Algoritmy umělé inteligence (VAI)

Řešení Sudoku pomocí UI

Jakub Zapletal

Jan Brzobohatý

Daniel Bujnovský

1 Program

Hlavním souborem programu je `main.py`. Zde se vytvoří zadání Sudoku ve formě 2D listu, na které se následně zavolá řešící metoda z jednoho ze tří importovaných souborů: `backtrack.py`, `bestfirstsearch.py`, `crook.py`.

Pro Crookův algoritmus se pouze zavolá řešící metoda. Metoda Best first search vrací vyřešenou mřížku - je jí potřeba přiřadit novou proměnnou. Pro metodu Backtrack je potřeba do vstupu zvolit, kterou verzi Backtracku chceme použít.

Program následně vyřeší zadání Sudoku, řešení vypíše a s ním i čas výpočtu, použitou metodu a v případě Backtracku i počet backtracků.

2 Backtrack algoritmus

V souboru `backtrack.py` se nachází čtyři různé verze Backtrack algoritmu.

2.1 Backtrack

Jedná se o klasickou *brute force* metodu. Pomocnou metodou nejprve najde následující prázdnou buňku a do ní se pokouší dosadit číslo. Jediné omezení je, aby toto číslo nekolidovalo s jiným číslem na řádku, sloupci, nebo v daném sektoru (3x3). Zkouší čísla od 1 do 9 a jakmile může jedno dosadit, učiní tak. Následuje rekurze - metoda bude volat sebe stále dokola. Takhle pokračuje dál, dokud nevyplní všechny buňky, nebo dokud nenarazí na problém (pravděpodobnější možnost). Tento problém je, že nemůže do další prázdné buňky dosadit žádné číslo od 1 do 9. V takovém případě vrátí metoda `False` a vrátí se o úroveň výš, tj. do předchozí buňky - provede *backtrack*. Zde vymaže dříve dosazenou hodnotu (dosadí 0) a zkouší vyplnit jiné číslo (o 1 větší, zkrátka pokračuje ve `for` cyklu). Pokud nemůže dosadit žádné, opět vrátí `False` a dělá to stejné znova. Takhle se může vrátit až k první buňce (což se většinou děje). Sudoku střední obtížnosti tak vyřeší za čas v řádu stovek milisekund a počet backtracků je přitom v řádu tisíců.

2.2 Reverzní backtrack

Jak název napovídá, algoritmus je převrácením původního Backtracku. Úprava je to drobná, rozdíl je pouze v tom, že nezačínáme vyplňovat buňky od začátku, ale od konce. Velice se vyplatí ji použít, když máme v zadání Sudoku více hodnot v druhé polovině mřížky než v té první. V takovém případě může metoda výrazně urychlit výpočet. Při porovnání řešení zadání `grid1` obyčejným Backtrackem je ten reverzní průměrně 3x rychlejší a backtracků potřebuje méně než tisíc.

2.3 Backtrack s implikacemi

Tato verze Backtracku se lehce blíží způsobu, jakým by Sudoku řešil člověk. Začíná jako běžný Backtrack, tzn. najde následující prázdnou buňku a pokouší se do ní dosadit hodnoty od 1 do 9. Najde-li vyhovující hodnotu (platí stále stejná omezení), dosadí ji.

Nyní však, ještě před rekurzí, provede několik implikací. Zavolá funkci `implicate()`, která ověří, zda tato nová hodnota způsobila nějaké změny v omezeních ve zbytku mřížky (pravděpodobně totiž ano).

Ověří to následujícím způsobem: projde jednotlivě všechny sektory, v každém vytvoří list prázdných buněk v tomto sektoru a množinu zbývajících možností tohoto sektoru (tj. doplněk množiny všech hodnot, které jsou v daném sektoru obsaženy). Následně projde všechny prázdné buňky tohoto sektoru. V každé této buňce zkontroluje podmínky řádku a sloupce - tedy vytvoří dvě množiny (pro řádek a pro sloupec) a přidá do nich všechny hodnoty, které jsou na daném řádku a sloupci (kde se nachází současná prázdná buňka). Následně provede rozdíl množiny zbylých možností v sektoru a omezení z řádku a sloupce. Pokud je výsledek jednoprvková množina, pak se jedná o

množinu obsahující jedinou možnost, kterou můžeme do dané prázdné buňky dosadit - tedy dosadíme ji. Toto provedeme (jak bylo řečeno) pro všechny prázdné buňky v daném sektoru a pro všechny sektory v mřížce.

