Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **Informatyka algorytmiczna (INA)**
Specjalność: **Algorytmika (ALG)**

PRACA DYPLOMOWA MAGISTERSKA

Zastosowanie szkiców danych w analizie dużych grafów

Application of data sketches in the analysis of large graphs

Paweł Polerowicz

Opiekun pracy
dr inż. Jakub Lemiesz

Słowa kluczowe: szkice danych, analiza grafów, strumienie danych

WROCŁAW, 2024

Streszczenie

Szkice danych stanowią potężne narzędzie w analizie wielkich zbiorów danych, w tym grafów. W niniejszej pracy definiujemy model strumieni grafowych i dokonujemy przeglądu istniejących rozwiązań w tej dziedzinie. Rozważamy metody polegające na tworzeniu zanurzeń wierzchołków grafu, w szczególności algorytm `NodeSketch`, który wykorzystuje próbki generowane z rozkładu wykładniczego do rekurencyjnego szkicowania wierzchołków na podstawie ich k-sąsiedztwa. Proponujemy także ulepszenie metody `NodeSketch` poprzez wykorzystanie schematu szkicowania opartego na algorytmie `ExpSketch`, co pozwala na wykonywanie szerszej gamy operacji na szkicach. Przeprowadzamy analizę teoretyczną złożoności obliczeniowej powstałego w ten sposób algorytmu `EdgeSketch` oraz przeprowadzamy liczne eksperymenty, aby ocenić jego skuteczność praktyce. Wyniki uzyskane w zadaniu rekonstrukcji grafu pokazują, że `EdgeSketch` przewyższa `NodeSketch` pod względem precyzji.

Słowa kluczowe: szkice danych, analiza grafów, strumienie danych

Abstract

Data Sketches are a powerful tool for analyzing large datasets, including graphs. In this thesis, we define the graph streaming model and review existing solutions in this field. We focus on methods that create embeddings of graph nodes. In particular, we discuss the `NodeSketch` algorithm, which uses samples generated from exponential distribution to recursively create sketches based on the k-neighborhood of a node. We then propose a way to augment `NodeSketch` with the different sketching scheme, based on `ExpSketch` algorithm, which allows more operations to be performed on data sketches. We call resulting algorithm `EdgeSketch`. We provide a theoretical analysis of its complexity and perform extensive experiments to evaluate it in practice. Results from node reconstruction task show that `EdgeSketch` consistently outperforms `NodeSketch` in terms of precision.

Keywords: data sketches, graph analysis, data streams

Spis treści

Spis rysunków

Spis tabel

Wstęp

W dobie internetu i powszechnej informatyzacji zachodzi potrzeba przetwarzania coraz większych zbiorów informacji. Wiele z nich wygodnie jest traktować jako dane grafowe. Stanowi to jednak wyzwanie dla tradycyjnych algorytmów i struktur danych, które często okazują się nieefektywne dla grafów o milionach, a nawet miliardach krawędzi, zarówno pod względem czasu obliczeń, jak i potrzebnych zasobów pamięciowych. W ostatnich latach temat ten stał się przedmiotem intensywnych badań, czego skutkiem było powstanie wielu efektywnych rozwiązań, pozwalających na analizowanie nawet bardzo dużych grafów w rozsądnym czasie. Wiele z nich wykorzystuje do tego celu szkice danych, czyli kompaktowe reprezentacje oryginalnych danych. W niniejszej pracy przyglądamy się takim rozwiązaniom, a w szczególności algorytmowi `NodeSketch`, tworzącemu szkice grafów poprzez rekurencyjne podsumowywanie otoczeń wierzchołków i generowanie na ich podstawie próbek z rozkładu wykładniczego. Na jego podstawie definiujemy algorytm `EdgeSketch`, wykorzystujący metodę `FastExpSketch` do szkicowania wierzchołków. Modyfikacja ta pozwala na wykonywanie bardziej złożonych operacji na szkicach. Przeprowadzone eksperymentu pokazują, że pomysł ten dobrze sprawdza się w praktyce, oferując lepszą precyzję przy rekonstrukcji grafu w stosunku do oryginalnego algorytmu `NodeSketch`, a także zmniejszając średnią liczbę drogich operacji, jak np. obliczanie wartości funkcji haszującej.

Struktura pracy

Praca rozpoczyna się od niniejszego wstępu, przedstawiającego ogólny zarys podejmowanej problematyki i skrótowo podsumowującego jej wkład badawczy. W pierwszym rozdziale znajduje się opis problemu wraz z formalną definicją i przedstawieniem różnych jego wariantów. Przedmiotem drugiego rozdziału jest przegląd literatury związanej z analizą wielkich grafów, z podziałem na zastosowane metodyki. W trzecim rozdziale szczegółowo omówione zostały algorytmy `ExpSketch` i `FastExpSketch`, a także operacje na szkicach danych. Czwarty rozdział zawiera ideę oraz dokładny opis działania algorytmu `EdgeSketch`, stanowiącego główny przedmiot pracy. W piątym i zarazem ostatnim rozdziale opisane zostały przeprowadzone eksperymenty, wraz z prezentacją wyników oraz wnioskami z nich płynącymi. Następnym elementem jest podsumowanie pracy. Zawarte zostały w nim ogólne konkluzje na temat pracy oraz możliwe kierunki dalszych badań. Pracy towarzyszy wykaz literatury oraz dodatek, zawierający instrukcję użytkowania części implementacyjnej.

Rozdział 1

Sformułowanie problemu

1.1. Analiza grafów

Analiza danych jest dynamicznie rozwijającą się dziedziną informatyki, znajdującą zastosowania w wielu gałęziach przemysłu i badaniach naukowych. Wielka różnorodność rozważanych zbiorów danych, pochodzących z odmiennych źródeł, indukuje potrzebę znajdowania wszechstronnych i efektywnych struktur danych i algorytmów, które mogą służyć do ich reprezentacji i przetwarzania. Grafy doskonale nadają się jako narzędzie do tego typu zadań ze względu na ich wrodzoną zdolność do modelowania złożonych relacji i struktur, od sieci społecznościowych i topologii Internetu po systemy biologiczne i sieci transportowe. Dzięki tej wszechstronności algorytmy grafowe znajdują dziś zastosowania w praktyce, napędzając innowacje i wspomagając przetwarzanie coraz bardziej obszernych zestawów informacji. Pomimo że grafy towarzyszą informatyce niemal od samych jej początków, to jednak rozwój tej dziedziny nie ustaje, zwłaszcza że ilość i złożoność danych stale rośnie. W dzisiejszej erze, w której bazy danych często osiągają ogromne rozmiary, istnieje potrzeba dostosowania metodologii opartych na grafach do bardziej efektywnego przetwarzania informacji.

W niniejszej pracy będziemy posługiwać się głównie pojęciem grafu prostego, określanego po prostu jako graf. Będziemy go oznaczać przez $G = (V, E)$ - graf, gdzie V - zbiór wierzchołków i $E \subseteq V \times V$ - zbiór krawędzi.

1.2. Główne sposoby modelowania problemu

W niniejszej pracy pochylamy się nad kwestią analizy wielkich zbiorów danych, przedstawionych w postaci grafów. Jednak przed przystąpieniem do omawiania istniejących lub konstrukcji nowych rozwiązań, należy zastanowić się nad istotą problemu, z którym się mierzymy oraz wymaganiami i ograniczeniami, które proponowane algorytmy powinny spełniać. Kluczową kwestią jest więc wybór sposobu modelowania problemu. W kontekście analizy grafów możemy wyróżnić kilka ważnych i użytecznych modeli.

1.2.1. Model klasyczny

W tradycyjnej analizie grafów przyjmuje się dość prosty model, gdzie cały graf reprezentujący zbiór danych jest nam dany na wejściu do algorytmu. W praktycznych zastosowaniach jest on zazwyczaj reprezentowany przez macierz sąsiedztwa lub listę sąsiedztwa, choć istnieją również alternatywne reprezentacje, jak macierz incydencji [43]. Charakterystyczną cechą tego modelu, odróżniającą go od omawianych dalej, jest fakt, że wiedza o grafie jest pełna i dostępna w dowolnym momencie działania algorytmu. Zazwyczaj zakładamy również, że

jest on niezmienny w czasie. Jest on niewątpliwie najprostszym i jednocześnie potężnym modelem, stąd też przez dekady to na nim opierały się badania w zakresie analizy grafów. Jednak zapamiętanie całego grafu wiąże się ze sporym narzutem pamięciowym, dla macierzy i listy sąsiedztwa odpowiednio rzędu $O(|V|^2)$ i $O(|E|)$. W świecie ogromnych grafów, gdzie rozmiary analizowanych zbiorów krawędzi mogą sięgać rzędu miliardów, taka złożoność może być nieakceptowalna.

1.2.2. Strumień grafowy

W odpowiedzi na charakterystykę problemu przetwarzania ogromnych zbiorów danych powstał model strumieniowy. Graf jest w nim reprezentowany przez strumień krawędzi, napływających stopniowo. Zakładamy, że zapisanie tego grafu w klasyczny sposób jest niepraktyczne lub niemożliwe ze względu na ograniczoną pamięć. Algorytmy oparte na tym modelu powinny więc działać on-line, na bieżąco aktualizując swój stan i będąc gotowe na obsługę zapytań w dowolnym momencie. Z uwagi na rozmiar, dynamikę i często nieznaną charakterystykę danych, takie metody muszą często pomijać niektóre, mniej istotne w danym kontekście informacje o grafie, ograniczając się do tych kluczowych. W związku z tym często dopuszcza się przybliżone odpowiedzi na zapytania, jednak najlepiej z rozsądnym ograniczeniem na możliwy błąd.

Dodatkowo możemy wyróżnić kilka podkategorii w ramach strumieni danych grafowych. Jeden z najważniejszych podziałów dotyczy, tego, jakiego typu zmiany mogą zachodzić w strukturze grafu. Z uwagi na tę kwestię będziemy wyróżniać dwa typy grafów. Graf nazwiemy statycznym, jeśli do grafu krawędzie są jedynie dodawane i raz ustanowione, nigdy nie znikną. W uproszczeniu możemy założyć, że graf, który badamy, jest stały i niezmienny, ale o kolejnych krawędziach dowiadujemy się stopniowo, gdy pojawiają się one w strumieniu. Z kolei grafy dynamiczne to takie, które dopuszczają szerszą gamę operacji, przede wszystkim usuwanie wcześniej istniejących krawędzi. Może to być przydatne przy reprezentowaniu szybko zmieniających się zbiorów danych, takich jak np. informacje o ruchu samochodowym czy podejrzanych aktywnościach na kontach bankowych. Strumienie grafowe będą głównym modelem rozważanym w ramach niniejszej pracy.

Definicja formalna

Niech $G = (V, E)$ - graf. Strumieniem grafowym nazywamy ciągłą sekwencję elementów, z których każdy ma postać trójki

$$e_i = (< s_i, d_i >; w_i, t_i),$$

gdzie s_i, d_i wierzchołki grafu G i przez parę $< s_i, d_i >$ oznaczamy krawędź pomiędzy nimi. Z kolei w_i i t_i to odpowiednio waga tej krawędzi i moment jej wystąpienia. Określona krawędź może powtarzać się w różnych momentach i z różnymi wagami. Zazwyczaj przyjmujemy, że wagi kolejnych wystąpień krawędzi są akumulowane. W literaturze można również spotkać nieco inne definicje, głównie różniące się dokładną postacią strumieniowanej krotki np. dla grafów dynamicznych może ona przybrać postać czwórki

$$e_i = (< s_i, d_i >; w_i, t_i, op),$$

gdzie $op \in \{+, -\}$ indykuje typ operacji, a więc czy dana krawędź jest dodawana, czy usuwana z grafu[32].

