Informatyka, studia dzienne, inż I st. semestr V

Sztuczna inteligencja i systemy ekspertowe **2018/2019**

Prowadzący: Przemysław Nowak Czwartek, 12:00

Data oddania: _____ Ocena: _____

Patryk Lisik 210254
Adam Sadowski 210310

Zadanie 1: Piętnastka

1. Cel

Implementacja kilku algorytmów przeszukiwania przestrzeni stanów oraz rozwiązanie i analiza wyników zadanych stanów układanki "piętnastka".

2. Wprowadzenie

2.1. Piętnastka

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Tablica 1: Rozwiązana piętnastka

1	2	3	4
5	6	7	8
9	10	11	12
13	14		15

Tablica 2: Układ wymagający jednego ruchu

Piętnastka jest grą logiczną wymyśloną pod koniec XIX wieku. Jej nazwa pochodzi od piętnastu ponumerowanych 1–15, które trzeba ułożyć tak jak w tablicy 1.

Stan końcowego musi musi zostać osiągnięty jedynie poprzez ruchy pustego pola w obrębie macierzy 4×4. Jeśli puste pole znajduje sie przy krawędzi niektóre ruchy mogą nie być dozwolone. W dalszej części ruchy będą oznaczane.
— L – (ang. left) w lewo
— R – (ang. right) w prawo
— D – (ang. down) w dół
— U – (ang. up) w górę

2.1.1. Złożonosć i rozwiązywalnosć

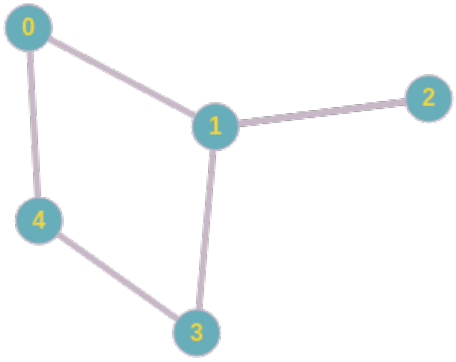
Ilość wszystkich stanów układanki można łatwo obliczyć znakując permutację bez powtórzeń zbioru elementów układanki($\{$"*puste_pole*", $1, 2 \ldots, 15\}$). Stany układanki można podzielić ze względu na ich parzystość. Nie można przejść z układu do nieparzystego, czego oczywistą implikacją jest nierozwiązywalność połowy układów[4]. Podsumowując ilość rozwiązywalnych stanów to $\frac{16!}{2} \approx 1.04 \cdot 10^{13}$. Zakładając, że nie jest możliwe wykonywanie ruchów przesuwających 3 pola naraz maksymalna głębokość rozwiązania wynosi 83, jeśli takie ruchy są dozwolone 43 [3].

2.2. Grafy

Grafem nazywamy strukturę danych modelującą zbiór wierzchołków i połączeń między nimi.

2.2.1. Reprezenatacja

Najbardziej oczywistą formą reprezentacji grafów sa rysunki podobne do ilustracji 1. Nie jest to jednak optymalny sposób przedstawiania grafu w pamięci komputera.



Rysunek 1: Przykładowy graf

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	0
2	0	1	0	0	0
3	0	1	0	0	1
4	1	0	0	1	0

Tablica 3: Przykładowa macierz sąsiedztwa

Macierz sąsiedztwa to sposób reprezentacji grafu w którym krawędzie reprezentowane są przez wartość 1 w komórce której pozycja odpowiada numerom wierzchołków. Na rysunku 1 widzimy połączenie 1–2. Jest ono reprezentowane w tablicy 3 poprzez 1 na pozycji 12 i 21. Redundancja informacji spowodowana jest tym, że graf na rysunku 1 jest nieskierowany(nie można wyróżnić kierunku połączenia). Wady tej metody reprezentacji ujawniają się w przypadku grafów o małej liczbie połączeń. Ilość elementów w macierzy jest proporcjonalna do kwadratu ilości wierzchołków, gdy ilość połączeń jest znikoma duża część przechowywanych informacji to 0.

0	1	4	
1	2	0	3
2	1		
3	1	4	
4	3	0	

Tablica 4: Przykładowa lista sąsiedztwa

Lista sąsiedztwa to sposób reprezentacji grafu przez wektor wektorów, gdzie indeksami pierwszej z nich(pogrubione cyfry w tablicy 4) oznaczany numer wierzchołka. Pod każdym z indeksów znajduje się lista połączeń każdego wierzchołka. Zaletą tego rozwiązania jest stały czas dostępu do listy połączeń danego wierzchołka.

2.2.2. Metody przeszukiwania

DFS Depth first search – przeszukiwanie w głąb
Opiera się na przeszukiwaniu od korzenia wzdłuż każdej gałęzi tak głęboko, jak to możliwe zanim zacznie się cofać

BFS Breath first search – przeszukiwanie w szerz
Opiera się na przeciwnej strategii do DFS i najpierw przeszukuje najpłycej położone węzły, zanim zejdzie głębiej. Z pośród rozważanych algorytmów jest to jedyny gwarantujący znalezienie najbliższego rozwiązania.

A* A star/gwiazdka – heurystyczne przeszukiwanie
Opiera się na zupełnie innym podejściu. W każdej iteracji wyznaczany jest koszt stanów, zgodnie z używaną metryką, i algorytm przeszukuje drzewo w pierwszej kolejności po węzłach o najniższym koszcie.
Wykorzystane metryki to:

Manhattan Dla każdego punktu liczona jest odległość od punktu docelowego

$$d = \sum_{i=1}^{n} |x_i - x_i'| + |y_i - y_i'|$$

Hamminga Koszt zwiększa się wraz z każdym elementem będącym na złej pozycji

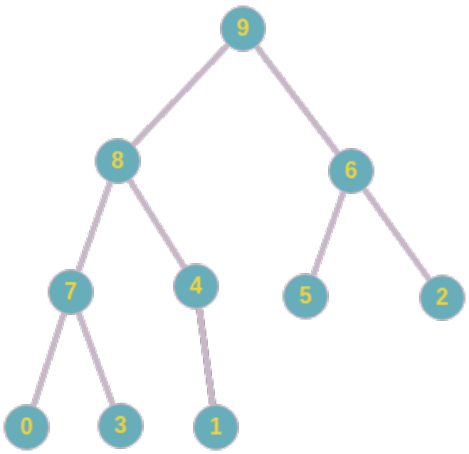
2.2.3. Pozostałe pojęcia

Cykliczność Graf można zaklasyfikować jako cykliczny kiedy daje się w nim wytyczyć ścieżkę która zaczyna i kończy się w tym samym wierzchołku[5]. Przykładowy graf z rysunku 1 ma cykl składający się z wierzchołków 1–0–4–3–1.

