PROLOG – SUDOKU

Referat

Trepa Grzegorz

IV Informatyka Stosowana

PWSZ Tarnów

Spis treści:

Opis pomysłu

W związku z dużą złożonością kombinatoryczną wszystkich możliwości wstawienia w każde pole cyfry, rozwiązanie takiego zadania jak Sudoku pochłonęło by dużo zasobów i czasu komputera. Postanowiłem więc zaimplementować algorytm napisany przeze mnie w języku C++ w języku PROLOG.

Pierwszym pomysłem w ograniczeniu złożoności obliczeniowej będzie nie tylko prosta reprezentacja danych typu:

KOD	
	:- dynamic(element/3). % nr: wiersz, kolumna, wartość

Dodatkowo będą zapisywane dynamicznie predykaty dla możliwości wstawienia elementu w każdej komórce, oraz możliwości wystąpienia konkretnej wartości w konkretnej kolumnie, kwadracie (3x3), wierszu:

KOD	
	:- dynamic(wiersz/2). % nr, wartość ***:- dynamic(kolumna/2). % nr, wartość*** ***:- dynamic(kwadrat/2). % nr wartość*** ***:- dynamic(mozliwosc/4). % wiersz, kolumna, kwadrat, wartość***

Wstawienie jakiejkolwiek wartości, będzie równocześnie ograniczać możliwości w wierszu, kolumnie, kwadracie (3x3) oraz możliwości poszczególnych komórek. Dzięki temu uniknie się próby wybierania takich wartości które z logicznego punktu widzenia już nie powinny być brane pod uwagę.

Jednak takie ograniczenia wg mnie nie były by wystarczające a złożoność obliczeniowa i tak była by spora chociaż o wiele mniejsza. W związku z tym postanowiłem zaimplementować 3 algorytmy których zadaniem było by wstawianie pewniaków. Algorytmy te to trzy podstawowe sposoby ręcznego rozwiązywania sudoku polegające na sprawdzaniu dodatkowych ograniczeń typu dwa wiersze i dwie kolumny blokują wstawienie wartości w innych polach, lub w danym wierszu, kwadracie, kolumnie tylko w określonym miejscu można wstawić wartość, lub też w konkretnym miejscu istnieje tylko jedna jedyna możliwość wstawienia wartości, pozostałe są z różnych powodów polokowane. Opis metod dokładniej w kolejnych działach referatu.

Istnieje klika poziomów sudoku (Łatwe, Średnie, Trudne i Piekielnie Trudne). Pierwsze trzy poziomy prawie w 99% można rozwiązać tylko tymi trzema algorytmami, jednak czasem to nie wystarcza, zwłaszcza przy ostatnim poziomie. Istnieją inne metody rozwiązywania, jednak ich wykorzystanie już nie daje takich efektów czasowych, w związku z tym w momencie nie rozwiązania zadania tymi trzema metodami, dla pierwszego pustego miejsca zostaną wypróbowane wszystkie możliwości w taki sposób że po włożeniu do sudoku pierwszej możliwości rekurencyjnie zostanie wywołany predykat liczenia sudoku, dzięki czemu znowu zadziałają algorytmy rozwiązywania (opisane zostało w dziale szczekania).

Predykat sudoku jako główny predykat

Istnieją dwa główne predykaty sudoku/1 i sudoku/2. Do używania powinno się korzystać z sudoku/1, drugi jest wykorzystywany przy rekurencyjnym wywoływaniu.

KOD

```
sudoku(L):- nl,nl,nl,
    not(czyszczenie_strzelen), not(czyszczenie),
    write('SUDOKU ZADANE: '),nl,nl,
    wypiszMacierz(L),nl,
    sudoku(L,W), liczba_elementow(N),stan_poczatkowy(M),
    A is N - M, retract(stan_poczatkowy(M)),
    write('Znaleziono elementow: '),write(A),nl,nl,
    write('SUDOKU WYNIKOWE: '),nl,nl,
    wypiszMacierz(W),
    not(czyszczenie),
    not(czyszczenie_strzelen),!.
```

Pomysłem w robieniu sodoku było by każdy predykat bez względu czy zakończył się pomyślnie czy też nie kończył się błędem (zwracał **false**), i był wywoływany w negacji ***not( )***.

Sudoku dostaje tutaj macierz w postaci listy list o wymiarach 9x9. Predykat ten głównie zajmuje się ładnym wyświetleniem wyników oraz wyczyszczeniem pamięci pod koniec działania.

Na początku wypisuje kilka znaków nowej lini, czyści ewentualne dynamiczne predykaty jeśli jakieś pozostały (zapamiętane strzelenia oraz możliwości, kolumny, wiersze …). Wypisuje macierz zadaną, Wyznacza rozwiązanie sudoku i drukuje wynik. Predykat kończy się wyczyszczeniem pamięci.

