy różnią się o:

     - 1 - w takim przypadku pola leżą po swojej lewej/prawej stronie,
     - Szerokość - w takim przypadku pola leżą powyżej/poniżej siebie.

     Funkcją odpowiedzialną za te czynności jest - `czy_sasiednie(ListaIndeksow, Szerokosc, Szer)`.

V. Użytkowanie i testowanie systemu:

1. Użytkowanie systemu.

Aby poprawnie używać tego programu należy postępować zgodnie z poniższymi krokami

a) Wybranie jednego z dwóch sposobów tworzenia puzzli:
   1. Wybranie puzzli z przygotowanych przykładów
   2. Ręczne wpisanie wszystkich wartości tj.:
      - szerokości oraz wysokości tabeli reprezentującej puzzle,
      - dobrego oraz najlepszego wyniku
      - ilości kroków do wykonania,
      - wartości w tabeli reprezentujących poszczególne pola puzzli

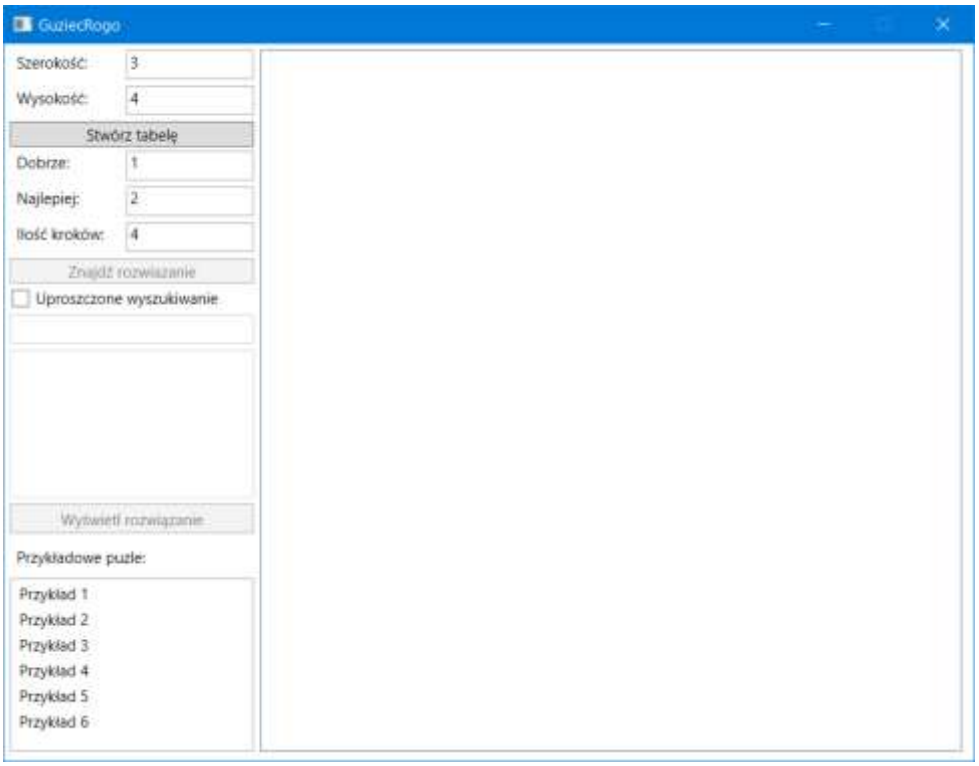
b) Opcjonalnie, zaznaczenie pola wyboru „Uproszczone wyszukiwanie", a następnie podanie numeru indeksu pola, z którego ma rozpoczynać się znalezione rozwiązanie.

c) Naciśniecie przycisku „znajdź rozwiązanie" uruchamia logikę programu oraz przystępuje do wyszukiwania rozwiązań