Droga

2.3. Struktury danych

2.3.1. Kopiec



Rysunek 2: Przykładowy graf drzewiasty

9	8	6	7	4	5	2	0	3	1

Tablica 5: Kopiec powstały z drzewa rys.2

Kopiec jest strukturą danych reprezentującą drzewo binarne(spójny graf acykliczny) w postaci wektora[6]. Kopce można podzielić na dwa typy[7]:
Kopce max gdzie wartość potomka jest zawsze mniejsza niż wartość rodzica.
Kopce min gdzie wartość potomka jest zawsze większa niż wartość rodzica.

Rysunek 2 przedstawia drzewo binarne które zostało przedstawione jako kopiec w tablicy 5. Kopiec można wyobrażać sobie jako kolejne poziomy drzewa binarnego dodawane na koniec tablicy. Dla ułatwienia w tablicy 5 kolejne poziomy drzewa zostały oddzielone dodatkową linią. Łatwo zauważyć, że pierwszy element nazywany też korzeniem(ang.root) jest w zależności od typu kopca jego największym(kopiec max) lub najniższym(kopiec min) elementem. Struktura jest używana aby móc szybko otrzymać największy lub najmniejszy element kolekcji[6].

2.3.2. HashSet

Set jest nieuporządkowanym zbiorem elementów bez duplikatów [8]. Jego główną zaletą jest szybka możliwość sprawdzenia czy obiekt znajduje się w nim. Średnia złożoność takiej operacji to O(1) [9]. Najgorszy przypadek sprawdzenia czy dany element jest w secie występuje gdy każdy element kolekcji powoduje kolizję tego samego haszu. W takim przypadku złożoność wynosi O(n)[9].

Set jest zwykle implementowany jako wektor wektorów gdzie index dodawanego elementu oblicza się ze wzoru $inex = hash(obj) \pmod{set\_len}$ Jeśli wystąpi kolizja

haszy, czyli funkcja haszująca zwróci tą samą liczbę lub hasze przystają modulo długość wektora setu, wtedy dwa lub więcej elementy znajdują w wektorze pod danym indeksem.

3. Opis implementacji

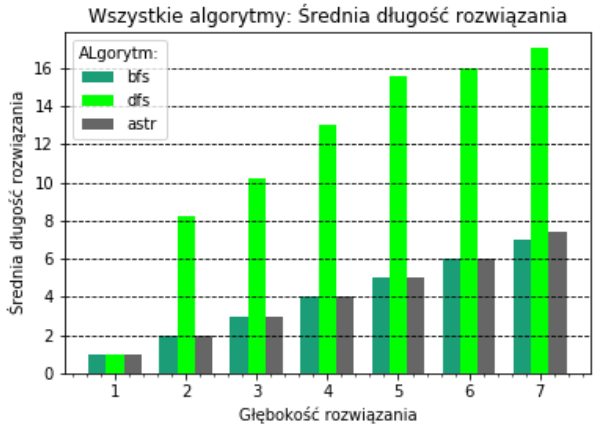
Implementację wykonano w języku Python3. Całość implementacji poza parserem parametrów i funkcją main() zamyka się w jednej klasie o długości ok. 200 linii, dlatego zrezygnowaliśmy z zamieszczania diagramu UML.

4. Materiały i metody

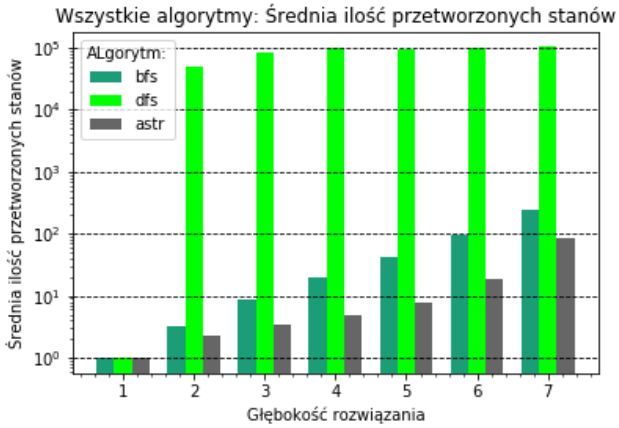
1. Przygotowanie danych
   — Pobranie z platformy WIKAMP generatora układanek 4×4
   — Wygenerowanie przypadków testowych
2. Uruchomienie przygotowanego programu
   — Pobranie z platformy WIKAMP skryptu uruchamiającego.
   — Modyfikacja polecenia uruchamiającego w skrypcie.
   — Uruchomienie skryptu
3. Analiza wyników i generowanie wykresów przy pomocy Jupyter Notobook'a
4. Profit

Celem zachowania wiarygodności pomiarów czasów program uruchomiono w środowisku tekstowym, jako process o największym priorytecie. Podczas przetwarzania nie doszło do użycia pamięci swap ani throttlingu procesora.