JL:cytowanie się nie pokazuje

1.2.3. Model półstrumieniowy

Model półstrumieniowy[10] (ang. *semi-streaming model*) różni się modelu strumieniowego w dwóch głównych kwestiach. Po pierwsze, narzuca on konkretne ograniczenia na pamięć

PP:U mnie pokazuje się normalnie, być może była to kwestia brakującego pliku na githubie, poprawiłem

wykorzystywaną przez algorytm, najczęściej $O(|V|polylog(|V|))$, a więc dla gęstych grafów znacznie mniejszą niż rozmiar grafu. Po drugie, wejście może być skanowane wielokrotnie, zwykle stałą lub logarytmiczną liczbę razy. Model ten można uznać więc za rodzaj pomostu między klasyczną analizą grafów, w których dane znane są od początku i nie istnieją ograniczenia na dostęp do nich, a modelem strumieniowym, który nie pozwala na wielokrotne przeglądanie wcześniejszych krawędzi. Model ten jest często wybierany przez badaczy analizujących konkretne, złożone problemy grafowe takie, jak np. wyznaczanie najkrótszych ścieżek [9] lub minimalnego drzewa rozpinającego [1] przy rygorystycznych ograniczeniach pamięciowych. Podobnie jak w przypadku strumieni grafowych, możemy w tym modelu rozważać grafy statyczne i dynamiczne.

1.2.4. Model rozproszony

W wielu praktycznych zastosowaniach, takich jak analiza sieci społecznościowych, dane napływają z różnych źródeł – np. serwerów rozsianych po świecie i obsługujących różne obszary. Kolejne paczki danych są często relatywnie niezależne od siebie i mogą być rozpatrywane oddzielnie. W takich przypadkach wygodnie jest rozważać model rozproszony analizy grafów. W tym modelu dane są dzielone pomiędzy wiele węzłów obliczeniowych. Takie podejście umożliwia przetwarzanie równoległe, skracając czas obliczeń i ograniczając wielkość przesyłanych danych. Większość obliczeń jest wykonywana lokalnie, bez konieczności angażowania jednej centralnej jednostki. Komunikacja między węzłami ogranicza się do niezbędnych w danym przypadku aktualizacji, zamiast obejmować wszystkie dane. Trzeba jednak pamiętać o wyzwaniach wynikających z często niepełnej wiedzy węzłów, która może utrudniać rozwiązywanie bardziej złożonych problemów. Obszar rozproszonej analizy grafów znalazł szerokie zastosowania w praktyce, czego dobrym przykładem są zaawansowane platformy ułatwiające pracę w tym modelu, takie jak Google Pregel[28], czy Apache Spark GraphX[46].

Rozdział 2

Przegląd literatury

Analiza wielkich grafów, zwłaszcza w ostatnich latach, przeżywa ogromny rozwój i budzi przy tym zainteresowanie grup badaczy z całego świata. Postępy w tej dziedzinie są naturalną odpowiedzią na potrzebę przetwarzania coraz większych zbiorów danych. Badanie interakcji w sieciach społecznościowych, zarządzanie ruchem internetowym, czy monitorowanie ruchu samochodowego to tylko niektóre z kluczowych w dzisiejszej rzeczywistości zastosowań. Grafy dobrze sprawdzają się jako modele do reprezentowania złożonych relacji między encjami, dzięki czemu odgrywają kluczową rolę w przetwarzaniu i wydobywaniu skondensowanych informacji ze strumieniowanych danych. Wybrane do tych celów algorytmy i metodologie w znacznym stopniu zależą od struktury badanych grafów, a także charakteru zapytań, które chcemy rozpatrywać. W zależności od wymagań dotyczących złożoności czasowej i pamięciowej, dokładności odpowiedzi, a także konkretnych informacji, na których zachowaniu nam zależy, inne metody mogą okazać się najlepszym wyborem. Przykładowo, odpowiedź na pytania o najkrótsze ścieżki między wierzchołkami może wymagać zapamiętania dodatkowych informacji o strukturze grafu, a więc potencjalnie użycia bardziej wyrafinowanego podejścia niż w przypadku zapytań wyłącznie o istnienie danej krawędzi.

W niniejszym przeglądzie literatury zagłębiamy się w sferę analizy dużych grafów ze szczególnym uwzględnieniem metod opartych na szkicach danych. Badamy ewoluujący krajobraz technik, algorytmów i aplikacji w tej dziedzinie, rzucając światło na metodologie stosowane w celu sprostania nieodłącznym wyzwaniom stawianym przez strumieniowe przesyłanie danych grafowych. Poprzez analizę najnowszych osiągnięć, staramy się zapewnić wgląd w znaczenie i potencjał analizy strumieni grafów w rozwiązywaniu złożonych zadań analitycznych w różnych dziedzinach, a także sformułować ogólne wnioski i wskazówki co do wyboru odpowiedniej metody do danego zastosowania. Dla lepszego ustrukturyzowania wiedzy omawiane algorytmy i struktury podzielone zostały na kilka kategorii, odpisanych dokładnie w dalszej części niniejszego rozdziału. Należy jednak pamiętać, że w niektórych przypadkach podział ten jest nieco umowny, gdyż różne podejścia i pomysły nierzadko przenikają się i inspirują wzajemnie, prowadząc do syntetycznych rozwiązań.

2.1. Szkice danych

Analiza strumieni danych jest szeroką dziedziną, nieograniczającą się oczywiście wyłącznie do danych grafowych. Istnieje wiele bardziej ogólnych, uniwersalnych metod, na których podstawie można budować rozwiązania bardziej wyspecjalizowane do konkretnych zadań. Doskonałym przykładem są szkice danych (ang. *sketch synopses*). Są to kompaktowe struktury zaprojektowane z myślą o wykorzystaniu ograniczonej ilości pamięci, umożliwiając jednocześnie aproksymację różnych statystyk i zapytań dotyczących strumienia danych,

takich jak zliczanie elementów, wyznaczanie mediany, czy wykrywanie wartości odstających. Utrzymując mały szkic o stałym rozmiarze, struktury te umożliwiają analizę strumieni danych w czasie rzeczywistym bez konieczności przechowywania wszystkich elementów strumienia, co zapewnia wysoką wydajność nawet przy przetwarzaniu ogromnych strumieni.

Istotną w kontekście budowy dalszych rozwiązań strukturą danych jest szkic Count-Min[7]. W przeciwieństwie do wielu proponowanych wcześniej szkiców zaprojektowanych do badania konkretnej statystyki, stanowi on dość uniwersalną strukturę, oferującą wsparcie dla kilku fundamentalnych zapytań. Konkretnie, wyrażając strumień danych jako $a = [a_1, a_2, \ldots, a_n]$, gdzie a_i oznacza liczbę wystąpień elementu i w strumieniu, Count-Min może zwrócić aproksymacje następujących wartości:

- $\mathcal{Q}(i) = a_i$ – częstość występowania pojedynczego elementu
- $\mathcal{Q}(l, r) = \sum_{i=l}^{r} a_i$ – łączna liczba wystąpień zakresu elementów
- $\mathcal{Q}(a, b) = a \cdot b = \sum_{i=1}^{n} a_i b_i$ – iloczyn skalarny dwóch wektorów.

Z kolei na ich podstawie można budować bardziej skomplikowane zapytania. Co ważne, Count-Min, choć zwraca przybliżone wyniki, to gwarantuje spełnienie pewnych założeń odnośnie dokładności. Konkretnie, wyraża się ją zazwyczaj w kontekście definiowanych przez użytkownika parametrów ϵ, δ, a więc wymagamy, aby błąd względny w odpowiedzi na zapytanie mieścił się w zakresie współczynnika ϵ z prawdopodobieństwem δ. Count-Min opiera się na zastosowaniu dwuwymiarowej tablicy o wymiarach $w \times d$, gdzie $w = \lceil \frac{e}{\epsilon} \rceil$ oraz $d = \lceil \ln \frac{1}{\delta} \rceil$. Nietrudno zauważyć, że zależą one od wymaganej dokładności. Struktura dalej wykorzystuje d niezależnych funkcji haszujących, mapujących elementy ze strumienia na kolumny tablicy. Każdy z napływających elementów wiąże się więc z aktualizacją jednej komórki w każdym wierszu tablicy. Wtedy, przykładowo częstotliwość elementu możemy aproksymować jako minimum z wartości odpowiadających mu komórek.

Do innych wartych wzmianki metod możemy zaliczyć np. Lossy Counting[29], ukierunkowany na wskazywanie szczególnie często występujących elementów. Z kolei AMS[2] skupia się na aproksymowaniu momentów częstotliwości (ang. *frequency moments*), a więc wartości postaci:

$$F_k = \sum_{i=1}^{n} a_i^k,$$

gdzie n jest liczbą unikalnych elementów, a_i jest liczbą wystąpień elementu i w strumieniu, a k jest numerem danego momentu. Momenty te niosą ze sobą istotne informacje statystyczne o przetwarzanych danych. Przykładowo F_0 to po prostu liczba unikalnych elementów, F_1 wyznacza długość strumienia, a F_2 można traktować jako miarę powtarzalności danych (ang. *repeat rate*)[11].

gSketch[51] stanowi jedną z pierwszych prób przeniesienia idei tych ogólnych metod do świata grafów. Obsługuje on proste zapytania dotyczących grafu, takie jak częstość występowania danej krawędzi w strumieniu lub gęstość wybranego podgrafu. Autorzy wychodzą od metody Count-Min, wykorzystując jej dwuwymiarową tablicę do składowania częstotliwości krawędzi. Dodatkowo, algorytm wykorzystuje próbkę testową do podziału zbioru krawędzi na podzbiory, bazując na ich częstotliwości w taki sposób, aby efektywnie wykorzystać dostępną pamięć, Następnie przetwarzany jest właściwy strumień danych. To przetwarzanie wstępne daje cenny wgląd w charakterystykę grafu, ale należy pamiętać, że wyznaczanie reprezentacyjnej dla danego zbioru próbki testowej nie zawsze jest trywialne. Szczególnie w kontekście strumieniowanych danych grafowych, jakiekolwiek istotne informacje o grafie mogą być niedostępne w momencie rozpoczęcia jego przetwarzania. Z kolei wybór podzbioru początkowych krawędzi ze strumienia jako próbki może nie reprezentować całościowej charakterystyki grafu. gSketch, podobnie jak Count-Min jest metodą stratną, a praktycznym

wyzwaniem jest odpowiednie wyważenie parametrów tak, aby zachować balans między zużytą pamięcią a dokładnością wyników.