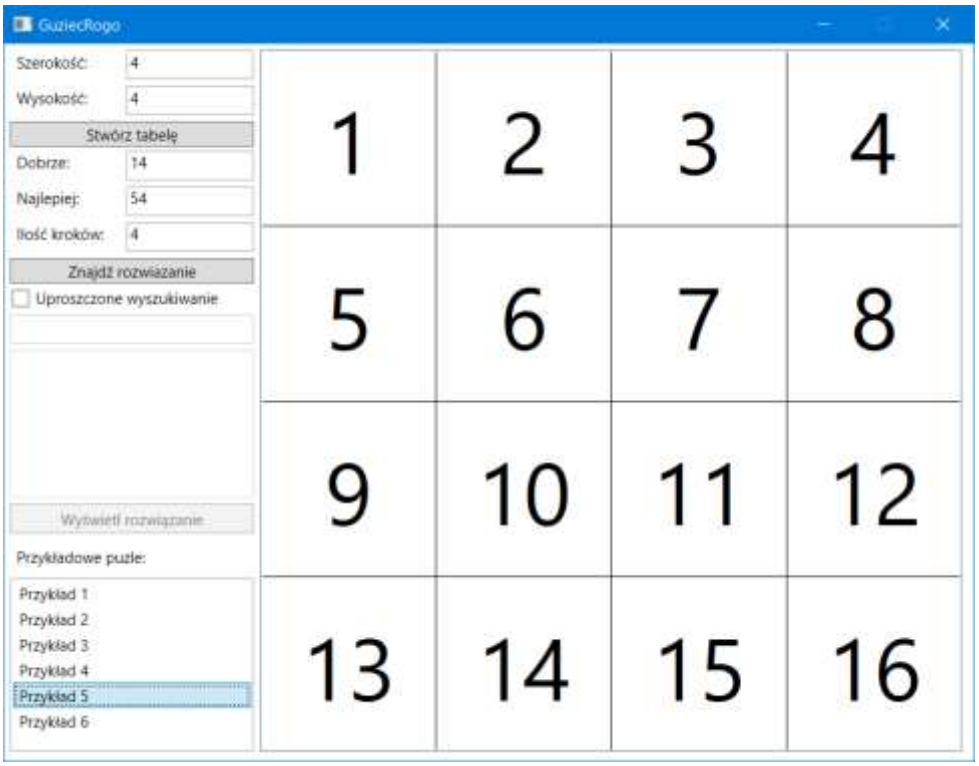
d) Znalezione rozwiązania zostają wyświetlone poniżej przycisku „znajdź rozwiązanie"

e) Użytkownik aplikacji może wybrać jedno ze znalezionych rozwiązań i wyświetlić je poprzez naciśnięcie przycisku „wyświetl rozwiązanie"