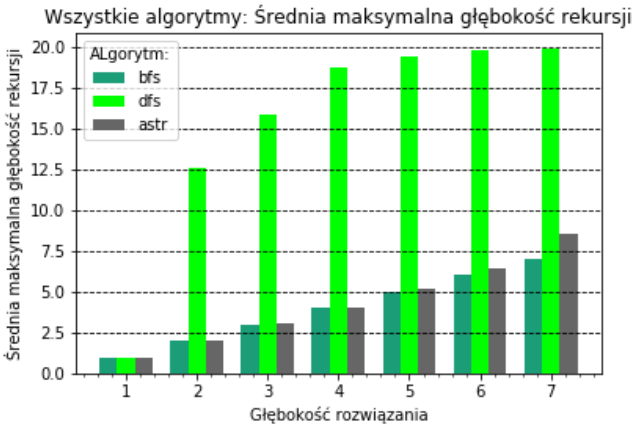
5. Wyniki



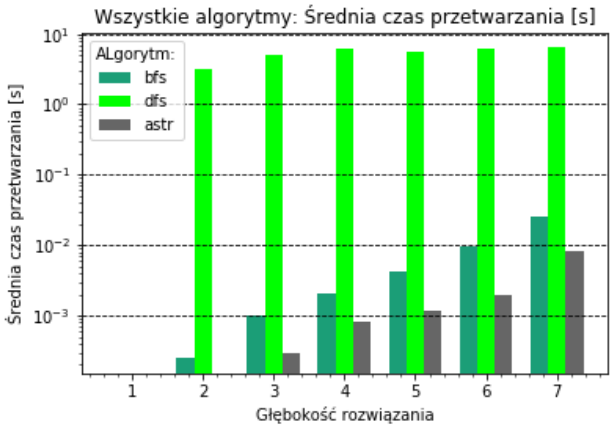
(a) Średnia długość rozwiązania



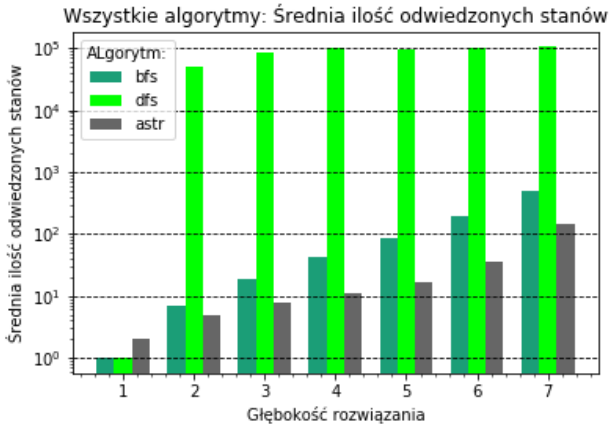
(b) Średnia ilość przetworzonych stanów



(c) Średnia maksymalna głębokość rekursji

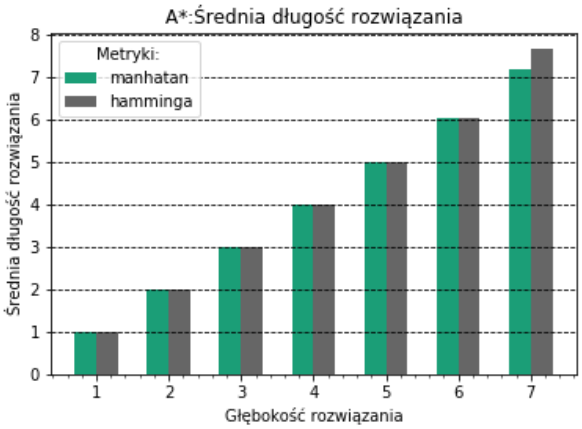


(d) Średni czas przetwarzania

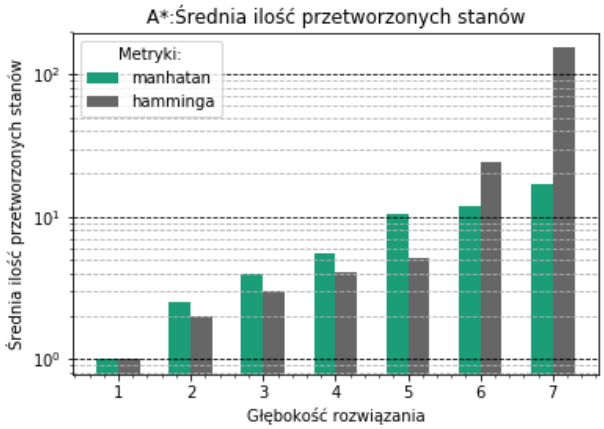


(e) Średnia ilość odwiedzonych stanów

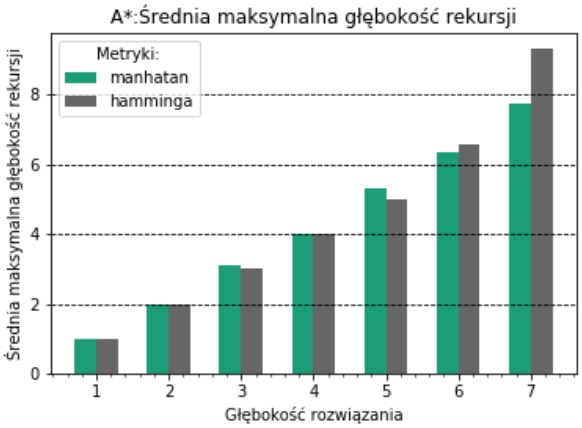
Rysunek 3: Porównanie wszystkich metod przeszukiwania



