Informatyka Stosowana, studia dzienne, inż I st. semestr IV

Sztuczna inteligencja i systemy ekspertowe 2023/2024

Prowadzący: Piotr Lipiński Poniedziałek, 12:00

Data oddania: _____ Ocena: _____

Piotr Polakowski 247768

Jakub Samek 247781

Filip Kobierski 242336

Zadanie Numer: Temat

1. Cel

Zadanie składa się z dwóch części: programistycznej i badawczej. Część programistyczna stanowi napisanie programu, który będzie rozwiązywał powyższą łamigłówkę przy użyciu różnych metod przeszukiwania przestrzeni stanów stategii:

— *wszerz* (breadth-first search);
— *w głąb* (depth-first search);
— *najpierw najlepszy*: A*, z następującymi heurystykami:
  — metryką Hamminga;
  — metryką Manhattan.

2. Wprowadzenie

2.1. Algorytm BFS

Algorytm BFS działa w taki sposób, że przeszukuje wszystkie możliwe stany idąc wszerz od węzła głównego – przeszukuje zatem wszystkie dostępne węzły na danej głębokości. Przekładając to na praktyczną implementację w „piętnastce" węzłem początkowym jest stan układanki, którą należy rozwiązać, a końcowym – układanka rozwiązana – ze wszystkimi elementami na odpowiedniej kolejności. Podczas każdej iteracji algorytmu sprawdza on dostępne możliwe ruchy, i jeżeli dane przesunięcie elementu zerowego da w rezultacie układ, który nie był do tej pory dla algorytmu znany, to staje się on kolejnym

węzłem i będzie bazą do generowania możliwych przesunięć w kolejnych iteracjach. Po wygenerowaniu możliwych stanów dla danego węzła algrytm przystępuje do generowania możliwych stanów dla kolejnego węzła na tej samej głębokości, o ile jest on dostępny. Stosowana w tym przypadku jest struktura FIFO(kolejka). Algorytm kończy się w momencie, w którym napotka na stan, który odpowiada rozwiązanej układance.

2.2. Algorytm DFS

Algorytm DFS natomiast wyszukuje „w głąb", czyli zamiast eksplorować wszystkie możliwe stany układanek na jednej głębokości zajmuje się jedną konkretną gałęzią – gdy znajdzie możliwy ruch w danej układance to wykonuje go, i dla układanki świeżo co zmienionej eksploruje kolejne możliwe przesunięcia. Algorytm będzie szedł „w głąb" tak długo, aż nie wyczerpie liczby możliwych stanów(co w przypadku układanki 4x4 zajęłoby bardzo dużo czasu) bądź gdy osiągnie zadaną liczbę stanów określoną manualnie w algorytmie przez programistę. Niezależnie od napotkanej przyczyny niemożności wykonania kolejnych przesunięć gdy taka sytuacja nastąpi, to algorytm cofa się do poprzedniego węzła i eksploruje kolejny możliwy stan tak samo jak w poprzednim przypadku. Algorytm DFS jest bardzo łatwy w implementacji za pomocą rekursji, natomiast można go też z powodzeniem zaimplementować iteracyjnie używając do tego celu stosu(LIFO). Algorytm kończy się, gdy znaleziona układanka będzie w takim stanie, jak układanka rozwiązana.

2.3. Algorytm A*

Algorytm A* jest w swoim działaniu podobny do algorytmu BFS, jednakże ma jedną sporą różnicę – nie wyznacza on „na ślepo" kolejnych możliwych sąsiadów – w naszym przypadku rozwiązań układanki, tylko posługuje się określoną metryką pozwalającą wyznaczyć najbardziej optymalny stan. W przypadku zaimplementowanego algorytmu zostały zastosowane dwie metryki: • Hamminga – bardzo prosta, jest to liczba pól układanki, które nie są na swoich miejscach • Manhattan – jest to suma odległości manhattan każdego z pól układanki od swojego miejsca docelowego Generalnie wzór opisujący obliczanie metryki dla kolejnego stanu wygląda następująco:

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

gdzie:
— $f(n)$ — aktualny priotytet danego węzła (układanki)
— $h(n)$ — wartość przyjętej metryki (w tym przypadku Hamminga lub Manhattan)
— $g(n)$ — głębokość aktualnie przetwarzanego węzła

W odróżnieniu od poprzednio opisanych algorytmów A* korzysta z tzw kolejki z priorytetem – jak sama nazwa wskazuje pierwszeństwo w przetwarzaniu ma zawsze węzeł, który posiada najmniejszy w tym wypadku priorytet(mniej = lepiej). Przetwarzanie jest realizowane do momentu, gdy zostanie odnaleziony właściwy układ planszy lub wyczerpią się możliwości ruchów.

3. Opis implementacji

Program rozwiązujący łamigłówkę składa się z następujących plików:

— main.py — plik główny służący do wywoływania programu, przyjmujący parametry określone w zadaniu
— Board.py — klasa reprezentująca obiekt układanki, pozwalająca na manupulację polami, operacje na planszy
— Solver.py — klasa odpowiedzialna za rozwiązywanie układanki, zawiera metody rozwiązujące z odpowiednimi algorytmami (dfs, bfs, A*)
— runprog.sh — skrypt uruchamiający przeszukiwanie układanek w trybie wsadowym
— runval.sh — skrypt walidujący rozwiązania układanek zamieszczonych w plikach tekstowych
— exdata.sh — skrypt pozwalający złączyć dane z wielu plików w jeden w celu wygodnego ich przetwarzania

Skrypty w języku bash zostały pobrane ze strony przedmiotu.

