PSZT - przeszukiwanie
Dokumentacja do projektu ”Labirynt”

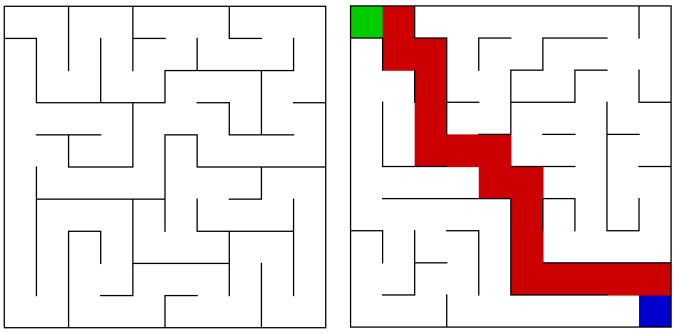
Łukasz Pokorzyński (300251), Adam Steciuk (nr 300263)
Listopad 2020

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). Budowany jest z komórek, których ułożenie jest przechowywane jako graf reprezentowany przez słownik, który każdej komórce przyporządkowuje jej (nieoddzielonych ścianą) sąsiadów. Wybrać 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. Żaden z algorytmów nie korzysta z rekurencji. Poniżej z lewej: labirynt wygenerowany z jedną ścieżką; z prawej: labirynt z wieloma ścieżkami i rozwiązaniem znalezionym przy pomocy BFS (start zielony, cel niebieski)



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 czasów analizy

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), itertools (count) oraz webbrowser i pathlib.

- **BFS**

Algorytm ten używa **kolejki FIFO** do odpowiedniej obsługi analizowanych ścieżek. Dopóki kolejka nie jest pusta, algorytm pobiera z kolejki ścieżkę i analizuje, czy ostatni węzeł grafu nie jest naszym celem lub został już odwiedzony. Jeżeli nie, to do kolejki dodawane są kolejne ścieżki, których ostatnimi elementami są węzły 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.