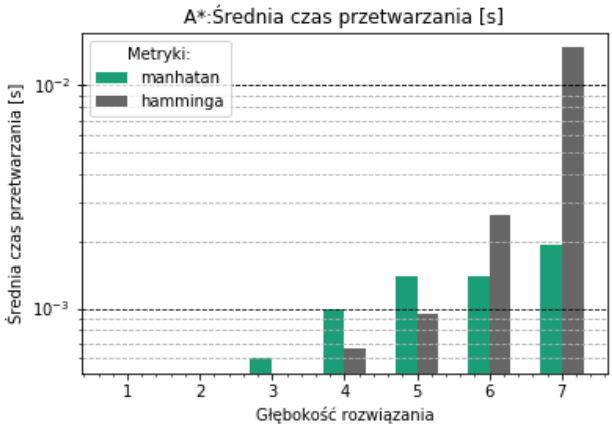
(a) Średnia długość rozwiązania



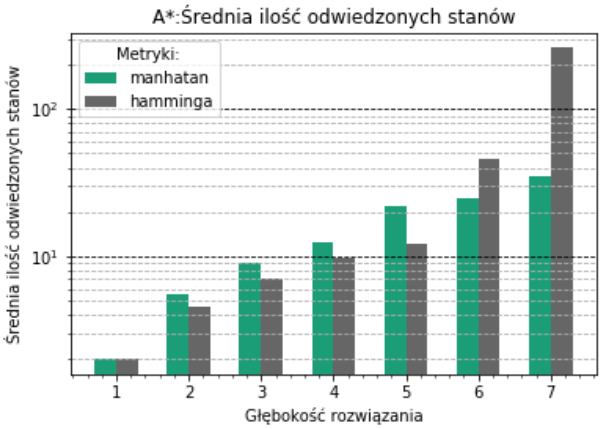
(b) Średnia ilość przetworzonych stanów



(c) Średnia maksymalna głębokość rekursji

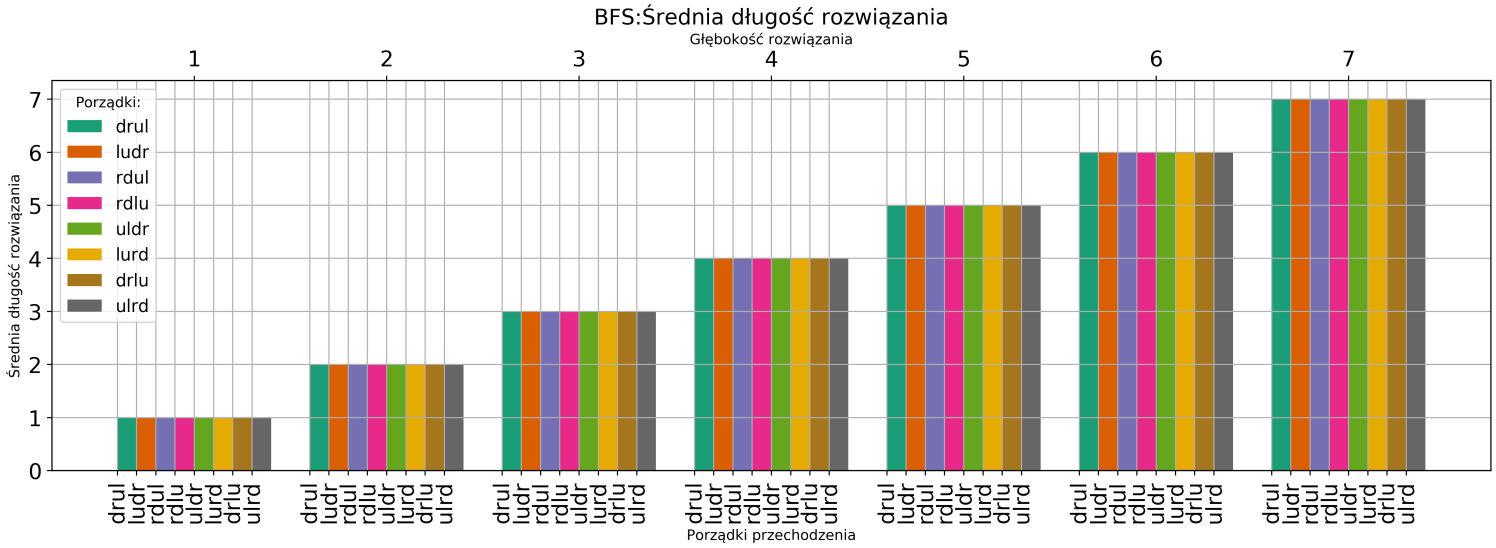


(d) Średni czas przetwarzania

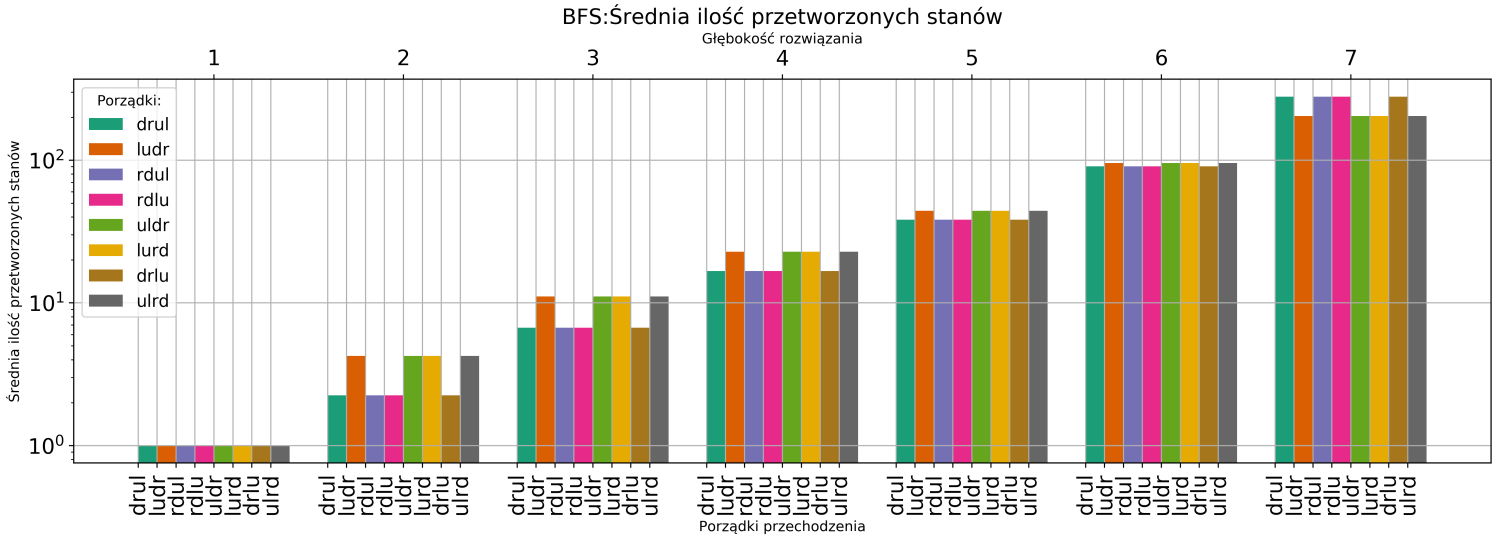


(e) Średnia ilość odwiedzonych stanów

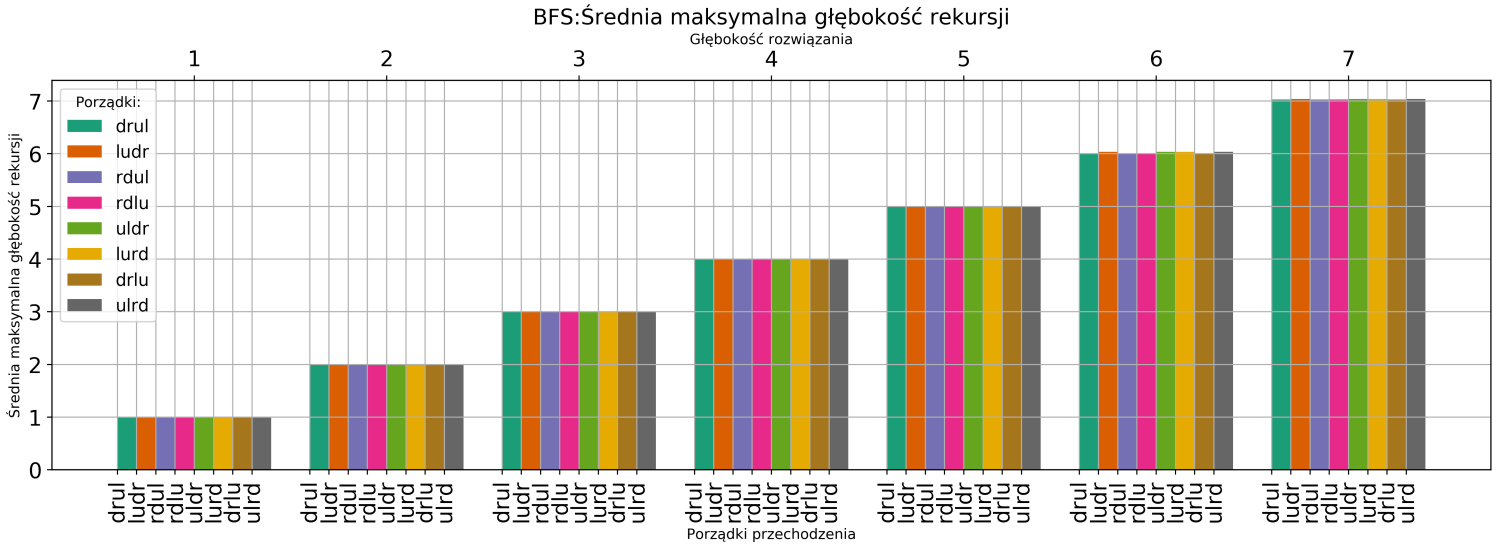
Rysunek 4: Porównanie wszystkich metod przeszukiwania