Program wywołuje się poprzez plik main przekazując do niego odpowiednie parametry zgodne z tymi umieszczonymi na stronie przedmiotu, są to:

— rodzaj algorytmu (bfs/dfs/astr)
— kolejność przeszukiwania w przypadku bfs/dfs lub metryka w przypadku A*
— nazwa pliku z układanką do rozwiązania
— nazwa pliku, do którego ma zostać zapisane rozwiązanie układanki
— nazwa pliku, do którego mają zostać zapisane statystyki dotyczące rozwiązania

Początkowo planowaliśmy napisać program w Javie, gdyż jest ona podobnie międzyplatformowa i kompilowana do bajtkodu. Nawet pomimo jej większej efektywności i mniej surowych reguł składni do wykonania tego zadania wykorzystaliśmy język Python. Ma on dynamiczne typy i jest interpretowany, czyli powolny, jednak jego kod ma mniej *boilerplate* — można wyrazić swoje myśli w kodzie mniejszą ilością znaków. rogram napisany został w języku programowania python.

4. Materiały i metody

Układanki do zbadania zostały wygenerowane przez program dostępny na stronie przedmiotu. Każda z układanek została zbadana każdym z algorytmów przy każdej możliwej kombinacji parametrów wywołania.

Do przeprowadzenia badania wszystkich wygenerowanych układów łamigłówek został wykorzystany skrypt dostępny na stronie przedmiotu. Umożliwiał on przetestowanie wszyskich możliwych kombinacji – wywoływał on w naszym przypadku program z odpowiednimi, ustandaryzowanymi wcześniej parametrami wywołania tak, aby zbadać wszyskie możliwe stany. Po uruchomieniu skryptu należało odczekać znaczną ilość czasu, aby ten wygenerował wszystkie niezbędne rozwiązania, które były dwoma rodzajami plików tekstowych zawierających:

— plik 1: długość znalezionego rozwiązania, ciąg liter odpowiadający kolejności przesunięć, które prowadziły do rozwiązania układanki

— plik 2: długość znalezionego rozwiązania, liczba stanów odwiedzonych, liczba stanów przetworzonych, maksymalna osiągnięta głębokość, czas wykonania(w ms)

Wynikiem wykonania programu było otrzymanie bardzo dużej liczby plików z danymi. W celu wygodnego ich przetwarzania został wykorzystany kolejny skrypt ze strony przedmiotu – exdata.sh, wyjście skryptu za pomocą potoku zostało przekierowane do pliku tekstowego, dzięki czemu można było na podstawie pojedynczego pliku utworzyć odpowiednie wykresy i analizować otrzymane w wyniku przetwarzania rozwiązań układanek dane.

5. Wyniki

Nasze wyniki można zobaczyć na rysunkach 1 do 16, które znajdują się na dedykowanych do tego stronach.

6. Dyskusja

Całkowite zestawienie rozwiązań układanek pokazuje, że wszystkie zaimplementowane przez nas algorytmy były w 100% skuteczne – udało się rozwiązać wszystkie układanki przygotowane przez program. Algorytm A-star wykazywał najmnijeszą liczbą przetworzonych stanów przy najkrótszym czasie znajdowania właściwej układanki. Metody heurystyczne okazały się tak samo skuteczne jak algorytmy przeszukiwania wszerz i w głąb w przypadku układanek o takich głębokościach, z jakimi mieliśmy do czynienia.

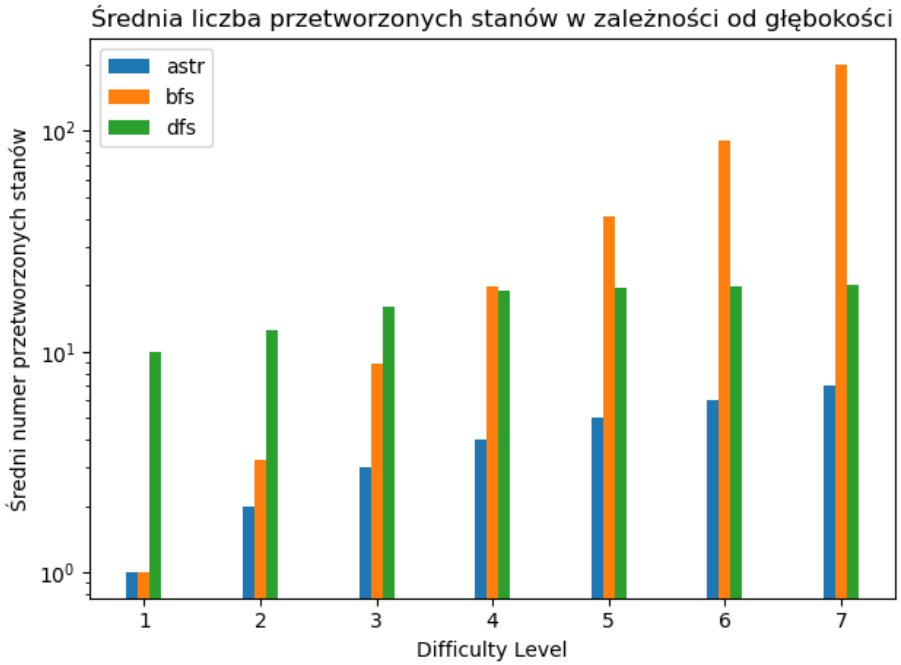
Niewątpliwą wadą algorytmów heurystycznych jest brak możliwości powrotu do poprzednich stanów i esksplorowania alternatywnych ścieżek rozwiązań, czego nie można powiedzieć o algorytmach bfs i dfs. BFS wykazał się w naszym przypadku większą szybkością i mniejszą liczbą przetwarzanych stanów niż dfs - nie mieliśmy bowiem wiedzy o rozlokowaniu grafu rozwiązań, więc algorytm bfs jest niewątpliwie bardziej optymalnym rozwiązaniem. Po odpowiedniej optymalizacji DFS jest w stanie rozwiązać wszystkie układanki, jednakże z zauważalnie większym czasem przetwarzania oraz znacząco większą liczbą przetworzonych stanów niż pozostałe algorytmy.

