Projekt - Podstawy Podejmowania Decyzji

Edge Matching Puzzle – TetraVex variation

Jakub Gołuch

Inżynieria Systemów

semestr 4, 2022/2023

May 15, 2023

1. Opis problemu

Nasz problem optymalizacyjny polega na znalezieniu optymalnego rozwiązania dla gry komputerowej TetraVex. Gra ta składa się z kwadratowej siatki oraz kolekcji płytek, domyślnie dziewięciu kwadratowych płytek dla siatki o wymiarach 3x3. Każda płyta ma cztery jednocyfrowe liczby, po jednej na każdej krawędzi.

Celem gry jest umieszczenie płytek w odpowiednich miejscach na siatce, aby ułożyć układankę jak najszybciej. Płytki nie mogą być obracane, a dwie płytki mogą być umieszczone obok siebie tylko wtedy, gdy liczby na sąsiadujących krawędziach się zgadzają.

Problem TetraVex został zainspirowany "problemem układania kafelków na płaszczyźnie", opisanym przez Donalda Knutha na stronie 382 w tomie „1: Fundamentalne algorytmy", pierwszej książce jego serii The Art of Computer Programming. Gra została nazwana przez Scotta Fergusona, lidera rozwoju i architekta pierwszej wersji Visual Basic, który stworzył ją dla pakietu Windows Entertainment Pack 3.

Możliwą liczbę plansz TetraVex można obliczyć. Na planszy o wymiarach n x n są (n-1) * n(n-1) poziomych i pionowych par, które muszą się zgadzać, oraz 4 * n * (n+1) liczb na krawędziach, które mogą być dowolnie wybierane. Oznacza to, że istnieje 2 * n(n+1) wyborów 10 cyfr, czyli 10^(2 * n(n+1)) możliwych plansz.

2. Model matematyczny

w – macierz parametrów wszystkich klocków

klocek = [g, p, d, l]

$$w = [klocek1 \quad klocek2 \quad \dots \quad klocekn]$$

Gdzie:

g- górny kolor klocka

p – prawy kolor klocka

d – dolny kolor klocka

l – lewy kolor klocka

n – liczba wszystkich klocków

r – rozmiar jednego z wymiarów planszy

p – macierz o wymiarach r x r, służąca jako plansza dla klocków

Przykładowe parametru modelu:

klocek1 = [4,5,6,7]

klocek2= [6,3,2,4]

klocek3 = [1,2,3,5]

w = [klocek1, klocek2, klocek3]

3. Zmienna decyzyjna

Zmienną decyzyjną jest przypisanie indeksu klocka do pozycji na planszy.

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{1r} \\ x_{21} & x_{22} & x_{2r} \\ x_{r1} & x_{r2} & x_{rr} \end{bmatrix}_{r\ x\ r}$$

$$x_{ij} - przypisanie\ indeksu\ klocka\ do\ pozycji\ ij$$

4. Funkcja celu

Maksymalizacja wykorzystania klocków uwzględniając dane ograniczenia.

$$argmax\big(f(X)\big)$$

$$f(X) = \sum_{i=1}^{r} \sum_{j=1}^{r} g(x_{ij})$$

$$\mathrm{g(x)} = \begin{cases} 1, gdy\ x \neq 0 \\ 0, gdy\ x = 0 \end{cases}$$

f(X) – liczba użytych klocków

X – zmienna decyzyjna o przypisaniu klocków do pozycji na planszy

5. Ograniczenia

- $\forall_{klocek\ \in\ w} \prod_{i=1}^{4} klocek[i]\ > 0\ -$
  $wszystkie\ ściany\ klocka\ muszą\ mieć\ kolor\ (0 = brak\ koloru)$
- $\sum_{i=1}^{r} \sum_{j=1}^{r} x_{ij} \leq \sum_{i=1}^{n} i$ – nie może być więcej klocków niż miejsc na planszy
- $w\big[x_{ij}\big][d] = w\big[x_{i+1,j}\big][g] \vee\ x_{ij} = 0 \vee\ x_{i+1,j}\ = 0\ -$
  $kolory\ stykające\ się\ w\ kolumnach\ muszą\ się\ zgadzać$ i = 1,2,…,r-1 j = 1,2,…,r
- $w\big[x_{ij}\big][p] = w\big[x_{i,j+1}\big][l] \vee\ x_{ij} = 0 \vee\ x_{i,j+1}\ = 0\ -$
  $kolory\ stykające\ się\ w\ rzędach\ muszą\ się\ zgadzać$ i = 1,2,…,r j = 1,2,…,r-1

- $\sum_{j=1}^{r}\sum_{k=1}^{r}\begin{cases}1, gdy\ x_{jk}=i\\0, gdy\ x_{jk}\neq i\end{cases} \leq 1$ – każdy klocek może być wykorzystany maksymalnie 1 raz i = 1,2,...,n