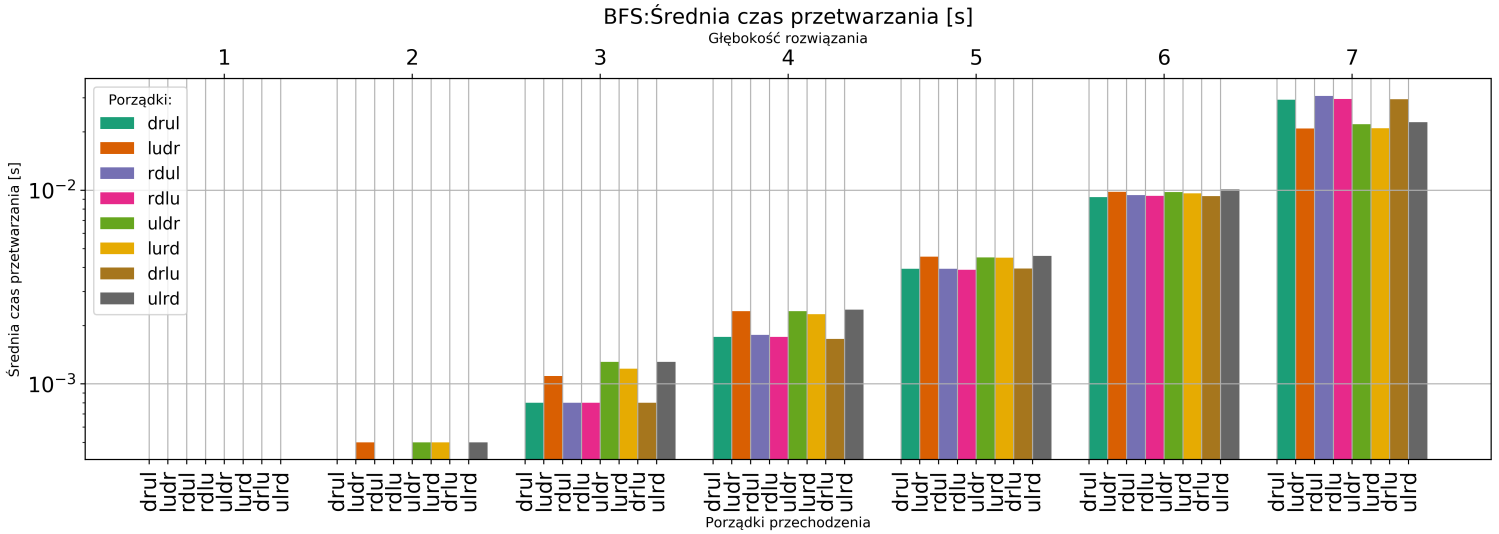
Rysunek 5: BFS – Średnia długość rozwiązania



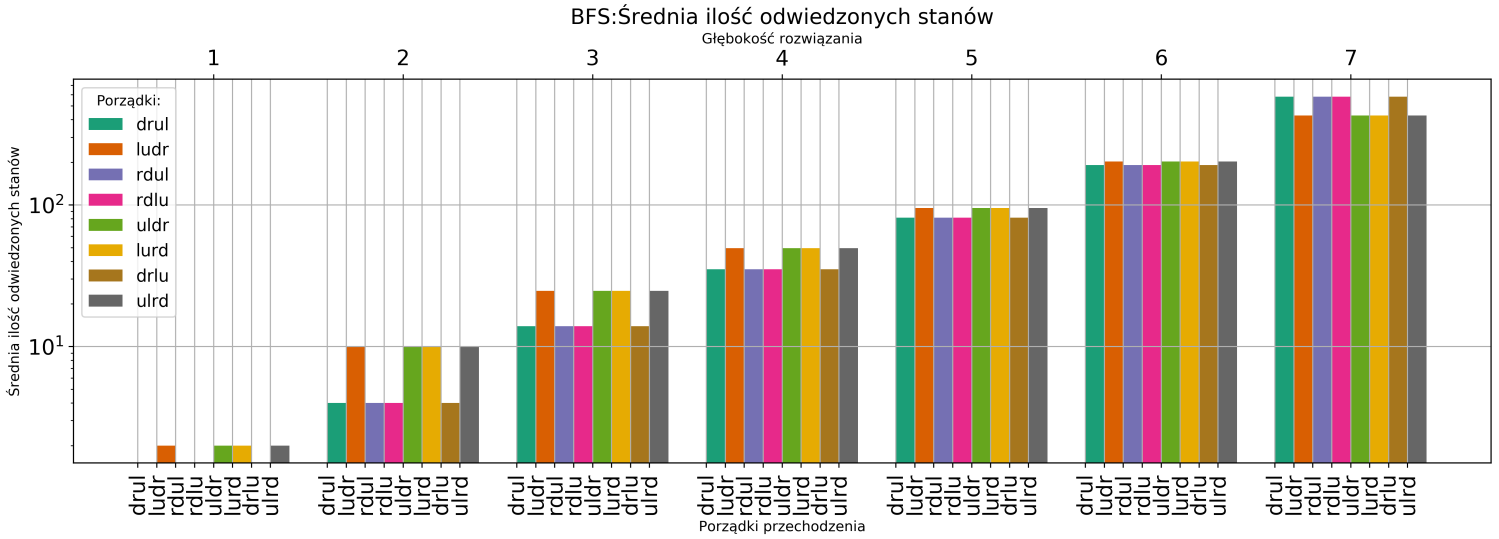
Rysunek 6: BFS – Średnia ilość przetworzonych stanów



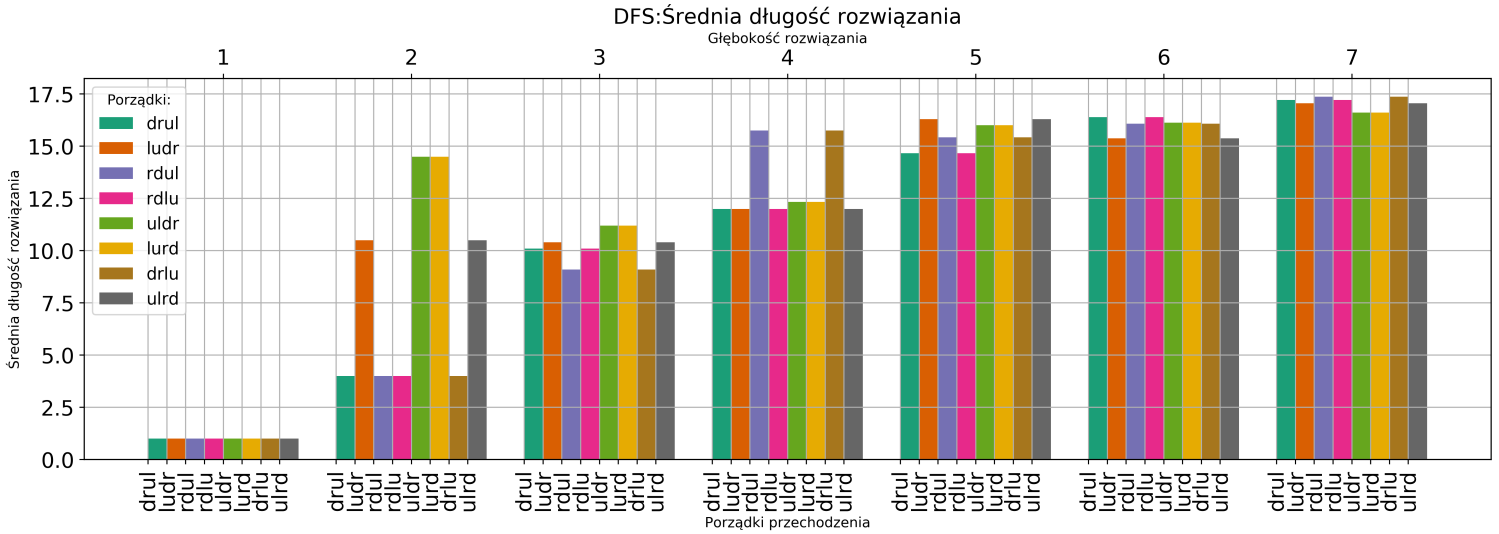
Rysunek 7: BFS – Średnia maksymalna głębokość rekursji