6.1. Globalnie

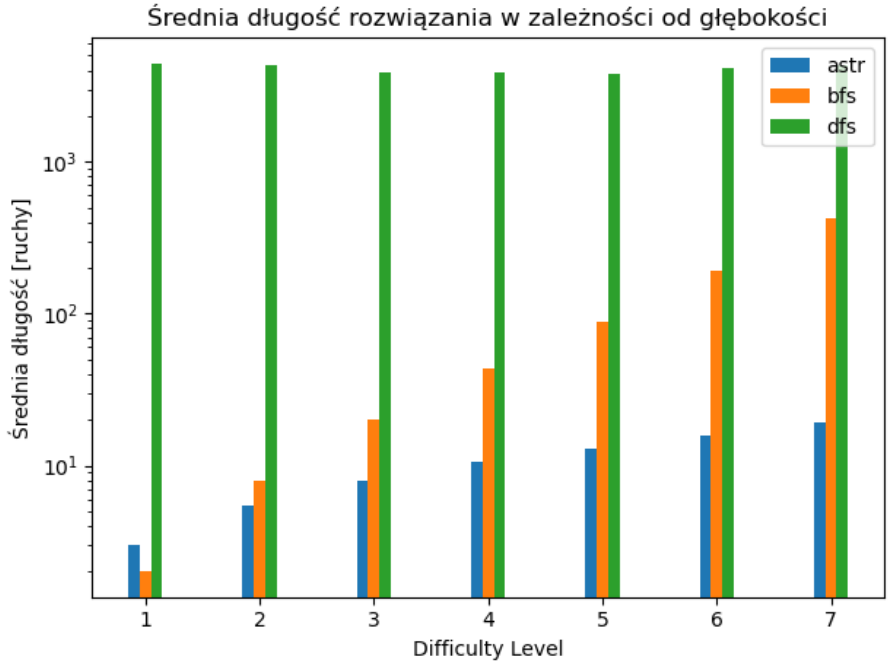
W przytłaczającej większości przypadków algrytm A* był najlepszy pod każdym względem. Przy bardzo prostych układankach czasami BFS szybciej znajdywał rozwiązanie, jednak różnicza jest niewielka

6.2. Heurystyki A*

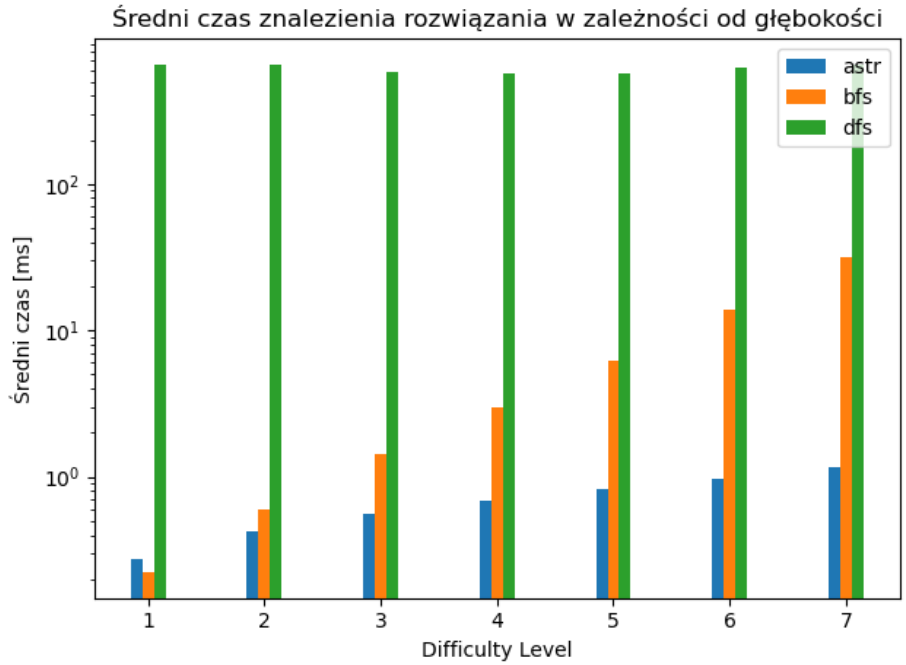
Nie można jednonacznie stwiedzić czy heurystyka Hamminga czy Manhattan są lepsze. Obydwa do rzwiązania potrzebują przetworzyć taką samą liczbę stanów, jednak Hamminga odwiedza ich więcej przy trudniejszych układankach i jest szybszy.



