Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych

Współczesne metody heurystyczne (WMH)

Algorytm przeszukiwania z tabu do rozwiązywania układanki Sudoku (PB5)

— Sprawozdanie cząstkowe —

Marcin Lembke
M.Lembke@stud.elka.pw.edu.pl

Marcin Malesa
M.Malesa@stud.elka.pw.edu.pl

Opiekun projektu:
dr inż. Piotr Bilski

25 stycznia 2017

1 Sudoku

Sudoku to gra logiczna, której celem jest uzupełnienie planszy, zazwyczaj o rozmiarze 9 na 9 pól, w taki sposób aby w każdym wierszu, kolumnie oraz wyznaczonym bloku 3 na 3 pola, znalazło się po jednej cyfrze z zakresu 1 do 9.

2 Generowanie plansz

Pierwszym problemem, który wyszczególniliśmy w sprawozdaniu wstępnym jest generowanie plansz Sudoku. Podaliśmy wtedy dwa pomysły na generowanie plansz: pierwszy polegajacy na generowaniu z wykorzystaniem bazy „dobrych" plansz, oraz drugi, bardziej złożony zaprezentowany w artykule [tutaj odnośnik do artykułu]. Ostatecznie zdecydowaliśmy się wzorować na metodzie zaprezentowanej w artykule, lecz z pewnymi zmianami:

1. Zdecydowaliśmy się przedstawić, na potrzeby generowania plansz, problem Sudoku jako problem spełniania ograniczeń (ang. *constraint satisfaction problem*, CSP). To podejście wydaje się być dobrym modelem układanki Sudoku.
2. Zrezygnowaliśmy z założenia o jednoznacznym rozwiązaniu planszy, ponieważ założenie to, mimo że jest czasami pożądane w przypadku układanki wygenerowanej do rozwiązania przez człowieka, nie jest wymagane do zrealizowania celu projektu, czyli zaprojektowania i implementacji algorytmu przeszukiwania z tabu, rozwiązującego układankę Sudoku. Innymi słowy interesuje nas sam fakt, znalezienia jakiegokolwiek rozwiązania przez zaimplementowany algorytm.
3. Uprościliśmy metodę oceny trudności planszy do jednego czynnika: liczby nieuzupełnionych pól.
4. Uprościliśmy metodę „robienia dziur" w układance do zwykłego losowania wypełnionych pól i ich usuwania.

2.1 Sudoku jako CSP

Problem spełnienia ograniczeń (CSP) można przedstawić jako trójkę (X, D, C):

$$X = \{X_1, \ldots, X_n\},$$
$$D = \{D_1, \ldots, D_n\},$$
$$C = \{C_1, \ldots, X_m\},$$

gdzie: X jest zbiorem zmiennych, D jest zbiorem dziedzin tych zmiennych oraz C jest zbiorem ograniczeń.

$$C \wedge x_1 \in D(x_1) \wedge \ldots \wedge x_n \in D(x_n)$$

Rozwiązanie układanki Sudoku o rozmiarze 9 na 9 można przedstawić jako CSP z następującymi ograniczeniami:

1. W każdym wierszu musi znaleźć się dokładnie po jednej cyfrze od 1 do 9.
2. W każdej kolumnie musi znaleźć się dokładnie po jednej cyfrze od 1 do 9.
3. W każdym kwadracie 3×3 musi znaleźć się dokładnie po jednej cyfrze od 1 do 9.

Oczywiście w tym wypadku zmiennych jest 81 i dziedzina każdej zmiennej to $\{1, 2, \ldots, 9\}$.

Algorytm generowania układanek Sudoku wygląda następująco:

1. Stwórz instancję problemu CSP.
2. Dodaj zmienne – wygeneruj pustą planszę.
3. Dodaj ograniczenia.
4. Wykorzystaj *solver* CSP do rozwiązania planszy.
5. Wylosuj uzupełnione pola do usunięcia.

Generowanie plansz zostało zaimplementowane w języku Python w wersji 3 z wykorzystaniem biblioteki *python-constraint*[1], dostarczającej m.in. *solver* CSP.