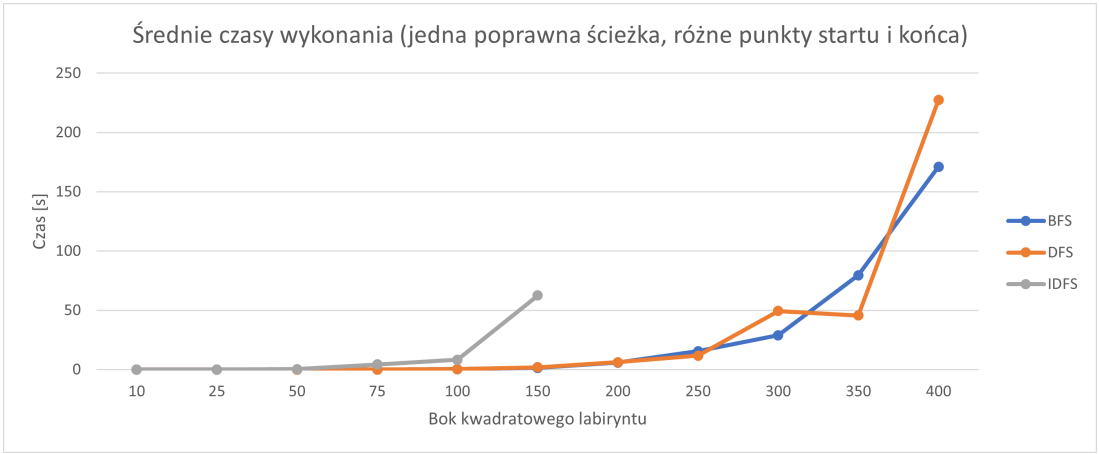
- **IDFS**

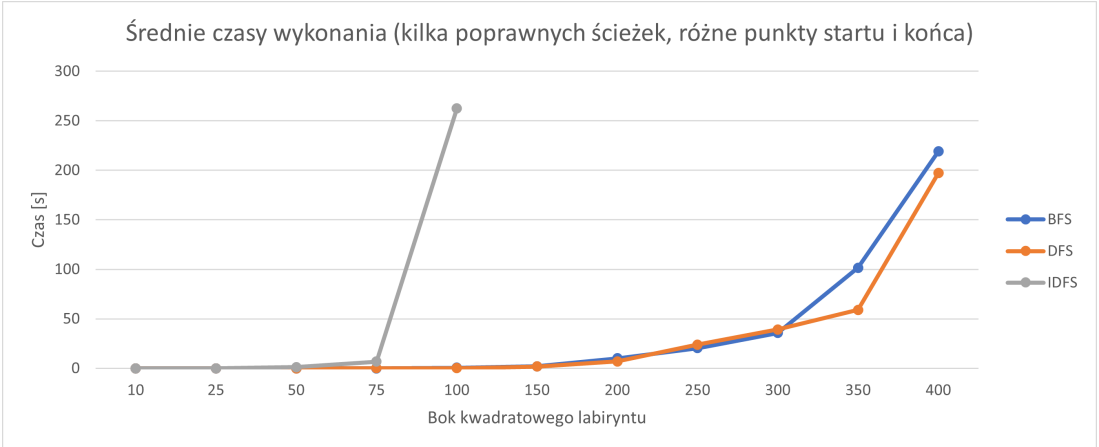
Wykonywany jest iteracyjnie z coraz głębszym eksplorowaniem grafu. Podzielony jest na dwie części. Pierwsza, *iter_deepening*, odpowiada za iteracyjne wywoływanie funkcji i zwiększanie możliwej głębokości co iterację. Druga, *depth_limited*, jest modyfikacją DFS kontrolującą na jakiej głębokości grafu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

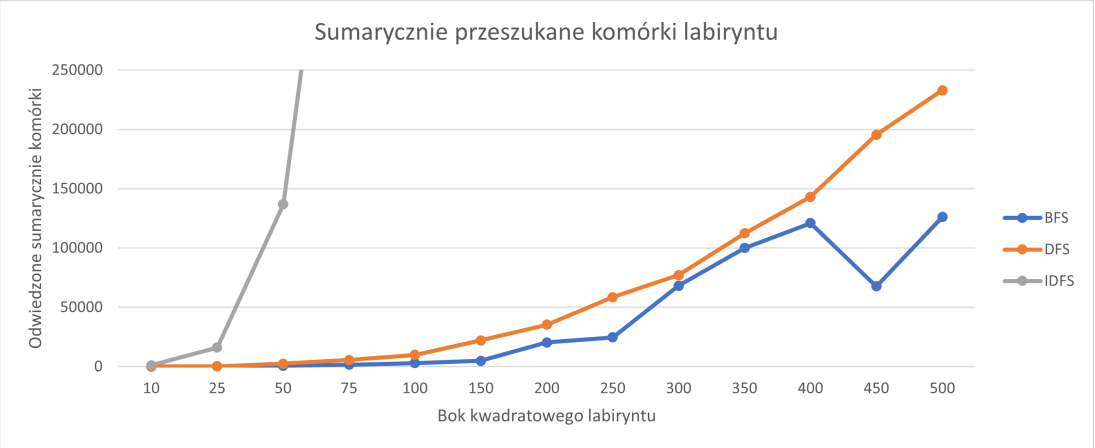
5 Założenia odnośnie przeprowadzania testów

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. Preferencja przeszukiwania zaimplementowana przez nas w programie powoduje, że algorytmy mają tendencję do przechodzenia labiryntu ”na północny zachód” (w takiej kolejności dodawane są do listy sąsiadów kolejne wierzchołki grafu). 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 czasu działania z 10 poszukiwań drogi o losowo wybieranych punktach startu i mety.