Všechny hodnoty, které jsme při implikaci do mřížky zapsali se zapisují i se svými souřadnicemi do listu implikací, který metoda vrací. Pokud se totiž později zjistí, že původní dosazená hodnota byla chybná a je potřeba se vrátit o buňku zpět, je potřeba nejen vynulovat tuto hodnotu, ale i všechny implikace - proto vracíme list s implikacemi.

Tato metoda je opravdu velice efektivní. Zadání `grid1` vyřeší v řádu jednotek milisekund, počet backtracků se přitom pohybuje v řádu desítek.

2.4 Backtrack s dopřednou kontrolou a MRV

Poměrně známá verze metody Backtrack, anglicky *Backtrack with forward checking and minimal remaining values (MRV)*.

Důležitou součástí tohoto algoritmu je list `remaining_values`, který obsahuje všechny zbylé možnosti všech prázdných buněk v celé mřížce.

Dopředná kontrola se provádí pokaždé, když vložíme do nějaké buňky novou hodnotu. Funkce `forward_check()` totiž zkontroluje jestli vložená hodnota nekoliduje s nějakou jinou, ještě nevloženou hodnotou a to tak, že projde všechny zbývající možnosti na řádku, sloupci a v sektoru, kde se právě dosazená hodnota nachází. Zjistí, zda někde nezbývá jen jedna možnost, která by byla právě naší hodnotou. Najde-li takovou možnost, vrátí `False` a je potřeba hledat jinou hodnotu, která by mohla být do prázdné buňky dosazena (klasický backtrack).

Tato metoda tedy zpřísňuje pravidla, podle kterých budou hodnoty do prázdných buněk vkládány a sníží tak počet chyb a nutných backtracků k jejich opravě. Časově by tedy měl být výhodnější. Bohužel to tak není, neboť list zbývajících možností `remaining_values` se musí vytvářet znova při každé další rekurzi (alespoň se mi nepodařilo najít způsob, jak to obejít a zjednodušit). Časově je na tom tedy podobně, průměrně i o něco hůž, než obyčejný Backtrack.

List `remaining_values` však můžeme využít ještě za jiným účelem. Zde nastupuje funkce MRV, která ze všech zbývajících možností vybere právě tu buňku, kde je zbývajících možností nejméně. Do této buňky pak dosazujeme hodnoty, ale pouze ty které jsou v možnostech buňky. Ideální případ je, že najdeme vždy takovou buňku, kde zbývá pouze jedna možnost - potom netřeba backtracků vůbec. Pokud není zadání příliš složité, skutečně k žádným backtrackům nedojde a metoda je tak velice rychlá.

Na testované zadání `grid1` nebyl zapotřebí ani jeden backtrack a čas výpočtu se pohybuje v řádu desítek milisekund.

Samotná dopředná kontrola velké zrychlení nepřináší, ale v kombinaci s MRV se jedná o velmi efektivní metodu silně konkurující verzi s implikacemi. Ve většině případů je však Backtrack s implikacemi stále nejrychlejší metodou.

3 Crookův algoritmus

Tento algoritmus řeší sudoku podobně jako člověk. Má několik logických postupů, jak vyřazovat čísla z políček, ve kterých být nemohou, a doplňovat správná čísla. Tyto logické postupy opakuje, dokud sudoku nevyřeší, nebo se nedostane do situace, že nemůže žádným logickým postupem doplnit nebo vyřadit číslo. V tuto chvíli začne „hádat“ pomocí backtracku.

Tento algoritmus kromě tabulky sudoku (listu `grid`) také sleduje, která čísla mohou ve kterém políčku být (dictionary `possibles`) – podobně jako člověk, který si do rožku vpisuje čísla. Celý algoritmus začne tím, že si vytvoří dictionary `possibles`. Každé pozici `(i,j)` přísluší list obsahující zatím nevyřazené hodnoty, tedy možná čísla, která mohou být v políčku `(i,j)`.