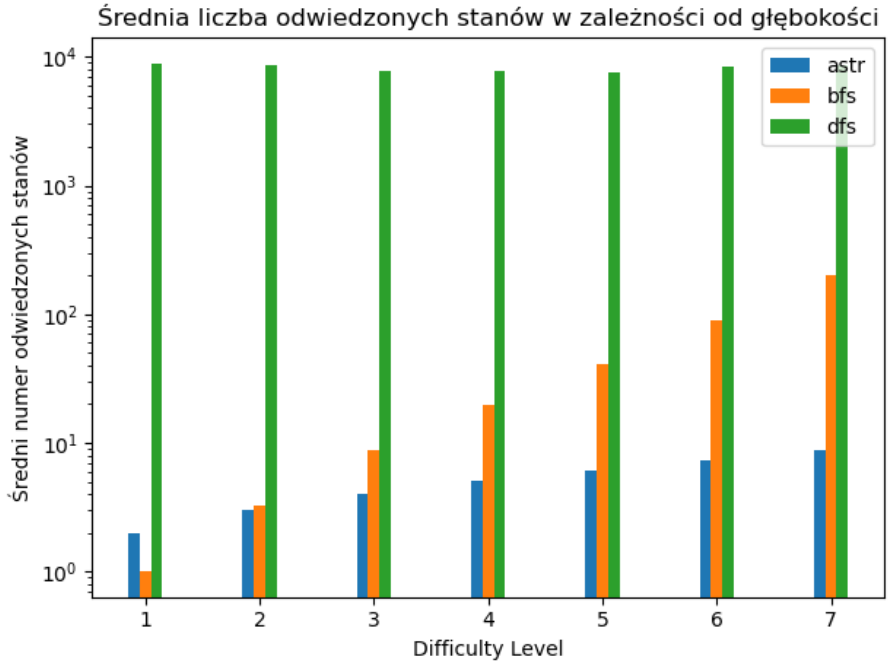
Rysunek 1. Globalnie: trudność a przetworzone stany



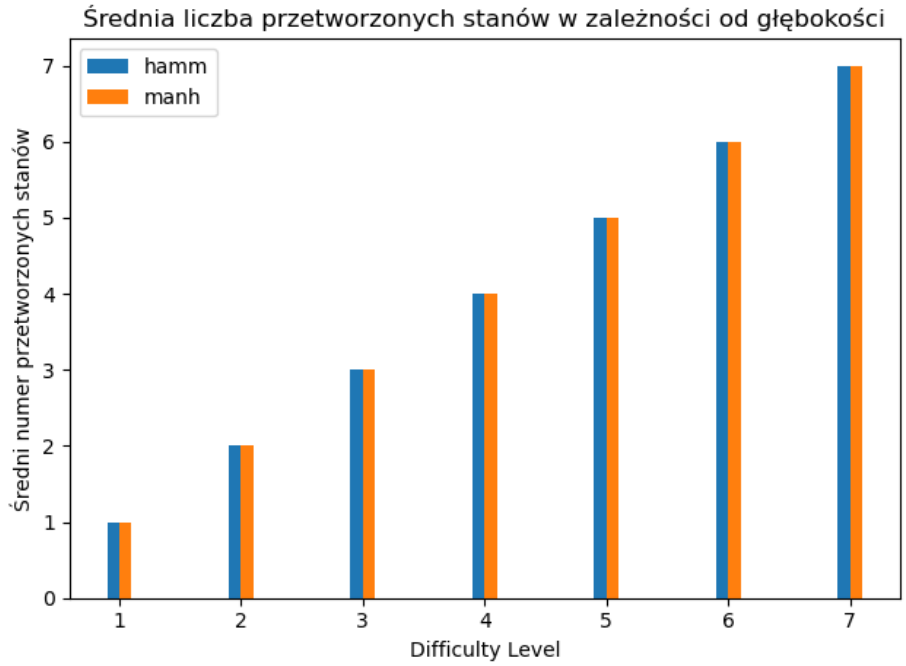
Rysunek 2. Globalnie: trudność a długość rozwiązania



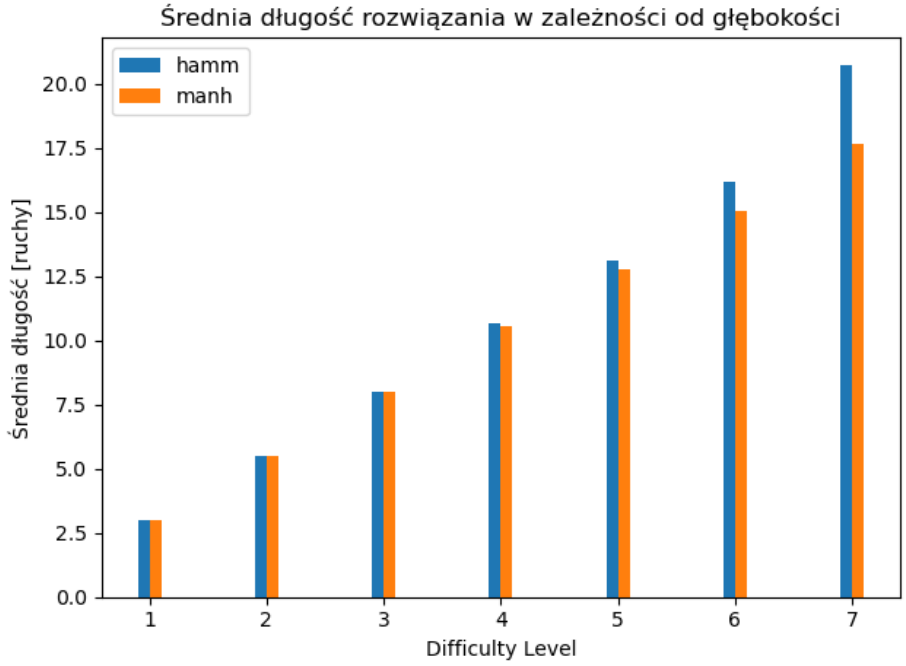
Rysunek 3. Globalnie: trudność a czas rozwiązania