Choć metody takie jak gSketch mogą efektywnie zapamiętywać informacje dotyczące częstotliwości występowania krawędzi, to tracą przy tym wiedzę o strukturze grafu. Przykładowo, trudno za ich pomocą odpowiedzieć, czy istnieje ścieżka między danymi dwoma wierzchołkami. Jedną z prób rozwiązania tego problemu jest struktura gMatrix[19]. Wykorzystuje ona, podobnie jak gSketch, zasadę działania Min-Count. Jednak w tym wypadku tablica zliczająca elementy wzbogaciła się o trzeci wymiar. Konkretnie, długość i szerokość tablicy odpowiadają wartościom funkcji haszującej dla wierzchołków, a jej głębokość związana jest ponownie z liczbą funkcji haszujących. Struktura może więc wspierać zapytania związane zarówno z krawędziami, jak i wierzchołkami. gMatrix wspiera również wykrywanie szczególnie często występujących krawędzi i wierzchołków, a więc tych, których częstotliwość przekracza dany parametr F. Wymaga to jednak, aby wybrane funkcje haszujące były odwracalne. Wtedy wystarczy wybrać komórki o odpowiednio dużych wartościach i obliczyć ich wartości odwrotne, aby odzyskać informacje o wierzchołkach. Podobnie, można rozważać osiągalność między wierzchołkami, wybierając krawędzie o częstotliwości występowania przekraczającej pewne F i przetwarzając je używając tradycyjnych algorytmów. Zwłaszcza w przypadku grafów o nierównej gęstości, możemy w ten sposób znacznie ograniczyć liczbę badanych krawędzi, zachowując wciąż odpowiednie ograniczenia na prawdopodobieństwo błędu.

2.2. MDL – minimalna długość opisu

Kluczowym problemem w analizie wielkich grafów jest rozmiar danych. Zasadne wydaje się więc pytanie, czy sposób zapisu analizowanych grafów jest efektywny. W wielu przypadkach może się okazać, że sama próba zmiany modelu opisującego dane przynosi znaczne oszczędności w kwestii wykorzystanej pamięci. Metoda minimalnej długości opisu (MDL - ang. *Minimum Description Length*) koncentruje się na znalezieniu prostego modelu, który najlepiej opisuje strukturę grafu. Techniki oparte na MDL identyfikują wzorce i kompresują graf, wybierając model, który minimalizuje całkowitą długość opisu modelu i danych, ułatwiając w ten sposób wydajne przechowywanie, przesyłanie i analizę dużych grafów. Bardziej formalnie, dla danych D i rodziny dostępnych modeli MF szukamy takiego modelu $M \in MF$, który minimalizuje $L(M) + L(D|M)$, gdzie $L(M)$ i $L(D|M)$ oznaczają odpowiednio długość opisu modelu M oraz zakodowanych w nim danych D. Wiele metod opartych na MDL kompresuje dane w sposób bezstratny, co jest niewątpliwą zaletą tego modelu.

Istnieje wiele bezstratnych metod kompresujących grafy do reprezentacji o mniejszym narzucie pamięciowym. Jednak większość z nich zakładała działanie na tradycyjnej postaci grafu, gdzie dane są skończone i znane na wejściu. Rzeczywistość analizy strumieni grafowych wymaga jednak bardziej elastycznego podejścia. Jedną z pierwszych inkrementacyjnych metod kompresji grafu jest MoSSo[21]. Reprezentacja wyjściowa tej metody składa się ze zbioru superwęzłów, a więc zbiorów wierzchołków oraz superkrawędzi. Każda taka superkrawędź oznacza połączenie wszystkich wierzchołków z danego superwęzła z wierzchołkami drugiego superwęzła. Dodatkowo, częścią zapisu jest także zbiór korekt krawędzi. Mają one postać pary zbiorów $C = (C^+, C^-)$, oznaczających krawędzie, które należy dodać i usunąć, aby otrzymać prawdziwe dane. W ogólności aktualizowanie struktury dla nowych krawędzi sprowadza się do przemieszczania wierzchołków między superwęzłami w taki sposób, aby zminimalizować długość zapisu. Oczywiście sprawdzenie wszystkich możliwości byłoby kosztowne, dlatego autorzy zakładają sprawdzanie za każdym razem pewnego losowego zbioru potencjalnych wierzchołków do przemieszczania, co pozwala na znaczną oszczędność czasu, przy zachowaniu zadowalającego zużycia pamięci.

W opisanym wyżej MoSSo losowy wybór potencjalnych zmian jest sterowany parametrami takimi jak prawdopodobieństwo utworzenia nowego superwęzła i liczba wierzchołków, których przesunięcie należy rozważyć w każdej rundzie. Jeśli wybrane wartości parametrów są zbyt małe lub zbyt wielkie, może to negatywnie wpływać na czas działania lub skuteczność kompresji. Jednak optymalne ich dobranie dla nieznanych wcześniej danych może być niemożliwe. Problem ten zauważają autorzy SGS[27]. Jest to metoda bezparametryczna, w każdym kroku rozważająca przemieszczenie jedynie wierzchołków indukujących obecnie rozważaną krawędź (u, v). Dodatkowo SGS opiera się na obserwacji, że w takim wypadku wystarczy rozważyć superwęzły bazując na wierzchołkach znajdujących się w odległości co najwyżej dwóch kroków od u lub v. Następnie najlepsza zmiana jest wyznaczana zachłannie, bazując na funkcji podobieństwa sąsiedztwa wierzchołka oraz superwęzła.

Przykładem nieco odmiennego podejścia może być z kolei metoda GS4[3]. O ile ona również wykorzystuje koncept grupowania wierzchołków w superwęzły, o tyle robi to, bazując zarówno na strukturze grafu, jak i pewnym zbiorze atrybutów wierzchołków. Jest to przydatne w przypadku, gdy wierzchołki grafu reprezentują bardziej złożone informacje. Przykładowo, w przypadku analizy danych pochodzącychc z portali społecznościowych, wierzchołkiem może być użytkownik, a atrybutami jego imię, wiek, ulubione zwierzę z gromady stułbiopławów itd. Niektóre z tych danych mogą być ważniejsze niż inne. GS4 pozwala na ustalenie wag dla atrybutów, dzięki czemu jest rozwiązaniem bardziej elastycznym, pozwalającym na sterowanie tym, jakie cechy grafu będą zachowywane priorytetowo. W przeciwieństwie do omawianych wcześniej algorytmów GS4 jest jednak metodą stratną. Dla oszczędności czasu przechowywany graf nie jest aktualizowany dla każdej nowej krawędzi, a raczej w wypadku istotnych różnic.

2.3. Metody oparte na modyfikacji macierzy sąsiedztwa

Jednym z najbardziej popularnych i być może najprostszym koncepcyjnie sposobem reprezentacji grafu jest macierz sąsiedztwa. Jej wiersze i kolumny odpowiadają poszczególnym wierzchołkom, a w komórkach przechowywane są wagi krawędzi pomiędzy nimi, o ile takowe krawędzie istnieją. W przypadku grafów nieważonych może to być np. wartość logiczna indykująca istnienie krawędzi lub ustalona stała. Ten sposób reprezentacji ma niewątpliwe zalety takie jak prostota implementacji i, przede wszystkim, stały czas dostępu do wag krawędzi. Z tego powodu niezaskakujący jest pomysł zachowania ogólnej zasady działania macierzy sąsiedztwa, przy jednoczesnej próbie zmniejszenia jej rozmiaru.

Jedną z pierwszych realizacji tej idei jest struktura TCM[38]. Ma ona postać macierzy o boku długości m, gdzie m jest pewną stałą. Podobnie jak w klasycznej macierzy sąsiedztwa, w jej komórkach składowane są wagi krawędzi. Zasadniczą różnicą jest natomiast sposób wyboru rzędu i kolumny odpowiadających danej parze wierzchołków. Są one bowiem wyznaczane przez wynik funkcji haszującej $H : V \to [1..m]$. Czas obliczania funkcji haszującej jest stały, a co za tym idzie, złożoność czasowa zapytań i dodawania nowych krawędzi również. Teoretyczna złożoność pamięciowa także jest stała i wynosi $O(m^2)$. W praktycznych zastosowaniach wybór m zależy jednak często od liczby krawędzi i przejmuje się najczęściej m rzędu $O(\sqrt{|V|})$. Dokładność rezultatów zależy od rozmiaru macierzy i może być niska ze względu na kolizje wartości funkcji haszującej. Łatwo zauważyć, że jeśli m jest istotnie mniejsze od $|V|$ to może do nich dochodzić często, co powoduje traktowanie różnych krawędzi jako kolejnych instancji tego samego połączenia. Autorzy, świadomi tego ograniczenia, proponują zastosowanie kilku parami niezależnych funkcji haszujących i stworzenie na ich podstawie wielu szkiców grafu. Przykładowo, jeśli badaną zmienną jest suma wag kolejnych instancji krawędzi między danymi dwoma wierzchołkami, to algorytm może sprawdzić odpowiednie komórki dla wszystkich szkiców, a następnie zwrócić minimalną wartość. Podejście to pozwala na analizę większych

grafów niż w przypadku pojedynczego szkicu, ale ostatecznie nie rozwiązuje całkowicie problemu. Użyteczność struktury TCM w bazowej formie jest dyskusyjna, stanowi ona jednak punkt wyjściowy dla bardziej zaawansowanych rozwiązań.

Strukturą opartą na koncepcie podobnym do TCM jest *Graph Stream Sketch* (GSS)[12]. Celem autorów było stworzenie metody oferującej lepszą skalowalność dla wielkich grafów. Podobnie jak w TCM, funkcja haszująca mapuje zbiór wierzchołków na pewien mniejszy zbiór M-elementowy. Wymiary skompresowanej macierzy sąsiedztwa wynoszą notomiast $m \times m$, $m < M$. Główną zmianą względem TCM jest wprowadzenie dodatkowych cech opisujących wierzchołki. Pierwszą z nich jest podpis $f(v), 0 \leq f(v) < F$ wierzchołka v, gdzie F jest stałą wyznaczającą maksymalną wartość liczbową podpisu, przy czym F i m są dobierane tak, aby spełnić $M = m \times F$. Dla każdego wierzchołka, jego podpis jest wyznaczany na podstawie wartości funkcji haszującej $H(v)$ jako:

$$f(v) = H(v) \bmod F.$$

W podobny sposób definiowany jest także adres wierzchołka. Ma on postać:

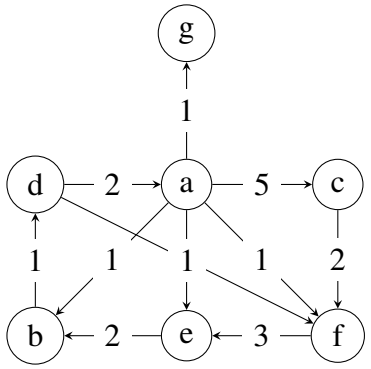
$$h(v) = \left\lfloor \frac{H(v)}{F} \right\rfloor.$$

