PSZT - przeszukiwanie
Dokumentacja do projektu ”Labirynt”

Łukasz Pokorzyński (300251), Adam Steciuk (nr 300263)
Listopad 2020

1 Projekt - MM.P4 Labirynt

Napisać program porównujący działanie algorytmów przeszukiwania BFS, DFS, IDFS dla problemu znalezienia drogi w labiryncie. Przestrzeń dyskretna - dozwolone ruchy to góra, dół, lewo, prawo.
WE: plik ze strukturą labiryntu z we/wy (dla większych użyć jakiegoś generatora).
WY: znaleziona najkrótsza ścieżka i/lub mapka z zaznaczoną ścieżką.

2 Przyjęte założenia

Przyjęliśmy, że labirynt może mieć nieograniczony rozmiar zarówno w wysokości, jak i w szerokości. W trakcie generacji dajemy możliwość ustalenia czy układ labiryntu ma być losowy, czy oparty na podanym ziarnie (seed). Wybrać także można dwa warianty generacji: z jedną poprawną ścieżką bądź z wieloma. Podczas rozwiązywania labiryntu możliwe jest ustalenie wielokrotnego rozwiązywania razem z losowo wybranymi początkiem i końcem szukanej ścieżki.

3 Podział zadań

- **Łukasz Pokorzyński** - Zaprojektowanie menu głównego, implementacja BFS, IDFS, rysowanie mapy i znalezionej ścieżki (plik SVG), testy
- **Adam Steciuk** - Generacja labiryntu, zapis/wczytywanie pliku reprezentującego labirynt, implementacja DFS, IDFS, modyfikacja solvera do uwzględniania średniej przebiegów

Dokumentacja jak i analiza wyników leżała w obowiązku obu członków zespołu.

4 Realizacja programu i algorytmów

Program otwiera się do menu głównego, gdzie użytkownik może dokonać wyboru z dostępnych opcji (generacja labiryntu, rozwiązanie labiryntu, zapis do pliku tekstowego struktury labiryntu, wyjście oraz pomoc odnośnie opcji). Korzysta z bibliotek statistics, time, random, collections (deque, defaultdict) i itertools (count).

- **BFS**

Algorytm ten używa **kolejki FIFO** do odpowiedniej obsługi analizowanych ścieżek. Dopóki kolejka nie jest pusta, algorytm pobiera z kolejki ścieżkę i analizuje, czy jej ostatnia komórka nie jest naszym celem lub została już odwiedzona. Jeżeli nie, to do kolejki dodawane są kolejne ścieżki, których ostatnimi elementami są komórki sąsiadujące (chyba, że sąsiad już został odwiedzony).

- **DFS**

Algorytm jest bliźniaczy do BFS z tą różnica, że używany jest **stos** do analizy kolejnych dodawanych ścieżek. Gwarantuje to przechodzenie ścieżki wgłąb w przeciwieństwie do BFS i kolejki, które powodują analizę wszerz.

- **IDFS**

Wykonywany jest iteracyjnie z coraz głębszym eksplorowaniem grafu. Podzielony jest na dwie części. Pierwsza, *iter_deepening*, odpowiada za iteracyjne wywoływanie funkcji i zwiększanie możliwej głębokości co iterację. Druga, *depth_limited*, jest modyfikacją DFS kontrolującą na jakiej głębokości grafu aktualnie znajduje się analizowana ścieżka. Do tego celu zapisywane są wartości głębokości dla węzłów.

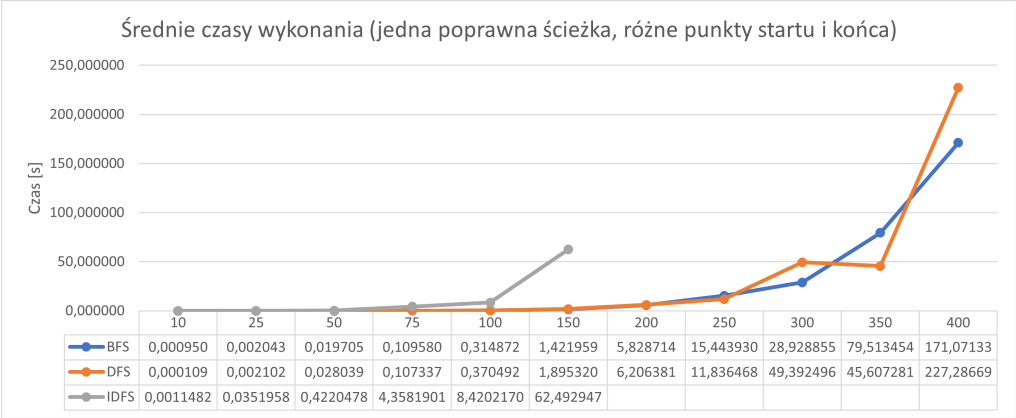
Algorytm ten, tak samo jak DFS, korzysta ze **stosu**.