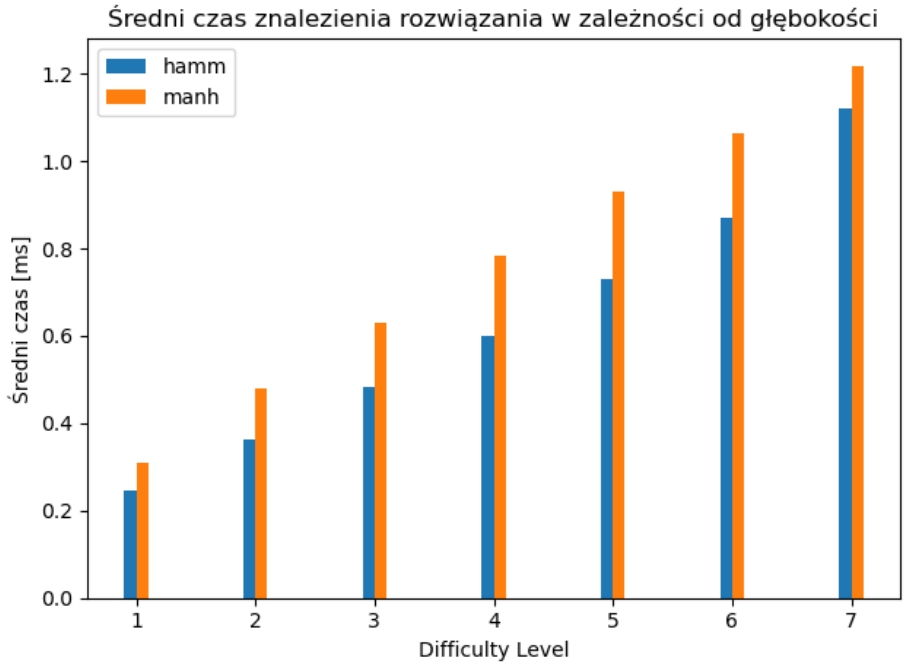
Rysunek 4. Globalnie: trudność a odwiedzone stany



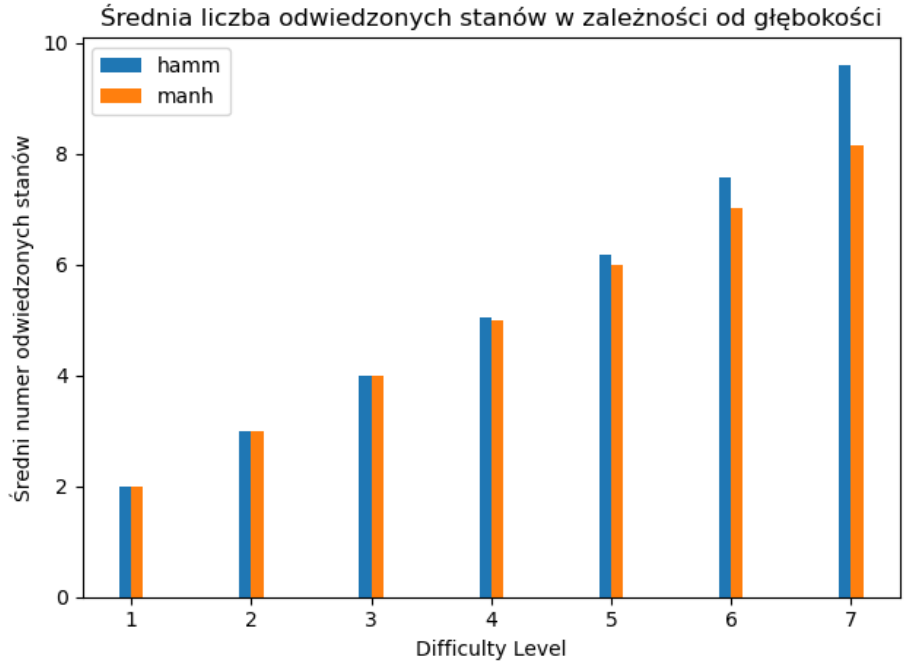
Rysunek 5. A*: trudność a przetworzone stany



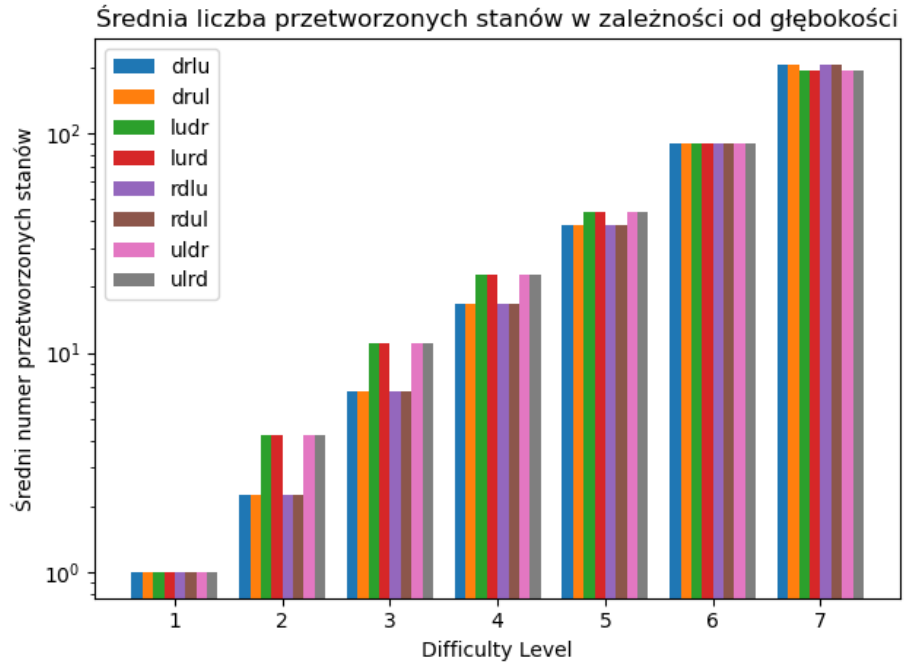
Rysunek 6. A*: trudność a długość rozwiązania