Właściwym predykatem liczącym jest :

KOD

```
sudoku(L,W):-
    inicjalizacja(L),!,
    pamiec_stanu_poczatkowego,
    algorytmy,
    strzelenie_liczb,
    rekonstrukcja(W).
```

On wywołuje kolejne predykaty. Najpierw inicjalizuje podaną listę na zestaw predykatów, następnie tylko do wyświetlenia elementów zapamiętuje przy pierwszym uruchomieniu ile wartości znajdowało się w sudoku.
Kolejnym krokiem jest zastosowanie algorytmów liczący, następnie algorytmu przepatrującego dla pierwszego nie zapełnionego pola wszystkich możliwości (strzelanie liczby) i końcowa rekonstrukcja wyniku.

Pamięć stanu początkowego jest realizowana następującym predykatem :

KOD	

```
pamiec_stanu_poczatkowego:-
        not(stan_poczatkowy(_)),
        liczba_elementow(X),
        assert(stan_poczatkowy(X)).

pamiec_stanu_poczatkowego:-
        stan_poczatkowy(_).
```

Jeżeli nie ma stanu początkowego to zlicza elementy a następnie zapamiętuje wartość, w przeciwnym wypadku nic nie robi – predykat drugi zrobiony w celu zwrócenia ***true*** w każdym wypadku.

Pamięć użyta na zapisanie tego stanu zostaje zwolniona w predykacie ***sudoku\1.***

Inicjalizacja danych z listy na zestaw informacji

Inicjalizacja listy na zestaw informacji jest realizowana następującym predykatem:

KOD	

```
inicjalizacja(L):-
        not(L = []),
        not(generuj_mozliwosci),!,
        not(generuj_pomoc),!,
        not(wstawiaj_dane(L,1)).
```

Na początku generowane są wszystkie możliwości nie patrząc na dane wejściowe czyli coś w formie macierzy 3D (9x9x9). Następnie generuje pomoc do algorytmów (w celu przyspieszenia) wszystkie kombinacje dla wierszy, kolumn i kwadratów (3x3) opisane poniżej. Jeśli macierz L jest pusta to nie inicjalizuje niczego.

KOD	

```
generuj_mozliwosci:-
        cyfra(A),cyfra(B),cyfra(C),
        KW is floor((A-1)/3)*3+ floor((B-1)/3)+1,
        assert(mozliwosc(A,B,KW,C)),
        fail.
```

Generowanie możliwości robi wszystkie kombinacje wartości 1 – 9 dodatkowo w celu przyspieszenia późniejszego algorytmu wylicza w którym kwadracie znajduje się wartość i dla tych wartości alokuje pamięć. Podobnie działa generowanie kombinacji w wierszach, kwadratach i kolumnach przedstawione poniżej:

KOD	

```
generuj_pomoc:-
        cyfra(A),cyfra(B),
        assert(wiersz(A,B)),
        assert(kolumna(A,B)),
        assert(kwadrat(A,B)),
        fail.
```

Jak można zaobserwować predykaty te, podobnie jak kolejny kończą się zawsze zwróceniem ***false***.

Poniższe predykaty wstawiają już podaną macierz (listę list) usuwając możliwości tworząc w ten sposób coraz większe ograniczenia. Samo wstawianie pojedynczej wartości opisane zostanie w kolejnym dziale.

KOD

```
wstawiaj_dane([H|T],W):-
        B is W + 1,
        not(wstawiaj_wiersz(H,W,1)),
        wstawiaj_dane(T,B),!.


wstawiaj_wiersz([H|T],W,K):-
        H > 0,
        B is K + 1,
        wstawiaj_wartosc(H,W,K),!,
        wstawiaj_wiersz(T,W,B).


wstawiaj_wiersz([H|T],W,K):-
        H =:= 0,
        B is K + 1,
        wstawiaj_wiersz(T,W,B).
```

Pierwszy z tych trzech predykatów działa rekurencyjnie. Jako argument dostaje macierz z której wydziela head (H) i tail (T) oraz numer wiersza W. Zwiększa numer wiersza, (numeracja wierszy, kolumn, kwadratów zaczyna się u mnie od 1). Wykorzystuje ona z kolei predykat służący do wstawiania z wiersza konkretnych już wartości. A następnie dla zmniejszonej o jeden wiersz macierzy wywołuje się rekurencyjnie.

Predykat wstawiaj wiersz przyjmuje jako argument konkretny wiersz który dzieli na head (H) i tail (T) oraz dostaje numer wiersza (W) i kolumny (K).
Bez względu na to czy Head jest zerem czy nie zwiększa numer kolumny i wywołuje się rekurencyjnie dla skróconego już wiersza. W przypadku gdy wartość Head nie jest zerem wywołuje predykat wstawiający konkretną wartość w podane miejsce (W, K).

Predykat ten zostanie opisany w kolejnym dziale.

