Informatyka, studia zaoczne, inż I st. semestr V

Sztuczna inteligencja i systemy ekspertowe 2017/2018

Prowadzący: mgr inż. Paweł Tarasiuk sobota, 14:45

Data oddania: _____ Ocena: _____

Natalia Mateuszuk 203940
Adrian Grzelak 200242

Zadanie 1
Przeszukiwanie przestrzeni stanów. Piętnastka

1. Wstęp

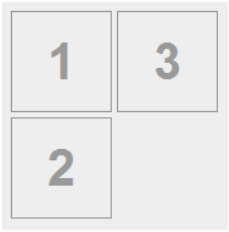
Piętnastka to popularna łamigłówka polegająca na przesuwaniu wolnego pola po planszy o rozmiarze 3×3 aż do momentu ułożenia całej układanki. W raporcie tym zostaną przedstawione odpowiedzi na kilka ciekawych pytań dotyczących tej łamigłówki, oraz analiza skuteczności algorytmów przeszukiwań drzew, na podstawie poszukiwania rozwiązania właśnie tej piętnastki. Ponadto przeanalizujemy jej pochodne, czyli plansze o rozmiarach niestandardowych jak np. 3×3, 2×2, 3×2 itp.

Rysunek Ułożona tablica piętnastka

2. Analiza rozwiązania

2.1. Czy zawsze można rozwiązać?

Nie. Łatwo można się o tym przekonać na tej tablicy 2×2, nie ważne co zrobimy nie uda nam się zamienić miejscami klocków 2 i 3 bez zmiany pozycji klocka nr 1.



Rysunek Nierozwiązywalna plansza

Każdą układankę o 1 wierszu lub 1 kolumnie da się ułożyć tylko wtedy gdy liczby są posortowane rosnąco. Co w przypadku tablicy $n \times m$ gdzie $n, m \geqslant 2$? W swojej pracy *Analysis of the Sixteen Puzzle*[1] Kevin Gong udowadnia że początkowy układ możemy stosunkowo łatwo sprawdzić pod względem rozwiązywalności. W tym celu zdefiniujemy **liczbę inwersji** dla uporządkowanego ciągu liczb.

Definicja 1. *Uporządkowany zbiór $A = (x_1, x_2, ..., x_n)$ posiada k-inwersji jeżeli $x_i < x_j$ w przypadku gdy $i > j$ dla k różnych par i,j.*

Przykładowo, permutacja $(1, 2, 4, 3)$ ma jedną inwersję $4 > 3$. Permutacja $(3, 1, 2, 4)$ ma dwie inwersje $3 > 1$, $3 > 2$. $(1, 2, 3, 4)$ nie ma żadnych.

Dla układanek o rozmiarach $n \times m$ gdzie $n, m \geqslant 2$ prawdziwe są następujące twierdzenia:

Twierdzenie 1. *[1] Jeżeli liczba kolumn w układance jest nieparzysta, to każda poprawna konfiguracja odpowiada permutacji z parzystą liczbą inwersji.*

Twierdzenie 2. *[1] Jeżeli liczba kolumn w układance jest parzysta, to każda poprawna konfiguracja która posiada pusty element w i-tym wierszu, a liczba wierszy pomniejszona o i $(m - i)$,jest liczbą parzystą - posiada parzystą liczbą inwersji.*

Twierdzenie 3. *[1] Jeżeli liczba kolumn w układance jest parzysta, to każda poprawna konfiguracja która posiada pusty element w i-tym wierszu, a liczba wierszy pomniejszona o i $(m - i)$,jest liczbą nieparzystą - posiada nieparzystą liczbą inwersji.*

Sprawdźmy zatem poniższą tablicę przy pomocy powyższych twierdzeń

Rysunek Plansza 2x4

Odpowiadająca jej permutacja to $(7, 2, 1, 4, 6, 3, 5)$. Liczba kolumn jest parzysta, zatem nie skorzystamy z Twierdzenia 1. Liczba wierszy pomniejszona indeks wiersza w którym znajduje się element pusty (m-i) jest równa 1 czyli jest liczbą nieparzystą (Twierdzenie 3). W takim przypadku oczekujemy że liczba inwersji również będzie nieparzysta, aby układanka dała się rozwiązać. Niestety $6 + 1 + 1 + 2 = 10$ zatem układanka ta nie da się rozwiązać.

