Informatyka, studia dzienne, inż I st. semestr V

Sztuczna inteligencja i systemy ekspertowe 2021/2022

Prowadzący: Krzysztof Lichy poniedziałek, 10:00

Data oddania: _____ Ocena: _____

Adam Kapuściński 229907
Damian Szczeciński 230016

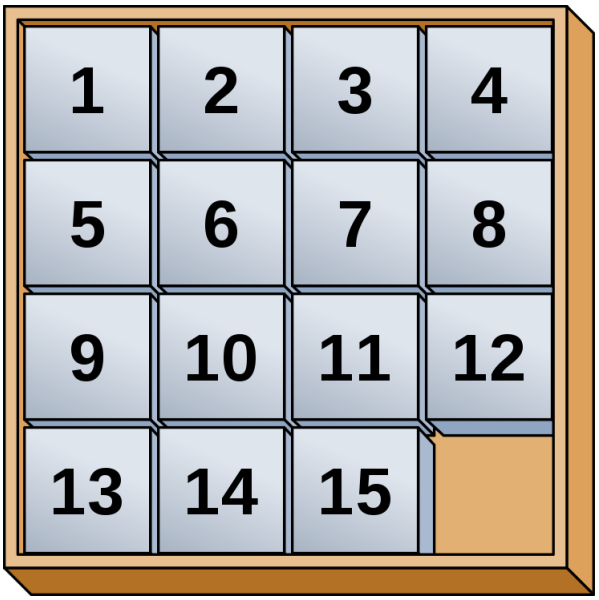
Zadanie 1: Piętnastka

1. Cel

Celem projektu było stworzenie programu do rozwiązywania zagadki "Piętnastki". Program potrafi przyjąć zarówno standardowe zagadki o wielkości planszy 4x4 jak i inne. Program może działać według 3 algorytmów:
— Przeszukiwanie wszerz,
— Przeszukiwanie w głąb,
— Przeszukiwanie algorytmem A*.

2. Wprowadzenie

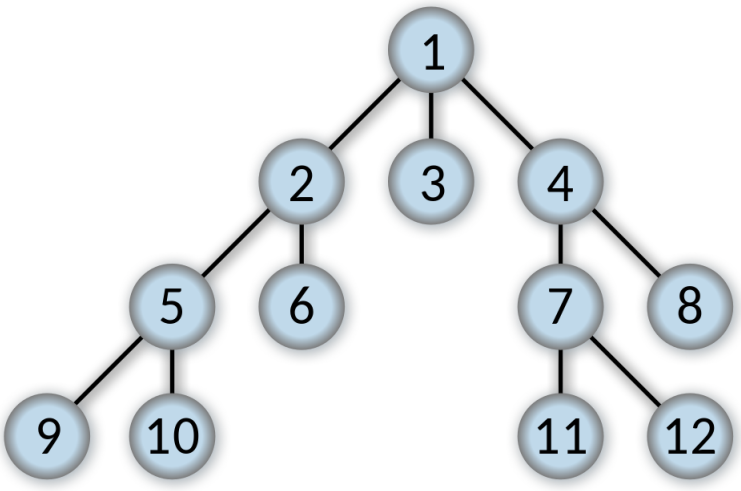
Tytułowa "Piętnastka" jest grą logiczną. W swojej podstawowej wersji posiada planszę 4 na 4 pola, gdzie każde z nich ma wartość z przedziału liczb całkowitych od 1 do 15. Jako, że łączna liczba pól wynosi 16 to otrzymujemy jeden pusty kafelek. Gracz za jego pośrednictwem może zamienić go miejscem z sąsiadującym innym polem z liczbą. Ostatecznym zadaniem gracza jest uzyskanie planszy, na której liczby są ustawione poziomami w kolejności rosnącej od lewego górnego rogu do prawego dolnego, gdzie ostatnie pole jest przewidziane na nasz pusty kafelek. W celu uzyskania takiego efektu gracz powinien odpowiednio przesuwać pusty kafelek z wykorzystaniem jak najmniejszej ilości ruchów.



Rysunek 1. Grafika przedstawiająca poprawne ułożenie "Piętnastki" [5]

2.1. Przeszukiwanie strategią wszerz (BFS)

Strategia przeszukiwania wszerz (ang. Breadth-first search) polega na odwiedzaniu wszystkich nieodwiedzonych sąsiadów danego wierzchołka. Zaczynając od pierwszego wierzchołka przechodzimy do następnych osiągalnych nieodwiedzonych wierzchołków, po czym kontynuujemy tą czynność aż uzyskamy oczekiwany efekt. Podczas tego procesy powstaje graf drzewa przeszukiwania wszerz.



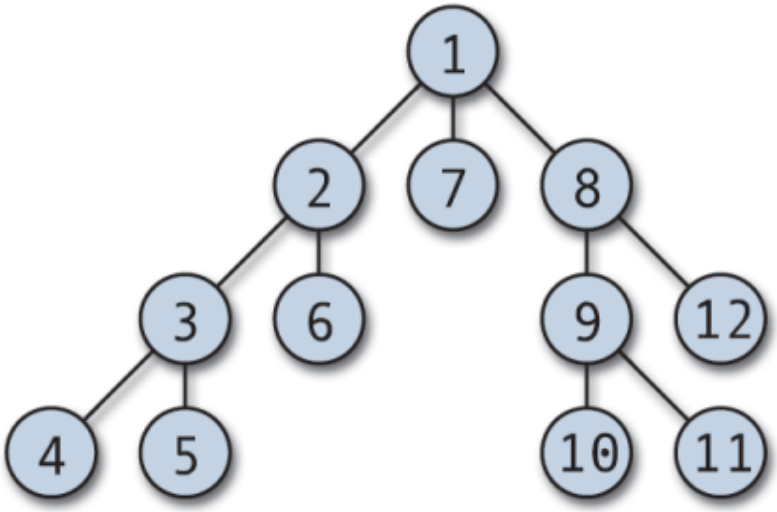
Rysunek 2. Grafika przedstawiająca przykładowy wygląd grafu drzewa, liczby przedstawiają kolejność odwiedzania wierzchołków [4]

Jako, że algorytm ten wymaga zapamiętywania już odwiedzonych pozycji to złożoność pamięciowa takiej strategi jest na poziomie $O(|V| + |E|)$ gdzie

V jest liczbą węzłów a E jest liczbą krawędzi. Taką samą wartość przyjmuje również złożoność czasowa. Wynika to z faktu możliwości zaistnienia sytuacji, gdzie algorytm musi przebyć wszystkie krawędzie prowadzące do wszystkich węzłów. Strategia ta cechuje się również kompletnością, czyli zasadą, że jeżeli rozwiązanie istnieje, to to owe przeszukiwanie je odnajdzie niezależnie od grafu.

2.2. Przeszukiwanie strategią w głąb (DFS)

Strategia przeszukiwania w głąb (ang. Depth-first search) polega na odwiedzeniu kolejnych wierzchołków leżących na danej krawędzi aż dotrze do jej końca. Następnym krokiem jest powrót o jeden wierzchołek wyżej i sprawdzanie następnej nieodwiedzonej wychodzącej z tego wierzchołka krawędzi. Podczas procesu odwiedzania najgłębszych węzłów z nieodwiedzonych powstaje graf drzewa przeszukiwania w głąb.



Rysunek 3. Grafika przedstawiająca przykładowy wygląd grafu w głąb, liczby przedstawiają kolejność odwiedzania wierzchołków [3]

Jako, że algorytm ten wymaga wymaga jedynie zapamiętania ścieżki od korzenia do obecnego węzła złożoność pamięciowa jest niewielka w porównaniu ze strategią *BFS*. Złożoność pamięciowa natomiast tak jak w *BFS* wynosi $O(|V| + |E|)$ gdzie V jest liczbą węzłów a E jest liczbą krawędzi. Wynika to z tego, że algorytm musi odwiedzić wszystkie wierzchołki oraz wszystkie krawędzie. Algorytm ten również cechuje się zupełnością dla drzew skończonych. Oznacza to, że strategia ta znajduje rozwiązanie lub informuje, że rozwiązanie nie istnieje.

2.3. Przeszukiwanie strategią najpierw najleszy A*

Strategia $A*$ jest jednym z wariantów strategii *pierwszy najlepszy*. Umożliwia ona znalezienie najkrótszej ścieżki pomiędzy wierzchołkiem początko-

wym a docelowym. Funkcja heurystyczna w strategii $A*$ wynosi:

$$f(n) = g(n) + h(n) \tag{1}$$

gdzie *g(n)* - jest kosztem drogi z wierzchołka początkowego do wierzchołka n a *h(n)* jest kosztem drogi z wierzchołka n do wierzchołka docelowego.

Wybór kolejnych wierzchołków odbywa się na podstawie strategii *"najpierw najlepszy"*. Najważniejszym elementem tej strategii jest wybór funkcji heurystycznej, oceniającej odległość do punktu docelowego.