Wstawianie wartości

Wstawianiem wartości zajmuje się poniższy predykat:

KOD	

```
wstawiaj_wartosc(H, W, K):-
        not(element(W,K,H)),
        KW is floor((W-1)/3)*3+ floor((K-1)/3)+1,
        assert(element(W,K,H)),!,
        not(usuwaj(H,W,K,KW)).
```

Jako argumenty przyjmuje wartość (H), numer wiersza (W) i kolumny (K) gdzie ma być wstawiony. Dla zabezpieczenia na początku sprawdzam czy taki element już nie istnieje, następnie wg znanego już wzoru wyliczam numer kwadratu. Wstawiam wartość oraz wywołuję predykat usuwaj który zajmuje się robieniem ograniczeń czyli usuwa wszystkie możliwości z całego wiersza, kolumny i kwadratu (3x3).

KOD	

```
usuwaj(H,W,K,KW):-
        retract(wiersz(W,H)),
        retract(kolumna(K,H)),
        retract(kwadrat(KW,H)),!,
        usuwaj_mozliwosci(H,W,K).
```

Jak widać usuwana jest podana wartość (H) z możliwości w wierszu (W), kolumny (K) i kwadratu (KW), a następnie wywoływany jest predykat usuwający możliwości wstawiania elementów.

Predykat usuwaj możliwość występuje w 5 wersjach, każda z nich przyjmuje 3 argumenty wartość usuwanej możliwości (X) numer wiersza (W) i kolumny (K).

KOD

```
usuwaj_mozliwosci(_,W,K):-
        mozliwosc(W,K,KW,A),
        retract(mozliwosc(W,K,KW,A)),
        fail.

usuwaj_mozliwosci(X,_,K):-
        mozliwosc(A,K,KW,X),
        retract(mozliwosc(A,K,KW,X)),
        fail.

usuwaj_mozliwosci(X,W,_):-
        mozliwosc(W,A,KW,X),
        retract(mozliwosc(W,A,KW,X)),
        fail.

usuwaj_mozliwosci(X,W,K):-
        KW is floor((W-1)/3)*3+ floor((K-1)/3)+1,!,
        mozliwosc(M,N,KW,X),
        retract(mozliwosc(M,N,KW,X)),
        fail.

usuwaj_mozliwosci(_,_,_).
```

Pierwsza wersja tego predykatu przelatuje wszystkie możliwości w danym wierszu i kolumnie i usuwa wszystko co się w tam kryje. Jest to usunięcie wszystkich możliwości dla miejsca gdzie została wstawiona wartość.

Druga wersja przelatuje przez całą kolumnę w której znajduje się nowo wstawiona wartość i z każdej komórki w tej kolumnie usuwa możliwość wstawienia podanej wartości

Trzeci predykat działa podobnie ale dla wiersza, zaś czwarty na podstawie wiersza i kolumny wyznacza w którym kwadracie się znajduje wstawiona wartość i w całym tym kwadracie usuwa możliwość ponownego wstawienia takiej wartości.

Piąty predykat zaś jest w celu zwrócenia pod koniec działania predykatu wartości ***true.***

Algorytmy i ich implementacja w prologu

Główny predykat

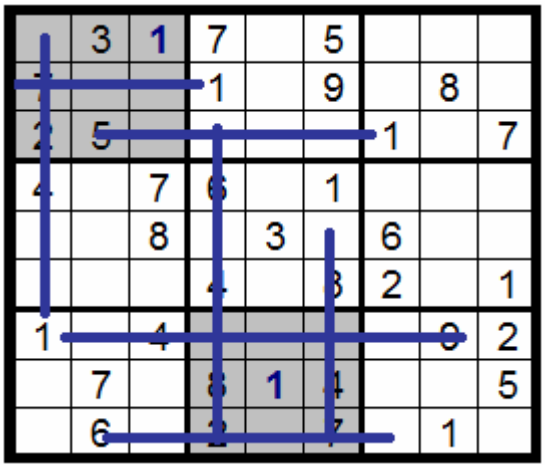
Predykatem obsługującym dział algorytmów jest :

KOD	
	algorytmy:- ***not(tylko_jedna_tu),*** ***not(pionowe_poziome),*** ***not(jeden_wiersz),*** ***not(jeden_kolumna),*** ***not(jeden_kwadrat).***

Po kolei wywołuje kolejne predykaty odpowiadające za dany algorytm.

Poziome i pionowe ograniczenia

Metoda ta polega na znalezieniu takiego pola oraz wartości w którym (w tym wypadku pole z niebieską 1) w zakresie jednego kwadratu (3x3) sąsiednie dwa wiersze i kolumny są blokowane bądź to przez taką wartość bądź wpisaną liczbę. I np. dla pierwszej jedynki dwa dolne wiersze (wiersz 2 i 3) są zablokowane bo już w nich jest jedynka, oraz pierwsza kolumna podobnie, natomiast w drugiej kolumnie nie ma jedynki, ale z pól (1,2) i (1,3) które jeszcze nie są zablokowane w drugiej właśnie kolumnie pole (1,2) jest już zajęte przez inną cyfrę.