6 Wyniki testów







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

7 Wnioski, obserwacje, przemyślenia

Złożoność czasowa widoczna na wykresach, pomimo odchyłów spowodowanych losowością, jest wykładnicza dla wszystkich trzech algorytmów, a wpływa na to współczynnik rozgałęzienia (maksymalnie wynoszący trzy) oraz głębokość grafu. Stąd wylosowany labirynt może bardziej lub mniej sprzyjać danemu sposobowi przeszukiwania - BFS będzie gorzej radził sobie w grafach o dużej szerokości rozgałęzienia, DFS w grafach o ścieżkach na znacznej głębokości.

Patrząc na czasy wykonania można stwierdzić, że IDFS dla problemu labiryntu nie jest najlepszym rozwiązaniem. Jego złożoność czasowa znacznie szybciej rośnie niż dla pozostałych dwóch sposobów, tak samo liczba przebadanych komórek jest znacznie wyższa niż dla DFS oraz BFS. Właściwości te wynikają 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 i IDFS, zwraca ścieżkę potwierdzającą istnienie połączenia punktu A i punktu docelowego B w grafie. Pozostałe algorytmy zwracają ścieżki optymalne o tej samej długości, które dla labiryntu o kilku możliwych drogach mogą nieznacznie się różnić. Różnica długości ścieżek bierze się z analizowania przez DFS ”do oporu”, czyli do momentu, aż któryś z węzłów grafu nie będzie posiadał połączeń z nieodwiedzonymi sąsiadami. IDFS, dzięki temu, że jest hybrydą BFS i DFS, jest w stanie zawsze znaleźć ścieżkę optymalną. Powoduje to wprowadzony współczynnik maksymalnej głębokości, który nie pozwala całkowicie na analizę w stylu DFS i wymusza na poruszanie się wszerz, tak jak BFS.

W ogólności DFS przechodzi po węzłach grafu więcej razy niż BFS, lecz nie musi to być koniecznie jego wada. Znów, dzięki właściwości przeszukiwania jednej ścieżki aż do napotkania ślepego zaułka, może on czasami znajdować rozwiązanie szybciej niż BFS, jeżeli podąży w odpowiednim kierunku, który jest dyktowany przez przyjętą preferencję kolejności przeszukiwania węzłów. Z tego powodu ważne dla nas było, by sytuacje szukania były jak najbardziej zróżnicowane.

Wszystkie algorytmy przez nas zaimplementowane mają podobną złożoność pamięcią - każdy z nich korzysta ze struktury składującej przeszukiwane ścieżki oraz struktury zapisującej węzły odwiedzone (w przypadku IDFS jest to tablica przechowująca głębokość, na której napotkany został węzeł).

Załącznik

Instrukcja obsługi

1. Program uruchamiamy przez interpreter Pythona z folderu zawierającego kod albo z terminala komendą:

   ```
   python main.py
   ```

2. Opcję w menu wybieramy wpisując odpowiednią wartość porządkową (1, 2, 3, 4 lub h) i zatwierdzając klawiszem Enter.
3. Po wybraniu opcji dokonywane są następujące czynności:
   1) Generacja labiryntu - buduje labirynt z pliku tekstowego lub pobiera od użytkownika parametry do jego utworzenia (x, y, seed, czy ma mieć tylko jedną ścieżkę)
   2) Szukanie rozwiązania - jeżeli labirynt został już stworzony, szuka rozwiązań oraz wyświetla czas wykonywania i liczbę kroków (przeszukanych węzłów grafu) dla algorytmu. Jeżeli szukanie ścieżki następuje po losowaniu startu i celu, wtedy program pyta ile takich losowych przeszukań ma przeprowadzić i wyświetla dodatkowo średnią z czasów wykonywania.
   3) Zapis do pliku .txt - zapisuje labirynt do pliku tekstowego jako mapa numeryczna z wartościami 0-3
   4) Wyjście - wyłącza działanie programu
   h) Pomoc - wyświetla opis opcji ”w pigułce”