2.3.1. Metryka Hamminga

Odległość Hamminga jest to liczba pozycji między dwoma ciągami o tej samej długości , w których odpowiadające im zawartości się różnią. To znaczy jest to liczba odzwierciedlająca odmienność dwóch ciągów o tej samej ilości pozycji pod względem zawartości. Dla naszego problemu wskazuje ona liczbę kafli niebędących na swoich miejscach.

2.3.2. Metryka Manhattan

Odległość ta jest sumą wartości bezwzględnych różnic ich współrzędnych. To znaczy jest to suma różnic między odpowiednimi składnikami dwóch obiektów na planszy kartezjańskiej. W naszym przypadku jest to odległość pomiędzy pozycjami kafli na planszy a ich miejscami docelowymi.

3. Opis implementacji

3.1. Opis struktury projektu

Program został napisany w języku *Python* w wersji 3.10. Struktura jego modułów wygląda następująco:

— *main.py* - plik główny programu
— *classes*
  — *a_star_strategy.py* - zawiera klasę *A_star*,
  — *bfs_strategy.py* - zawiera klasę *BFS*,
  — *dfs_strategy.py* - zawiera klasę *DFS*,
  — *puzzle_reader.py* - zawiera klasę *PuzzleReader*, służy ona do wczytywania zagadek z plików *.txt,
  — *puzzle.py* - zawiera klasę *Puzzle*. Zawiera ona metody do przesuwania planszy i zapamiętywania kombinacji wykonanych ruchów,
  — *tools.py* - zawiera funkcję *eq_2d_array*, która sprawdza równość dwóch tablic dwu-wymiarowych.

wszystkie klasy zawarte w plikach **_strategy.py* wykonują obliczenia prowadzące do rozwiązania zagadki i przechowują informacje o statystykach wykonania.

3.2. Opis zliczania stanów odwiedzonych i przetworzonych

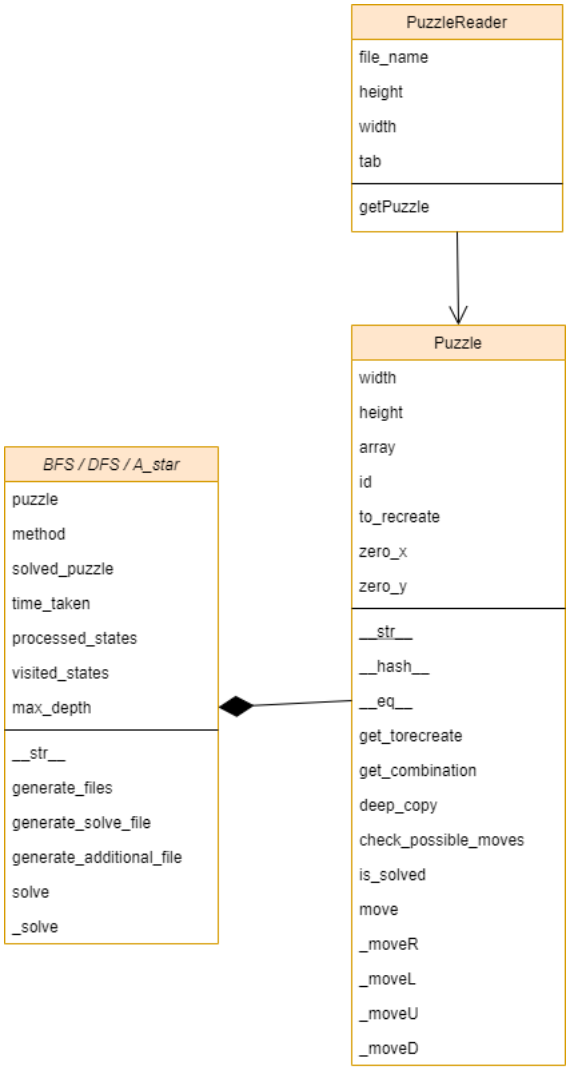
— BFS:

- stan odwiedzony - każdy stan, do którego doszedł program przed zakończeniem działania. Jeżeli stan nie był stanem odwiedzonym, to zostaje dodany do listy stanów odwiedzonych,
- stan przetworzony - stan, gdzie doszło do sprawdzenia poprawności ułożenia zagadki.

— DFS:
- stan odwiedzony - każde rekurencyjne wywołanie metody rozwiązującej zagadkę. Jeżeli stan nie był stanem odwiedzonym, to zostaje dodany do listy stanów odwiedzonych,
- stan przetworzony - stan gdzie doszło do sprawdzenia poprawności ułożenia zagadki,
- dodatkowe informacje - podczas pracy nad algorytmem zauważyliśmy, że przetwarzanie jest dużo szybsze jeżeli nie będziemy sprawdzać czy dany stan już odwiedziliśmy, czy nie. Dlatego zdecydowaliśmy się zamieścić wyniki dla przetwarzania z uwzględnieniem sprawdzania stanów odwiedzonych oraz bez uwzględniania. Nazwa metody ze sprawdzaniem stanów to w naszym sprawozdaniu "*slow.dfs*", natomiast bez sprawdzania to "*dfs*".

— A*:
- stan odwiedzony - każdy stan dodany do kolejki,
- stan przetworzony - stan wybrany z kolejki do przetwarzania.

3.3. Diagram UML



Rysunek 4. Diagram klas

3.4. Dodatkowe informacje

Program może działać z dowolnymi rozmiarami planszy pod warunkiem, że jest to plansza prostokątna.

4. Materiały i metody

Wyniki programu zostały uzyskane poprzez rozwiązanie 413 zagadek wygenerowanych programem *puzzlegen.jar*, udostępnionym na platformie *WIKAMP*. Wykorzystaliśmy wszystkie 3 zaimplementowane algorytmy, gdzie w algorytmie$A*$ wykorzystaliśmy heurystyki *Hamminga* oraz *Manhattan*, natomiast przy *BFS* i *DFS* sprawdziliśmy przetwarzanie w kolejnościach:

— prawo-dół-góra-lewo
— prawo-dół-lewo-góra

— dół-prawo-góra-lewo
— dół-prawo-lewo-góra
— lewo-góra-dół-prawo
— lewo-góra-prawo-dół
— góra-lewo-dół-prawo
— góra-lewo-prawo-dół

Dodatkowo sprawdziliśmy osobno *DFS* ze sprawdzaniem odwiedzonych stanów (*slow.dfs*) oraz bez sprawdzania (*dfs*). Jako domyślny wybraliśmy sposób bez sprawdzania, ponieważ gwarantował dużo szybsze przetwarzanie a jako, że nadany został limit głębokości rekursji nie musieliśmy martwić się o zapętlenie algorytmu i tym samym jego nieskończoną pracę.

Do wygenerowania danych użyliśmy skryptu "*runprog.ps1*", a do weryfikacji poprawności solucji wygenerowanych przez nasz program użyliśmy skryptu "*runval.ps1*" oraz dołączonego do niego programu "*puzzleval.jar*", wszystkie udostępnione na platformie *WIKAMP*.

Program dla każdego wykonania wytwarza plik ze statystykami o następującej budowie:

— pierwsza linia - długość znalezionego rozwiązania,
— druga linia - liczba stanów odwiedzonych,
— trzecia linia - liczba stanów przetworzonych,
— czwarta linia - maksymalna głębokość rekursji,
— piąta linia - czas trwania procesu obliczeniowego w milisekundach z dokładnością do 3 miejsc po przecinku.

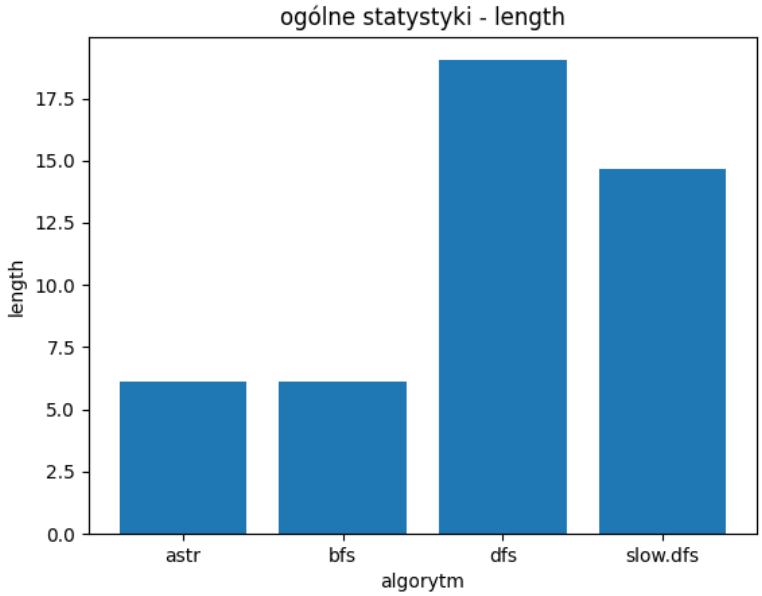
Dodatkowo w nazwie pliku znajdują się informacje o tym jakiego użyto algorytmu, z jakim parametrem, a także wymiary zagadki, oryginalna odległość od układu początkowego oraz numer porządkowy.