Powyższe twierdzenia przekładają się na następujący algorytm sprawdzający:

```
bool Board::isSolvable() {
	//Get permutation
	std::vector<int> permutation = this->values;
	//Get rid of 0 element (represents empty puzzle)
	int emptyPuzzleRow;
	for (int i = 0; i < permutation.size(); i++) {
		if (permutation[i] == 0) {
			permutation.erase(permutation.begin()+ i);
			//Getting row of empty puzzle
			emptyPuzzleRow = i / this->cols + 1;
			break;
		}
	}
	//Trivial case
	if (this->cols == 1 || this->rows == 1) {
		return std::is_sorted(permutation.begin(),
			permutation.end());
	}
	//Getting inversions count
	int inversions = 0;
	for (int i = 0; i < permutation.size(); i++) {
		for (int j = i+1; j < permutation.size(); j++) {
			if (permutation[i] > permutation[j])
				inversions++;
		}
	}

	if (this->cols % 2 == 1 && inversions%2 == 0) { //Thrm 1
		return true;
	}
	if (this->cols %2 == 0) { //Theorem 2 and 3
		if ((this->rows - emptyPuzzleRow) % 2 ==
			0 && inversions % 2 == 0) return true;//2
		else if ((this->rows - emptyPuzzleRow) % 2 ==
			1 && inversions % 2 == 1) return true;//3
	}
	return false;
}
```

2.2. Ilość dostępnych permutacji tablicy

Twierdzenie 4. *[1] W przypadku tablicy o jednej kolumnie lub jednym wierszu, oczywistym jest że liczba możliwych stanów jest równa liczbie wszystkich elementów tablicy. W przypadku gdy tablica ma rozmiar $n \times m$ gdzie $n, m > 1$ Istnieje dokładnie $\frac{(n \times m)!}{2}$ rozwiązywalnych konfiguracji.*

Poniższa tabela podaje nam liczbe rozwiązywalnych stanów w poszczególnych rozmiarach układanki.

Rozmiar Tablicy	Rozwiązywalne stany
$1 \times n$	n
2×2	12
2×3	360
2×4	20 160
2×5	1 814 400
2×6	239 500 800
2×7	43 589 145 600
2×8	10 461 394 944 000
3×3	181 440
3×4	239 500 800
3×5	653 837 184 000
4×4	10 461 394 944 000

Tabela. Liczba możliwych stanów w zależności od rozmiaru

Oczywiście tabela rozszerza się na większe rozmiary, jednakże w naszym programie ze względu na ograniczenia pamięci i sprzętowe, rozpatrujemy tablice o maksymalnie 16 elementach.

Warto wspomnieć że w przypadku losowania tablicy do dalszej części eksperymentu, około połowa przypadków była nierozwiązywalna. Wynika to z faktu, że istnieje dokładnie tyle samo stanów nierozwiązywalnych co rozwiązywalnych. Dowiedzione to zostało w pracy [1]*Analysis of the Sixteen Puzzle.*

3. Algorytm BFS

Algorytm przeszukiwania grafu wszerz polega na pobieraniu najpierw najbliższych sąsiadów w grafie , sprawdzanie ich stanów i dopiero potem pobierania "poziom niżej". Tak zdefiniowany sposób zawsze znajdzie nam najkrótszą drogę, jednakże musi wykonać bardzo dużą liczbę porównań. Poniżej w tabeli przedstawione są wyniki z kilkunastu prób algorytmu BFS z losowego stanu początkowego.

Rozmiar Tablicy	Długość Ścieżki	Odwiedzone stany	Czas rozwiązania
2×2	5	10	0ms
2×2	6	12	0ms
2×2	6	12	0ms

3×2	14	219	0ms
3×2	14	215	0ms
2×3	9	61	0ms
2×3	13	157	0ms
2×4	23	8 707	12ms
2×4	19	4 906	6ms
2×4	25	13 776	20ms
3×3	24	119 896	269ms
3×3	22	92 739	150ms
3×3	18	20 101	24ms
3×3	20	54 782	73ms
3×3	18	19 819	34ms
3×3	24	116 132	202ms
3×3	26	158 738	314ms
3×3	23	103 843	171ms
3×3	25	155 378	291ms
2×6	Nie osiągnięto	48 814 626	$\sim$ 75s
2×6	Nie osiągnięto	48 814 626	$\sim$ 75s
2×6	Nie osiągnięto	48 665 982	$\sim$ 75s
4×4	Nie osiągnięto	43 740 510	$\sim$ 75s
4×4	Nie osiągnięto	43 749 238	$\sim$ 75s
4×4	Nie osiągnięto	43 747 957	$\sim$ 75s
4×4	Nie osiągnięto	43 746 019	$\sim$ 75s

Tabela. Wyniki dla algorytmu BFS.