Przykładowy graf wraz z podpisami oraz adresami wierzchołków przedstawiono na rysunku 2.1. Adresy służą do wyznaczania rzędu i kolumny komórek. Komórki te mają postać krotki lub, bardziej obrazowo, kubełka, w którym przechowywana jest para podpisów wierzchołków tworzących krawędź oraz kumulatywna waga krawędzi. Przechowywanie podpisów w komórkach pozwala zredukować ryzyko scalenia ze sobą krawędzi między różnymi parami wierzchołków ze względu na kolizje. Łatwo bowiem zauważyć, że nawet jeśli dwa różne wierzchołki mają taki sam adres, to prawdopodobieństwo, że ich podpisy również są identyczne, jest niewielkie. Z tego względu nowa krawędź jest dodawana do kubełka tylko w wypadku, gdy jest on pusty lub gdy istniejące w nim podpisy są zgodne z podpisami wierzchołków krawędź tą tworzących. W przeciwnym przypadku jest ona zapisywana w dodatkowym buforze, mającym postać listy sąsiedztwa pełnych wartości funkcji haszującej. Pozwala on na dodawanie nowych krawędzi z niskim ryzykiem kolizji, nawet jeśli sama macierz jest już zapełniona. Przykład częściowo zapełnionej struktury GSS wraz z buforem przedstawia rysunek 2.2. Należy natomiast zauważyć, że część macierzowa struktury jest bardziej efektywna czasowo, oferując stały czas odpowiedzi na zapytanie, podczas gdy dla bufora jest on liniowy względem liczby wierzchołków. Dokładność odpowiedzi w części macierzowej zależy od długości podpisów. Potencjalnym problemem GSS jest niskie wykorzystanie pamięci w macierzy. Przy kolizji adresów nowe krawędzie mogą trafiać do bufora, mimo, że w samej macierzy pozostaje wiele pustych komórek. Aby temu zaradzić, autorzy proponują haszowanie krzyżowe (ang. *square-hashing*). Zakłada ono obliczanie dla każdego wierzchołka sekwencji niezależnych adresów. Podczas wstawiania nowych krawędzi algorytm sprawdza nie jedną komórkę macierzy, a kilka, zgodnie z sekwencją adresów i wybiera pierwszą spełniającą wymagania co do zgodności podpisów. Autorzy przedstawiają probabilistyczne ograniczenie na błąd względny zapytań postaci:

$$Pr(\tilde{f}(s,d) - f(s,d)/\bar{w} > \delta) \leq \frac{|E|}{\delta m^2 4^f},$$

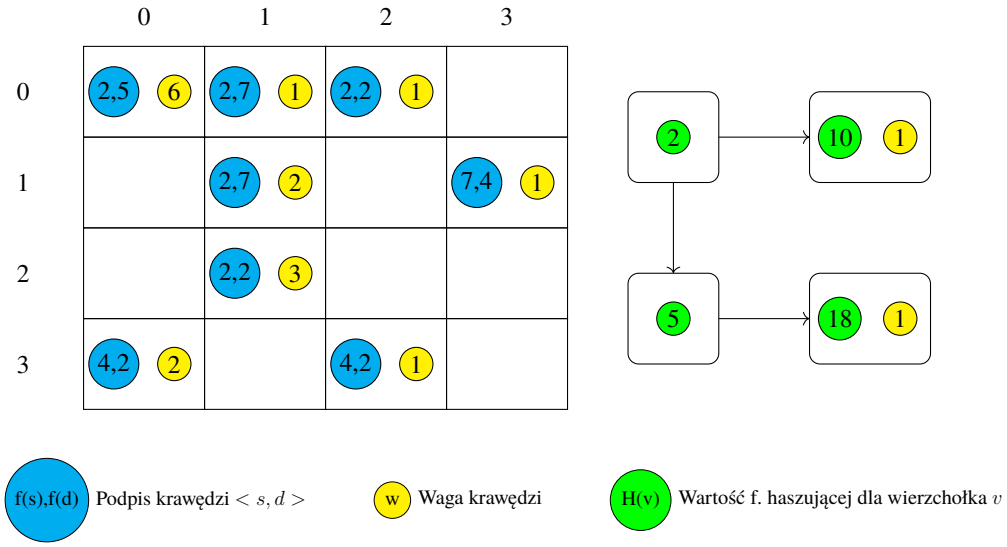
gdzie $\tilde{f}(s,d)$ jest zwróconą sumą wag krawędzi (s,d), $f(s,d)$ jej rzeczywistą wartością, $\bar{w}$ średnią wagą krawędzi, a f długością podpisu.

Większość struktur służących analizie strumieniowanych grafów nie przechowuje informacji o czasie wystąpienia krawędzi. Nie wspierają one więc zapytań z zakresem czasowym, a więc np., czy dana krawędź wystąpiła w zakresie $[t, t + L)$. Tego typu zapytania mogą być kluczowe



Node	a	b	c	d	e	f	g
H(v)	2	15	5	28	10	18	5
f(v)	2	7	5	4	2	2	5
h(v)	0	1	0	3	1	2	0

Rys. 2.1: Graf wraz z odpowiadającymi wierzchołkom wartościami haszy, podpisami oraz adresami. W tym wypadku ustalono $F = 8$, $m = 4$ oraz $M = 32$. Przykład został zaczerpnięty z [12].



Rys. 2.2: Struktura GSS dla grafu przedstawionego na rysunku 2.1. Skompresowana macierz sąsiedztwa składa się z kubełków zawierających podpisy wierzchołków tworzących krawędź oraz łączną wagę krawędzi pomiędzy nimi. Dodatkowy bufor przechowuje krawędzie, które nie mogły zostać wstawione do macierzy. Przykładowo, informacja o krawędzi $< a, b >$ jest zawarta w komórce $M_{0,1}$, gdyż adresy wierzchołków a i b to odpowiednio 0 i 1. Krawędź $< a, e >$ ma ten sam adres, ale inny podpis, dlatego została ona przekierowana do bufora.

np. w przypadku analizy danych dotyczących rozprzestrzeniania się wirusów. Problem ten podejmuje praca proponująca strukturę Horae[6]. W jej wypadku krawędź:

$$e_i = (< s_i, d_i >, w_i, t_i)$$

jest wstawiana do komórki o adresie:

$$(h(s_i|\gamma(t_i)), h(d_i|\gamma(t_i))),$$

gdzie $\gamma(t_i) = \lfloor \frac{t_i}{gl} \rfloor$ i gl jest długością przedziałów czasowych. Intuicyjnie, zapytanie o pojawienie się krawędzi w zakresie czasowym $[T_b, T_e]$ może być transformowane w sekwencję zapytań o pojedyncze zakresy, których wyniki są sumowane, a więc $Q([T_b, T_e]) = Q([T_b]), Q([T_{b+1}]), \ldots Q([T_e])$. Jednak dla takiego algorytmu złożoność czasowa jest liniowa względem liczby zakresów. Autorzy starają sie poprawić ten aspekt, zauważając, iż przedział długości L może zostać zdekomponowany do co najwyżej $2 \log L$ podprzedziałów podsiadających dwie szczególne cechy. Po pierwsze, wszystkie zakresy czasowe w danym podprzedziale mają wspólny prefiks binarny. Po drugie, prefiksy różnych podprzedziałów mają

różne długości. Z tego względu Horae zapamiętuje $O(\log(T))$ identycznych skompresowanych macierzy, gdzie T jest liczbą rozróżnialnych zakresów czasowych. Każda z nich jest utożsamiana z jedną warstwą struktury. Warstwy odpowiadają z kolei różnym długościom prefiksów. Dzięki temu zamiast wykonywać liniową względem długości przedziału czasowego liczbę zapytań, wystarczy zdekomponować przedział na podprzedziały i na ich podstawie wykonać co najwyżej jedno zapytanie na warstwę.

Metody oparte na macierzach w większości przypadków nie czynią założeń co do struktury grafu. Takie ogólne podejście oczywiście zapewnia wysoką uniwersalność, jednak w niektórych przypadkach może być nieefektywne. Przykładowo, jeśli wierzchołki w grafie są mocno zróżnicowane pod względem stopnia, a więc bardziej obrazowo, da się wyróżnić obszary gęste i rzadkie w grafie, to kolizje haszy mogą zdarzać się często. Struktura Scube[5] używa probabilistycznego zliczania do identyfikacji wierzchołków wysokiego stopnia. Przeznaczane jest dla nich więcej kubełków w macierzy niż dla wierzchołków o niskich stopniach, co pozwala bardziej efektywnie zarządzać zapełnieniem macierzy.

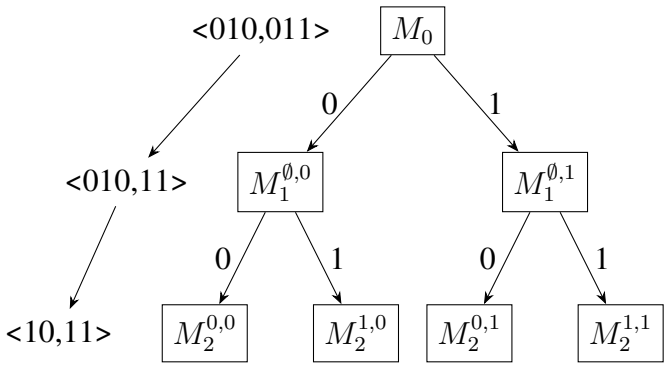
Metody takie jak GSS czy Horae, choć często dają przyzwoite wyniki przy odpowiednim dobraniu parametrów do badanego grafu, to ostatecznie cierpią z uwagi na ograniczoną skalowalność. Jedną z prób odpowiedzi na ten problem jest struktura AUXO[17]. Korzysta ona z macierzy przechowujących podpisy wierzchołków, podobnie jak GSS. Jednak zamiast wstawiać nadmiarowe krawędzie do bufora o liniowym czasie dostępu, AUXO wykorzystuje wiele macierzy ustawionych w strukturę drzewa. Konkretnie, jest to binarne lub czwórkowe drzewo prefiksowe, w którego strukturę zaszyte zostały prefiksy podpisów wierzchołków. W ten sposób na każdym kolejnym poziomie drzewa podpisy przechowywane w komórkach mogą być coraz krótsze, gdyż informacja ta jest wbudowana w kształt struktury. Pozwala to osiągnąć logarytmiczny względem liczby krawędzi czas odpowiedzi na zapytania. Rysunek 2.3 ukazuje ogólny schemat działania Auxo. Autorzy pokazują, że koszt pamięciowy struktury Auxo o l poziomach można wyrazić za pomocą wzoru:

$$M_A = m^2 b(2^l - 1) - \sum_{i=0}^{l-1} m^2 i 2^i,$$

gdzie b jest rozmiarem kubełka na zerowym poziomie drzewa. Składnik $m^2b(2^l - 1)$ odpowiada za koszt przechowywania wszystkich macierzy z pełnymi podpisami, a $\sum_{i=0}^{l-1} m^2 i 2^i$ to pamięć zaoszczędzona przy użyciu prefiksów zaszytych w strukturze drzewa. Przyjmując średni współczynnik zapełnienia macierzy α, otrzymujemy wysokość drzewa $l = \log_2 \frac{|E|}{m^2\alpha}$. Złożoność pamięciową można wtedy sprowadzić do $O(|E|(1 - \log(E)))$. Należy jednak pamiętać, że w rzeczywistości złożoność pamięciowa AUXO jest ograniczona przez długość podpisów, która wyznacza maksymalną głębokość drzewa. Dlatego podane obliczenia mają zastosowanie pod warunkiem wyboru podpisów odpowiednio długich w stosunku do liczby krawędzi. Niemniej jednak liczba możliwych do przetworzenia krawędzi jest eksponencjalna w stosunku do liczby bitów podpisu, więc w praktycznych zastosowaniach stosunkowo łatwo można dobrać wystarczające wartości. W porównaniu do TCM i GSS, AUXO osiąga lepszą efektywność pamięciową i skalowalność kosztem zwiększenia złożoności czasowej, co może być potencjalną wadą tego rozwiązania.

