Projekt 1

Zadanie 1 - Konwersje reprezentacji grafów

Aby skonwertować graf do innej reprezentacji, trzeba użyć klasy “wrappującej” grafy nieskierowane: ***GraphRepresentation*** (core.graph_representation).

Jeśli chcemy stworzyć obiekt klasy przy pomocy struktur ‘pythonowych’ ***GraphRepresentation***, klasa w konstruktorze, pod parametrem ***repr_type*** oczekuje struktury:
- W przypadku macierzy incydencji, oraz sąsiedztwa jest to po prostu macierz jako lista list.
- W przypadku listy sąsiedztwa, jest to słownik, gdzie klucze to numery węzłów, a wartości to listy oznaczające sąsiadów danego węzła.

Dane potrzebne z do stworzenia grafu, możemy wczytać z dysku za pomocą metod:
(utilites.read_from_file)

incidence_matrix_from_file
adjacency_matrix_from_file
ajacency_list_from_file

Przykład takich pliku jakiego oczekują powyższe funkcje, możemy znaleźć w docstringach tych funkcji. Oczekiwany format pliku wejściowego to tekstowy (.txt, .dat).

Metody
incidence_matrix_from_file, adjacency_matrix_from_file,
oczekują macierzy w formacie kolumn oddzielonych spacjami, zaś wierszy oddzielonych znakami nowej linii.
Metoda ***ajacency_list_from_file*** oczekuje w pierwszej kolumnie numeru opisywanej listy sąsiedztwa, a w kolejnych elementów tej listy. Należy pamiętać, że listy indeksujemy od 0. Listy oddzielone są znakami nowej linii, a elementy listy od indeksów, oraz indeksy od samych siebie, spacjami.

Do realizowania konwersji, służy metoda ***GraphRepresenation.convert(GraphRepresentationType)***.. Metoda aktualną konwertuje reprezentacje matematyczną, na tą podaną w argumencie.

Przykłady użycia podane w tests.Task_1_1

Zadanie 2 - Wyświetlanie grafów

Do wyświetlania grafów służy metoda ***GraphRepresentation.display().***
Uwaga! Metoda działa tylko na typie listy sąsiedztwa, więc przed wywołaniem tej metody, wymagana jest konwersja na ten typ:

GraphRepresentation.convert(GraphRepresentationType.ADJACENCY_LIST)
GraphRepresentation.display()

Przykłady użycia znajdują się w tests.Task_1_2

Zadanie 3 - Losowanie grafów

Do losowania grafów w tym zestawie, służą metody:
(random_generation.graph_generators)

generate_with_edges
generate_with_probability
Przykłady użycia znajdują się w tests.Task_1_3

Opis argumentów które należy podać dla ich działania znajduję się w docstringach.
Uwaga! przy dużej liczbie węzłów, wyświetlanie za pomocą metody “display” może być niedokładne (wiele węzłów na kole powoduje zapełnianie się koła i w konsekwencji, nakładanie się na siebie węzłów, mimo to, że jest mechanizm, który zmniejsza wielkość węzłów, wraz ze wzrostem ich liczby).

Projekt 2

Zadanie 1 - Konstruowanie grafu z ciągu graficznego

Aby stworzyć graf na podstawie ciągu graficznego, możemy skorzystać z klasy ***GraphRepresentation*** i pola klasy ***GraphRepresentationType.GRAPHIC_SEQUENCE***.
W momencie tworzenia obiektu klasy ***GraphRepresentation*** , konstruktor tej klasy oczekuje aby reprezentacja matematyczna grafu, za pomocą struktury Pythonowej, była listą liczb całkowitych, która reprezentuje właśnie ciąg graficzny.
Sprawdzanie czy podana lista liczb całkowitych jest reprezentuje ciąg graficzny, jest realizowane przez funkcję ***check_if_seq_is_graphic*** (algorithms.checkers).

Przykłady użycia dla tego zadania możemy znaleźć w tests.Task_2_1.