5. Wyniki

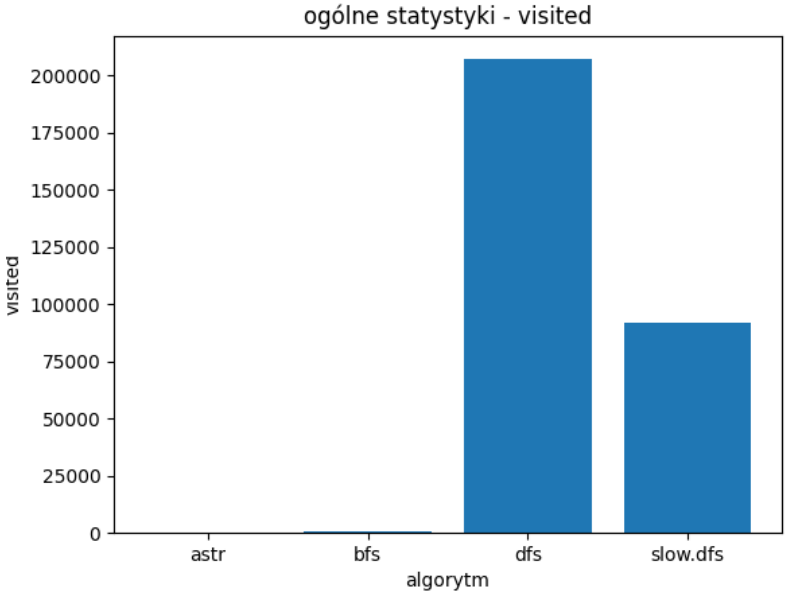
Przedstawione poniżej wykresy przedstawiają średnią wartość prezentowanej charakterystyki dla danego algorytmu lub parametru algorytmu.

5.1. Porównanie algorytmów

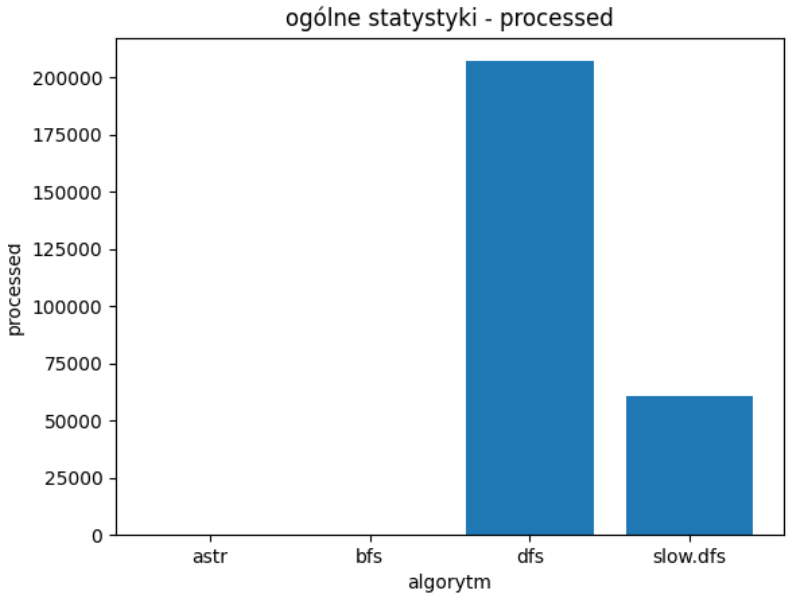
5.1.1. Wszystkie algorytmy



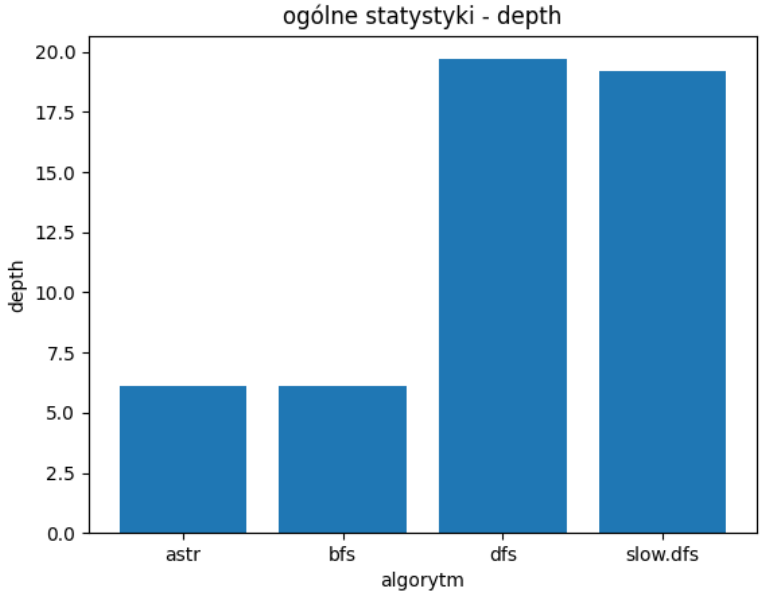
Rysunek 5. Wykres prezentujący średnią długość rozwiązania przy użyciu danego algorytmu



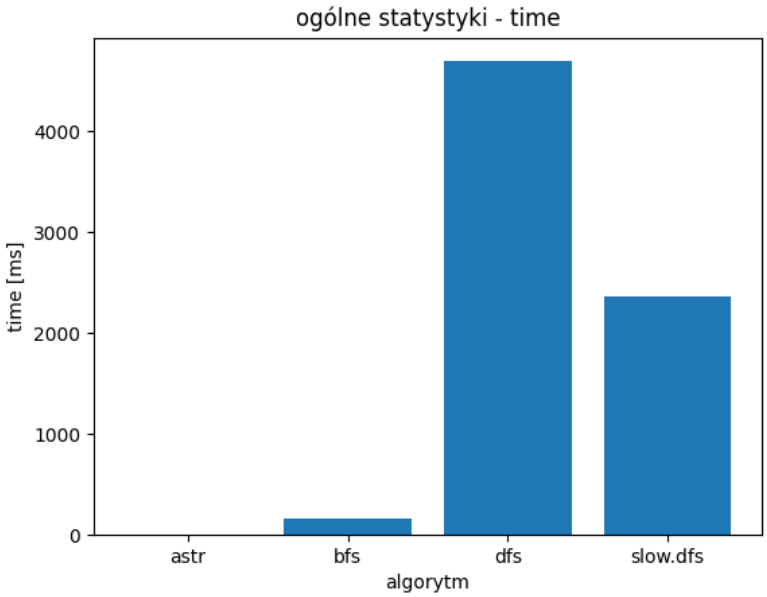
Rysunek 6. Wykres prezentujący średnią ilość stanów odwiedzonych dla danego algorytmu



Rysunek 7. Wykres prezentujący średnią ilość stanów przetworzonych dla danego algorytmu

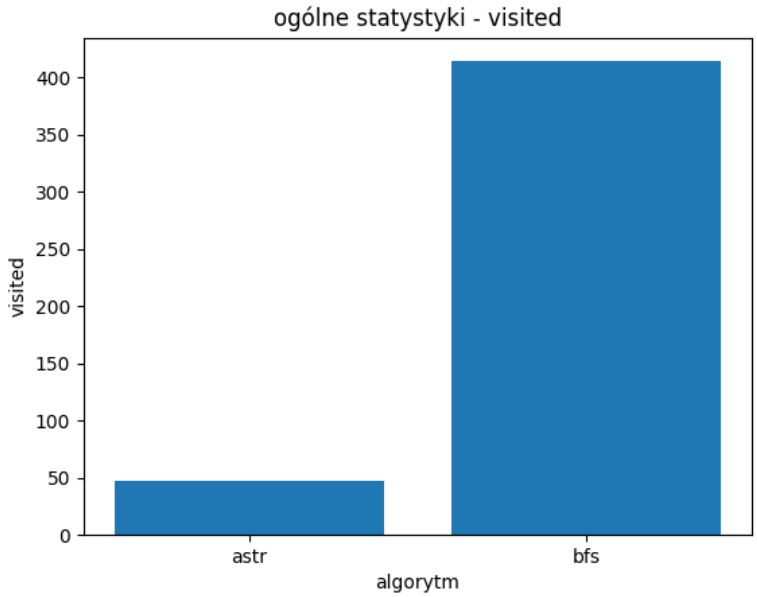


Rysunek 8. Wykres prezentujący średnią maksymalną głębokość rekursji dla danego algorytmu