2.4. Znurzenia grafów

Kolejną, dość rozległą i rozwijającą się metodologią w dziedzinie analizy grafów są zanurzenia grafów (ang. *graph embeddings*). Nazwa odnosi się do reprezentowania wierzchołków przez wektory cech, a więc bardziej obrazowo, zanurzania grafu w niskowymiarowych przestrzeniach



Rys. 2.3: Drzewo prefiksowe dla struktury AUXO o 3 poziomach. W tym wypadku zastosowano drzewo binarne, gdzie prefiksy wierzchołka docelowego i startowego są wbudowywane w strukturę naprzemiennie. Obok ukazano, jak para podpisów w kubełku różni się, zależnie od poziomu, na którym dana krawędź jest przechowywana.

wektorowych. Wektory te mogą zachowywać między innymi kluczowe informacje topologiczne związane z połączeniami danego wierzchołka z jego sąsiadami. Przyjmują one najczęściej wartości rzeczywiste, a podobieństwo między wektorami może być mierzone na różne sposoby, np. za pomocą podobieństwa cosinusów (ang. *cosine similarity*) lub odległości Hamminga[26]. W tym kontekście, przez podobieństwo lub bliskość wierzchołków rozumieć będziemy ich wzajemne położenie w grafie. W szczególności, podobieństwo pierwszego rzędu zależy od liczby i czasem również wagi bezpośrednich połączeń między wierzchołkami, podczas gdy bliskość drugiego rzędu bierze pod uwagę połączenia poprzez jeden wierzchołek pośredni. Co istotne, tego typu reprezentacja pozwala na łatwe zastosowanie uczenia maszynowego do analizy danych. W ogólności zanurzenia są stosowane szczególnie często w takich zadaniach jak klasyfikacja węzłów, przewidywanie połączeń między nimi oraz rekonstrukcja grafu. Wśród algorytmów opartych na tej metodologii możemy wyróżnić kilka unikalnych podkategorii, cechujących się odmiennymi podejściami do tego, jak cechy są wybierane i przetwarzane. Praca [47] stanowi stosunkowo aktualny i rozbudowany przegląd tego typu metod, ukazując przy okazji wpływ hiperparametrów na uzyskiwane wyniki, zwłaszcza dla metod opartych na faktoryzacji i próbkowaniu. Dla tych dwóch kategorii rozwiązań, autorzy proponują też uogólnione algorytmy, których szczegółowe działanie może być łatwo regulowane za pomocą parametrów. Autorzy dzielą się również pewnymi wskazówkami co do wyboru hiperparametrów dla konkretnych scenariuszy.

2.4.1. Próbkowanie

Techniki oparte na próbkowaniu koncentrują się na wyborze reprezentatywnego zbioru par węzłów z grafu wejściowego, w celu uchwycenia jego podstawowych właściwości strukturalnych. Na ich podstawie określony model uczy się reprezentowania wierzchołków poprzez optymalizację stochastyczną, często wykorzystując SGD (*stochastic gradient descent*) do iteracyjnego poprawiania rezultatów. Funkcja straty może przybierać różne postaci, ale w ogólności powinna ona zachowywać informacje o sąsiedztwie wierzchołków. Zapewnienie zadowalającej jakości reprezentacji wyjściowej wymaga często próbkowania znacznej liczby par węzłów, a zatem również dużych zasobów obliczeniowych, w szczególności czasu procesora.

Znanym przykładem tego typu techniki jest DeepWalk[33], wywodzący się, jak sama nazwa wskazuje, z uczenia głębokiego (ang. *deep learning*). Wykorzystuje on sekwencję krótkich, losowych spacerów po wierzchołkach grafu do zapamiętywania ich tzw. reprezentacji społecznej (ang. *social representation*), zawierającej informacje o podobieństwie sąsiedztwa wierzchołków i przynależności do większych klastrów. DeepWalk wykorzystuje algorytm SkipGram[30] do

efektywnego wyznaczania wektorów cech. Co ciekawe, można na ten schemat działania patrzeć jak na uogólnienie metod przetwarzania języka naturalnego do przetwarzania języka losowych spacerów, traktowanych jako zdania.

Node2Vec[13] rozwija ideę DeepWalk, wprowadzając dodatkowe parametry sterujące spacerem. Pozwala to na wybranie, jak szybko spacer może oddalać się od startowego wierzchołka, efektywnie regulując, jak bardzo spacer zbliżony jest do przeszukiwania wszerz lub przeszukiwania wgłąb. Odpowiednie regulowanie tych parametrów sprzyja lepszemu poznaniu struktury grafu, która mogłaby być trudna do uchwycenia przy zwykłym spacerze losowym.

Kolejną wartą uwagi metodą tego typu jest LINE[37]. Opiera się ona na badaniu bliskości zarówno pierwszego, jak i drugiego rzędu między wierzchołkami. Bliskość pierwszego rzędu zależy od wagi połączeń między wierzchołkami. Z kolei bliskość drugiego rzędu mierzy podobieństwo struktury sąsiedztwa dwóch wierzchołków. Wiąże się to z założeniem, że wierzchołki o podobnym sąsiedztwie z dużym prawdopodobieństwem są podobne do siebie. LINE próbkuje bezpośrednio pary węzłów z grafu, ucząc się zanurzeń oddzielnie dla bliskości pierwszego i drugiego rzędu, a następnie łącząc je razem.

VERSE[40] jest uniwersalną metodą, zdolną dostosować się do dowolnej miary podobieństwa węzłów, co pozwala na lepsze jej dostosowanie do konkretnych przypadków. Trenuje ona sieć neuronową złożoną z jednej warstwy, próbkując pary węzłów i starając się zachować wybraną miarę podobieństwa.

2.4.2. Faktoryzacja

Metody opierające się na faktoryzacji zakładają dekompozycję reprezentacji macierzowej grafu w taki sposób, aby uchwycić ukryte relacje między węzłami w przestrzeni o niższym wymiarze. Ma to na celu zachowanie właściwości strukturalnych grafu, takich jak bliskość węzłów i struktura sąsiedztwa. Jednak koszt obliczeniowy faktoryzacji macierzy może być bardzo wysoki w przypadku dużych grafów, zarówno w kontekście wykorzystania pamięci, jak i czasu obliczeń.

Jedną z istotniejszych metod opartych na faktoryzacji jest GraRep[4]. Wyróżnia się ona uwzględnieniem nie tylko lokalnych, ale i globalnych informacji strukturalnych w zanurzeniach, co ma na celu dokładniejsze uchwycenie charakterystyki grafu. Wykorzystywany model bada sąsiedztwo wierzchołków, rozważając różnej długości ścieżki pomiędzy nimi. Czyni to poprzez manipulowanie globalnymi macierzami przejścia zdefiniowanymi na grafie, bez angażowania procesów próbkowania. Autorzy proponują naprzemienne wykorzystanie różnych wartości długości ścieżek, a także definiują funkcję straty zależną od tychże długości. Ostateczna reprezentacja dla każdego wierzchołka ma charakter globalny i powstaje z połączenia wielu wyuczonych modeli. Wykorzystanym sposobem dekompozycji jest rozkład według wartości osobliwych, szerzej znany jako SVD (*singular value decomposition*). Sprowadza się on do przedstawienia macierzy A w postaci:

$$A = U\Sigma V^T,$$

gdzie U i V - macierze ortogonalne i Σ - macierz diagonalna.

HOPE[31] działa na podobnej zasadzie, faktoryzując macierze odpowiadające sąsiadom w odległości co najwyżej k od danego wierzchołka z wykorzystaniem SVD. Autorzy biorą sobie jednak za cel zachowanie asymetrycznej przechodniości. Przedstawia ona korelacje między skierowanymi krawędziami, a więc, istnienie skierowanej ścieżki z u do v może sugerować istnienie skierowanej krawędź z u do v. Prawdopodobieństwo ich istnienia jest wyliczane na podstawie indeksu Katza[18]. Indeks ten jest definiowany jako suma ważona ścieżek między dwoma wierzchołkami. Waga ścieżki jest natomiast wykładniczą funkcją jej długości.

NetMF[35] jest rozwiązaniem syntetycznym, generalizującym metody próbkujące takie, jak DeepWalk, Node2Vec i LINE. Autorzy zauważają, że sposób działania tych metod można

przedstawić jako niejawne faktoryzacje macierzy. Na podstawie tej obserwacji wyznaczają oni jawne postaci macierzy, które są aproksymowane i faktoryzowane przez omawiane algorytmy. Dodatkowo, bazując na relacji między metodą DeepWalk i macierzy Laplace'a dla grafu (ang. *Laplacian matrix*), autorzy proponują algorytm efektywnie aproksymujący jawną postać tej macierzy.

Z kolei ProNE[50] jest metodą, składającą się z dwóch zasadniczych kroków. Pierwszy z nich polega na sprowadzeniu generowania zanurzeń wierzchołków do faktoryzacji rzadkich macierzy. W drugim kroku uzyskane wyniki są poprawiane w celu lepszego ukazania globalnych zależności w grafie. Aby to osiągnąć, ProNE stosuje nierówność Cheegera wyższego rzędu[22] do propagacji zanurzeń. Jak twierdzą autorzy, taki sposób przeprowadzania obliczeń może istotnie zwiększyć wydajność w praktycznych zastosowaniach.

2.4.3. Metody oparte na sieciach neuronowych

Większość algorytmów opisywanych w tym podrozdziale wykorzystuje sieci neuronowe w mniejszym lub większym stopniu. Jednak często nie stanowią one istoty rozwiązania. Przykładowo, mogą one być eksploatowane tylko na dalszym etapie rozwiązania, np. do samej klasyfikacji wierzchołków, podczas gdy właściwe wektory zanurzeń są generowane w inny sposób. Nic nie stoi jednak na przeszkodzie, aby i do tego zadania zaprząc moc sieci neuronowych. Istnieje wiele rozwiązań, które budują zaawansowane modele sieci, zdolne radzić sobie z danymi grafowymi, bez konieczności użycia kosztownych faktoryzacji macierzy, czy próbkowania.

Jednym z przykładów tego podejścia jest SDNE[41]. Jest to metoda wykorzystująca głęboki, wielowarstwowy model dostosowany do uczenia pół-nadzorowanego. Konkretnie, część nadzorowana bada sąsiedztwo pierwszego rzędu między wierzchołkami, ucząc się lokalnej struktury grafu. Z kolei w uczeniu nienadzorowanym, mającym na celu wychwycenie globalnych informacji, SDNE używa sąsiedztwa drugiego rzędu. Autorzy proponują również użycie autoenkodera, aby minimalizować błędy rekonstrukcji danych. Taki model pozwala na zachowanie kompleksowej struktury grafu.

DVNE[52] wyróżnia się na tle innych rozwiązań tym, że generuje zanurzenia wierzchołków w przestrzeni Wassersteina. Dokładniej, autorzy wykorzystują dystans 2-Wassersteina jako miarę podobieństwa między rozkładami prawdopodobieństwa, a więc, w tym przypadku, wygenerowanymi zanurzeniami oraz prawdziwymi danymi. Miara ta dobrze zachowuje tranzytywność oraz może modelować niepewność wierzchołków, powiązaną z wariancją rozkładu. Proces uczenia postulowany w DVNE jest uczeniem nienadzorowanym.

GCN[20] jest kolejnym przykładem uczenia pół-nadzorowanego. W tym przypadku autorzy przyglądają się głównie zadaniu klasyfikacji wierzchołków. W tym celu definiują zasadę propagacji w konwolucyjnej sieci neuronowej, pozwalającą na jej działanie bezpośrednio na macierzy sąsiedztwa. Inspiracją są w tym przypadku metody aproksymacji spektralnych splotów grafów (ang. *spectral graph convolutions*)[16].