Zadanie 2 - Randomizacja grafu

Aby zrandomizować zadany graf, należy stworzyć obiekt klasy ***GraphRepresentation*** podając dowolną reprezentacje matematyczną, można to zrobić z pliku, lub ręcznie, pythonową strukturą (opisane w Projekcie 1), następnie użyć funkcji ***randomize_graph*** (algorithms.graph_randomization).
Uwaga! Algorytm randomizacji, użyty na innej postaci grafu niż macierz sąsiedztwa, **sam go skonwertuje to macierzy sąsiedztwa**, należy o tym pamiętać, jeżeli zachodzi potrzeba wyświetlenia zrandomizowanego grafu (konwersja do listy sąsiedztwa).

Przykłady użycia dla tego zadania możemy znaleźć w tests.Task_2_2.

Zadanie 3 - Spójne składowe

Aby znaleźć spójne składowe grafu, należy skorzystać z klasy ***CohenretComponentFinder*** (algorithms.coherenet_component)
Znajdowanie spójnej składowej realizowane jest przez
CohenretComponentFinder.find metoda zwraca liste węzłow, którym przyporządkowane są konkretne składowe tzn.indeksy listy odpowiadają za numer węzła a wartość elementu w liście, za numer składowej do której przynależy węzeł.
zaś znajdowanie największej spójnej składowej przez
CohenretComponentFinder.most_common_component metoda zwraca listę numerów węzłów, które przynależą do największej znalezionej spójnej składowej.
Obie funkcje oczekują na wejściu obiektu klasy ***GraphRepresentation*** która posiada reprezentacje matematyczną w postaci macierzy sąsiedztwa.

Przykłady użycia dla tego zadania możemy znaleźć w tests.Task_2_3.

Zadanie 4 - Grafy eulerowskie

Do generowania grafów eulerowskich, można skorzystać z funkcji ***generate_random_euler_graph*** (random_generation.graph_generators), zaś do znajdowania w nim cyklu Eulera, służy klasa ***EulerCycleFinder*** (algorithms.euler_cycle).
EulerCycleFinder.find wypisuje ścieżke eulera wskazując na kolejne numery wierzchołków, po których należy przechodzić aby przejść ścieżką eulera, zwraca listę wierzchołków należących do ścieżki eulera.

Przykłady użycia dla tego zadania możemy znaleźć w tests.Task_2_4.

Zadanie 5 - Generacja grafów k-regularnych

Do generowania grafów k-regularnych, służy funkcja ***k_regular_graph*** (random_generation.graph_generators). Należy pamiętać, że drugi argument, czyli stopień danego węzła, musi być mniejszy lub równy od liczby węzłów - 1.

Przykłady użycia dla tego zadania możemy znaleźć w tests.Task_2_5.

Zadanie 6

Do znajdowania cyklu Hamiltona w grafie nieskierowanym, służy klasa ***HamiltionianGraphChecker .*** Tworząc obiekt klasy, należy przekazać w konstruktorze obiekt klasy ***GraphRepresentationType.***
Uwaga! Algorytm działa na podstawie macierzy sąsiedztwa, w konsekwencji przekonwertuje on graf podczas jego działania właśnie na tę postać.

Przykłady użycia dla tego zadania możemy znaleźć w tests.Task_2_6.

Projekt 3

Zadanie 1 - Generacja grafów spójnych

Generacja grafów spójnych, z ważonymi krawędzami, zrealizowana jest za pomocą klasy ***generate_connected_graph*** (random_generation.graph_generation). Funkcja ta zwraca obiekt klasy nx.Graph. Możemy odzyskać macierz sąsiedztwa z tego obiektu, za pomocą metody ***nx.to_numpy_array,*** wynikowo dostaniemy ***np.array*** który jest macierzą sąsiedztwa.

Przykłady użycia dla tego zadania możemy znaleźć w tests.Task_3_1.