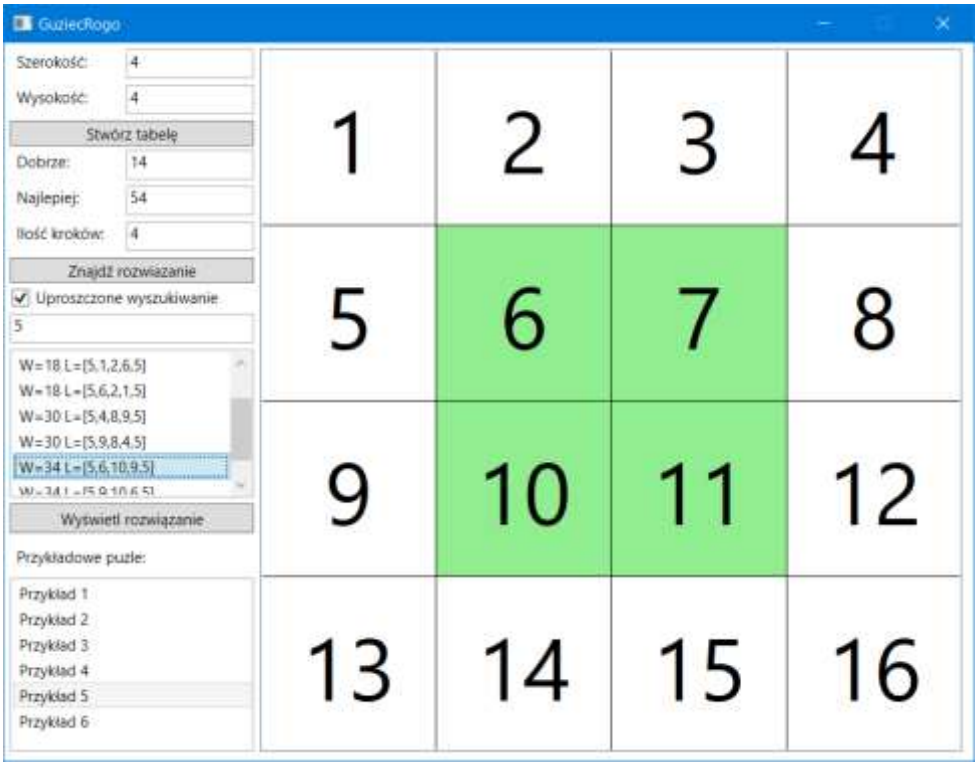
Interfejs graficzny dostępny dla użytkownika składa się z 2 głównych części. Pierwszą z nich jest lewa strona okna odpowiadająca za wprowadzanie danych konfiguracyjnych oraz za sterowanie wyświetlaniem wyników. Resztę okna zajmuje tworzona dynamicznie tablica reprezentująca planszę do gry Rogo, do której wprowadzane są wartości poszczególnych pól.



Rys. 1 - interfejs aplikacji widoczny natychmiast po uruchomieniu



Rys. 2- interfejs użytkownika widoczny po załadowaniu przykładu piątego.



Rys. 3 - puzzle zostały rozwiązane przy użyciu "uproszczonego wyszukiwania" oraz zostało wyświetlone jedno ze znalezionych rozwiązań.

2. Testowanie aplikacji.

Proces testowania systemu był złożony, testowany był zarówno interfejs by zapobiec wprowadzaniu niepoprawnych danych, jak i logika programu aby znaleźć jak najmniejsze (często trudne do zauważenia) błędy i/lub problemy. Końcowe testy finalnej wersji odbywały się podstawie zaimplementowanych przykładów. Testy jednostkowe poszczególnych funkcji napisanych w języku Prolog były testowane także pod względem wydajności, bardzo pożyteczną funkcją okazała się funkcja `time()`. Zwraca ona czas jak i ilość operacji wykonanych przez daną funkcję, dzięki niej możliwe było sprawdzenie dosłownie każdej linii kodu. Dobrym przykładem jest zmiana funkcji `all_distinct` na `all_different`, spowodowała ona około 4 krotny wzrost wydajności, poniżej przedstawione są dane zwracane przez funkcję `time`:

Funkcja	**Liczba wnioskowań (Inferences)**	**Czas procesora**	**Czas wykonywania [s]**	**Średnie obciążenie procesora**	**Liczba wnioskowań na sekundę**
`all_different`	5,442,375	0.344	0.453	76%	15,832,364
`all_distinct`	19,311,257	1.703	1.777	96%	11,338,720

Powyższe wyniki zostały uzyskane dla przykładu piątego (zaimplementowanego w aplikacji) uruchomionego na systemie Windows 10 wspieranego przez 8GB pamięci operacyjnej oraz przy użyciu 8 wątkowego procesora Intel Core i7. Niestety, biblioteka SwiPlCs nie wykorzystuje pełnej mocy języka Prolog, zauważalny jest spadek wydajności oraz niższe obciążenie procesora – wykorzystywany jest tylko jeden wątek. Ta biblioteka nie pozwala również na asynchroniczną pracę, wywołanie funkcji `PlQuery()` powoduje rozpoczęcie poszukiwań wszystkich rozwiązań, co przy rozbudowanych przykładach powoduje bardzo długi czas oczekiwania, podczas którego nie są dostępne żadne informacje zwrotne., nie jest możliwe też Wszystko to przekłada się na nie najlepszą wydajność aplikacji.

