12.06.2020

Piotr Kopycki
Patryk Bryła
Adrian Lorenc
SiL-L3-Z1

Sztuczna inteligencja - laboratorium

Sprawozdanie końcowe wariant 1

Peaceful chess queen armies

I Opis zadania

Celem naszego zadania było rozwiązanie problemu peaceful chess queen armies jako problemu spełniania ograniczeń (problem CSP). Sednem problemu pokojowych armii królowych jest takie ułożeń dwóch armii (czarnej i białej) złożonych z x królowych, na szachownicy o rozmiarach y na y, w taki sposób by figury z przeciwnych armii nie atakowały się na żadnej pozycji. Wszystko musi odbywać się w zgodzi z zasadami gry w szachy.

Dane wejściowe do systemu to tylko rozmiar armii oraz boku szachownicy (szachownica jest kwadratem), więc muszą to być oczywiście dodatnie liczby całkowite. Ograniczeniem rozmiaru planszy od dołu (najmniejsza wartość) jest tylko rozwiązywalność problemu. Najmniejszy rozmiar planszy dla którego problem da się rozwiązać to 3x3. Górną granicę określa czas obliczeń, przyjęliśmy w naszych warunkach 10x10, gdzie czas jest już dość długi. Rozmiar armii są ograniczane tylko przez rozwiązywalność. Nie da się na planszy 2x2 ustawić 100 figur.

Wyniki mają postać planszy o danym rozmiarze z rozłożonymi na niej, w zgodzie z zasadami problemu, figurami. W przypadku braku rozwiązania, zwracany jest odpowiedni komunikat.

II Założenia realizacyjne

1. Założenia dodatkowe (opcjonalne)

1. graficzna reprezentacja wyników (rysowanie planszy i figur)
2. graficzny interfejs użytkownika pobierający dane wejściowe

2. Metody, strategie oraz algorytmy wykorzystywanie do rozwiązania zadania

W rozwiązaniu naszego problemu używamy paradygmatu programowania ograniczeń (CP - Constraint programming). Najważniejszą metodą rozwiązująca nasz problem z użyciem Gecode jest konstruktor, którego kod zaprezentowany jest w punkcie 7. Funkcja ta przyjmuje jako argument wejściowy SizeOptions. Na początku przypisywane są wartości zmiennym:

- size_of_board - rozmiar planszy,
- unattacked_squares - nieatakowane pozycje,
- white_squares, black_squares - pola atakowane przez odpowiednio białe i czarne,
- white_placement, black_placement - pozycja odpowiednio białych i czarnych,
- nr_queens_placed - liczba rozłożonych królowych.

Następnie ustalane są podstawowe zasady modelu dla każdego pola planszy, są to:

- białe i czarne nie mogą być na tych samych polach,

- czarne nie mogą być na polach atakowanych przez białe,
- nieatakowane pozycje przydzielane są czarnym.

Potem zmienna optymalizująca dla białych i czarnych łączona jest z liczbą królowych. Następnie łączona jest liczność nieatakowanych pól z liczbą białych królowych. Na końcu ustawiane są wybory rozgałęzień.

W naszej pracy posłużyliśmy się paroma gotowymi przykładami, które musieliśmy w większym lub mniejszym stopniu dostosować do naszych założeń:

1. Gecode - użyty został przykładowy kod z oficjalnej strony [4] służący do rozwiązania podobnego problemu nie-pokojowych królowych.
2. Do rysowania szachownicy użyliśmy kodu ze źródła [3], kod ten tworzył szachownicę o sztywnym rozmiarze 8x8, musieliśmy więc zmodyfikować go tak by było możliwe tworzenie różnej ilości pól oraz odpowiednie skalowanie okna do rozmiaru planszy.

3. Języki programowania, narzędzia informatyczne i środowiska używane do implementacji systemu

Język programowania: C++
Narzędzia:
Visual Studio Community 2019
Visual Studio 2017
Biblioteki:
freeglut 3.0.0-1 (wersja 64 bitowa) - graficzna reprezentacja wyników
Gecode 6.2.0 (wersja 64 bitowa) - przedstawienie problemu jako rozwiązanie problemu CSP

III Podział prac

Autor	Podzadanie
Patryk Baryła	Rozwiązanie problemu z użyciem Gecode
Piotr Kopycki	Graficzna reprezentacja wyników
Adrian Lorenc	Interfejs użytkownika

IV Opis implementacji

1. Struktury danych wykorzystywanie w programie

- IntSet* A - pozycja pionka na kwadratowej szachownicy,
- int pos(int i, int j, int n) - pozycja pionka na szachownicy.