GraphSAGE[14] wykorzystuje uczenie z nadzorem. Bierze pod uwagę nie tylko topologię grafu, ale także cechy wierzchołków. W przeciwieństwie do większości podobnych metod GraphSAGE nie generuje unikalnego wektora zanurzenia dla każdego wierzchołka. Zamiast tego, trenuje zbiór funkcji agregujących informacje o cechach z lokalnego sąsiedztwa węzła. Mogą one być potem stosowane dla zupełnie nowych wierzchołków.

2.4.4. Szkice

Szczególnie istotne z perspektywy niniejszej pracy są techniki zanurzania oparte na szkicach danych. Ich głównym założeniem jest tworzenie kompaktowych szkiców dla oryginalnych

wielowymiarowych danych, przy jednoczesnym zachowaniu pewnej miary podobieństwa między wierzchołkami. Często stosowanym pomysłem jest generowanie reprezentacji wierzchołków w przestrzeni Hamminga[15], a więc wykorzystanie kodów binarnych. Metody te opierają się na zastosowaniu zachowujących podobieństwo technik haszowania w celu generowania zanurzeń węzłów. Wykorzystywane przez nie haszowanie może być zależne albo niezależne od danych. W literaturze te warianty bywają również opisywane jako odpowiednio *learning-to-hash* i *locality sensitive hashing*[42].

INH-MF[26] to pierwsza tego typu technika zaproponowana dla problemu zanurzeń grafu. Wykorzystuje ona haszowanie zależne od danych i uczy się reprezentowania szkicu grafu poprzez faktoryzację macierzy, zachowując jednocześnie wysokopoziomowe informacje o bliskości węzłów w grafie. W celu poprawy wydajności procesu uczenia autorzy proponują także zastosowanie uczenia w podprzestrzeni Hamminga. Przy przetwarzaniu nowych krawędzi INH-MF aktualizuje tylko niektóre części wyjściowej reprezentacji.

Większość opisywanych w tym podrozdziale metod skupia się na zachowaniu informacji strukturalnych dotyczących badanego grafu. O ile w wielu scenariuszach jest to wystarczające, o tyle istnieją przypadki, gdy przydatne może się okazać bardziej elastyczne podejście do definiowania cech wierzchołków, które algorytm powinien brać pod uwagę przy wyznaczaniu zanurzeń. Naprzeciw tym wymaganiom wychodzi algorytm NetHash[44]. Korzysta on ze zrandomizowanej techniki haszowania MinHash. Za jej pomocą koduje płytkie drzewa, z których każde jest zakorzenione w wierzchołku grafu. Właściwe szkice powstają poprzez rekurencyjne szkicowanie drzewa od dołu do góry, a więc od predefiniowanych sąsiednich węzłów najwyższego rzędu do węzła głównego. W ten sposób kodowane są zarówno atrybuty, jak i informacje o strukturze każdego węzła. NetHash zachowuje w szczególności informacje zawarte bliżej węzła głównego.

GNN[36] (*Graph Neural Network*) jest popularną metodą w zadaniach związanych z użyciem uczenia maszynowego dla danych grafowych. Zakłada ona zastosowanie konwolucyjnej sieci neuronowej do pół-nadzorowanych zadań klasyfikacji węzłów, a więc takich wykorzystujących zarówno etykietowane, jak i nieetykietowane dane. Jednak jej efektywne wykorzystanie zazwyczaj wymaga znacznych zasobów obliczeniowych ze względu na konieczność uczenia się dużej liczby parametrów. Sprawia to, że algorytmy oparte na GNN mogą być niepraktyczne dla problemu przetwarzania ogromnych grafów. #GNN[45] modyfikuje GNN w celu znalezienia kompromisu między dokładnością a wydajnością. Autorzy proponują wprowadzenie zrandomizowanego haszowania do implementacji przekazywania wiadomości i przechwytywania informacji o sąsiedztwie wysokiego rzędu w sieci GNN.

NodeSketch

Ważnym i interesującym, także w kontekście niniejszej pracy, krokiem w rozwoju analizy wielkich grafów z wykorzystaniem szkiców jest algorytm NodeSketch[49]. Autorzy wykorzystują rekursywne szkicowanie niezależne od danych, aby tworzyć szkice, które dla każdego wierzchołka zachowują informacje o jego sąsiedztwie wysokiego rzędu. Bardziej konkretnie, zaczynają oni od przedstawienia idei spójnego ważonego próbkowania (ang. *consistent weighted sampling*).

Niech $V^a, V^b \in \mathbb{R}_+^D$ - nieujemne wektory D-elementowe. Wtedy ich podobieństwo min-max, zwane też ważonym indeksem Jaccarda możemy zdefiniować jako

$$Sim_{MM}(V^a, V^b) = \frac{\sum_{i=1}^{D} min(V_i^a, V_i^b)}{\sum_{i=1}^{D} max(V_i^a, V_i^b)}. \tag{2.1}$$

Możemy wobec tej wartości zastosować normalizację sum-to-one $\sum_{i=1}^{D} V_i^a = \sum_{i=1}^{D} V_i^b = 1$ i oznaczyć takie znormalizowane podobieństwo przez Sim_{NMM}. Jest to efektywny sposób mierzenia podobieństwa pomiędzy wektorami nieujemnych liczb rzeczywistych, co ukazano w [25]. Przykładowy graf wraz z macierzą SLA oraz odpowiadającą jej macierzą znormalizowanych podobieństw min-max ukazuje rysunek 2.4. Zasadniczo spójne ważone próbkowanie sprowadza się do generowania próbek danych w taki sposób, aby prawdopodobieństwo wylosowania identycznych wartości dla obu wektorów było równe ich podobieństwu min-max. Próbki te są następnie traktowane jako szkic wektora wejściowego. Proponowany proces generowania próbek dla wektora $V = [V_1, V_2, \ldots, V_D]$ jest dość prosty. Zaczyna się on od wyboru funkcji haszującej h_j, takiej, że $h_j(i) \sim Uniform(0, 1)$. Za jej pomocą próbka S_j wyznaczana jest jako:

$$S_j = \underset{i \in \{1,2,\ldots,D\}}{\arg\min} \frac{-\log h_j(i)}{V_i}. \tag{2.2}$$

Wybierając m ($m \ll D$) niezależnych funkcji haszujących i generując na ich podstawie próbki otrzymujemy szkic S o rozmiarze m wektora V. Ponadto, wynikowe szkice zachowują własność:

$$Pr[S_j^a = S_j^b] = Sim_{NMM}(V^a, V^b), j = 1, 2, \ldots, m. \tag{2.3}$$

Na podstawie tego schematu, autorzy budują właściwy algorytm NodeSketch. Jego działanie rozpoczyna się od wygenerowania zanurzeń dla wierzchołków na podstawie sąsiedztwa pierwszego i drugiego stopnia. Wykorzystuje do tego macierz sąsiedztwa SLA (*Self-Loop-Augmented*) grafu. Różni się ona od zwykłej macierzy tym, że zawiera połączenia z każdego wierzchołka do niego samego, co pozwala prawidłowo zachowywać sąsiedztwo pierwszego rzędu. Innymi słowy, jeśli A jest oryginalną macierzą sąsiedztwa, to macierz SLA ma postać:

$$\tilde{A} = I + A,$$

gdzie I jest macierzą jednostkową. zanurzenia k-tego rzędu są generowane rekurencyjnie na podstawie macierzy SLA i zanurzeń $(k-1)$-wszego rzędu. Ważną cechą spójnego ważonego próbkowania jest jednostajność generowanych próbek, co oznacza, że prawdopodobieństwo wybrania wartości i jest proporcjonalne do V_i, a więc

$$Pr(S_j = i) = \frac{V_i}{\sum_n V_n}.$$

Własności ta implikuje, że udział elementu i w szkicu S stanowi nieobciążony estymator V_i, z czego można wywnioskować, że poprzez empiryczną obserwację rozkładu elementów w szkicu można aproksymować wektor V.

Bazując na powyższych obserwacjach, autorzy przechodzą do właściwego sformułowania algorytmu. Dla każdego węzła r, obliczany jest przybliżony wektor SLA k-tego rzędu. Odbywa się to poprzez połączenie oryginalnego wektora SLA z wektorem (k-1)-wszego rzędu z odpowiednią wagą:

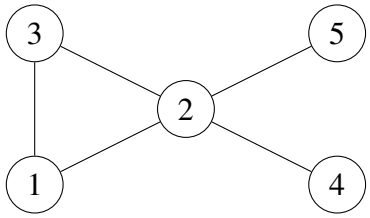
$$\tilde{V}_i^r(k) = \tilde{V}_i^r + \sum_{n \in \Gamma(r)} \frac{\alpha}{m} \sum_{j=1}^{m} \mathbb{1}_{[S_j^n(k-1)=i]}, \tag{2.4}$$

gdzie $\Gamma(r)$ to zbiór sąsiadów wierzchołka r, a $S^n(k-1)$ jest szkicem $(k-1)$-wszego rzędu dla wierzchołka n. Dodatkowo parametr α steruje tym, jaki wpływ ma na wagę połączenia między wierzchołkami ma rząd ich sąsiedztwa, czyli liczba wierzchołków pośrednich. Dzięki temu krótkie połączenia są traktowane z wyższym priorytetem niż te długości bliskiej k. Następnie,

szkic wierzchołka r, oznaczany jako $S^r(k)$ jest wyznaczany z równania 2.2. Ostatecznym rezultatem wykonania algorytmu jest szkic

$$S(k) = [S^1(k), S^2(k), \ldots, S^D(k)]^T, S(k) \in \mathbb{R}_+^{D \times m}.$$

Działanie algorytmu ukazuje pseudokod 1.



1	1	1	0	0
1	1	1	1	1
1	1	1	0	0
0	1	0	1	0
0	1	0	0	1

1.0	0.43	1.0	0.2	0.2
0.43	1.0	0.43	0.25	0.25
1.0	0.43	1.0	0.2	0.2
0.2	0.25	0.2	1.0	0.33
0.2	0.25	0.2	0.33	1.0

Rys. 2.4: Przykładowy graf wraz z odpowiadającą mu macierzą SLA oraz macierzą wartości podobieństw min-max, obliczonych przy pomocy równania 2.1. Można spostrzec, że np. dla wierzchołka 4, jego sąsiedztwo jest najbardziej zbliżone do sąsiedztwa wierzchołka 5, czemu odpowiada wartość 0.33 podobieństwa min-max. Warto też zauważyć, że $Sim_{NMM}(V, V)$, czyli podobieństwo wierzchołka do siebie samego jest zawsze równe 1.