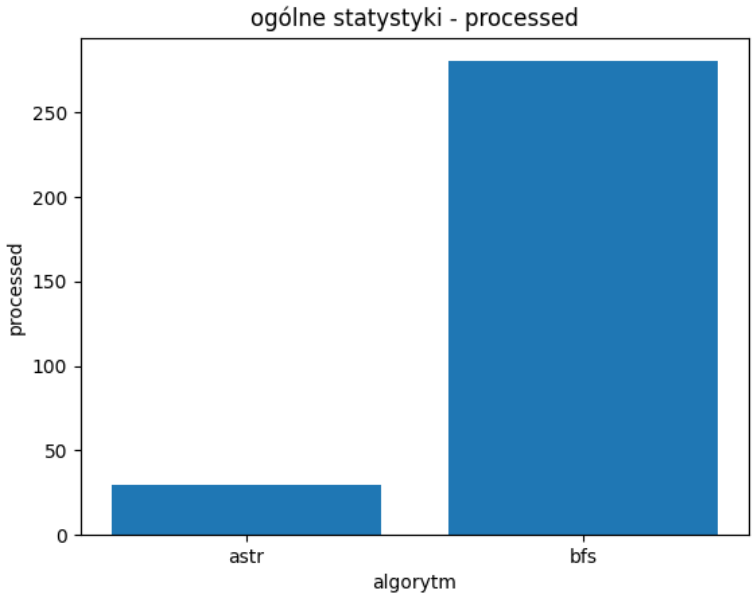


Rysunek 9. Wykres prezentujący średni czas wykonywania obliczeń dla danego algorytmu

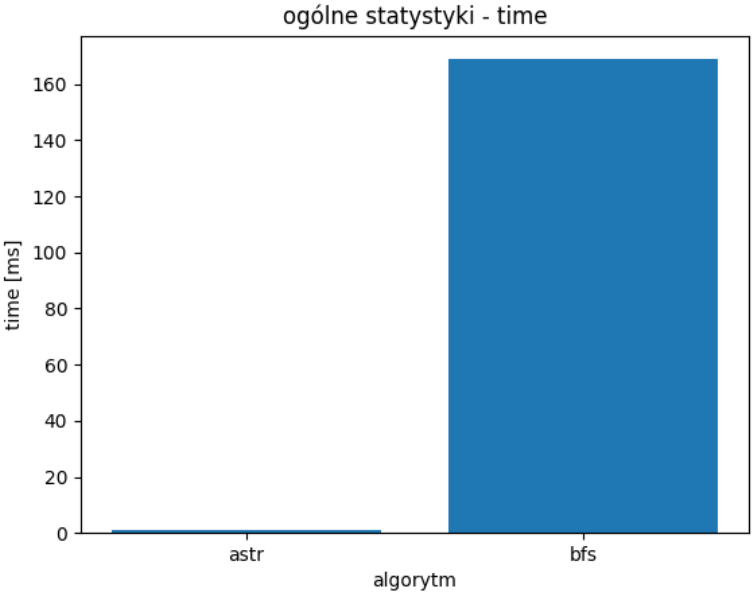
5.1.2. Wybrane wykresy porównujące algorytmy BFS i A*



Rysunek 10. Wykres prezentujący średnią ilość stanów odwiedzonych dla danego algorytmu



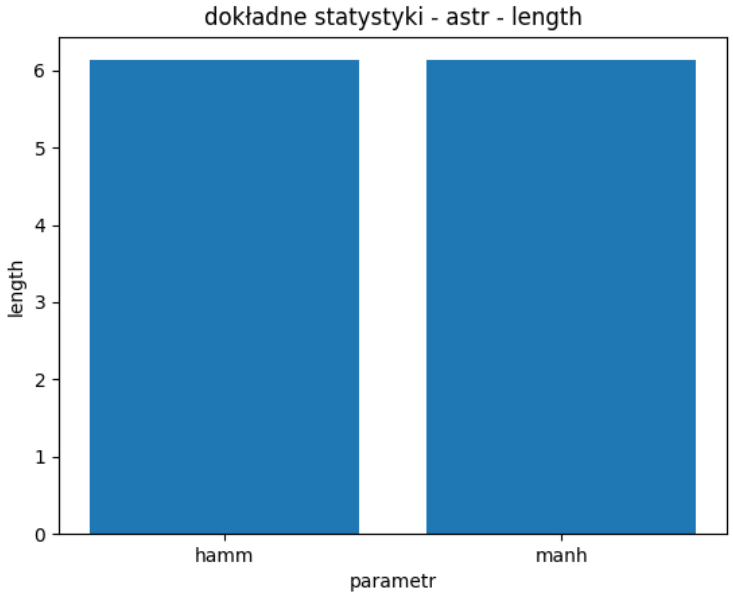
Rysunek 11. Wykres prezentujący średnią ilość stanów przetworzonych dla danego algorytmu



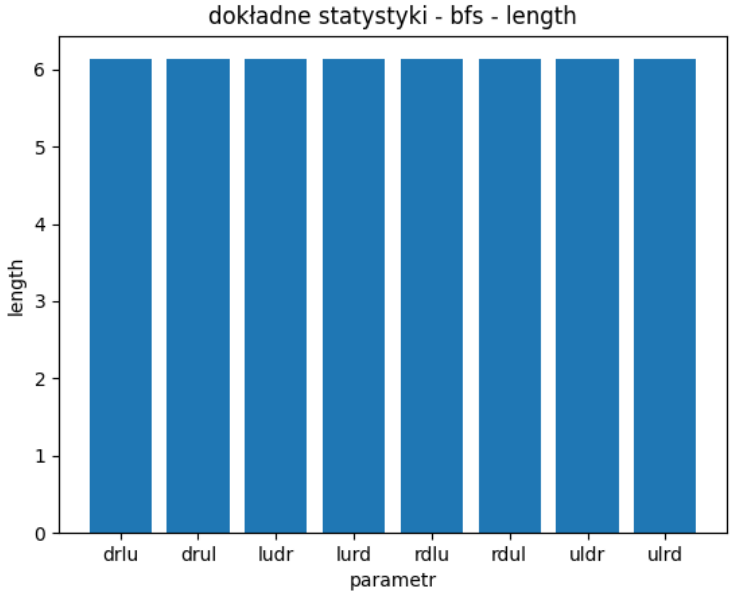
Rysunek 12. Wykres prezentujący średni czas wykonywania obliczeń dla danego algorytmu

5.2. Porównanie parametrów algorytmów

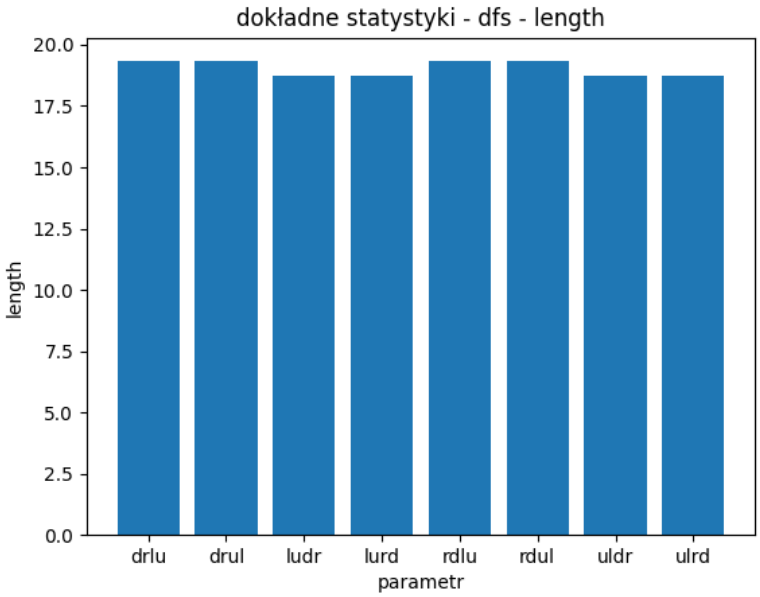
5.2.1. Długość rozwiązania



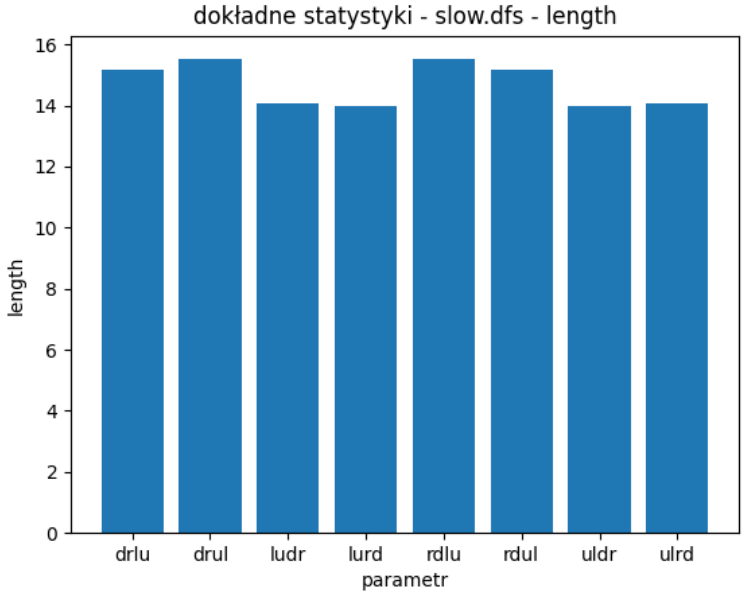
Rysunek 13. Wykres prezentujący średnią długość rozwiązania dla algorytmu A* w zależności od wybranej heurystyki