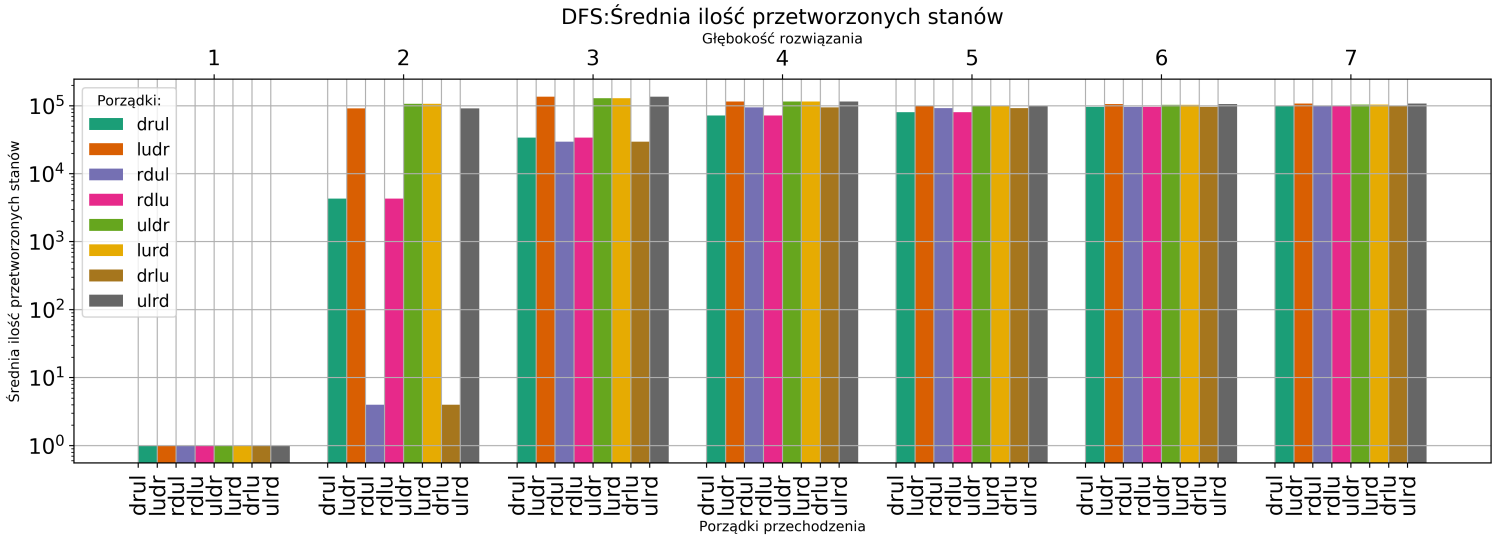
Rysunek 8: BFS – Średni czas przetwarzania



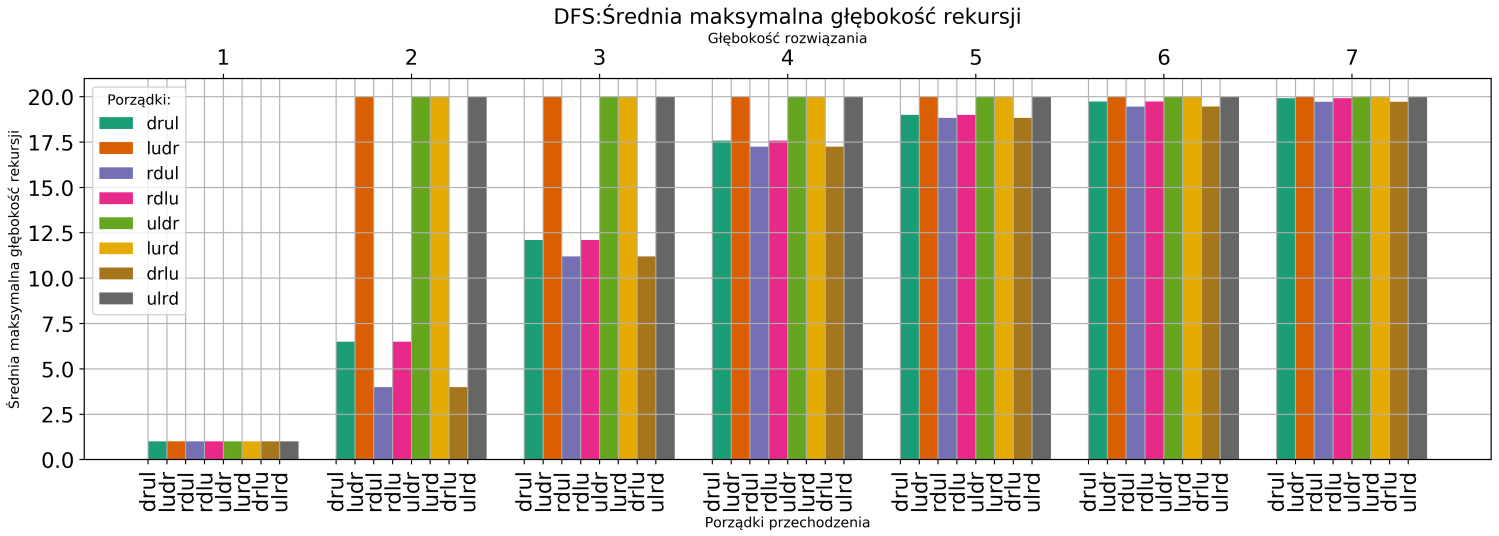
Rysunek 9: BFS – Średnia ilość odwiedzonych stanów