45689	468	23589	34589	35689	389	258	7	1
5678	1	3578	358	3568	2	9	4	358
4589	48	23589	134589	7	1389	258	23	6
1478	5	178	23789	2389	6	1248	239	23489
148	9	18	2358	2358	38	7	236	2348
2	3	6	789	1	4	8	5	89
156789	678	15789	123789	2389	13789	245	29	2459
1579	2	4	6	9	179	3	8	59
3	8	89	289	4	5	6	1	7

Jediné možné číslo Algoritmus najde políčka, do který jde dosadit jen jedno číslo.

45689	468	23589	34589	35689	389	258	7	1
5678	1	3578	358	3568	2	9	4	358
4589	48	23589	134589	7	1389	258	23	6
1478	5	178	23789	2389	6	1248	239	23489
148	9	18	2358	2358	38	7	236	2348
2	3	6	789	1	4	8	5	89
156789	678	15789	123789	2389	13789	245	29	2459
1579	2	4	6	9	179	3	8	59
3	8	89	289	4	5	6	1	7

Toto číslo do nich dosadí. Potom musí aktualizovat hodnoty `possibles` políček ležících ve stejném sektoru, řádku, nebo sloupci.

45689	468	23589	34589	35689	389	258	7	1
5678	1	3578	358	3568	2	9	4	358
4589	48	23589	134589	7	1389	258	23	6
1478	5	178	23789	2389	6	1248	239	23489
148	9	18	2358	2358	38	7	236	2348
2	3	6	789	1	4	8	5	89
156789	678	15789	123789	2389	13789	245	29	2459
1579	2	4	6	9	179	3	8	59
3	8	89	289	4	5	6	1	7

Tento postup se opakuje, dokud nedoplní všechna takto doplnitelná políčka (i ta nově vzniklá). V ukázkovém sudoku se dostaneme do následující situace.

589	6	238	34589	358	389	25	7	1
578	1	378	358	3568	2	9	4	38
589	4	238	13589	7	1389	25	23	6
478	5	178	389	238	6	124	23	234
48	9	18	358	2358	38	7	236	234
2	3	6	7	1	4	8	5	9
6	7	5	138	38	138	24	29	24
1	2	4	6	9	7	3	8	5
3	8	9	2	4	5	6	1	7

Doplnění čísla do jediného možného políčka Pokud jde v některém řádku, sloupci nebo sektoru doplnit číslo do právě jednoho políčka, doplní se tam. Po doplnění se musí aktualizovat hodnoty `possibles`.

589	6	238	4	358	389	25	7	1
578	1	378	358	3568	2	9	4	38
589	4	238	13589	7	1389	25	23	6
478	5	178	389	238	6	124	23	234
48	9	18	358	2358	38	7	236	234
2	3	6	7	1	4	8	5	9
6	7	5	138	38	138	24	29	24
1	2	4	6	9	7	3	8	5
3	8	9	2	4	5	6	1	7

Tímto postupem se algoritmus dostane do následující situace.

589	6	238	4	358	389	25	7	1
57	1	37	35	6	2	9	4	8
589	4	28	158	7	189	25	3	6
478	5	78	9	238	6	1	2	234
48	9	1	358	2358	38	7	6	234
2	3	6	7	1	4	8	5	9
6	7	5	138	38	138	24	9	24
1	2	4	6	9	7	3	8	5
3	8	9	2	4	5	6	1	7

Interakce sektoru a sloupce/sektoru a řádku Pokud v některém sektoru leží políčka, do kterých jde dosadit vybrané číslo, v jednom řádku nebo sloupci, nemohou se vyskytovat v tomto řádku nebo sloupci mimo daný sektor. Proto je algoritmus zakáže.

589	6	238	4	358	389	25	7	1
57	1	37	35	6	2	9	4	8
589	4	28	158	7	189	25	3	6
478	5	78	9	238	6	1	2	234
48	9	1	358	2358	38	7	6	234
2	3	6	7	1	4	8	5	9
6	7	5	138	38	138	24	9	24
1	2	4	6	9	7	3	8	5
3	8	9	2	4	5	6	1	7

Na tato políčka by šel použít i Crookův algoritmus, protože obě obsahují pouze dvě stejná čísla. Kdyby v jednom z těchto políček bylo i jiné číslo, které by v druhém nebylo, stále by šel tento postup použít, ale Crookův algoritmus už ne.