Klasy:

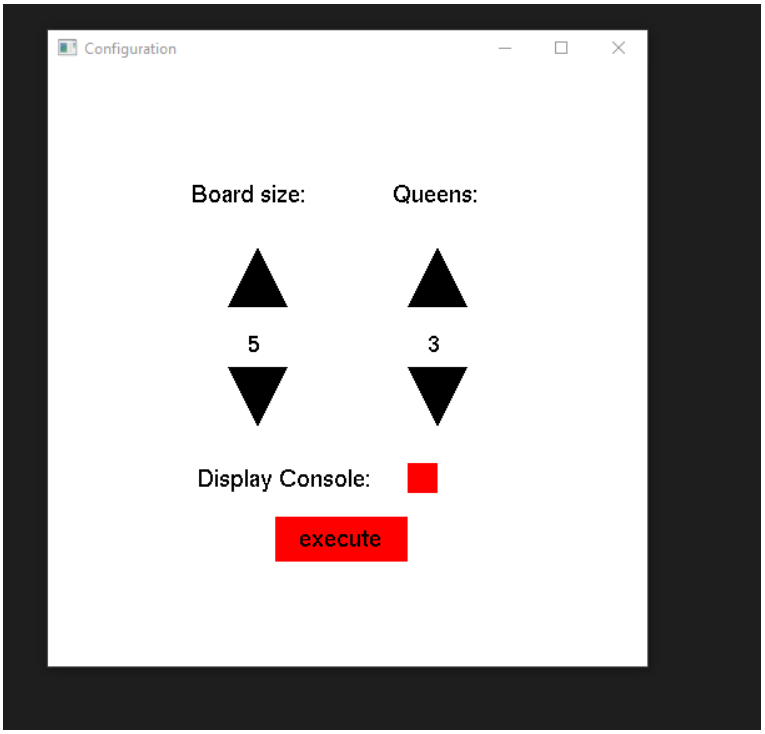
- QueenArmies - klasa mają na celu rozmieszczenie białych i czarnych królowych na podanej planszy w sposób pokojowy z maksymalną możliwą liczbą rozwiązań,
  - const int size_of_board - rozmiar szachownicy,
  - SetVar unattacked_squares - nieatakowane pola,
  - SetVar white_squares i SetVar black_squares - pola atakowane przez białe i czarne królowe,
  - BoolVarArray white_placement i BoolVarArray black_placement - pozycja białych i czarnych,
  - IntVar nr_queens_placed - liczba rozłożonych królowych.
- QueenArmies::QueenBranch - rozgałęzienie próbujące wypełnić szachowicę jak najweksza możliwą liczbą królowych.

2. Podprogramy zdefiniowane w programie (funkcje, procedury, metody lub predykaty)

- int pos(int i, int j, int n) - argumenty to numer kolumny , numer wiersza i rozmiar szahcownicy , zwraca pozycję na liście.
- QueenArmies::QueenArmies (const SizeOptions opt) - konstruktor przyjmuje jako argument SizeOption,
- QueenArmies::QueenArmies (QueenArmies s) - kontrsuktor dla klonowania,
- QueenArmies::QueenArmies (QueenArmies s) - zwraca kopię podczas klonowania,
- virtual IntVar cost(void) const - zwraca koszt rozwiązania,
- virtual bool QueenArmies::QueenBranch::status (const Space home) const - atrybut wejściowy to przestrzeń obliczeniowa, zwraca True jeżeli bracher ma alternatywy,
- virtual Gecode::Choice* QueenArmies::QueenBranch::choice (Space home) - atrybut wejściowy to przestrzeń obliczeniowa, zwraca wybór,
- virtual Choice* QueenArmies::QueenBranch::choice (const Space ,Archive e) - atrybuty wejściowe to przestrzeń obliczeniowa oraz archiwum rozwiązań, zwraca wybór,
- virtual ExecStatus QueenArmies::QueenBranch::commit (Space home, const Gecode::Choice _c,unsigned int a) - atrybuty wejściowe to przestrzeń obliczeniowa, wybór, oraz alternatywa, potwierdza wybór i alternatywę,
- virtual Actor* copy(Space home) - atrybut to przestrzeń obliczeniowa kopiuje rozgałęzienie podczas klonowania,
- static void post(QueenArmies home) - postuje rozgałęzienie argumentem postowany jest rozgałęzienie,
- virtual size_t dispose(Space) - usuwa rozgałęzienie i zwraca jego rozmiar.