[1] `https://github.com/python-constraint/python-constraint`

3 Przeszukiwanie tabu

Po obmyśleniu sposobu na generowanie rozwiązywalnych plansz Sudoku, pora przystąpić do zaprojektowania algorytmu przeszukiwania z tabu. Pierwszym, aczkolwiek trywialnym zadaniem jest ustalenie sposobu reprezentacji plansz. Zdecydowaliśmy się robić to za pomocą zwykłej dwuwymiarowej tablicy.

```
class Node  // Klasa reprezentujaca pojedyncze pole na planszy
{
unsigned int value;   // Wartosc pola
bool startingNode;   // Czy pole jest polem poczatkowym?

(...)
};

class Sudoku   // Klasa reprezentujaca pojedyncza plansze Sudoku
{
Node map[9][9];

(...)
};
```

Kolejnym ważnym zadaniem było zdefiniowanie relacji sąsiedztwa na parach elementów w przestrzeni rozwiązań, która będzię obejmować całą jej dziedzinę. Zdecydowaliśmy się na funkcję swap, która zamienia miejscami cyfry o pozycjach x i y w bloku 3x3 o numerze blockNo.

```
bool swap(unsigned int blockNo, unsigned int x, unsigned int y)
{
(...)
}
```

Dodatkowo, powinniśmy także zdefiniować strukturę, która będzie reprezentować możliwe do wykonania ruchy.

```
class PossibleMove
{
unsigned int blockNo;
unsigned int x;
unsigned int y;
};
```

Po wygenerowaniu planszy uruchamiamy algorytm przeszukiwania z tabu, którego uproszczona wersja jest przedstawiona w poniższym kodzie. Jeszcze niżej jest natomiast opisane co dzieje się w kolejnych krokach algorytmu.

```
Sudoku tabuSearchSudokuSolver(Sudoku inputSudoku)
{
Sudoku currentSolution = inputSudoku.fillHolesRandomly();    // 1.
Sudoku bestSolution = currentSolution; // 2.
queue<PossibleMove> tabuList; // 2.
while (!stopCondition())       // 3.
{
Sudoku bestCandidateSolution;
PossibleMove bestCandidateMove;
vector<pair<PossibleMove, Sudoku>> currentSolutionNeighbourhood =
currentSolution.getNeighbourhood(); // 4.
for (auto candidate : currentSolutionNeighbourhood)
{
PossibleMove candidateMove = candidate.getKey();
Sudoku candidateSolution = candidate.getValue();
if ( !tabuListContainsMove(tabuList, candidateMove)
&& (candidateSolution.fitness() > bestCandidateSolution.fitness()) )
{
bestCandidateSolution = candidateSolution;    // 5.
bestCandidateMove = candidateMove;  // 5.
}
```

```
}

if (bestCandidateSolution.fitness() > bestSolution.fitness())
{
bestSolution = bestCandidateSolution;  // 6.
}
currentSolution = bestCandidateSolution;  // 7.
tabuList.push(bestCandidateMove);    // 7.
if (tabuList.size() > MAX_TABU_LIST_SIZE)
{
tabuList.pop();        // 8.
}
}
return bestSolution; // 9.
}
```

1. Wypełniamy puste pola planszy wejściowej losowymi liczbami, tak aby w każdym bloku 3x3 planszy znajdowawły się różne cyfry od 1 do 9. Jest to nasze obecne najlepsze rozwiązanie (oczywiście najprawdopodobniej nieprawidłowe).

2. Tworzymy pustą listę tabu, reprezentowaną jako kolejka FIFO.

3. Sprawdzamy warunek stopu. W naszym algorytmie funkcja stopCondition() zwróci wartość true wtedy i tylko wtedy, gdy liczba kolizji (czyli liczba powtarzających się cyfr w kolumnach i wierszach) będzie równa 0.

4. Zczytujemy do wektora sąsiędztwo aktualnego rozwiązania, reprezentowane jako lista par - możliwy ruch i utworzone przez ten ruch sudoku.

5. Dla każdego elementu z sąsiedztwa, sprawdzamy czy ruch zawiera znajduje się na liście tabu i czy jego fitness jest większy niż fitness najlepszego dotychczasowego kandydata. Funkcja fitness() zwraca tym większą wartość im mniej kolizji występuje. Jeśli tak, to nadpisujemy najlepszego dotychczasowego kandydata znalezionym rozwiązaniem.

6. Jeśli fitness naszego najlepszego kandydata w danej iteracji jest większy niż fitness najlepszego znalezionego rozwiązania od początku działania algorytmu, to nadpisujemy to najlepsze rozwiązanie.

7. Nadpisujemy obecne rozwiązanie najlepszym kandydatem i wrzucamy wykonany ruch na listę tabu.

8. Jeśli lista tabu jest już pełna to usuwamy ostatni jej element;

9. Zwracamy najlepsze dotychczasowe rozwiązanie. Kończymy algorytm.