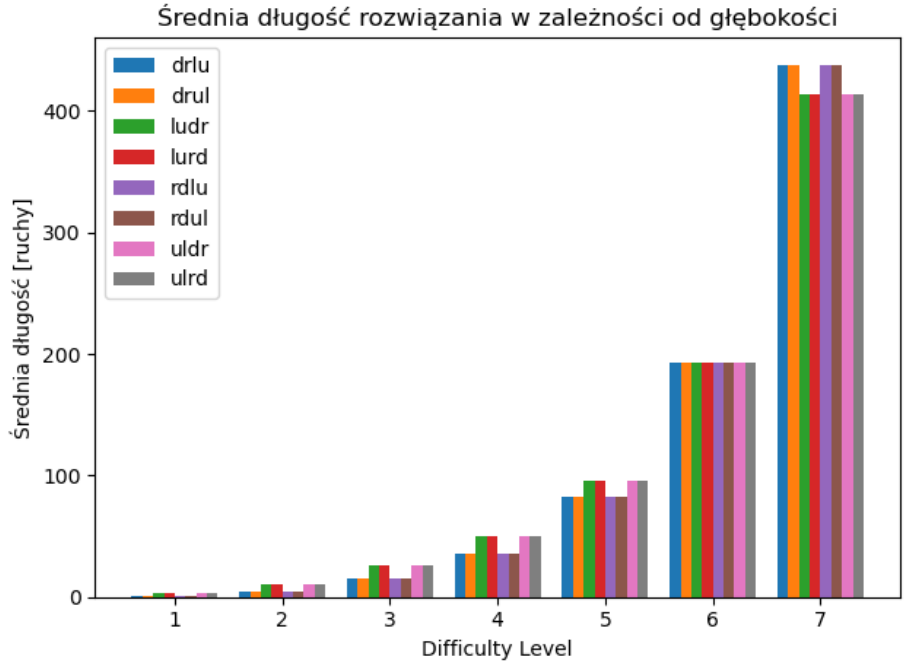
Rysunek 7. A*: trudność a czas rozwiązania



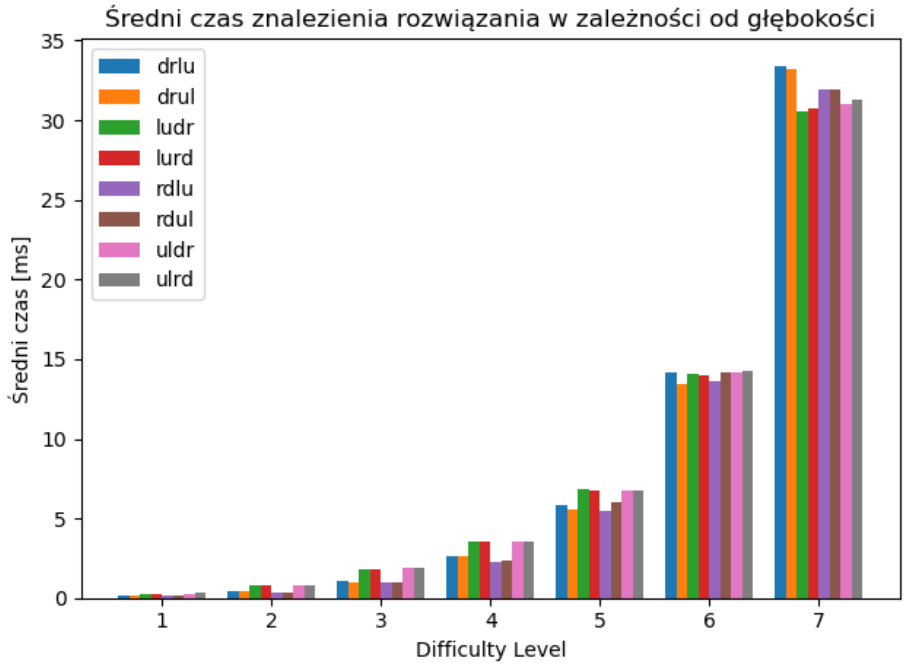
Rysunek 8. A*: trudność a odwiedzone stany



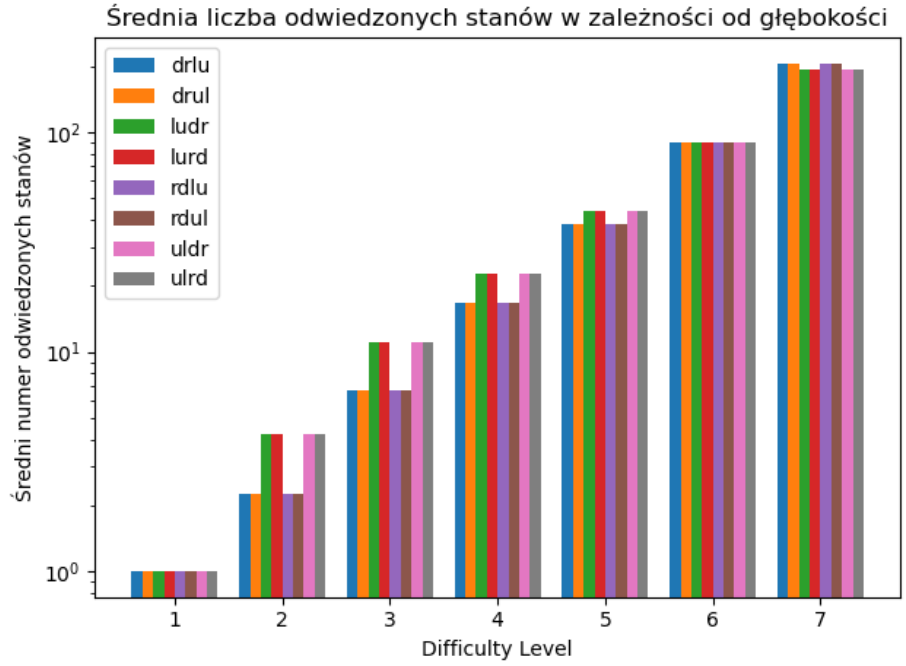
Rysunek 9. BFS: trudność a przetworzone stany