Crookův algoritmus Crookův algoritmus funguje na úvaze, že když se v n políčkách v řádku, sloupci nebo boxu opakuje právě n stejných čísel, nemůžou se tato čísla objevovat jinde. Kdyby totiž bylo jinde, nemohlo by být v zmíněné n-tici, takže by jedno políčko „zůstalo na ocet“.

589	6	238	4	358	389	25	7	1
57	1	37	35	6	2	9	4	8
589	4	28	158	7	189	25	3	6
478	5	78	9	38	6	1	2	34
48	9	1	358	2358	38	7	6	34
2	3	6	7	1	4	8	5	9
6	7	5	138	38	138	4	9	2
1	2	4	6	9	7	3	8	5
3	8	9	2	4	5	6	1	7

V obrázku jsou vymazány zároveň 3 i 8, tento program by je však odstranil po jednom.

Tyto logické postupy se opakují tak dlouho, dokud není sudoku vyřešeno, nebo dokud žádná z metod nedokáže vyloučit ani dosadit žádné číslo. Ukázkové sudkou by takto vyřešit šlo. Pokud však zadání nejde těmito postupy vyřešit, algoritmus zkusí uhádnout pomocí backtracku jedno číslo a pokračovat v logických postupech, dokud nenarazí na řešení nebo na spor.

4 Algoritmus best-first search

Princip algoritmu best-first search spočívá ve výběru takového následníka aktuálního stavu (mřížky), který má nejlepší hodnotu hodnotící funkce. V případě problému řešení sudoku onou hodnotící funkcí můžeme rozumět: "vygenerujme nový stav/doplňme prázdné pole takovým způsobem, který bude pro nás nejméně komplikovaný". Nejlepší způsob je tedy zaměřit se vždy na takové pole, které má nejmenší počet možných doplnění. Postupujeme tedy v duchu hladového algoritmu, kdy vždy vybíráme tu aktuálně nejlepší možnost.

Hlavní části programu je metoda `solve_sudoku(grid)`, která rekurzivně volá sama sebe vždy s jiným parametrem grid(mřížka). V prvním kroku metoda `isSolved` zkontroluje, zda zkoumaná mřížka není řešením zadání. Pokud není pokračujeme dalším krokem.

V druhém kroku algoritmus najde metodou `getCandidates()` buňku s nejvíce omezeními, tedy nejméně možnostmi na doplnění - takovou buňku je nejjednodušší vyřešit. Pokud našel větší počet takovýchto buněk, volí první nalezenou.

Následně pro tuto buňku určí metodou `getNumberPossibilitiesForCell()` počet přípustných možností na doplnění na základě kontroly sloupců, řádků a sektorů a následně metoda `getPossibilitiesForCell` určí ony přípustné možnosti a nahraje je do listu `possible_values`.

Do listu `candidates` nahraje algoritmus kopie původní mřížky, každou s doplněnou jinou hodnotou z `possible_values` v uvažované buňce. Pokud je v `possible_values` více než jedna hodnota, dostaneme tedy více kandidátů tzn. dojde k dělení uzlu příslušného stavu na více větví. Algoritmus si ze všech možných kandidátů (mřížek) vybere první v pořadí a pokračuje v prohledávání této větvě (opět zavolá rekurzivně sám sebe s parametrem aktuálního kandidáta). Takto pokračuje principem deep-first search dokud nenarazí na problém - prázdnou buňku, která nemá žádnou přípustnou možnost. V tom případě se vracíme do uzlu, ve kterém došlo k větvení (bylo v něm vícero mřížek, kterými bylo možno pokračovat) a pokračujeme s jinou možností.

Je-li mřížka zcela vyplněná, nahraje se do globální proměnné `promenna` a úvodní metoda `solve()` ji vrátí spouštěcímu souboru. Následně se mřížka vypíše a porovná se známým řešením.