Jak widzimy w przypadku tego algorytmu (Możemy to głównie zaobserwować z przypadków 3×3) algorytm zwykle odnajduje ścieżkę po przejrzeniu znacznej ilości dostępnych stanów. Można wywnioskować że pomimo bardzo atrakcyjnych długości ścieżek (najkrótszych możliwych) algorytm jest mało wydajny.

4. Algorytm DFS

Algorytm przeszukiwania grafu w głąb przeszukuje w sposób analogiczny do algorytmu BFS, z tą różnicą że zamiast przejrzeć najpierw cały poziom, to przegląda najpierw całą ścieżkę. Dopiero gdy ta mu się kończy, to wraca do ostatniego rozgałęzienia i od kolejnej krawędzi rozpoczyna poszukiwanie znowu. Tak zdefiniowany sposób znajduje niejednokrotnie ekstremalnie długie ścieżki. Poniżej w tabeli przedstawione są wyniki z kilkunastu prób algorytmu DFS z losowego stanu początkowego.

Rozmiar Tablicy	Długość Ścieżki	Odwiedzone stany	Czas rozwiązania
2×2	2	3	0ms
2×2	9	10	0ms

2×2	9	10	0ms
2×3	1	360	0ms
2×3	120	161	0ms
3×2	138	307	0ms
4×2	2667	2 979	11ms
4×2	8242	13 911	37ms
4×2	5290	6 016	13ms
3×3	54051	130 784	303ms
3×3	59936	72 816	231ms
3×3	19510	20 438	52ms
3×3	56646	126 239	309ms
3×3	38910	148 557	367ms
3×4	Nie osiągnięto	23 841 458	$\sim$ 75s
2×6	Nie osiągnięto	28 334 635	$\sim$ 75s
2×6	Nie osiągnięto	28 334 887	$\sim$ 75s
4×4	Nie osiągnięto	22 492 606	$\sim$ 75s
4×4	Nie osiągnięto	22 493 316	$\sim$ 75s

Tabela. Wyniki dla algorytmu DFS.

Podobnie jak algorytm BFS algorytm ten również przegląda znaczną liczbę stanów przed odnalezieniem rozwiązania. W dodatku ścieżki otrzymane w jego wyniku są bardzo długie. W kontekście rozwiązywania piętnastki, trudno dostrzec zalety tego algorytmu w porównaniu do innych. Algorytm sprawdzi się lepiej tylko w bardzo szczególnych przypadkach.

5. Algorytm A*

Algorytm A* w celu usprawnienia przeszukiwania przyjmuje funkcję heurystyczną. Funkcja ta definiuje jakąś miarę podobieństwa między elementem odwiedzanym a poszukiwanym. W każdym kolejnym kroku wybieramy element który jest najbardziej podobny do poszukiwanego rozwiązania. Algorytmy A* mają znaczną przewagę nad BFS i DFS jeżeli tylko taką miarę da się zdefiniować i elementy podobne są blisko siebie.

5.1. Heurestyka Tile On Place

Heurystyka Tile On Place daje 1 punkt za to że element przebywa na swoim miejscu. Policzmy heurystykę dla poniższej tablicy.

Jak widzimy jedynie element ”6” jest na właściwym miejscu. Zatem funkcja zwróci nam wartość 1. Do dalszego przeszukiwania wybierane są elementy ze zmaksymalizowaną wartością heurystyki. Poniżej w tabeli przedstawione są wyniki z kilkunastu prób algorytmu A* z heurystyką Tile on Place z losowego stanu początkowego.

Rozmiar Tablicy	Długość Ścieżki	Odwiedzone stany	Czas rozwiązania
2×2	0	1	0ms
2×2	2	3	0ms
2×2	3	4	0ms
2×2	8	9	0ms
3×3	146	4 755	8ms
3×3	57	297	0ms
3×4	323	384 135	857ms
3×4	189	21 642	43ms
3×4	108	10 543	21ms
3×4	298	384 058	997ms
3×4	311	384 410	1105ms
3×4	306	384 389	970ms
4×4	Nie Osiagnieto	31 708 484	$\sim$ 75s
4×4	Nie Osiagnieto	30 575 910	$\sim$ 75s

Tabela. Wyniki dla algorytmu A* z heurystyką Tile On Place.

To już są dużo lepsze pod względem liczby odwiedzonych stanów wyniki niż było to w przypadku algorytmów DFS i BFS. Ponadto udało nam się znaleźć rozwiązania układanek 3x4! (Wcześniejsze algorytmy nie znalazły, ze względu na ograniczenia sprzętowe) Z powodzeniem można stwierdzić że algorytmy wykorzystujące heurystykę do przeszukiwania przestrzeni stanów piętnastki zdecydowanie się nadają.