Algorytm 1: NodeSketch($\tilde{A}, k, \alpha$)

if $k > 2$ **then**
  $S(k-1) \leftarrow NodeSketch(\tilde{A}, k-1, \alpha)$;
  foreach rząd r w $\tilde{A}$ **do**
    $\tilde{V}_i^r(k) \leftarrow \tilde{V}_i^r + \sum_{n \in \Gamma(r)} \frac{\alpha}{m} \sum_{j=1}^{m} \mathbb{1}_{[S_j^n(k-1)=i]}, i \in \{1, 2, \ldots, D\}$;
    $S_j^r(k) \leftarrow \arg\min_{i \in \{1,2,\ldots,D\}} \frac{-\log h_j(i)}{\tilde{V}_i^r(k)}, j \in \{1, 2, \ldots, m\}$;
else if $k = 2$ **then**
  foreach rząd r w $\tilde{A}$ **do**
    $S_j^r(2) \leftarrow \arg\min_{i \in \{1,2,\ldots,D\}} \frac{-\log h_j(i)}{\tilde{V}_i^r(2)}, j \in \{1, 2, \ldots, m\}$;
return $S(k)$

SGSketch

Choć NodeSketch okazuje się efektywny w wielu scenariuszach testowych, to nie da się ukryć, że jego możliwości, podobnie jak wielu omawianych wcześniej metod generowania zanurzeń wierzchołków, są ograniczone. Po pierwsze, nie bierze on pod uwagę wag krawędzi, a tylko ich istnienie. Tymczasem waga połączenia może nieść ze sobą cenne informacje. Po drugie, zakłada on, że cała macierz sąsiedztwa SLA jest znana w chwili wywołania algorytmu. Fakt ten sprawia, że NodeSketch nie jest przystosowany do działania w formie On-Line, a więc reagowania na zmiany w grafie, jak np. ustanawianie nowych krawędzi. Ta obserwacja, choć z pozoru nieco rozczarowująca, może stanowić inspirację do budowy kolejnych, bardziej elastycznych rozwiązań.

Jedną z prób stworzenia metody mogącej radzić sobie z ewoluującym strumieniem krawędzi jest, oparty na NodeSketch, algorytm SGSketch[48]. Jego główna idea polega na stopniowym zapominaniu dawnych krawędzi i wykorzystaniu efektywnego, inkrementacyjnego mechanizmu aktualizacji wygenerowanych zanurzeń.

Niech SG będzie strumieniowanym grafem. Typowa implementacja stopniowego zapominania krawędzi polega na obliczaniu jej wagi bazując na momencie wystąpienia w strumieniu. Starszym krawędziom przypisywane będą mniejsze wartości, zgodnie z prawem rozpadu naturalnego (ang. *exponential decay*). Możemy więc przyjąć wagę krawędzi, która pojawiła się w chwili t jako $w_t = e^{\lambda(t_n - t)}$, gdzie t_n jest momentem wystąpienia najnowszej krawędzi, a λ jest parametrem sterującym szybkością zapominania. Stąd elementy macierzy sąsiedztwa wyznaczane są ze wzoru:

$$A_{i,j} = A_{j,i} = \underset{t \leq t_n}{max}\, e^{-\lambda(t_n - t)} \mathbb{1}_{[(r_i, r_j)^t \in SG]}. \tag{2.5}$$

Taka forma zapominania ma charakter globalny, co oznacza, że waga wszystkich istniejących krawędzi zmniejsza się w wyniku przetwarzania każdej nowej krawędzi. Może to prowadzić do kosztownych i niepotrzebnych zmian dla wierzchołków znajdujących się daleko od nowych krawędzi. Dlatego autorzy proponują zastosowanie lokalnej techniki zmniejszania wag. Formalnie, aby obliczyć wagę krawędzi (r_i, r_j) zaobserwowanej w chwili t dla wierzchołka r_i, wyznaczamy najpierw liczbę krawędzi wchodzących i wychodzących z r_i w przedziale czasowym $(t, t_n]$:

$$\phi_{(t,t_n]}(r_i) = |\{(r_p, r_q)^{t'} | r_i \in (r_p, r_q), t' \in (t, t_n]\}|. \tag{2.6}$$

Ostatecznie, elementy macierzy mają postać:

$$A_{i,j} = \underset{t \leq t_n}{max}\, e^{-\lambda \phi_{(t,t_n]}(r_i)} \mathbb{1}_{[(r_i, r_j)^t \in SG]}. \tag{2.7}$$

Działanie właściwego algorytmu SGSketch rozpoczyna się od wygenerowania początkowego szkicu. Odbywa się to w sposób w zasadzie identyczny jak w algorytmie NodeSketch, z tą różnicą, że wykorzystywana jest macierz SLA z lokalnym zapominaniem krawędzi. Zasada działania procedury aktualizacji szkicu ukazana jest w algorytmie 2. Argumentami wejściowymi jest macierz sąsiedztwa SLA $\tilde{A}$, aktualny szkic S, zbiór wierzchołków indukujących nową krawędź lub krawędzie Ω, rząd k oraz współczynnik α. Zasadniczo, mechanizm jest podobny do pierwszego generowania szkicu, ale w tym przypadku większość wartości jest już znana, a przeliczane są zanurzenia tylko tych elementów, które znajdują się w sąsiedztwie nowych krawędzi. Pozwala to na znaczną oszczędność w kwestii czasu obliczeń.

Algorytm 2: SGSketchUpdate($\tilde{A}, S, \Omega, k, \alpha$)

if $k > 2$ **then**
  zaktualizuj $S(k-1)$ i zbiór Ω: $SGSketchUpdate(\tilde{A}, S, \Omega, k-1, \alpha)$;
  foreach $r \in \Omega$ **do**
    $\tilde{V}^r(k) \leftarrow \tilde{V}_i^r + \sum_{n \in \Gamma(r)} \frac{\alpha}{m} \sum_{j=1}^{m} \mathbb{1}_{[S_j^n(k-1)=i]}, i \in \{1, 2, \ldots, D\}$;
    $S_j^r(k) \leftarrow \arg\min_{i \in \{1,2,\ldots,D\}} \frac{-\log h_j(i)}{\tilde{V}_i^r(k)}, j \in \{1, 2, \ldots, m\}$;
else if $k = 2$ **then**
  foreach $r \in \Omega$ **do**
    $S_j^r(2) \leftarrow \arg\min_{i \in \{1,2,\ldots,D\}} \frac{-\log h_j(i)}{\tilde{V}_i^r(2)}, j \in \{1, 2, \ldots, m\}$;
foreach $r \in \Omega$ **do**
  $\Omega \leftarrow \Omega \cup \Gamma(r)$;

Rozdział 3

Operowanie na szkicach danych

W niniejszym rozdziale przedstawimy szerzej operacje, które mogą być wykonywane na szkicach danych oraz skomentujemy ich praktyczne zastosowania. Omówimy także algorytmy `ExpSketch` i `FastExpSketch`, generujące szkice, na których możliwe jest wykonywanie między innymi działań teoriomnogościowych.

3.1. Szkice danych

Jak już wspomniano w poprzednich rozdziałach, szkice danych to kompaktowe struktury, które pozwalają na efektywną reprezentację kluczowych informacji o dużych zbiorach danych. Są one najczęściej generowane przez algorytmy online, przetwarzające strumienie danych i wykorzystujące funkcje haszujące do przypisywania identycznym elementom strumienia tych samych, pseudolosowych wartości. Formalnie, możemy zdefiniować strumień danych jako multizbiór $\mathfrak{M} = (\mathbb{S}, m)$, gdzie $\mathbb{S}$ to zbiór unikalnych elementów, a funkcja $m : \mathbb{S} \to \mathbb{N}_{\geq 1}$ wyznacza ich liczność w strumieniu.

Szkice danych były początkowo wykorzystywane głównie do estymacji prostych statystyk, takich jak liczba unikalnych elementów, czy częstość ich występowania. Tego typu wiedza bywa wystarczająca w wielu zastosowaniach, ale w niektórych przypadkach może okazać się niewystarczająca. Użyteczniejsze są szkice przechowujące dodatkowe informacje o elementach, takie jak ich wagi, czy dodatkowe atrybuty, a także umożliwiające wykonywanie działań na wyznaczonych szkicach. Przykładowo, posiadając szkice dwóch zbiorów danych, przydatna byłaby możliwość efektywnego wyznaczenia szkicu ich sumy lub przecięcia.

3.1.1. ExpSketch

Szczególnie ciekawym przykładem algorytmu mającego na celu zwiększyć liczbę możliwych do wykonania operacji, a tym samym użyteczność szkicu, jest `ExpSketch`[23]. Operuje on na elementach postaci (i, λ_i), gdzie i jest identyfikatorem elementu, a λ_i jego wagą. Wynikiem jego działania jest natomiast wektor zanurzeń o pewnym ustalonym rozmiarze m. W praktyce `ExpSketch` może reprezentować wiele różnych cech zbioru danych poprzez dodanie dodatkowego wymiaru w zanurzeniach elementów. W takim wypadku zamiast jednej wagi λ_i do każdego elementu i przypisany jest wektor wag $\lambda_i = (\lambda_{i,1}, \lambda_{i,2}, \ldots, \lambda_{i,d})$, a szkic elementu przyjmuje formę macierzy o rozmiarze $d \times m$. Jednak w kontekście szkicowania grafów jedyna cecha, którą będziemy rozważać, jest związana z wagami krawędzi. Dlatego przedstawimy wersję algorytmu dla $d = 1$.

Schemat działania algorytmu `ExpSketch` jest dość prosty. Przetwarza on elementy w strumieniu sekwencyjnie. Dla każdego elementu (i, λ_i) obliczane jest m wartości funkcji

haszującej:

$$h(i||1), h(i||2), \ldots, h(i||m),$$

gdzie $h : \mathfrak{M} \to [0, 1]$ jest funkcją haszującą, a $||$ oznacza konkatenację reprezentacji binarnych liczb o ustalonej długości. Każda z wartości funkcji haszującej jest następnie przekształcana przy użyciu odwrotności funkcji rozkładu wykładniczego z parametrem λ_i:

$$F^{-1}(u) = -\frac{\ln u}{\lambda_i}.$$

Zgodnie z *inverse transform sampling theorem*[8], otrzymujemy w ten sposób zmienną losową o rozkładzie wykładniczym:

$$E = -\frac{\ln(h(i||k))}{\lambda_i} \sim Exp(\lambda_i).$$

Każda z otrzymanych w ten sposób wartości jest porównywana z odpowiadającą jej pozycją w szkicu i zapisywana, jeśli jest mniejsza. Zasadę działania całej procedury ilustruje algorytm 3. Co istotne, na tak zdefiniowanych szkicach można w prosty sposób wykonywać operacje teoriomnogościowe.

Algorytm 3: `ExpSketch`$(\mathfrak{M}, m)$

```
1 M = (M_1, M_2, ..., M_m) ← (∞, ∞, ..., ∞);
2 foreach (i, λ_i) ∈ 𝔐 do
3     foreach k ∈ 1, 2, ..., m do
4         U ← h(i||k);
5         E ← −ln(U/λ_i);
6         M_k ← min{M_k, E};
7 return M
```

3.1.2. Operacja teoriomnogościowe na szkicach

Rozważmy dwa zbiory $\mathbb{A}$ i $\mathbb{B}$ oraz odpowiadające im szkice:

$$A = (A_1, A_2, \ldots, A_m) \quad \text{i} \quad B = (B_1, B_2, \ldots, B_m).$$

Wiemy, że elementy A_k i B_k szkiców A i B reprezentują minimalne wartości zmiennych losowych o rozkładzie wykładniczym:

$$A_k \sim Exp(|\mathbb{A}|_w) \quad \text{i} \quad B_k \sim Exp(|\mathbb{B}|_w),$$

gdzie $|\mathbb{S}|_w = \sum_{i \in \mathbb{S}} \lambda_i$ jest sumą wag elementów w zbiorze $\mathbb{S}$.