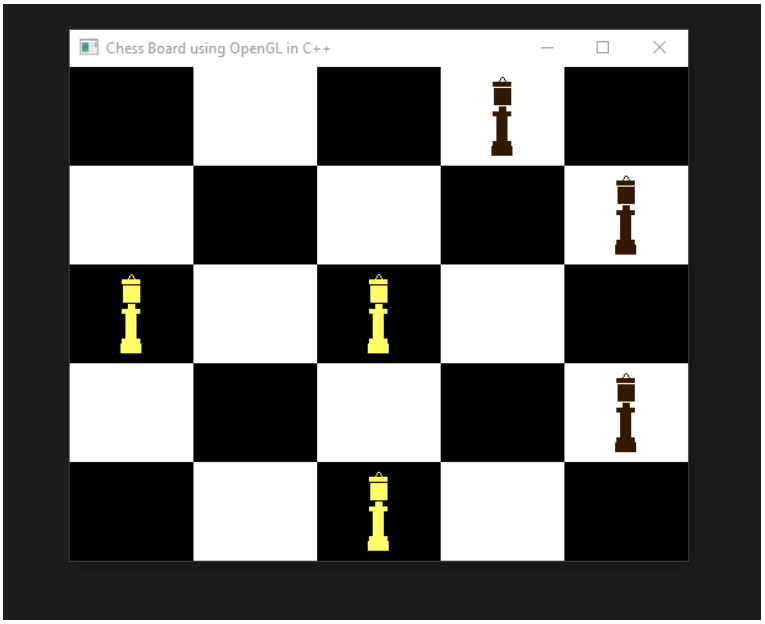
V Użytkowanie i testowanie systemu

Użytkowanie programu rozpoczynamy od podania za pomocą graficznego interfejsu danych potrzebnych do obliczeń tzn: ilość pól z których składa się każdy wiersz szachownicy oraz ile pionków ma posiadać każda armia królowych.



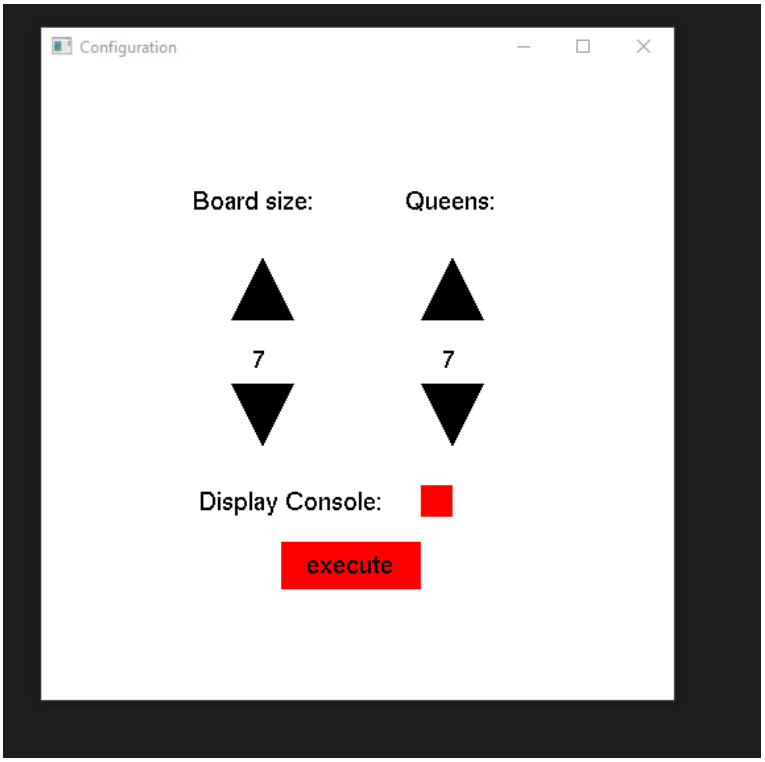
Rys. 1: Dane wejściowe dla planszy o wymiarach 5x5 i 3 pionków w każdej armii

Po podaniu danych rozpoczynamy wykonywanie obliczeń klikając w przycisk execute. Zaznaczenie pola Dispaly Console powoduje uzyskanie w oknie konsoli dodatkowych rozwiązań dla planszy o zadanej wielkości. Po wykonaniu obliczeń program wyświetla okno z graficzną reprezentacją przykładowego rozwiązania dla problemu o zadanych parametrach.