5 Założenia odnośnie przeprowadzania testów

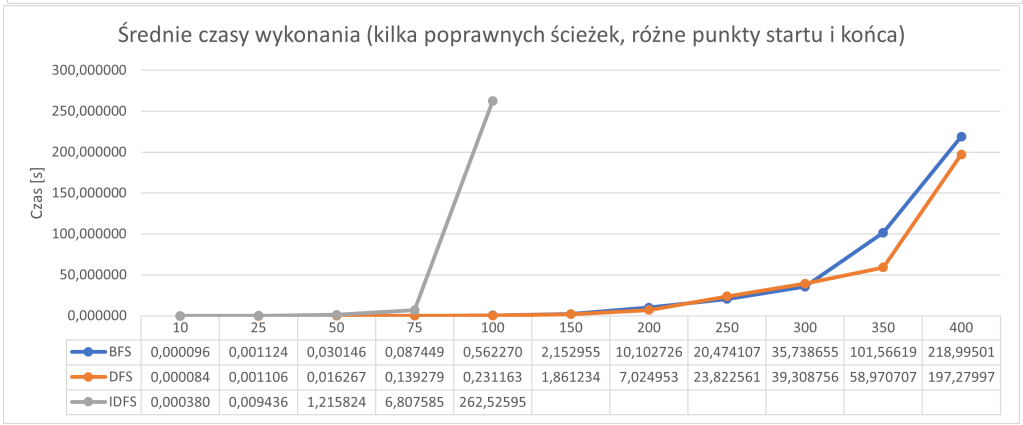
Aby móc rzetelnie porównać skuteczność zadanych algorytmów dla problemu znajdowania drogi w labiryncie, należy przeprowadzić testy dla sytuacji zarówno optymistycznych jak i pesymistycznych dla każdego algorytmu. Natura naszej implementacji labiryntu powoduje, że algorytmy mają tendencję do przeszukiwania labiryntu ”na północny zachód” (w takiej kolejności dodawani są do listy sąsiadów kolejne komórki). Rozwiązanie to albo dyskryminowało DFS zawsze starający się maksymalnie trawersować gałęzie idące we wskazanych kierunkach (w przypadku startu w pobliżu lewego górnego narożnika i mety w pobliżu prawego dolnego) albo dawało mu ”niesprawiedliwą” przewagę (w przypadku odwrotnym). Aby uśrednić wyniki i móc porównywać rzeczywiste skuteczności algorytmów zdecydowaliśmy dla każdego rozpatrywanego rozmiaru problemu wyznaczać średnią z 10 poszukiwań drogi o losowo wybieranych punktach startu i mety.

6 Wyniki testów

Dane dla labiryntów o danym boku (oś X na wykresach):