Rysunek 14. Wykres prezentujący średnią długość rozwiązania dla algorytmu BFS w zależności od kolejności przetwarzania sąsiadów

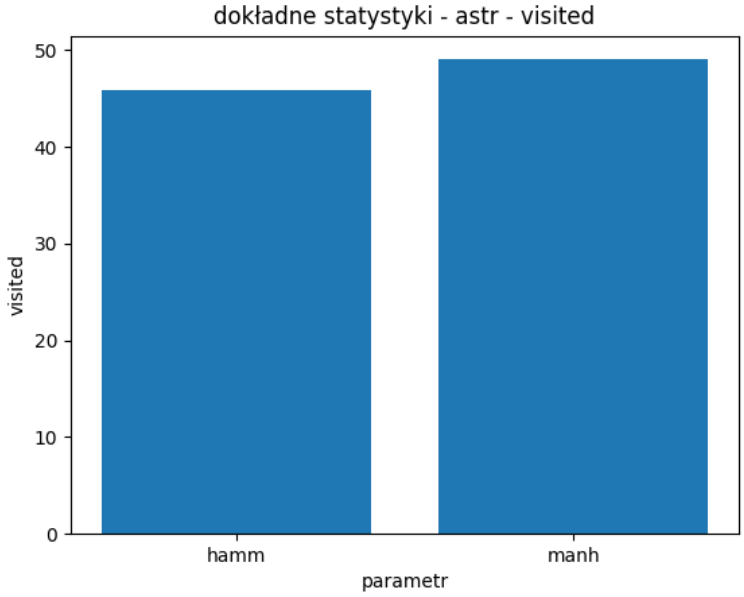


Rysunek 15. Wykres prezentujący średnią długość rozwiązania dla algorytmu DFS w zależności od kolejności przetwarzania sąsiadów

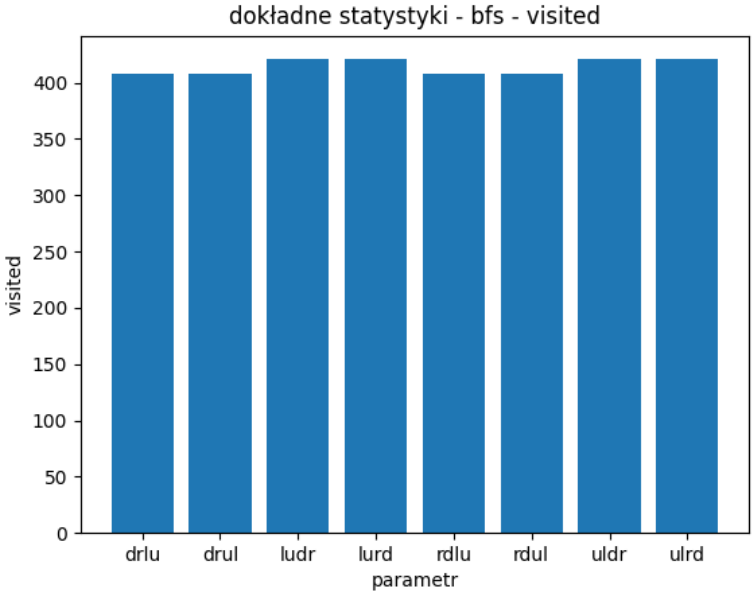


Rysunek 16. Wykres prezentujący średnią długość rozwiązania dla algorytmu slow.DFS w zależności od kolejności przetwarzania sąsiadów

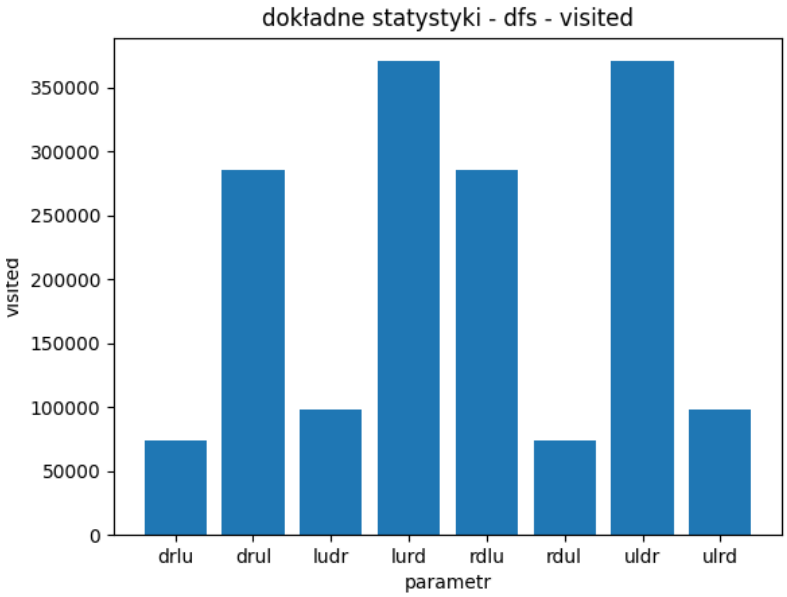
5.2.2. Liczba odwiedzonych stanów



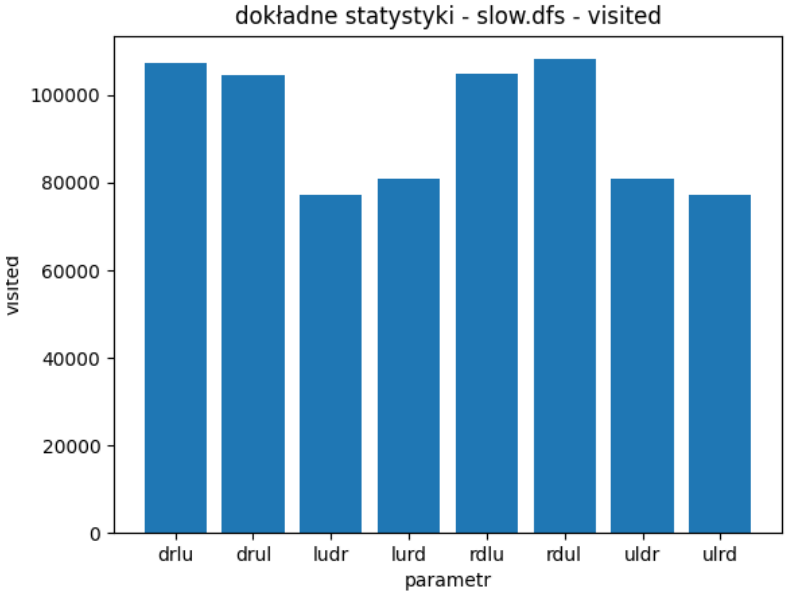
Rysunek 17. Wykres prezentujący średnią liczbę odwiedzonych stanów dla algorytmu A* w zależności od wybranej heurystyki



Rysunek 18. Wykres prezentujący średnią liczbę odwiedzonych stanów dla algorytmu BFS w zależności od kolejności przetwarzania sąsiadów

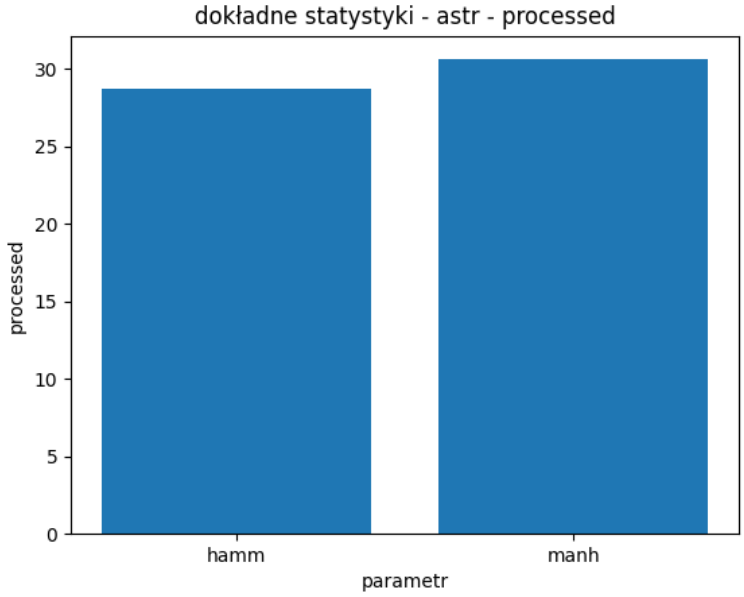


Rysunek 19. Wykres prezentujący średnią liczbę odwiedzonych stanów dla algorytmu DFS w zależności od kolejności przetwarzania sąsiadów