Zadanie 2 - Algorytm znajdowania najkrótszej ścieżki

Klasa realizująca znajdowanie najkrótszej ścieżki za pomocą algorytmu Dijkstry, to ***DijkstrasAlgorithm*** (algorithms.Dijkstras_algorithm). Obiekt klasy realizującej ten algorytm, wymaga w konstruktorze macierzy sąsiedztwa, oraz macierzy krawędzi, którą uzyskamy
za pomocą funkcji ***generate_branch_matrix*** (algorithms.Dijkstras_algorithm).
Następnie za pomocą metody ***DijkstrasAlgorithm.all_shortest_paths*** podając jako argument, węzeł początkowy, od którego chcemy znaleźć najkrótsze ścieżki do innych węzłów. W konsoli wypiszą się najkrótsze ścieżki od węzła podanego z argumentu do pozostałych węzłów, wraz z najmniejszymi odległościami do nich.

Przykłady użycia dla tego zadania możemy znaleźć w tests.Task_3_2.

Zadanie 3 - Macierz odległości węzłów

To zadanie opiera się na użyciu funkcjonalności opisanych instrukcji do poprzedniego zadania, oraz metody ***DijkstrasAlgorithm.create_distance_matrix*** . Metoda ta zwraca liste list, która tworzy macierz, z jedynkami na diagonali (ponieważ odległość węzła od samego siebie to 0).

Przykłady użycia dla tego zadania możemy znaleźć w tests.Task_3_3.

Zadanie 4 - Węzeł centralny, węzeł centralny minimax

Aby uzyskać węzły minmax, oraz centralny, należy na otrzymanej macierzy, którą zwraca opisana wcześniej funkcja ***DijkstrasAlgorithm.create_distance_matrix*** , zastosować metody ***center_node_minmax, center_node*** (algorithms.center_node). Metody zwracają indeksy węzłów, które zostały wyznaczone jako, odpowiednio, centrum minimax, oraz węzeł centralny.

Przykłady użycia dla tego zadania możemy znaleźć w tests.Task_3_4.

Zadanie 5 - Minimalne drzewo rozpinające

W tym zadaniu, do wyznaczenia minimalnego drzewa rozpinającego, została skonstruowana funkcja ***find_minimum_spanning_tree*** (algorithms.minimum_spanning_tree). Metoda przyjmuje i zwraca obiekt klasy ***networkx.Graph*** . Obiekt zwracany przez metodę to właśnie minimalne drzewo rozpinające grafu wejściowego.

Przykłady użycia dla tego zadania możemy znaleźć w tests.Task_3_5.

Adnotacja do korzystania z części biblioteki operującej na grafach ważonych, skierowanych. Biblioteka tylko w przypadku grafów nieskierowanych, korzysta z ***GraphRepresentation*** (core.graph_representation). W innych wypadkach, używana jest klasa ***networkx.Graph / networkx.DiGraph*** . Stworzono szereg udogodnień dla użytkownika, mających na celu ułatwić wyświetlanie obiektów klas wymienionych wyżej (visualization.nx_graph). Wszystkie metody ułatwiające wyświetlenie tych grafów, mają parametr opcjonalne ***filename***, jeśli zostanie on podany, graf nie tylko zostaniewyświetlony na ekran, ale też zostanie zapisany do pliku o nazwie ***filename*** (należy pamiętać o dobrym rozszerzeniu), w głównym katalogu projektu.

W przypadku algorytmów operujących na tych klasach, obsługa tych obiektów, spoczywa na bibliotece ***networkx*** , wyświetlanie ich jest wyjątkiem. Dlatego np. wczytywanie danych do obiektów tej klasy powinno być zrealizowane zgodnie ze wskazówkami producenta biblioteki.
Link do dokumentacji:
https://networkx.org/documentation/stable/index.html
Tworzenie obiektów klas, za pomocą danych pochodzących z różnych źródeł:
https://networkx.org/documentation/stable/reference/readwrite/index.html