I właśnie ten algorytm będzie miał za zadanie wykrywać pola w których spełnione są powyższe właściwości (blokada wierszy i kolumn bądź to zapełnienie sąsiedniego pola liczbą).

KOD

```
pionowe_poziome:-
      mozliwosc(W,K,_,X),
      modyf(W,W1,W2),
      modyf(K,K1,K2),
      (
       (
         (not(wiersz(W1,X));element(W1,K,_)),
         (not(wiersz(W2,X));element(W2,K,_)),
         not(kolumna(K1,X)),
         not(kolumna(K2,X))
       );
       (
         (not(kolumna(K1,X));element(W,K1,_)),
         (not(kolumna(K2,X));element(W,K2,_)),
         not(wiersz(W1,X)),
         not(wiersz(W2,X))
       )
      ),
      wstawiaj_wartosc(X,W,K),!,
      pionowe_poziome.
```

Pierwszą rzeczą jaka tu się rzuca w oczy jest zastosowanie ‘;’ i to aż 5 razy. Można by to zastąpić kilkoma predykatami jednak nie zrobiłem tego z kilku względów, po pierwsze chciałem aby warunek dotyczący właściwości wyżej wymienionych został podkreślony, a po drugie uważałem że taki zapis jest bardziej czytelny niż w kilku predykatach.

Predykat ten leci po wszystkich możliwościach do napotkania warunku. Rozpoznawanie numerów wierszy i kolumn sąsiednich zostaje wyznaczone dzięki modyfikatorom:

KOD

```
%%%    modyfikatory rozpoznawania sąsiadów
modyf(1,2,3).  modyf(2,1,3).  modyf(3,1,2).
modyf(4,5,6).  modyf(5,4,6).  modyf(6,4,5).
modyf(7,8,9).  modyf(8,7,9).  modyf(9,7,8).
```

Modyfikator jako pierwszą liczbę ma numer obecnej kolumny/wiersza a pozostałe dwie numery sąsiadów do sprawdzenia.

Jeżeli warunek jest spełniony to wstawia wartość i wywołuje siebie rekurencyjnie, w przeciwnym wypadku sprawdza kolejną możliwość.

Tylko jedna wartość w kratce

Metoda ta polega na znalezieniu takiego pola w którym możliwa jest do wstawienia tylko jedna cyfra. Jak widać na przykładzie w polu z niebieską trójką tylko trójka mogła być wstawiona bo jak widać zaznaczone na szaro liczby blokują wstawienie innych wartości.

	3		7		5			
7			1		9		8	
2	5		3			1		7
4		7	6		1			
		8		3		6		
			4		8	2		1
1		4					9	2
	7		8		4			5
	6		2		7		1	

Ten algorytm realizuje poniższy predykat. Przechodzi on po kolei po wszystkich możliwościach i dla danego pola o współrzędnych (W, K) sprawdza ile jest takich możliwości przypisanych dla tego pola. Jeżeli liczba tych możliwości (A) jest równa jeden to wstawia wartość i wywołuje siebie rekurencyjnie

KOD

```
tylko_jedna_tu:-
    mozliwosc(W,K,_,X),
    liczba_mozliwosci(W,K,A),
    A =:= 1,
    wstawiaj_wartosc(X,W,K),
    tylko_jedna_tu.
```

Predykat liczba_mozliwosci wykorzystuje dynamiczną zmienną :

KOD

```
:- dynamic(licz_el/1).
```

Liczba możliwości jest ustalana poprzez poniższą parę predykatów:

KOD

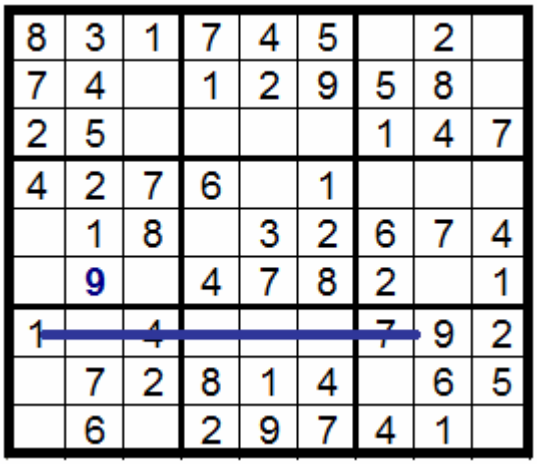
```
liczba_mozliwosci(W,K,A):-
        assert(licz_el(0)),
        not(zliczaj_mozliwosci(W,K)),!,
        licz_el(A),!,
        retract(licz_el(A)).

zliczaj_mozliwosci(W,K):-
        mozliwosc(W,K,_,_),
        licz_el(X),
        E is X + 1,
        retract(licz_el(X)),
        assert(licz_el(E)),
        fail.
```

Pierwszy z nich inicjalizuje wartość dynamiczną licz_el na zero, następnie wywołuje kolejny predykat zliczaj_mozliwosci, który z kolei przechodzi po wszystkich możliwościach dla danego wiersza (W) i kolumny (K) i zlicza je modyfikując wartość w licz_el. Po skończeniu działania tego predykatu odczytywana jest wartość licz_el (A) i czczona jest zajęta pamięć.