VI. Tekst programu:

```
1.    ?- use_module(library(clpfd)).
2.    /*
3.    zapewnienie odpowiedniej długości listy oraz warunku
      OstatniElement=Glowa
4.    */
5.    inicjalizuj_liste(Dlugosc, JakasLista):-
6.          last(JakasLista, Element),
7.          nth0(0,JakasLista, Element),
8.          length(JakasLista, Dlugosc).
9.
10.
11.   /*
12.   sprawdzenie czy pola w liście znajdują się obok siebie;
13.   pole poniżej/powyżej rożni się o szerokość;
14.   pole po lewej/prawej różni się o 1;
15.   */
16.   czy_sasiednie([H1, H2], Szerokosc, Szer):-
17.         S1 is H2 - Szerokosc,   /* Góra */
18.         S2 is H2 - 1,           /* Lewo */
19.         S3 is H2 + 1,           /* Prawo */
20.         S4 is H2 + Szerokosc,   /* Dół */
21.         X is H1 mod Szerokosc,
22.         (X = 0 -> member(H1, [S1, S2, S4]);
23.         X = Szer -> member(H1, [S1, S3, S4]);
24.         X \= 0 -> member(H1, [S1, S2, S3, S4])),
25.         !.
26.   czy_sasiednie([H1, H2 | T], Szerokosc, Szer):-
27.         S1 is H2 - Szerokosc,   /* Góra */
28.         S2 is H2 - 1,           /* Lewo */
29.         S3 is H2 + 1,           /* Prawo */
30.         S4 is H2 + Szerokosc,   /* Dół */
31.         X is H1 mod Szerokosc,
32.         (X = 0 -> member(H1, [S1, S2, S4]);
33.         X = Szer -> member(H1, [S1, S3, S4]);
34.         X \= 0 -> member(H1, [S1, S2, S3, S4])),
35.         czy_sasiednie([H2 | T], Szerokosc, Szer).
36.
37.
38.   /*
39.   sumowanie wartości wybranych indeksów
40.   */
41.   sprawdz_sume([],_,0):-
42.         abort.
43.   sprawdz_sume([H | T], ListaWartosci, Suma):-
44.         (length(T, 0),
45.         nth0(H, ListaWartosci, Glowa),
46.         Suma is Glowa
47.         ;
48.         sprawdz_sume(T, ListaWartosci, SumaOgona),
49.         nth0(H, ListaWartosci, Glowa),
50.         Suma is SumaOgona + Glowa).
51.
```

```
52.
53.   /*
54.   Funkcja filtrująca indeksy, dla których wartość drugiej listy
      przyjmuje wartość dodatnią
55.   */
56.   wybierz_niezerowe([], [], []).
57.   wybierz_niezerowe([H | T], [_ | T1], S) :-
58.         H=<0,
59.         wybierz_niezerowe(T, T1, S).
60.   wybierz_niezerowe([H | T], [H1 | T1], [H1 | S]) :-
61.         H>0,
62.         wybierz_niezerowe(T, T1, S).
63.
64.
65.   /*
66.   Funkcja filtrująca indeksy, dla których wartość drugiej listy
      przyjmuje wartość dodatnią lub zerową
67.   */
68.   wybierz_nieujemne([], [], []).
69.   wybierz_nieujemne([H | T], [_ | T1], S) :-
70.         H<0,
71.         wybierz_nieujemne(T, T1, S).
72.   wybierz_nieujemne([H | T], [H1 | T1], [H1 | S]) :-
73.         H>=0,
74.         wybierz_nieujemne(T, T1, S).
75.
76.
77.   /*
78.   funkcja do konwersji list na domain
79.   */
80.   list_to_domain([H], H..HT) :-
81.         HT is H + 0.
82.   list_to_domain([H | T], '\\/'(H .. HT, TDomain)) :-