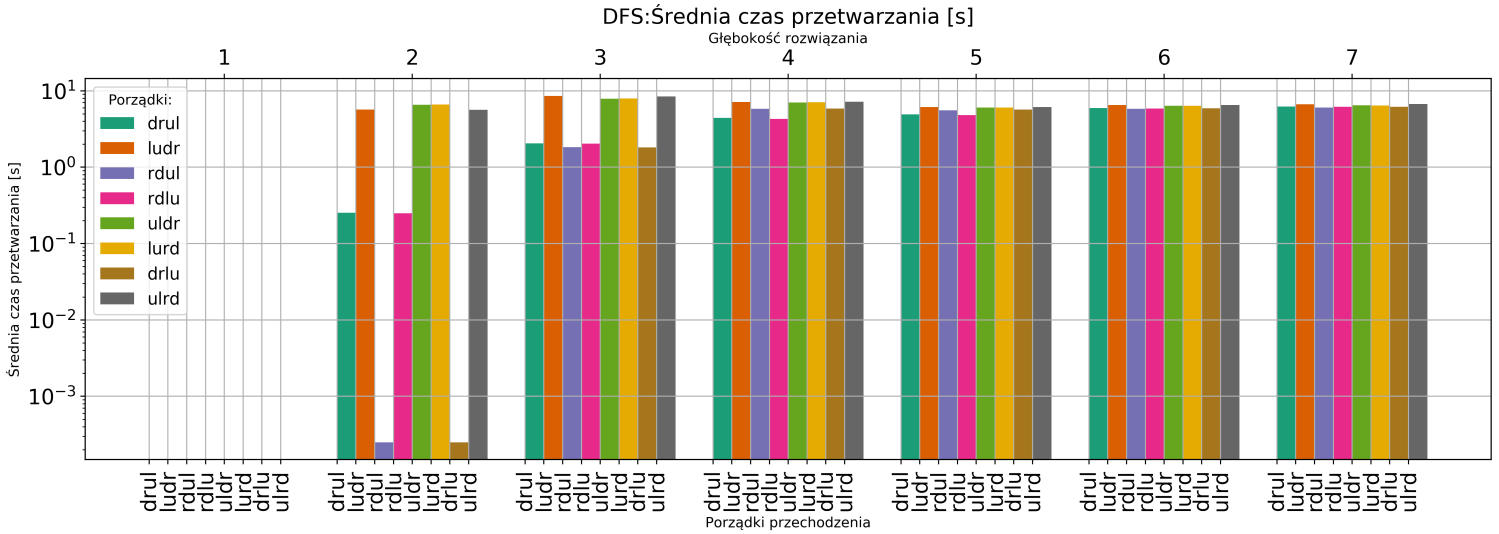
Rysunek 10: DFS – Średnia długość rozwiązania



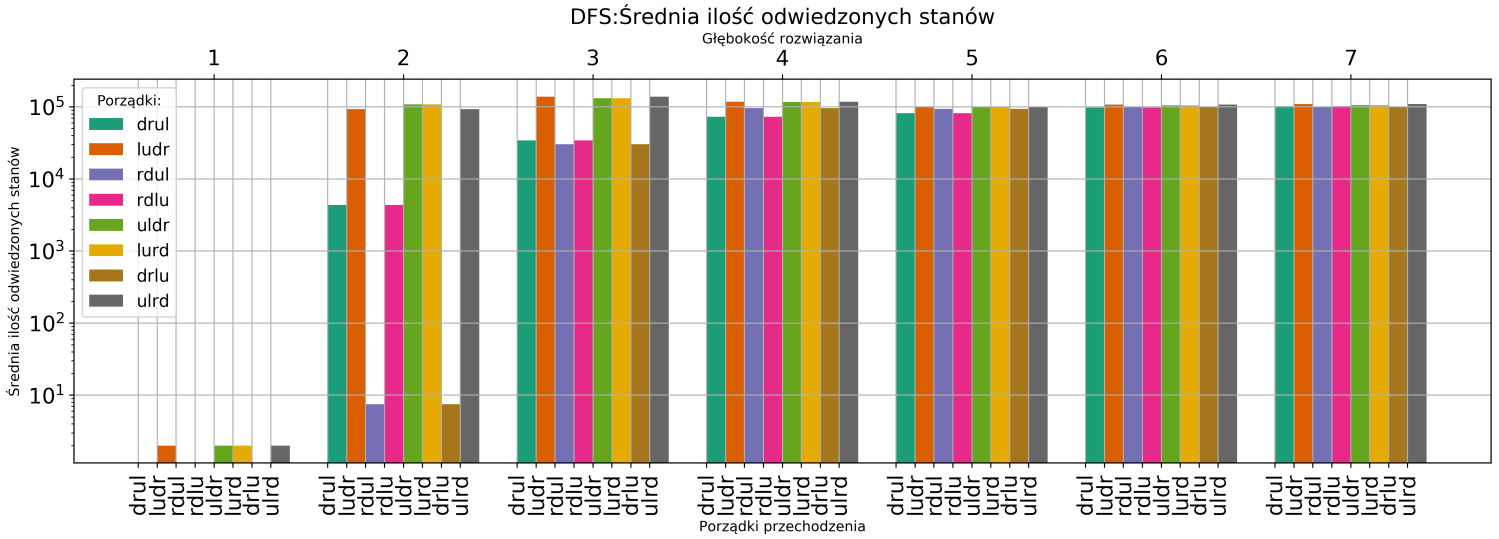
Rysunek 11: DFS – Średnia ilość przetworzonych stanów



Rysunek 12: DFS – Średnia maksymalna głębokość rekursji



Rysunek 13: DFS – Średni czas przetwarzania



Rysunek 14: DFS – Średnia ilość odwiedzonych stanów

Rysunek 3 pokazuje, że A* jest najlepszy jeśli chodzi o ilość przetworzonych stanów i czas rozwiązania. DFS jest zawsze najmniej optymalny. Przedstawione na rysunku 4 wyniki metryk algorytmu A* wskazują iż metryka Hamminga działa lepiej dla rozwiązań położonych płycej niż 6. Wyniki zaprezentowane na rysunkach 5, 6, 7, 9 i 8 wskazują na brak korelacji między wynikami a użytym porządkiem sąsiedztwa. W przypadku algorytmu DFS w 1184 przypadkach rozwiązanie nie zostało odnalezione przyjęto wtedy, że długość ścieżki jest równa głębokości rekurencji, czyli 20.

6. Dyskusja

6.1. Porównanie wszystkich algorytmów

Najbardziej oczywistym wnioskiem płynącym z rysunku 3 jest to iż, z pośród rozważanych algorytmów DFS jest średnio najmniej optymalny. Otrzymane rezultaty tego sposobu rozwiązania, wymusiły użycie skali połlogarytmicznej, aby utrzymać czytelność wykresów. Ilustracja 3a pokazuje, że DFS średnio odnajduje rozwiązanie dalekie od optymalnego. Na diagramie 3c widać, że osiągana głębokość przeszukiwania nie zwiększa się znacząco, jeśli rozwiązanie położone jest w odległości 4 lub większej. Powodem tego jest odgórnie nałożony limit maksymalnej głębokości rekursji wynoszący w naszym przypadku 20. Na rysunkach 3b i 3e widać, że znacznie większa głębokość rekursji koreluje z wielokrotnie wyższymi statystykami wierzchołków odwiedzonych i przetworzonych. Potrzeba przetworzenia znacznie większej części przestrzeni stanów objawi się też znacznie dłuższym czasem wykonania widocznymi na 3d. Nieoptymalność przeszukiwania w głąb jest zgodna z intuicją autorów. Szanse na to, że rozwiązanie położone jest głębiej, niż szerzej wydają się być małe, biorąc pod uwagę jak szybko rozszerza graf przestrzeni stanów.