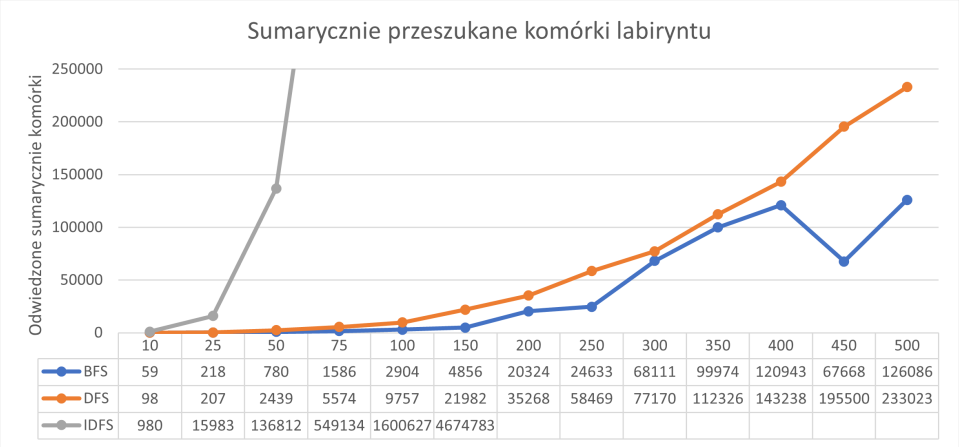


	10	25	50	75	100	150	200	250	300	350	400
BFS	0,000950	0,002043	0,019705	0,109580	0,314872	1,421959	5,828714	15,443930	28,928855	79,513454	171,07133
DFS	0,000109	0,002102	0,028039	0,107337	0,370492	1,895320	6,206381	11,836468	49,392496	45,607281	227,28669
IDFS	0,0011482	0,0351958	0,4220478	4,3581901	8,4202170	62,492947					



	10	25	50	75	100	150	200	250	300	350	400
BFS	0,000096	0,001124	0,030146	0,087449	0,562270	2,152955	10,102726	20,474107	35,738655	101,56619	218,99501
DFS	0,000084	0,001106	0,016267	0,139279	0,231163	1,861234	7,024953	23,822561	39,308756	58,970707	197,27997
IDFS	0,000380	0,009436	1,215824	6,807585	262,52595						

Długości znalezionych ścieżek przy wielu możliwych rozwiązaniach				
Wielkość labiryntu	10x10	25x25	50x50	75x75
BFS	21	63	119	187
DFS	43	273	975	1353
IDFS	21	63	119	187



	10	25	50	75	100	150	200	250	300	350	400	450	500
BFS	59	218	780	1586	2904	4856	20324	24633	68111	99974	120943	67668	126086
DFS	98	207	2439	5574	9757	21982	35268	58469	77170	112326	143238	195500	233023
IDFS	980	15983	136812	549134	1600627	4674783							

7 Wnioski

Złożoność czasowa widoczna na wykresach, pomimo odchyłów spowodowanych losowością, to złożoność wykładnicza dla wszystkich trzech algorytmów. W zależności od wylosowanego labiryntu może on bardziej lub mniej sprzyjać danemu sposobowi przeszukiwania. BFS będzie gorzej radził sobie w grafach o dużej szerokości rozgałęzienia, DFS w grafach o ścieżkach bardzo głębokich.
Patrząc na czasy wykonywania można stwierdzić, że IDFS dla problemu labiryntu nie jest najlepszym rozwiązaniem. Jego złożoność czasowa znacznie szybciej rośnie wykładniczo niż dla pozostałych dwóch sposobów. Tak samo liczba przebadanych komórek jest znacznie wyższa niż dla DFS oraz BFS, co wynika z iteracyjnej natury algorytmu i przechodzenia tych samych punktów kilkanaście razy (odwiedzone komórki nie są zapisywane między iteracjami).
Łatwy do zauważenia jest fakt, że DFS, w przeciwieństwie do BFS oraz IDFS, zwraca ścieżkę potwierdzającą istnienie połączenia punktu A i punktu docelowego B w labiryncie. Pozostałe algorytmy zwracają ścieżki optymalne o tej samej długości, które dla labiryntu o kilku możliwych drogach mogą nieznacznie się różnić.

8 Instrukcja obsługi

Program uruchamiamy przez plik main.py przy pomocy interpretera Pythona. Użytkownik wybiera jedną z opcji: 1) generację labiryntu (z pliku lub na podstawie parametrów użytkownika), 2) znalezienia rozwiązania labiryntu, 3) zapisu struktury badanego labiryntu do pliku, 4) wyjścia oraz h) opcji pomocy. Wpisując odpowiednią wartość porządkową (1, 2, 3, 4 lub h) przechodzimy do danego podmenu. Opcje 3, 4, h wykonują się samoczynnie wyświetlając stosowne informacje na ekranie, po czym przenoszą do menu głównego. Opcja 1 pyta o ewentualny plik tekstowy ze strukturą labiryntu, jeżeli go nie uzyska, to pyta o parametry labiryntu (x, y, seed, czy ma mieć tylko jedną ścieżkę). Opcja 2 pyta czy szukanie ścieżki ma nastąpić między punktami (0,0), a (x-1, y-1), czy ma losować komórki startu i celu, wtedy też pyta ile takich losowych przeszukań ma przeprowadzić.