5.2. Heurestyka Manhattan

Heurystyka Manhattan daje każdej cyfrze na planszy liczbę punktów która odpowiada odległości w jakiej znajduje się od swojego miejsca w rozwiązaniu (Mówimy oczywiście o odległości w metryce Manhattan). Policzmy heurystykę dla poniższej tablicy. Poniżej w tabeli przedstawione są wyniki z kilkunastu prób algorytmu A* z heurystyką Manhattan z losowego stanu początkowego.

I tak kolejno policzmy dla każdej cyfry:
0. (Puste Pole) Raz w dół i 3 razy w prawo. Zatem odległość 4.

1. 3x w lewo. Odległość 3.
2. 1x w lewo. Odległość 1.
3. 1x do góry. Odległość 1.
4. 3x w prawo, raz do góry. Odległość 4.
5. 3x w lewo. Odległość 3.
6. Na swoim miejscu. Odległość 0.
7. Raz w prawo i Raz w dół. Odległość 2.

Zatem całkowita odległość dla tej tablicy wynosi $4+3+1+1+4+3+0+2=18$. Do dalszego przeszukiwania tablicy wybieramy stan o najmniejszej odległości.

Rozmiar Tablicy	Długość Ścieżki	Odwiedzone stany	Czas rozwiązania
2×2	1	2	0ms
2×2	2	3	0ms
2×2	1	2	0ms
2×2	5	8	0ms
2×2	4	5	0ms
3×3	46	378	5ms
3×3	65	196	3ms
3×3	190	1722	30ms
3×3	113	690	13ms
3×3	128	1374	23ms
3×3	58	274	5ms
3×3	54	743	12ms
3×3	128	1012	18ms
3×3	59	323	4ms
3×3	111	640	10ms
3×3	80	1619	37ms
3×3	82	350	5ms
3×3	58	302	4ms
3×4	121	1015	27ms
3×4	175	1400	41ms
3×4	148	1814	43ms
3×4	510	7348	251ms
4×4	231	10361	401ms
4×4	641	29118	968ms
4×4	412	27857	905ms
4×4	436	12377	478ms
4×4	577	11094	351ms

Tabela. Wyniki dla algorytmu A* z heurystyką Manhattan.

W porównaniu do poprzednich algorytmów, A* z heurystyką Manhattan wypada zdecydowanie najlepiej! Znajdujemy rozwiązania dużych tablic przy sprawdzaniu nie wielkiej liczby stanów w krótkim czasie.

6. Podsumowanie

Pierwszym rzucającym się w oczy faktem, którego potwierdzenie widać w ilości przeszukanych stanów dla poszczególnych algorytmów, jest to że dodanie do planszy kolejnych wierszy lub kolumn diametralnie zmienia liczbę dostępnych rozwiązań. I w ten sposób jeżeli byśmy postanowili przeszukać algorytmem BFS tablice o rozmiarze 5×5, to mielibyśmy do przeszukania 'tylko' 7 755 605 021 665 492 992 000 000. To 'jedynie' 741 354 768 000 razy więcej niż w przypadku 4×4. Tą drugą liczbę idzie nawet przeczytać, chociaż zakładam że czytelnik i tak tego nie zrobił :). Taka własność powoduje konieczność zastosowania odpowiednich heurystyk dla tablic wyższych rzędów.

DFS pomimo jasno zdefiniowanych zasad układania szuka nijako "na ślepo", nie wnosząc niemalże niczego nowego w stosunku do tego co dawał nam BFS. Dużą zaletą BFS'a jest fakt że zawsze znajdziemy najkrótszą ścieżkę do rozwiązania - co może być niejednokrotnie dla nas cenniejszą informacją niż dłuższa ścieżka znaleziona szybciej. Z porównywanych algorytmów zdecydowanie najszybszym jest algorytm A* z heurystyką Manhattan. Jednakże nawet on w przypadkach większych niż 4x4 okazałby się niewystarczająco szybki

Warto zauważyć że wszystkie przedstawione algorytmy są zupełne. Oznacza to że w przypadku istnienia rozwiązania z pewnością zostanie ono przez algorytm znalezione. W tabelach w przypadku gdy mamy informację 'Nie Osiągnięto' oznacza to ograniczenia sprzętowe, a nie ograniczenia algorytmu.

Literatura

[1] Kevin Gong. Analysis of the Sixteen Puzzle. `http://kevingong.com/Math/SixteenPuzzle.html`, 2000. [Online; Ostatni dostęp 09.12.2017].