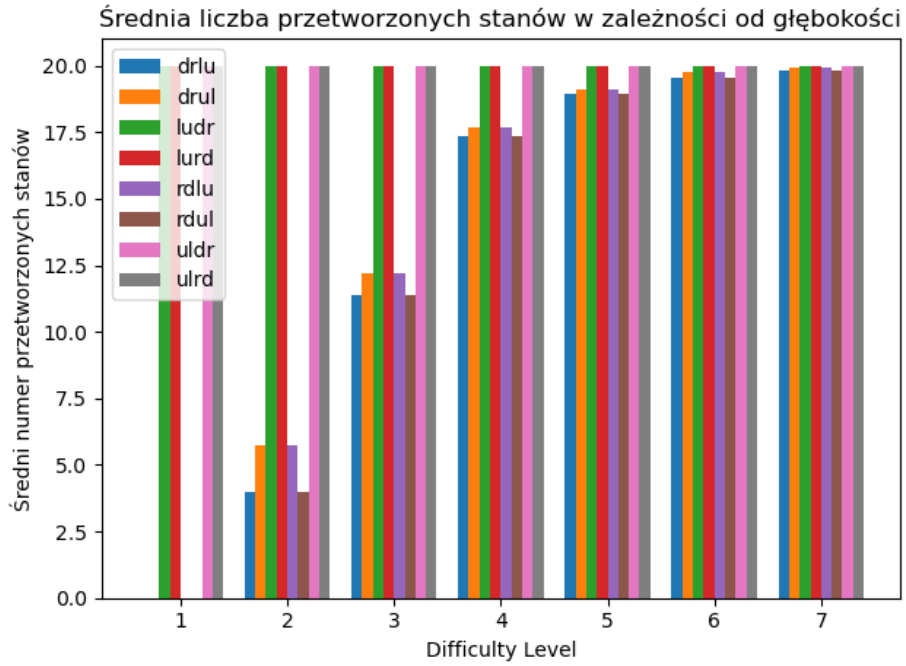
Rysunek 10. BFS: trudność a długość rozwiązania



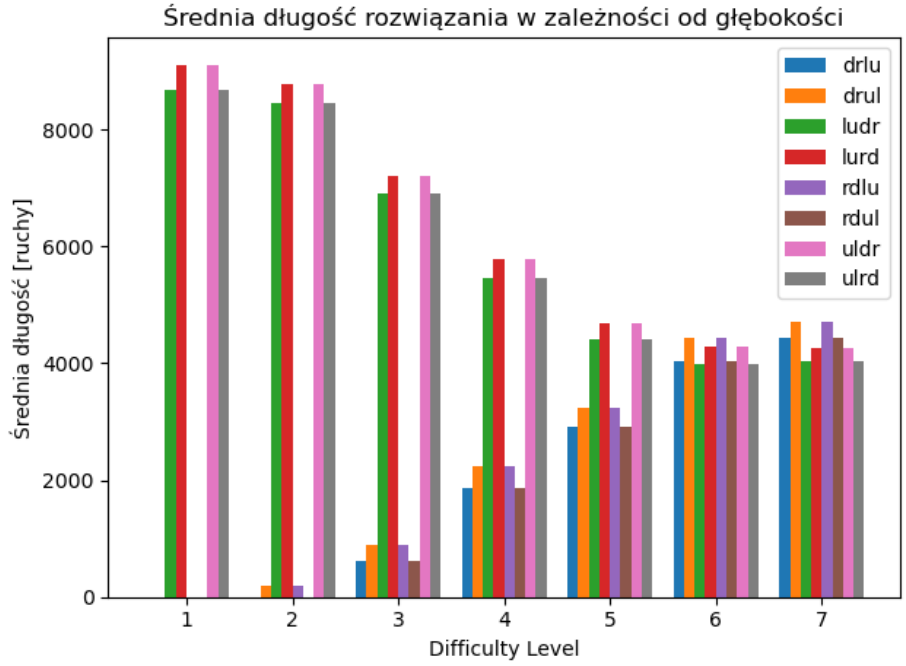
Rysunek 11. BFS: trudność a odwiedzone stany