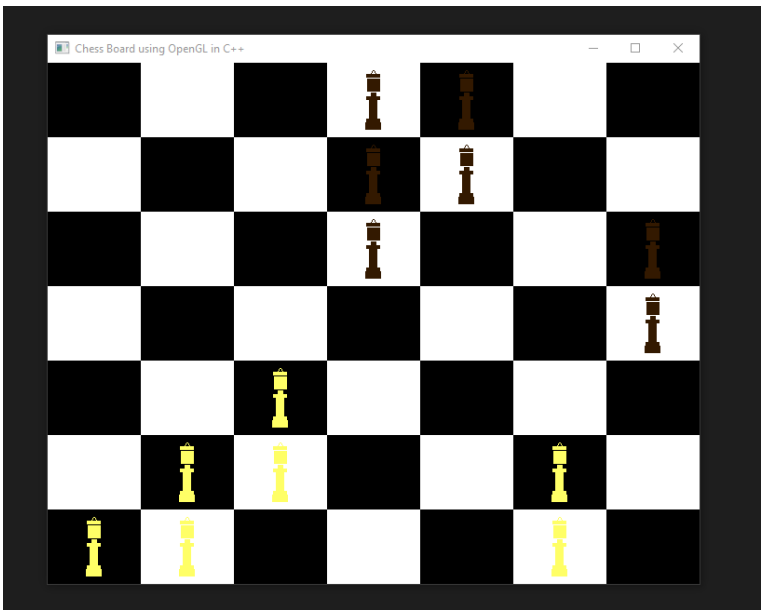


Rys. 2: Wynik obliczeń dla planszy o wymiarach 5x5 i 3 pionków w każdej armii

Zamknięcie okna z graficznym wynikiem spowoduje ponowne otwarcie okna interfejsu i umożliwi ponowne wykonanie całej procedury.



Rys. 3: Dane wejściowe dla planszy o wymiarach 7x7 i 7 pionków w każdej armii



Rys. 4: Wynik obliczeń dla planszy o wymiarach 7x7 i 7 pionków w każdej armii

Zamknięcie okien powoduje zakończenie pracy programu.

VI Literatura

[1] Meissner A., Sztuczna Inteligencja Podstawy programowania z ograniczeniami., http://users.man.poznan.pl/ameis/Si-CSP.pdf (dostęp 13.06.2020)

[2] Schulte Ch., Tack G., Lagerkvist M., Modeling and Programming with Gecode., https://www.gecode.org/doc-latest/MPG.pdf (dostęp 13.06.2020)

[3] https://www.openglprojects.in/2018/08/opengl-chess-board-in-c-with-source-code_26.htmlgsc.tab=0

[4] https://www.gecode.org/doc/6.1.1/reference/queen-armies_8cpp_source.html

VII Tekst programu

```
QueenArmies(const SizeOptions& opt) :
            IntMaximizeScript(opt),
            size_of_board(opt.size()),
            unattacked_squares(*this, IntSet::empty, IntSet(0, size_of_board*size_of_board)),
            white_squares(*this, IntSet::empty, IntSet(0, size_of_board*size_of_board)),
            black_squares(*this, IntSet::empty, IntSet(0, size_of_board*size_of_board)),
            white_placement(*this, size_of_board*size_of_board, 0, 1),
            black_placement(*this, size_of_board*size_of_board, 0, 1),
            nr_queens_placed(*this, 0, size_of_board*size_of_board)
    {
            // Basic rules of the model
            for (int i = size_of_board * size_of_board; i--; ) {
                    // white_placement[i] means that no blacks are allowed on A[i]
                    rel(*this, white_placement[i] == (unattacked_squares || A[i]));
                    // Make sure blacks and whites are disjoint.
                    rel(*this, !white_placement[i] || !black_placement[i]);
                    // If i in unattacked_squares, then black_placement[i] has a piece.
                    rel(*this, black_placement[i] == (singleton(i) <= unattacked_squares));
            }

            // Connect optimization variable to number of pieces
            linear(*this, white_placement, IRT_EQ, nr_queens_placed);
            linear(*this, black_placement, IRT_EQ, nr_queens_placed);

            // Connect cardinality of unattacked_squares to the number of white pieces.
            IntVar unknowns = expr(*this, cardinality(unattacked_squares));
            rel(*this, nr_queens_placed <= unknowns);
            linear(*this, white_placement, IRT_EQ, unknowns);
```

```
        if (opt.branching() == BRANCH_NAIVE) {
            branch(*this, white_placement, BOOL_VAR_NONE(), BOOL_VAL_MAX());
            branch(*this, black_placement, BOOL_VAR_NONE(), BOOL_VAL_MAX());
        }
        else {
            QueenBranch::post(*this);
            assign(*this, black_placement, BOOL_ASSIGN_MAX());
        }
    }
```