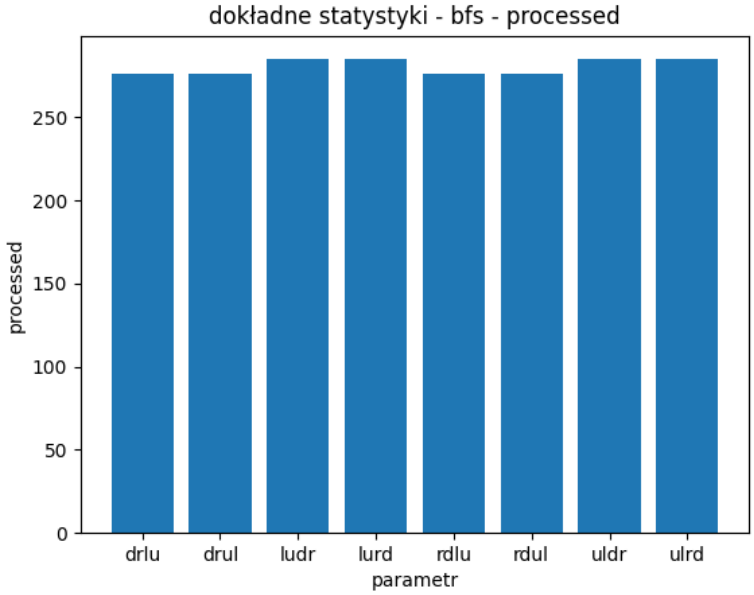


Rysunek 20. Wykres prezentujący średnią liczbę odwiedzonych stanów dla algorytmu slow.DFS w zależności od kolejności przetwarzania sąsiadów

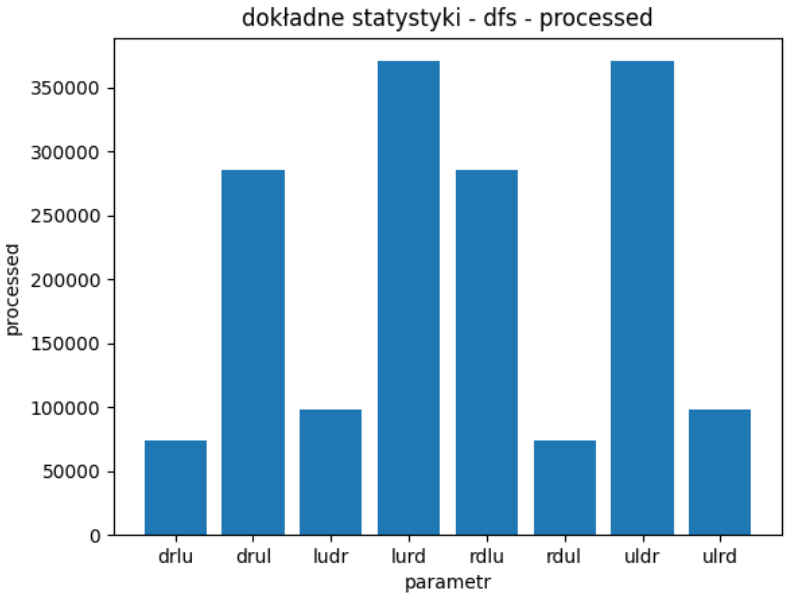
5.2.3. Liczba przetworzonych stanów



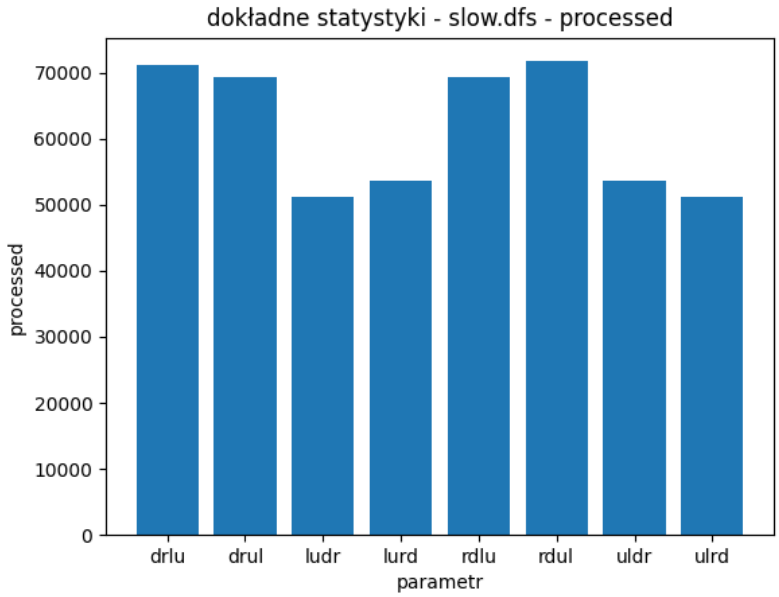
Rysunek 21. Wykres prezentujący średnią liczbę przetworzonych stanów dla algorytmu A* w zależności od wybranej heurystyki



Rysunek 22. Wykres prezentujący średnią liczbę przetworzonych stanów dla algorytmu BFS w zależności od kolejności przetwarzania sąsiadów

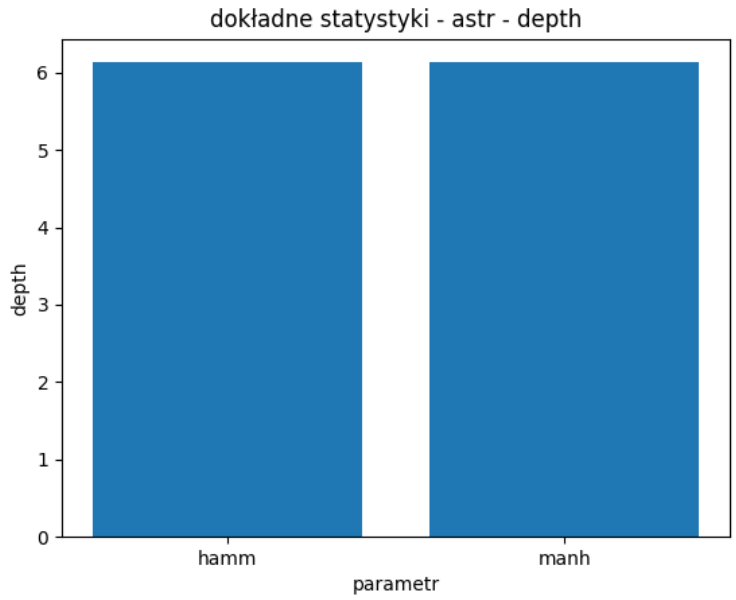


Rysunek 23. Wykres prezentujący średnią liczbę przetworzonych stanów dla algorytmu DFS w zależności od kolejności przetwarzania sąsiadów

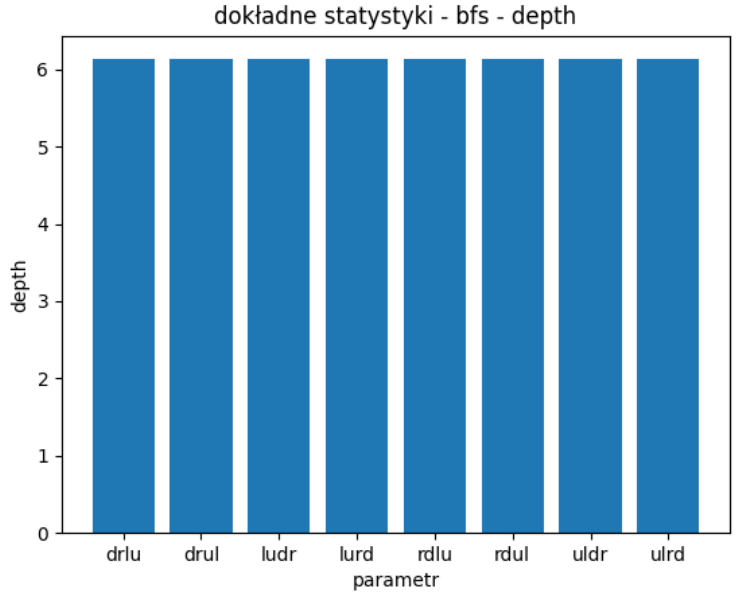


Rysunek 24. Wykres prezentujący średnią liczbę przetworzonych stanów dla algorytmu slow.DFS w zależności od kolejności przetwarzania sąsiadów