83.         HT is H + 0,
84.         list_to_domain(T, TDomain).
85.
86.
87.   /*
88.   Główna funkcja wywoływana przez GUI
89.   */
90.   szukaj(Szerokosc, Wysokosc, LiczbaKrokow, Dobrze, Najlepiej,
      ListaWartosci, ListaIndeksow):-
91.         IloscPol is Szerokosc*Wysokosc,
92.         LK is LiczbaKrokow+1,/*LK to długość listy wynikowej,
            gdzie element pierwszy jest dodany jako ostatni*/
93.         Szer is Szerokosc-1,/*Szer jest używane w sprawdzaniu
            sąsiedniości elementów*/
94.         inicjalizuj_liste(LK, ListaIndeksow),
95.         nth0(0, ListaIndeksow, _, SciezkaBezGlowy),
96.         MaxIndeks is IloscPol-1,
97.         numlist(0, MaxIndeks, ZakresIndeksow),
98.         wybierz_nieujemne(ListaWartosci, ZakresIndeksow,
            ListaNieujemnych),
99.         list_to_domain(ListaNieujemnych, ZestawNieujemnych),
```

```
100.        ListaIndeksow ins ZestawNieujemnych,/*zapewnienie, że
      elementy znalezionego rozwiązania nie będą zawierać
      ujemnych (nieprzechodnich) pól*/
101.        wybierz_niezerowe(ListaWartosci, ZakresIndeksow,
            ListaGlow),
102.        list_to_domain(ListaGlow, ZestawGlow),
103.        nth0(0,ListaIndeksow, Glowa),
104.        Glowa in ZestawGlow,
105.        all_different(SciezkaBezGlowy),
106.        sprawdz_sume(SciezkaBezGlowy, ListaWartosci, Suma),
107.        Suma#>=Dobrze,
108.        Suma#=<Najlepiej,
109.        czy_sasiednie(ListaIndeksow, Szerokosc, Szer).
110.
111.
112.  /*
113.  Dodatkowa funkcja wywoływana przez GUI umożliwia wyszukiwanie
      ścieżek rozpoczynających się z podanego pola
114.  */
115.  szukaj_po_glowie(Szerokosc,  Wysokosc,  LiczbaKrokow,  Dobrze,
      Najlepiej, ListaWartosci, Glowa, ListaIndeksow):-
116.        IloscPol is Szerokosc*Wysokosc,
117.        LK is LiczbaKrokow+1,/*LK to długość listy wynikowej,
            gdzie element pierwszy jest dodany jako ostatni*/
118.        Szer is Szerokosc-1,/*Szer jest używane w sprawdzaniu
            sąsiedniości elementów*/
119.        inicjalizuj_liste(LK, ListaIndeksow),
120.        nth0(0, ListaIndeksow, _, SciezkaBezGlowy),
121.        MaxIndeks is IloscPol-1,
122.        numlist(0, MaxIndeks, ZakresIndeksow),
123.        wybierz_nieujemne(ListaWartosci, ZakresIndeksow,
            ListaNieujemnych),
124.        list_to_domain(ListaNieujemnych, ZestawNieujemnych),
125.        ListaIndeksow ins ZestawNieujemnych,/*zapewnienie, że
            elementy znalezionego rozwiązania nie będą zawierać
            ujemnych (nieprzechodnich) pól*/
126.        nth0(0, ListaIndeksow, Glowa),
127.        all_different(SciezkaBezGlowy),
128.        sprawdz_sume(SciezkaBezGlowy, ListaWartosci, Suma),
129.        Suma#>=Dobrze,
130.        Suma#=<Najlepiej,
131.        czy_sasiednie(ListaIndeksow, Szerokosc, Szer).
```