Zliczaj możliwości jest wywołane w not( ) ze względu że zawsze zliczaj_mozliwosci zwraca ***false***.

Tylko jedno pole dla wartości w wierszu, kwadracie, kolumnie

Ten algorytm składa się z trzech osobnych algorytmów działających podobnie tylko że jeden dla wiersza drugi dla kolumny a trzeci dla kwadratu. Rozpatrzmy przypadek kolumny. Jak można zobaczyć na przykładzie w drugiej kolumnie brakuje dwóch wartości, jednak jak widać 9 nie może się znaleźć w drugim pustym polu bo jest to pole zablokowane. W związku z tym w tej kolumnie jedynym możliwym polem jest to zaznaczone na niebiesko.



Za powyższy algorytm w przypadku wiersza odpowiedzialny jest poniższy predykat:

KOD

```
jeden_wiersz:-
        wiersz(W,X),
        assert(licz_el(0)),
        not(zlicz_mozliwosci(W,_,_,X)),
        licz_el(A),
        retract(licz_el(A)), A =:= 1,
        mozliwosc(W,K,_,X),
        wstawiaj_wartosc(X,W,K),
        fail.
```

Przechodzi kolejno po wierszach znajdując konkretne wartości, alokuje dynamiczną zmienną licz_el na zero i wywołuje predykat zlicz_mozliwosci. Predykat ten dostaje cztery wartości wiersz, kolumna, kwadrat, wartość. W tym przypadku ważna jest tylko wartość i numer wiersza. Po zliczeniu możliwości zczytuje wartość licz_el do zmiennej A czyści pamięć. Jeśli liczba tych możliwości jest równa 1 to wtedy wstawia wartość we wskazanym miejscu.

Predykat kończy się failem by wykonywać się wielokrotnie.

KOD	

```
zlicz_mozliwosci(W,K,KW,X):-
        mozliwosc(W,K,KW,X),
        licz_el(M),
        E is M + 1,
        retract(licz_el(M)),
        assert(licz_el(E)),
        fail.
```

Zlicz możliwości przelatuje po wszystkich istniejących możliwościach które spełniają zadane parametry W, K, KW, X. następnie jeśli znajdzie daną możliwość modyfikuje wartość zmiennej dynamicznej licz_el.

KOD	

```
jeden_kolumna:-
        kolumna(K,X),
        assert(licz_el(0)),
        not(zlicz_mozliwosci(_,K,_,X)),
        licz_el(A),
        retract(licz_el(A)), A =:= 1,
        mozliwosc(W,K,_,X),
        wstawiaj_wartosc(X,W,K),
        fail.
```

Zarówno jeden_kolumna i jeden_kwadrat działają podobnie, jedyną różnicą są wartości dostarczone dla funkcji zlicz_mozliwosci. W pierwszym przypadku ważna jest wartość i kolumna zaś w drugim ważny jest numer kwadratu i wartość.

KOD	

```
jeden_kwadrat:-
        kwadrat(KW,X),
        assert(licz_el(0)),
        not(zlicz_mozliwosci(_,_,KW,X)),
        licz_el(A),
        retract(licz_el(A)),  A =:= 1,
        mozliwosc(W,K,KW,X),
        wstawiaj_wartosc(X,W,K),
        fail.
```

Strzelanie wartości – przeglądanie możliwości

W związku z tym że same algorytmy nie wystarczają do rozwiązania bardzo trudnych sudoku potrzeby będzie algorytm który będzie zajmował się strzelaniem. Głównym predykatem służącym do tego jest :

KOD

```
strzelenie_liczb:-
        liczba_elementow(H), H < 81,
        mozliwosc(W,K,_,_),
        zapamietaj_mozliwosc(W,K),
        rekonstrukcja(Q),
        strzelanie_liczb2(W,K,Q),!.

strzelenie_liczb:-
        liczba_elementow(W),W =:= 81,!.
```

Pierwszy pobiera liczbę elementów, jeśli liczba ta jest mniejsza od 81 to znaczy że sudoku nie jest rozwiązane. Wtedy pobiera pierwszą lepszą możliwość, a dokładniej jej wiersz i kolumnę. Dla tego wiersza i kolumny zapamiętuje wszystkie możliwości. Oraz zamienia predykaty z powrotem na macierz (listę list). Po zrekonstruowaniu wywołuje predykat który tak naprawdę zajmuje się strzelaniem z zapamiętanych wartości.