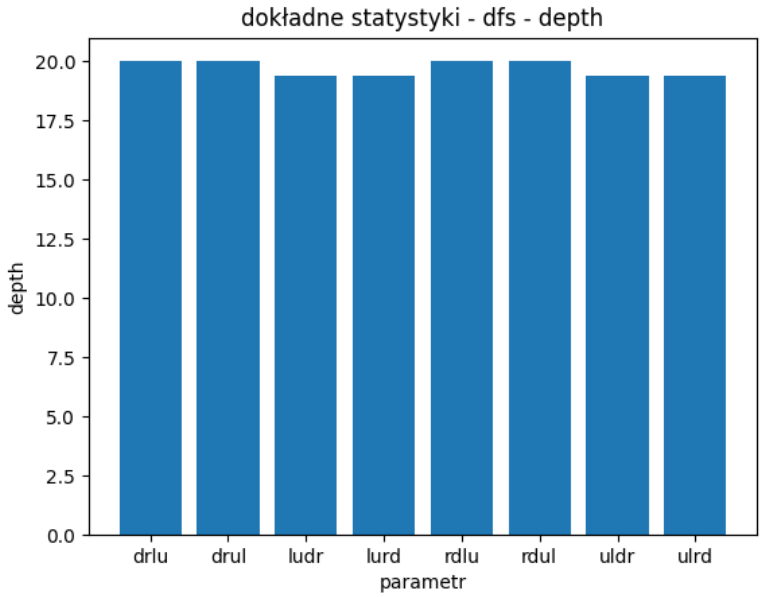
5.2.4. Głębokość rekursji



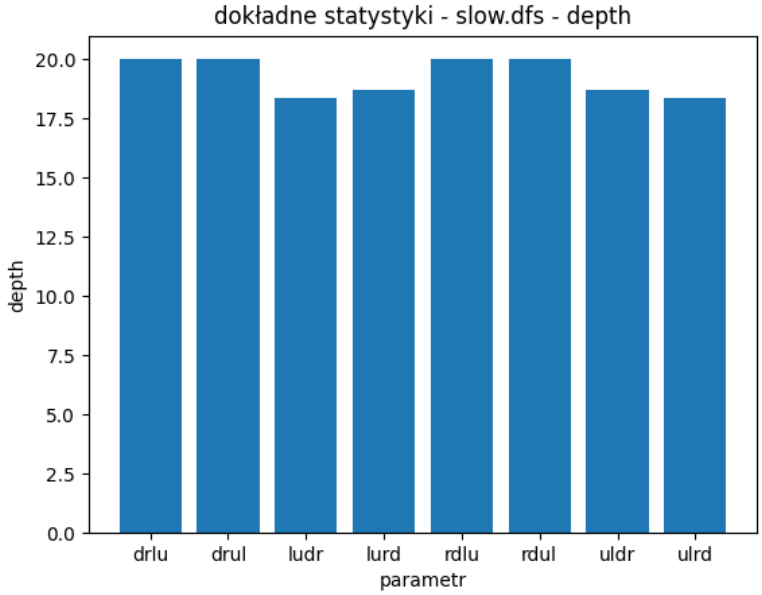
Rysunek 25. Wykres prezentujący średnią maksymalną głębokość rekursji dla algorytmu A* w zależności od wybranej heurystyki



Rysunek 26. Wykres prezentujący średnią maksymalną głębokość rekursji dla algorytmu BFS w zależności od kolejności przetwarzania sąsiadów

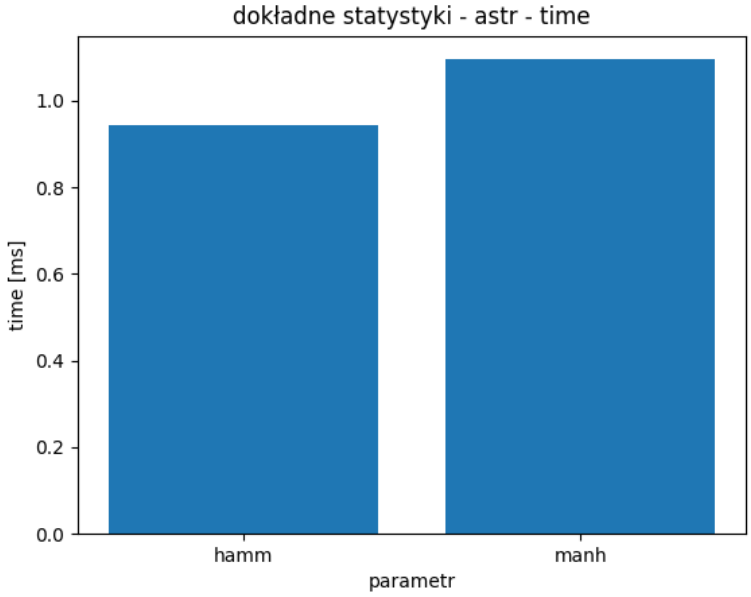


Rysunek 27. Wykres prezentujący średnią maksymalną głębokość rekursji dla algorytmu DFS w zależności od kolejności przetwarzania sąsiadów

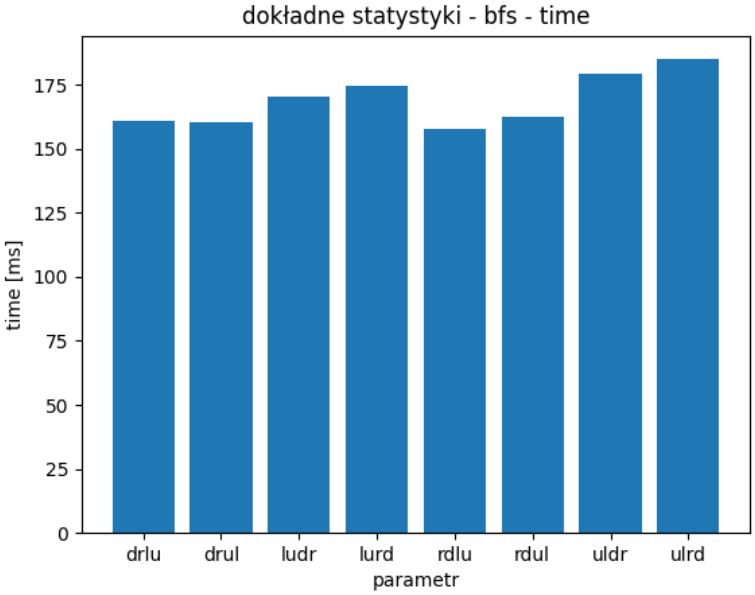


Rysunek 28. Wykres prezentujący średnią maksymalną głębokość rekursji dla algorytmu slow.DFS w zależności od kolejności przetwarzania sąsiadów

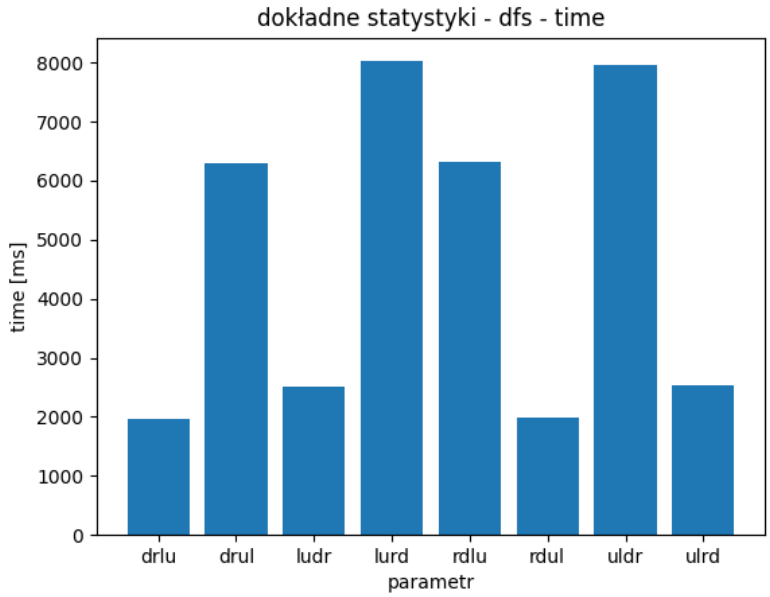
5.2.5. Czas przetwarzania



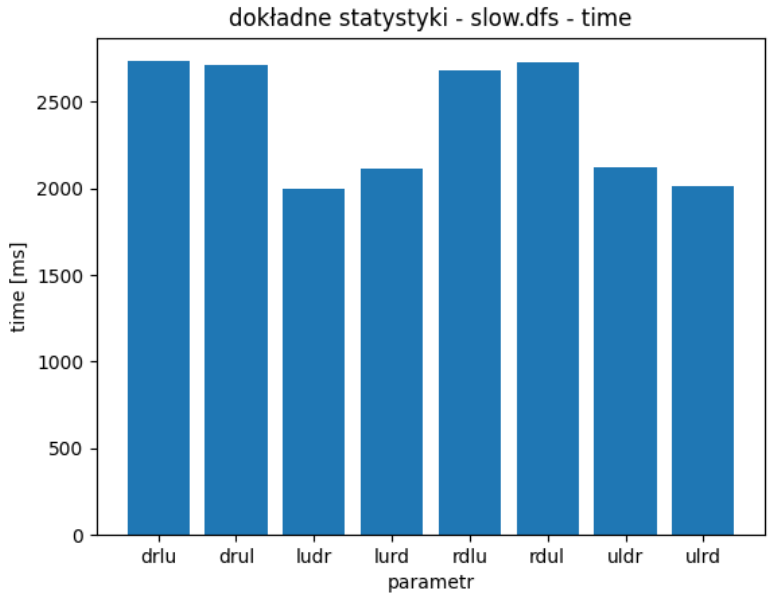
Rysunek 29. Wykres prezentujący średni czas przetwarzania dla algorytmu A* w zależności od wybranej heurystyki



Rysunek 30. Wykres prezentujący średni czas przetwarzania dla algorytmu BFS w zależności od kolejności przetwarzania sąsiadów



Rysunek 31. Wykres prezentujący średni czas przetwarzania dla algorytmu DFS w zależności od kolejności przetwarzania sąsiadów



Rysunek 32. Wykres prezentujący średni czas przetwarzania dla algorytmu slow.DFS w zależności od kolejności przetwarzania sąsiadów

6. Dyskusja

6.1. Długość rozwiązania

Na rysunku 5 zauważyć możemy, że średnia długość rozwiązania dla algorytmów *A** oraz *BFS* jest praktycznie identyczna. Natomiast algorytm *DFS* nie poradził sobie dobrze zarówno w wariancie ze sprawdzaniem odwiedzonych stanów (*slow.dfs*) jak i w wariancie bez ich sprawdzania (*dfs*), chociaż ten pierwszy poradził sobie minimalnie lepiej.