6. Implementacja w IBM Cplex

6.1 Kod

```
using CP;

int liczba_klockow = 8;

int rozmiar_planszy = 3;

// klocek o wartości [0,0,0,0] to klocek widmo, bo cplex wywala
brak wartości przy odwołaniu się do indeksu 0, który wcześniej nie
istniał
int w[0..liczba_klockow][1..4] = [[0,0,0,0], [1,2,3,4], [2, 4, 7,
2], [1, 5, 5, 4], [3, 4, 6, 5], [5, 7, 2, 1], [6, 6, 2, 3], [6, 7,
6, 6], [2, 1, 8, 7]];

dvar int klocki[1..rozmiar_planszy][1..rozmiar_planszy];

maximize sum(i in 1..rozmiar_planszy, j in 1..rozmiar_planszy)
klocki[i][j];

subject to {

  forall(i in 1..liczba_klockow) {
    count(all(j in 1..rozmiar_planszy, k in 1..rozmiar_planszy)
klocki[j][k], i) <= 1;
  }

  forall(i in 1..rozmiar_planszy, j in 1..rozmiar_planszy)
klocki[i][j] <= liczba_klockow;

  forall(i in 1..rozmiar_planszy, j in 1..rozmiar_planszy-1)
klocki[j][i] == 0 || klocki[j+1][i] == 0 || w[klocki[j][i]][3] ==
w[klocki[j+1][i]][1];

  forall(i in 1..rozmiar_planszy, j in 1..rozmiar_planszy-1)
klocki[i][j] == 0 || klocki[i][j+1] == 0 || w[klocki[i][j]][2] ==
w[klocki[i][j+1]][4];


}
```

```
float liczba_nieuzytych_klockow = liczba_klockow -
rozmiar_planszy^2 + count(all(j in 1..rozmiar_planszy, k in
1..rozmiar_planszy) klocki[j][k], 0);

int czy_sie_da_uzupelnic = 0;

execute {

   liczba_nieuzytych_klockow;

   if(liczba_nieuzytych_klockow == 0) {
     czy_sie_da_uzupelnic = 1
   };
   czy_sie_da_uzupelnic;
```

6.2 Wyniki

- klocki = [[1,2,3], [4,0,5], [6,7,8]]
- czy_sie_da_uzupelnic = 1
- liczba_nieuzytych_klockow = 0

1..rozmiar_planszy (wielkość 3)	1..rozmiar_planszy (wielkość 3)		
	1	2	3
1	1	2	3
2	4	0	5
3	6	7	8

7. Implementacja w Python

Do rozwiązania problemu optymalizacji użyłem algorytmu Tabu Search, w którym występuje dynamiczna zmiana rozwiązania, zależna od ilości iteracji. Lista tabu zawiera zestaw zakazanych rozwiązań dopuszczalnych, jest ona uaktualniana po każdej iteracji.

Kroki działania TS:

1. Wybieramy losowo przyporządkowanie klocków w tablicy oraz ustalamy listę dozwolonych ruchów w każdej iteracji i wymiar listy tabu. Definiujemy warunek stopu; może to być określona liczba iteracji.

2. Dopóki nie jest spełniony warunek stopu wykonuj:

3. Wybierz ruch (zamianę wartości przypisanych klocków do miejsc w tablicy), który maksymalizuje rozwiązanie.

4. Jeśli ruch jest na liście tabu to wybierz pozostały krok.

5. Zaktualizuj listę tabu i wprowadź nowy ruch.

6. Jeśli znaleziono lepsze rozwiązanie, niż aktualne to zaktualizuj je.

7. Po spełnieniu warunku stopu, wyświetlamy rozwiązanie optymalne

Kod na Colabie:
https://colab.research.google.com/drive/1gi0an49UvyO9kMTsLy6jI54b-3tZLG_-#scrollTo=yHflB2Ba6hml

Rozwiązanie znalezione przez Tabu Search:

```
[Block(0, 0, 0, 0), Block(0, 0, 0, 0), Block(1, 2, 3, 4)]
[Block(0, 0, 0, 0), Block(0, 0, 0, 0), Block(0, 0, 0, 0)]
[Block(2, 4, 7, 2), Block(1, 5, 5, 4), Block(3, 4, 6, 5)]
```

Czyli:

klocki = [[0,0,1],[0,0,0],[2,3,4]]

8. **Wnioski**

Na podstawie danych generatywnych dla 8 klocków i planszy o wymiarach 3 x 3, wynik końcowy modelu jest bardziej satysfakcjonujący dla wersji z Cplexa. Wersja z algorytmem tabu search przedstawia zły wynik.