KOD

```
zapamietaj_mozliwosc(W,K):-
        mozliwosc(W,K,_,X),not(pamiec_strzelen(W,K,X)),
        assert(pamiec_strzelen(W,K,X)),
        fail.
zapamietaj_mozliwosc(W,K):-
        mozliwosc(W,K,_,_).
```

Zapamiętanie możliwości odbywa się w prosty sposób. Do puki istnieje możliwość dla zadanych parametrów W, K sprawdza czy wartość już nie jest w pamięci strzeleń i jeśli nie jest to ja zapamiętuje jako wartość do strzelania. Drugi predykat służy by została zwrócona na końcu wartość true.

KOD

```
strzelanie_liczb2(W,K,Q):-
        liczba_elementow(H), H =\= 81,
        pamiec_strzelen(W,K,X),
        retract(pamiec_strzelen(W,K,X)),
        not(czyszczenie),
        inicjalizacja(Q),
        wstawiaj_wartosc(X,W,K),
        rekonstrukcja(P),
        not(czyszczenie),
        sudoku(P,_),
        strzelanie_liczb2(W,K,Q),!.

strzelanie_liczb2(_,_,_):-
        liczba_elementow(A), A =:= 81.
```

Ten predykat zajmuje się właściwym strzelaniem. Na początku sprawdza ile jest elementów. Jeśli jest ich 81 to kończy, w przeciwnym razie pobiera wartość z pamięci strzeleń dla zadanego wiersza i kolumny. Wartość ta po wybraniu zostaje usunięta.

Kolejnym elementem jest wyczyszczenie wszystkich dynamicznych danych związanych z reprezentacją macierzy, i od początku podana macierz Q zostaje zainicjalizowana, z wybranej wartości strzeleń zostaje wstawiona wartość i zrekonstruowana nowa macierz zawierająca strzeloną wartość. Dane dynamiczne zostają ponownie wyczyszczone a dla tak zrekonstruowanej macierzy wywołuje się predykat sudoku. Pod koniec rekurencyjnie wywołuje się ponowne strzelanie, tak że jeśli poprzednie sudoku znalazło wynik to drugi predykat przerwie działanie, a jeśli nie to zmieni się strzelony element.

Rekonstrukcja sudoku z predykatów

Rekonstrukcja polega na odzwierciedleniu tego co zapisane jest w dynamicznych zmiennych „możliwość” na macierz wynikową:

KOD

```
rekonstrukcja(L):-
        wczytaj_dane(L,1),!.
```

Wczytaj dane jest zdefiniowane następująco:

KOD

```
wczytaj_dane([H|T],W):-
        W < 9,
        B is W + 1, B < 10,
        wczytaj_wiersz(H,W,1),
        wczytaj_dane(T,B),!.

wczytaj_dane([H],W):-
        W =:= 9,
        wczytaj_wiersz(H,W,1).
```

Dla wierszy mniejszych od 9 zwiększa numer kolejnego wiersza, za pomocą predykatu wczytaj_wiersz wczytuje kolejne wiersze a następnie rekurencyjnie wywołuje samego siebie. Dla wiersza 9 nie wywołuj już siebie rekurencyjnie.
Wczytywanie wiersza działa podobnie tylko tym razem różnic jest w wyołanej funkcji czytaj element:

KOD

```
wczytaj_wiersz([H|T],W,K):-
        K < 9,
        B is K + 1, B < 10,
        czytaj_element(W,K,H),
        wczytaj_wiersz(T,W,B),!.
wczytaj_wiersz([H],W,K):-
        K =:= 9,
        czytaj_element(W,K,H),!.
```

Gdy nie ma elementu wpisywana jest wartość 0, inaczej wartość znaleziona.

KOD

```
czytaj_element(W,K,X):-
        element(W,K,X).
czytaj_element(W,K,0):-
        not(element(W,K,_)).
```

Pozostałe predykaty (pomocnicze)

Czyszczenie przechodzi po wszystkich dynamicznych wartościach i je usuwa

KOD

```
czyszczenie:- mozliwosc(A,B,C,D),retract(mozliwosc(A,B,C,D)),fail.
czyszczenie:- element(A,B,C),retract(element(A,B,C)),fail.
czyszczenie:- wiersz(A,B),retract(wiersz(A,B)),fail.
czyszczenie:- kolumna(A,B),retract(kolumna(A,B)),fail.
czyszczenie:- kwadrat(A,B),retract(kwadrat(A,B)),fail.
czyszczenie:- licz_el(X),retract(licz_el(X)),fail.
czyszczenie_strzelen :- pamiec_strzelen(W,K,X), retract(pamiec_strzelen(W,K,X)),fail.
```

Wypisywanie macierzy polega na podziale na Head i Tail a następnie wypisania listy Head i rekurencyjnie wypisaniu macierzy Tail

KOD

```
wypiszMacierz([H|T]):-
        wypiszListe(H),nl,
        wypiszMacierz(T).
wypiszMacierz([]):-nl.
```

Dzieli na Head i Tail. Head wypisuje za pomocą write a Tail wypisuje rekurencyjnie.