6.2. Liczba stanów odwiedzonych

Na rysunku 10 zauważyć możemy, że średnia liczba stanów odwiedzonych dla algorytmu *A** wynosi nieco ponad 50 a dla *BFS* jest to około 400, więc 8 razy więcej. Według rysunku 6 algorytm *DFS* odwiedził średnio 100 000 stanów przy omijaniu wcześniej odwiedzonych stanów, a bez ich omijania aż 200 000, to odpowiednio około 600 i 1200 razy więcej niż przy strategii *BFS*.

6.3. Liczba stanów przetworzonych

Na rysunku 11 widać, że algorytm *A** przetwarzał średnio około 30 stanów, natomiast *BFS* nieco ponad 250, co daje wynik oscylujący w granicach 8,5 razy większej liczby stanów przetworzonych. Ciekawą różnicę możemy także zaobserwować na rysunku 7 gdzie widzimy, że dodanie omijania odwiedzonych stanów zmniejszyło średnią liczbę odwiedzonych stanów z około 200 000 do 70 000, a więc prawie 3 krotnie.

6.4. Maksymalna głębokość rekursji

Na rysunku 8 widzimy, że zarówno algorytm *A** jak i *BFS* poradziły sobie bardzo dobrze w kwestii głębokości rekursji. W przypadku *BFS* jest to cecha tego algorytmu, jednak przy *A** znaczy to, że algorytm ten bardzo dobrze radzi sobie ze znajdowaniem najbardziej optymalnych ścieżek. W przypadku *dfs* i *slow.dfs* głębokość ta jest bliska, prawie równa, maksymalnej głębokości na jaką zezwala nasz program, co jest cechą algorytmu *DFS*. Należy jednak zwrócić uwagę na fakt, że *DFS* uwzględniający sprawdzanie odwiedzonych stanów poradził sobie nieznacznie lepiej od wariantu nie sprawdzającego tego warunku.

6.5. Czas wykonania

Na rysunku 12 możemy zauważyć, że średni czas wykonywania obliczeń algorytmem *A** jest bardzo niski i nie przekraczający 2ms. Średni czas wykonywania dla strategii *BFS* przekracza aż 160ms. Mimo, że pod względem ludzkich możliwości wydaje się to być dobrym wynikiem, to jednak jest on znacznie, bo co najmniej 80 razy, wyższy od przetwarzania *A**. Ciekawym z naszego punktu widzenia jest wykres przestawiony na rysunku 9.Można na nim zauważyć, że średni czas przetwarzania algorytmem *DFS* bez sprawdzania stanów odwiedzonych to ponad 4 sekundy. Ciekawym jest, że nasze

wstępne testy podczas tworzenia kodu wykazujące szybsze działanie algorytmu *DFS* bez sprawdzania czy stan został już odwiedzony, nie są odzwierciedlone przy testach w szerszej skali. Nazwany przez nas *slow.dfs* wcale nie jest wolniejszy od algorytmu *DFS* bez sprawdzania stanów. Średnio jest około 2 razy szybszy. Faktem jest, że to nadal są nieco ponad 2 sekundy przetwarzania, które w stosunku do możliwości algorytmu $A*$ są mizernym wynikiem, jednakże uważamy to za warte wspomnienia.

6.6. Porównanie parametrów - A*

Niestety z rysunków 13, 17, 21, 25 oraz 29 nie możemy jednoznacznie wywnioskować, ze względu na zbyt małe różnice w ich wynikach, która z heurystyk jest lepszą. W celu uzyskania lepszych wyników niezbędne były by testy na bardziej skomplikowanych ułożeniach układanki. Możemy się jedynie pokusić o stwierdzenie, że przy wykonaniu tych testów to prawdopodobnie heurystyka Hamminga okazałaby się bardziej odpowiednia do rozwiązywania tej zagadki. Wynika to z obserwacji, że na wykresach, na których pojawiają się różnice, są one z korzyścią dla tej właśnie heurystyki.

6.7. Porównanie parametrów - BFS

Na podstawie rysunków 14 oraz 26 zauważyć możemy, że algorytm *BFS* zachowuje się zgodnie z oczekiwaniami. Oznacza to, że zawsze znajduje rozwiązanie najlepsze (najkrótsze). Również zgodnie z oczekiwaniami zachowują się wykresy przedstawione na rysunkach 18, 22 oraz 30. Widzimy na nich, że kolejność przetwarzania ma niewielkie znaczenie dla prędkości znajdowania rozwiązania.

6.8. Porównanie parametrów - DFS

Na rysunkach 15 oraz 27 możemy zaobserwować, że kolejność przetwarzania nie ma w praktycznie żadnego znaczenia dla długości znalezionego rozwiązania oraz tego, jak "głęboko" algorytm zejdzie. Dzięki rysunkom 19, 23 oraz 31 zauważyć możemy, że kolejność przetwarzania ma duże znaczenie na liczbę stanów odwiedzonych, tym samym na liczbę stanów przetworzonych a co za tym idzie na czas wykonywania.

6.9. Porownanie parametrów - slow.DFS

Na rysunku 16 widzimy, że kolejność przetwarzania ma niewielkie znaczenie na to jaką długość będzie miało rozwiązanie zagadki. Podobnie jest z maksymalną głębokością rekursji, co widać na rysunku 28. Na rysunkach 20, 24 oraz 32 można zaobserwować, że kolejność przetwarzania ma wpływ na te same statystyki jednak w stopniu mniejszym niż przy algorytmie *DFS*. Jednocześnie maksymalne wartości tych statystyk dla *slow.DFS* są zdecydowanie mniejsze niż te dla *DFS*.

że podobnie jak ma to miejsce przy algorytmie *DFS* nie uwzględniającym sprawdzania stanów przetworzonych, kolejność przetwarzania ma wpływ na te same statystyki, jednakże jest on dużo mniejszy. Jednocześnie maksymalne

wartości tych statystyk dla *slow.DFS* są zdecydowanie mniejsze niż te dla *DFS*.

7. Wnioski

— Algorytm *DFS*, w obu wariantach, jest najmniej odpowiednim wyborem do rozwiązywania zagadki "piętnastki". Otrzymane wyniki pokazują, że jest najwolniejszy, zwraca najdłuższe rozwiązania a także jest najmniej przewidywalny pod względem tego, jak długi będzie czas przetwarzania w zależności od podanej jej kolejności.
— Algorytm *BFS* jest bardzo dobrym wyborem, pod względem prostoty implementacji. Pomimo faktu, że jest znacznie wolniejszy od algorytmu *A**, to w momencie, gdy znaczenie ma przejrzystość kodu i czytelność intencji programisty, to *BFS* jest najlepszym wyborem.
— Algorytm *A**, jest idealnym wyborem gdy skupiamy się na szybkości przetwarzania, oszczędności pamięci (względem pozostałych testowanych algorytmów) oraz jego modularności (możliwość zmiany funkcji aproksymującej odległość do celu).
— Metryka Hamminga dla algorytmu *A** cechuje się prawdopodobnie większą optymalnością do szukania rozwiązań układanki "piętnastki". Jest ona także znacznie prostsza w implementacji w stosunku do metryki Manhattan.
— Mimo wstępnych testów przy pracy nad programem, ostatecznie okazało się, że stosowanie sprawdzanie listy stanów odwiedzonych przynosi wymierne korzyści takie jak skrócony czas przetwarzania, czy znacząca redukcja liczby stanów odwiedzonych i przetworzonych.

Literatura

[1] dr ing. Witold Beluch. *Metody Heurystyczne, Przeszukiwanie grafów cz.1 - strategia ślepa*, 2012, dostępny online.
[2] dr ing. Witold Beluch. *Metody Heurystyczne, Przeszukiwanie grafów cz.2 - strategie heurystyczne*, 2011, dostępny online.
[3] Wikipedia, Wolna encyklopedia. *Przeszukiwanie w głąb, graf drzewa*, 2021, dostępny online.
[4] Wikipedia, Wolna encyklopedia. *Przeszukiwanie wszerz, graf drzewa*, 2021, dostępny online.
[5] Wikipedia, Wolna encyklopedia. *Piętnastka (układanka), plansza*, 2021, dostępny online.