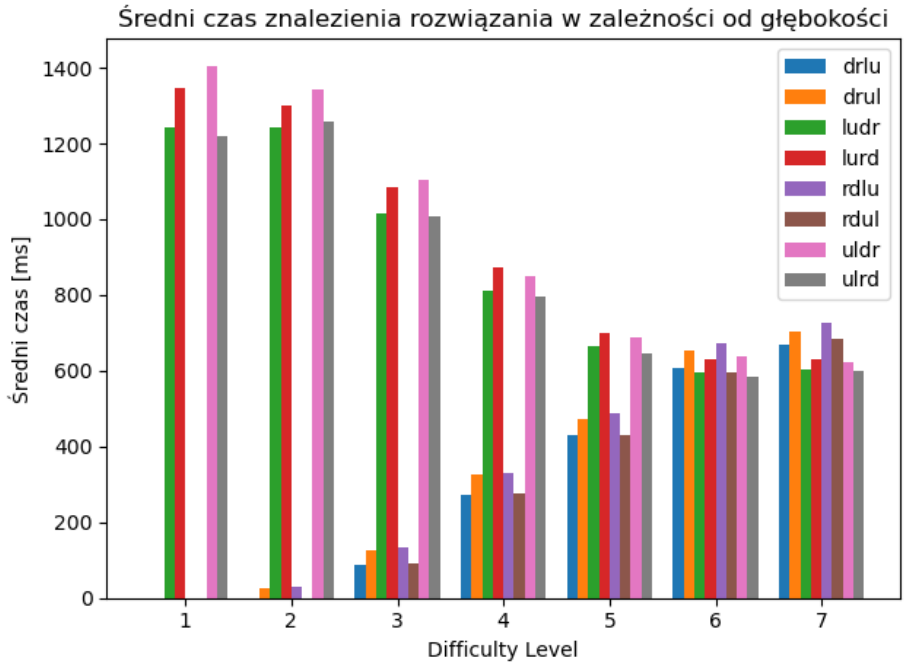
Rysunek 12. BFS: trudność a czas rozwiązania



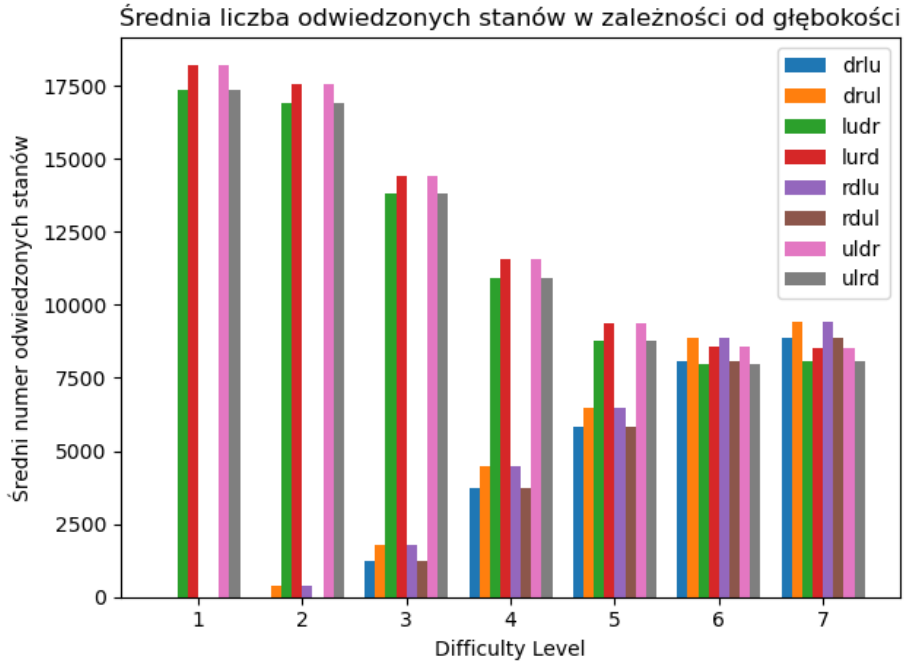
Rysunek 13. DFS: trudność a przetworzone stany



Rysunek 14. DFS: trudność a długość rozwiązania



Rysunek 15. DFS: trudność a czas rozwiązania



Rysunek 16. DFS: trudność a odwiedzone stany

7. Wnioski

Algorytm A* okazał się najszybszy, ale w teorii nie zawsze może dać rozwiązanie w przypadku bardziej skomplikowanych układanek. W przypadku tego algorytmu w naszych eksperymentach czas wykonania programu jak i ilosć odwiedzonych stanów pozostawała na bardzo niskim poziomie w miarę zwiększania się trudności poszczególnych układanek.

Algorytm bfs jest w teorii algorytmem, który zawsze znajdzie rozwiązanie, nie zawsze jednak w tak optymalnym czasie, jak algorytm wykorzystujący heurystyki. W tym przypadku czas rozwiązywania układanek przez algorytm bfs był na niskim poziomie, rozwiązane zostały wszysktie układanki. BFS jest algorytmem optymalnym, gdy nie znamy rozkładu danych i chcemy mieć pewność uzyskania rozwiązania.

DFS okazał się w tym zestawieniu najwolniejszy, i choć rozwiązał wszystkie zadane układanki, robił to z największą liczbą przetworzonych i odwiedzonych stanów, a co za tym idzie z największym czasem wykonywania. Algorytm ten jeżeli źle, nieoptymalnie będzie wybierał kolejne węzły znacznie wydłuża swój czas wykonania i liczbę przetwarzanych/odwiedzanych stanów. Sytuacja wygląda inaczej, gdy znamy rozkład danych – wtedy DFS przy dobraniu optymalnych węzłów przetwarzających może okazać się naprawdę wydajnym rozwiązaniem.

— l2short T. Oetiker, H. Partl, I. Hyna, E. Schlegl. *Nie za krótkie wprowadzenie do systemu LaTeX2e*, 2007, dostępny online.
— `https://pl.wikipedia.org/wiki/Przeszukiwanie_wszerz`
— `https://pl.wikipedia.org/wiki/Przeszukiwanie_w_g%C5%82%C4%85b`
— `https://pl.wikipedia.org/wiki/Algorytm_A*`
— `https://www.algorytm.edu.pl/grafy/bfs.html`
— `https://www.algorytm.edu.pl/grafy/przeszukiwanie-w-glab.html`