KOD

```
wypiszListe([]).
wypiszListe([0|T]):-write(0),write('    '), wypiszListe(T),!.
wypiszListe([H|T]):- H =\= 0,write(H),write('    '), wypiszListe(T),!.
```

Allokuje pamięć dla licz element, a następnie przechodząc po elementach zmienia licz_element.

KOD

```
liczba_elementow(W):-
        assert(licz_el(0)),
        not(zlicz_elementy),!,
        licz_el(W),
        retract(licz_el(W)),!.

zlicz_elementy:-
        element(_,_,_),
        licz_el(X),
        E is X + 1,
        retract(licz_el(X)),
        assert(licz_el(E)),
        fail.
```

Testowanie dla sudoku o różnym poziomie trudności

Testowanie zostanie przeprowadzone nie tylko pod względem sprawdzenia wyniku, ale także pod względem czasu wykonania przy wykorzystaniu predykatu ***time()***.

Łatwy

Wywołanie :

KOD	***time(sudoku([[0,0,6,0,0,0,9,0,0], [0,2,0,6,9,7,0,3,0], [7,0,0,0,8,0,0,0,6], [0,8,0,9,6,3,0,2,0], [0,0,0,0,0,0,0,0,0] ,[0,1,0,2,4,5,0,7,0], [6,0,0,0,5,0,0,0,1], [0,5,0,8,3,9,0,6,0], [0,0,9,0,0,0,8,0,0]])).***

```
SUDOKU ZADANE:

0   0   6   0   0   0   9   0   0
0   2   0   6   9   7   0   3   0
7   0   0   0   8   0   0   0   6
0   8   0   9   6   3   0   2   0
0   0   0   0   0   0   0   0   0
0   1   0   2   4   5   0   7   0
6   0   0   0   5   0   0   0   1
0   5   0   8   3   9   0   6   0
0   0   9   0   0   0   8   0   0

Znaleziono elementow: 51

SUDOKU WYNIKOWE:

3   4   6   5   2   1   9   8   7
8   2   1   6   9   7   5   3   4
7   9   5   3   8   4   2   1   6
4   8   7   9   6   3   1   2   5
5   6   2   1   7   8   3   4   9
9   1   3   2   4   5   6   7   8
6   3   8   7   5   2   4   9   1
1   5   4   8   3   9   7   6   2
2   7   9   4   1   6   8   5   3

% 66,177 inferences, 0.11 CPU in 0.49 seconds (22% CPU, 606013 Lips)

Yes
```

Średni

Wywołanie :

KOD	***time(sudoku([[3,9,1,0,0,0,6,5,0],[0,2,0,0,5,0,0,0,3],[0,0,5,0,0,4,1,2,0], [0,0,0,2,0,0,0,0,9],[0,0,0,8,0,1,0,0,0] ,[7,0,0,0,0,3,0,0,0],[0,1,6,3,0,0,4,0,0], [5,0,0,0,6,0,0,8,0],[0,4,8,0,0,0,7,3,6])).***

```
SUDOKU ZADANE:

3   9   1   0   0   0   6   5   0
0   2   0   0   5   0   0   0   3
0   0   5   0   0   4   1   2   0
0   0   0   2   0   0   0   0   9
0   0   0   8   0   1   0   0   0
7   0   0   0   0   3   0   0   0
0   1   6   3   0   0   4   0   0
5   0   0   0   6   0   0   8   0
0   4   8   0   0   0   7   3   6

Znaleziono elementow: 51

SUDOKU WYNIKOWE:

3   9   1   7   2   8   6   5   4
8   2   4   1   5   6   9   7   3
6   7   5   9   3   4   1   2   8
1   6   3   2   7   5   8   4   9
4   5   2   8   9   1   3   6   7
7   8   9   6   4   3   5   1   2
2   1   6   3   8   7   4   9   5
5   3   7   4   6   9   2   8   1
9   4   8   5   1   2   7   3   6

% 67,835 inferences, 0.16 CPU in 0.49 seconds (32% CPU, 434837 Lips)
```

Trudny

Wywołanie :