Przeszukiwanie w szerz będące metodą brute-force szukającą najpłytszego rozwiązania przeszukując graf stanów poziom po poziomie, zawsze osiąga lepsze wyniki niż DFS. Na rysunku 3a i 3c widać, że BFS zawsze odnajduje najpłytsze rozwiązanie. Na wykresach 3b i 3e widać, że ilość operacji jest znacznie mniejsza niż w przypadku metody w głąb i tylko nieznacznie wyższa niż przy użyciu algorytmu A*.

Diagramy 3d, 3b pokazują, że heurystyki dobrze rozwiązują problem przeszukiwania przestrzeni stanów. Czas przetwarzania jest najkrótszy ze wszystkich rozpatrywanych algorytmów. Jak pokazuje 3a szybkie odnajdywanie rozwiązania okupione zostało nieco dłuższą ścieżką do niego prowadzącą.

Jeśli pominiemy wyniki otrzymane przy pomocy algorytmu DFS na ilustracjach 3b, 3d i 3e widzimy liniowy wzrost analogicznych słupków, co w połączeniu z logarytmiczną osią y sugeruje wykładniczą złożoność problemu rozwiązania piętnastki.

6.2. A*

Wykorzystana implementacja algorytmu A* wykorzystuje metryki Hamminga i Manhattan. Jak widać na 4a do głębokości 6 obie metryki znajdują najlepsze rozwiązanie. W przypadku głębokości 7 nieco lepsze wyniki osiąga metryka Manhattan.

Diagram 4b pokazuje, że metryka Hamminga pozwala odnaleźć rozwiązanie, przy mniejszej ilości przetworzonych, jeśli jest ono położone płytko(5 lub wyżej). Metryka Manhattan, pomimo osiągania nieco gorszych wyników, przy małych głębokościach przy dużych wygrywa znacznie. Gdy rozwiązanie wymaga co najmniej 7 ruchów, jego znalezienie wymaga ok 8 razy mniej przetworzonych stanów. Wyniki te bardzo mocno korelują z czasem przetwarzania(rys.4d) i ilością odwiedzonych stanów(rys.4e). Mimo znacznie większej ilości przetworzonych stanów na rysunku 4c widać rezultaty podobne do poprzednich. Do głębokości 5 metryka Hamminga wygrywa lub wyniki są równe. W przypadku głębszych rozwiązań tak jak poprzednio optymalniejsza jest metryka Manhattan, jednak jej przewaga nie jest tak przemożna jak w ilości przetworzonych stanów(rys.4b) i czasie przetwarzania(rys.4d).

Metryka Manhattan, mimo większej złożoności szacowania pojedynczego stanu, osiąga wyniki akceptowalnie dobre wyniki dla płytkich rozwiązań, będąc zwykle nie gorsza niż 20% jednocześnie osiągając znacznie krótsze czasy przetwarzania(rys.4d) i przetworzenia(rys.3b) dla głębszych rozwiązań.

6.3. BFS

Przeszukiwanie w szerz jest algorytmem optymalnym. Oznacza to, że zawsze znajduje rozwiązanie położone tak płytko jak to możliwe. Widać to na wykresach 5 i 7, gdzie głębokość znalezionego rozwiązania jest zawsze najmniejsza, a przeszukiwania nigdy nie osiąga głębokości rekursji większej niż rozwiązanie.

Diagram 6 pokazuje, że testowane porządki nie mają większego wpływu na ilość przetworzonych stanów. Czas wykonania(rys.8) i ilość odwiedzonych stanów(rys.9) wyraźnie korelują z ilością przetworzonych stanów(rys.6), nie różnicując się zbytnio w zależności od porządku.

6.4. DFS

Rozpatrując długość rozwiązania jako miarę jego jakości rysunek 10 ciężko wskazać najlepszy porządek dla algorytmu DFS. Porządki postaci *UL jak RDUL, DRUL i DRLU osiągają lepsze rezultaty dla bardzo płytkich rozwiązań(3 i płytsze),

jednak już na głębokości 4 są wyraźnie gorsze(RDUL i DRLU) lub porównywalne z resztą(RDLU). Na wykresach czasu przetwarzania(rys.8), ilości odwiedzonych(rys.9) i przetworzonych(rys.6) węzłów widoczna jest podobna zależność. Wyniki przeszukiwania w głąb spłaszczone są przez narzucony przez autorów maksymalny limit rekursji wynoszący 20. Osiągnięcie tej wartości powoduje koniec przetwarzania i zwrócenie braku odnalezienia rozwiązania. 1184 przypadki zakończyły się takim rezultatem. Jest to szczególnie widoczne na diagramie średniej maksymalnej rekursji(rys.12), gdzie porządki idące szybko w lewo jak LUDR,ULDR,LURD,UlDR już na głębokości 2 średnio nie znajdują rozwiązania.

7. Wnioski

1. Nie używać DFS
2. Jeśli istnieje potrzeba znalezienia najpłytszego rozwiązania należy wykorzystać BFS.
3. Jeśli czas jest najważniejszym czynnikiem determinującym jakość rozwiązania należy użyć A*.
4. Złożoność problemu rośnie wykładniczo wraz z odległością rozwiązania.
5. Metryka Manhattan lepiej sprawdza się jeśli rozwiązanie jest położone głębiej.
6. Porządek nie ma wpływu na BFS.

Literatura

[1] T. Oetiker, H. Partl, I. Hyna, E. Schlegl. *Nie za krótkie wprowadzenie do systemu LaTeX2e*, 2007, dostępny online.

[2] *15 Puzzle* `http://mathworld.wolfram.com/15Puzzle.html`

[3] *The Fifteen Puzzle can be solved in 43 "moves"* `http://cubezzz.duckdns.org/drupal/?q=node/view/223`

[4] *The Fifteen Puzzle* `http://www.math.ubc.ca/~cass/courses/m308-02b/projects/grant/fifteen.html`

[5] J.Wałaszek *Algorytmy i struktury danych* `https://eduinf.waw.pl/inf/alg/001_search/0132.php`

[6] J. Boccara *105 STL Algorithms in Less Than an Hour* `https://youtu.be/2olsGf6JIkU`

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein *Wprowadzenie do algorytmów*, 2007

[8] *Dokumentacja języka Python 3.7* rozdział 5.4 Sets, dostępna `https://docs.python.org/3/tutorial/datastructures.html`

[9] *Wiki języka Python3* `https://wiki.python.org/moin/TimeComplexity`