KOD	***time(sudoku([[5,0,0,2,0,4,0,7,0],[0,0,0,9,3,0,0,0,0],[8,3,0,0,0,0,0,0,0],[0,5,0,0,0,9,6,0,0], [6,0,7,0,0,0,2,0,8],[0,0,2,1,0,0,0,5,0],[0,0,0,0,0,0,0,6,2],[0,0,0,0,4,1,0,0,0],[0,8,0,5,0,2,0,0,1]])).***

```
SUDOKU ZADANE:

5   0   0   2   0   4   0   7   0
0   0   0   9   3   0   0   0   0
8   3   0   0   0   0   0   0   0
0   5   0   0   0   9   6   0   0
6   0   7   0   0   0   2   0   8
0   0   2   1   0   0   0   5   0
0   0   0   0   0   0   0   6   2
0   0   0   0   4   1   0   0   0
0   8   0   5   0   2   0   0   1

Znaleziono elementow: 55

SUDOKU WYNIKOWE:

5   6   9   2   1   4   8   7   3
2   7   1   9   3   8   5   4   6
8   3   4   6   7   5   1   2   9
1   5   8   7   2   9   6   3   4
6   9   7   4   5   3   2   1   8
3   4   2   1   8   6   9   5   7
4   1   5   8   9   7   3   6   2
9   2   6   3   4   1   7   8   5
7   8   3   5   6   2   4   9   1

% 296,132 inferences, 1.09 CPU in 1.15 seconds (95% CPU, 271181 Lips)
```

Piekielnie trudny

Wywołanie :

KOD	***time(sudoku([[0,0,0,0,0,9,0,7,0],[0,0,0,1,0,0,0,4,5],[0,3,0,0,0,5,0,0,6],[9,0,0,0,0,0,0,3,4], [0,8,0,9,0,2,0,6,0],[4,5,0,0,0,0,0,0,8],[8,0,0,7,0,0,0,2,0],[6,9,0,0,0,4,0,0,0],[0,1,0,8,0,0,0,0,0]])).***

```
SUDOKU ZADANE:

0   0   0   0   0   9   0   7   0
0   0   0   1   0   0   0   4   5
0   3   0   0   0   5   0   0   6
9   0   0   0   0   0   0   3   4
0   8   0   9   0   2   0   6   0
4   5   0   0   0   0   0   0   8
8   0   0   7   0   0   0   2   0
6   9   0   0   0   4   0   0   0
0   1   0   8   0   0   0   0   0

Znaleziono elementow: 55

SUDOKU WYNIKOWE:

5   6   8   4   3   9   1   7   2
2   7   9   1   8   6   3   4   5
1   3   4   2   7   5   9   8   6
9   2   1   6   5   8   7   3   4
3   8   7   9   4   2   5   6   1
4   5   6   3   1   7   2   9   8
8   4   5   7   9   1   6   2   3
6   9   3   5   2   4   8   1   7
7   1   2   8   6   3   4   5   9

% 402,536 inferences, 2.01 CPU in 1.96 seconds (103% CPU, 200027 Lips)
```

Wywołanie :

KOD	***time(sudoku([[0,1,0,0,2,0,4,0,0],[0,0,4,0,0,3,2,0,5],[8,2,0,0,5,4,0,6,0],[1,4,0,0,0,9,0,0,2], [0,3,0,2,0,6,0,4,0],[0,8,2,4,0,0,0,0,7],[0,5,0,7,9,2,0,0,4],[2,0,8,3,4,0,7,0,0],[4,0,0,0,0,0,0,2,0]])).***

```
SUDOKU ZADANE:

0   1   0   0   2   0   4   0   0
0   0   4   0   0   3   2   0   5
8   2   0   0   5   4   0   6   0
1   4   0   0   0   9   0   0   2
0   3   0   2   0   6   0   4   0
0   8   2   4   0   0   0   0   7
0   5   0   7   9   2   0   0   4
2   0   8   3   4   0   7   0   0
4   0   0   0   0   0   0   2   0

Znaleziono elementow: 45

SUDOKU WYNIKOWE:

5   1   3   6   2   7   4   8   9
9   6   4   1   8   3   2   7   5
8   2   7   9   5   4   3   6   1
1   4   5   8   7   9   6   3   2
7   3   9   2   1   6   5   4   8
6   8   2   4   3   5   1   9   7
3   5   6   7   9   2   8   1   4
2   9   8   3   4   1   7   5   6
4   7   1   5   6   8   9   2   3

% 121,323 inferences, 0.47 CPU in 0.74 seconds (63% CPU, 259236 Lips)
```

Wnioski

Jak można zauważyć występują różne czasy wykonania nawet w zakresie jednego poziomu trudności. Poziom łatwy i średni został rozwiązany w stosunkowo krótkim czasie, jednak poziom trudny i pierwszy przykład z poziomu „piekielnie trudny” przekroczył 1 sekundę. Wzięło się to stąd że prawdopodobnie predykat wszedł w dział strzelania i musiał wykonać kilka prób by dobrze trafić. W przypadku ostatniego przykładu, korzystając z debugera prześledziłem jak tutaj przebiegał algorytm strzelania. Dla pola w = 1, k = 1 były możliwości 3, 5, … strzał na cyfre 3 okazał się błędny dopiero strzał na 5 był poprawny i przynosił rozwiązanie. W trakcie rozwiązywania algorytm wykluczał możliwości, ale jeszcze 2 razy trzeba było strzelać. Na strzęście strzały były od razu poprawne.