Suma

Z własności rozkładu wykładniczego wiemy, że dla dwóch zmiennych losowych X, Y o rozkładzie wykładniczym i parametrach λ_X, λ_Y, $\min(X, Y)$ ma także rozkład wykładniczy, którego parametr wynosi $\lambda = \lambda_X + \lambda_Y$. Stąd, chcąc otrzymać szkic C, gdzie

$$C_k \sim Exp(|\mathbb{A} \cup \mathbb{B}|_w),$$

możemy zastosować operację minimum na odpowiadających sobie elementach szkiców A i B:

$$A \cup B = (\min\{A_1, B_1\}, \min\{A_2, B_2\}, \ldots, \min\{A_m, B_m\}).$$

Taka konstrukcja jest oczywiście intuicyjna. Gdyby algorytm `ExpSketch` otrzymał na wejściu zbiór $\mathbb{A} \cup \mathbb{B}$, to każdy z elementów obu zbiorów zostałby wykorzystany do wygenerowania wartości funkcji haszującej, a wynikowy szkic zawierałby najmniejsze z otrzymanych wartości.

Z kolei samą wartość $|\mathbb{A} \cup \mathbb{B}|_w$ możemy estymować jako:

$$\hat{U}(A, B) = \frac{m-1}{\sum_{k=1}^{m} \min\{A_k, B_k\}}.$$

Jako pokazano w [24], tak zdefiniowany estymator jest nieobciążony i ma wyznaczoną wariancję, podobnie jak kolejne, przedstawione dalej estymatory.

Ważone podobieństwo Jaccarda

Podobieństwo Jaccarda to miara podobieństwa dwóch zbiorów, zdefiniowana jako stosunek liczby elementów wspólnych do liczby elementów w sumie zbiorów:

$$J(\mathbb{A}, \mathbb{B}) = \frac{|\mathbb{A} \cap \mathbb{B}|}{|\mathbb{A} \cup \mathbb{B}|}.$$

Można także zdefiniować jego ważony wariant, wykorzystując sumy wag elementów zbiorów:

$$J_w(\mathbb{A}, \mathbb{B}) = \frac{|\mathbb{A} \cap \mathbb{B}|_w}{|\mathbb{A} \cup \mathbb{B}|_w}.$$

Jak pokazano w [23],

$$J_w(\mathbb{A}, \mathbb{B}) = Pr[A_k = B_k].$$

Stąd można łatwo wyznaczyć nieobciążony estymator ważonego podobieństwa Jaccarda zbiorów $\mathbb{A}$ i $\mathbb{B}$ jako:

$$\hat{J}_w(A, B) = \frac{1}{m} \sum_{k=1}^{m} \mathbb{1}[A_k = B_k].$$

Przecięcie

Statystyki dotyczące przecięcia zbiorów można łatwo estymować na podstawie sumy i podobieństwa Jaccarda. Dla $|\mathbb{A} \cap \mathbb{B}|_w$ otrzymujemy nieobciążony estymator w postaci:

$$\hat{I}(A, B) = \hat{J}_w(A, B) \cdot \hat{U}(A, B).$$

Estymator sumy wag elementów

Oznaczmy sumę wag wszystkich elementów w strumieniu jako $\Lambda = \lambda_1, \lambda_2, \ldots, \lambda_n$ i roważmy k-tą pozycję w szkicu. Dla każdego elementu i generowana jest próbka $E_i \sim Exp(\lambda_i)$, a w ostatecznym szkicu zapisywana jest najmniejsza z tych próbek:

$$M_k = \min\{E_1, E_2, \ldots, E_n\}.$$

Oczywiście minimum zmiennych losowych o rozkładzie wykładniczym również ma rozkład wykładniczy, z parametrem równym Λ:

$$M_k \sim Exp(\Lambda).$$

Warto zauważyć, że wartości $M_1, M_2, \ldots, M_m$ są niezależne od siebie. Wiadomo również, że suma niezależnych zmiennych losowych o rozkładzie wykładniczym i o tej samej wartości oczekiwanej ma rozkład gamma. Korzystając z tego faktu, oznaczmy $G_m = M_1 + M_2 + \cdots + M_m$. Wtedy

$$G_m \sim \Gamma(m, \Lambda).$$

Korzystając z własności rozkładu gamma, możemy wyznaczyć estymator sumy wag elementów w strumieniu jako:

$$\hat{\Lambda} = \frac{m-1}{G_m}.$$

Jest to estymator nieobciążony.

3.1.3. FastExpSketch

Łatwo zauważyć, że `ExpSketch` dla każdego elementu w strumieniu wykonuje m iteracji pętli, po jednej dla każdej wartości przechowywanej w szkicu. Pojedyncza iteracja polega na obliczeniu funkcji haszującej, wygenerowaniu liczby losowej, obliczeniu wartości E i zaktualizowaniu szkicu. Każda z tych operacji wykonywana jest w czasie stałym, co daje złożoność czasową $\Omega(m)$ dla pojedynczego elementu w strumieniu. W przypadku przetwarzania krawędzi grafu oznacza to łączną złożoność czasową $\Omega(m|E|)$, gdzie $|E|$ to liczba krawędzi w grafie. Taka liczba operacji może być w praktyce zbyt duża, dlatego rozważać będziemy zoptymalizowaną wersję algorytmu, `FastExpSketch` [24]. Jego idea opiera się na wykorzystaniu następującego twierdzenia [8]:

Twierdzenie 1 *Niech $E_1, E_2, \ldots, E_m$ będą niezależnymi zmiennymi losowymi o rozkładzie wykładniczym. Oznaczmy przez:*

$$E_{(1)} \leq E_{(2)} \leq \cdots \leq E_{(m)}$$

ich statystyki pozycyjne. Wtedy dla każdego $k \in \{1, 2, \ldots m\}$ zachodzi równość rozkładów:

$$E_{(k)} \stackrel{d}{=} E_{(k-1)} + \frac{E_k}{m-k+1}.$$

W oryginalnym algorytmie `ExpSketch`, dla każdego elementu generowanych jest m wartości $E_1, E_2, \ldots, E_m$, które są następnie porównywane z analogicznymi wartościami w szkicu. Można zauważyć, że jeśli dla danego k zachodzi:

$$E_k > \max\{M_1, M_2, \ldots, M_m\},$$

to oczywiście E_k nie zostanie zapisane w szkicu. Ten fakt pozwala na zmniejszenie liczby operacji wykonywanych przez algorytm, przy utrzymaniu identycznych wyników. Konkretnie, zamiast wartości $E_1, E_2, \ldots$, generowane są ich statystyki pozycyjne $E_{(1)}, E_{(2)}, \ldots$, dopóki nie zajdzie warunek $E_{(i)} > \max\{M_1, M_2, \ldots, M_m\}$. Następnie wygenerowane wartości $E_{(1)}, E_{(2)}, \ldots, E_{(i-1)}$ są porównywane z wartościami w szkicu na losowo wybranych pozycjach.

Działanie `FastExpSketch` ilustruje Algorytm 4. Algorytm rozpoczyna się od inicjalizacji szkicu, oraz zmiennych pomocniczych. Dla każdego elementu w strumieniu ustawiamy zmienną S przechowującą aktualną statystykę pozycyjną na 0. W linijkach $9-11$ obliczamy kolejne statystyki pozycyjne, korzystając z Twierdzenia 1 i zapisujemy je na losowej pozycji j w szkicu, o ile są mniejsze od aktualnie zapisanej tam wartości. Pozycje w szkicu wyznaczamy linijkach $14-16$ na podstawie losowej permutacji P, za pomocą kroków analogicznych do algorytmu Fischera-Yatesa. Cała procedura jest przerywana, jeśli wygenerowana wartość S przekroczy

aktualne maksimum MAX wartości w szkicu. Na końcu wartość MAX jest aktualizowana, o ile zaszła taka potrzeba.

Rozważając złożoność algorytmu, możemy skupić się na liczbie porównań w linii 19. Oznaczmy przez C_i zmienną losową wyznaczającą liczbę takich porównań dla elementu i. Rozważymy twierdzenie, którego dowód można znaleźć w [24]:

Twierdzenie 2 *Dla $i \geq 2$, $\Lambda_i = \lambda_1, \lambda_2, \ldots, \lambda_i$ oraz $r_i = \frac{\lambda_i}{\Lambda_{i-1}}$, zachodzi:*

$$\mathbb{E}[C_i] = m \left(1 - \prod_{j=1}^{m} \frac{j}{j + r_i} \right).$$

W przypadku ogólnym wartość oczekiwana zależy więc od wag elementów w strumieniu. W dalszych rozważaniach przyjmiemy, że są one jednakowe, co ma uzasadnienie np. w przypadku grafów nieważonych. W takim przypadku wartość oczekiwana liczby porównań wynosi:

$$\mathbb{E}[C_i] = m \left(1 - \prod_{j=1}^{m} \frac{j}{j + \frac{1}{i-1}} \right).$$

Korzystając z własności liczb Stirlinga pierwszego rodzaju, możemy pokazać, że:

$$\lim_{i \to \infty} i\mathbb{E}[C_i] = mH_m \quad \text{i} \quad \mathbb{E}[C_i] \leq \frac{mH_m}{i},$$

gdzie H_m jest m-tą liczbą harmoniczną. Stąd otrzymujemy:

$$\mathbb{E}[C] \leq \sum_{i=1}^{n} \frac{mH_m}{i} = mH_mH_n,$$

gdzie $H_n = \ln n + \gamma + O(\frac{1}{n})$, więc wartość oczekiwana liczby porównań jest logarytmiczna względem liczby elementów w strumieniu.

Algorytm 4: `FastExpSketch`$(\mathfrak{M}, m)$

```
1  permInit ← (1, 2, 3, ..., m);
2  M = (M_1, M_2, ..., M_m) ← (∞, ∞, ..., ∞);
3  MAX ← ∞;
4  foreach (i, λ_i) ∈ 𝔐 do
5      S ← 0;
6      updateMAX ← false;
7      P ← permInit;
8      foreach k ∈ 1, 2, ..., m do
9          U ← h(i||k);
10         E ← −ln(U/λ_i);
11         S ← S + E/(m − k + 1);
12         if S > MAX then
13             break;
14         r ← RandomInteger([k, m], seed = i);
15         swap(P[k], P[r]);
16         j ← P[k];
17         if M_j = MAX then
18             updateMAX ← true;
19         M_j ← min{M_j, S};
20     if updateMAX then
21         MAX ← max{M_1, ..., M_m};
22 return M
```

Rozdział 4

Ulepszenie algorytmu NodeSketch z wykorzystaniem FastExpSketch

4.1. Motywacja

Algorytm NodeSketch cechuje się wysoką efektywnością pamięciową i czasową. Eksperymenty pokazują, że osiąga on także dobre rezultaty na rzeczywistych danych [49]. Niemniej jednak, gama operacji, które można wykonać na wynikowych szkicach jest dość ograniczona. Ponadto, NodeSketch w swej bazowej postaci nie uwzględnia wag krawędzi w zanurzeniach, co potencjalnie ogranicza jego użyteczność dla grafów ważonych. W niniejszym rozdziale przyglądamy się generalizacji algorytmu NodeSketch, polegającej na wykorzystaniu metody FastExpSketch do szkicowania wierzchołków. Wyjaśniamy zasadę działania tej metody, przedstawiamy jej implementację i analizujemy jej złożoność. Zwracamy także uwagę na korzyści płynące z zastosowania takiego podejścia, takie jak możliwość wykonywania operacji teoriomnogościowych na szkicach, czy uwzględnienie wag krawędzi.

4.2. Idea

NodeSketch wykorzystuje *inverse sampling theorem* do generowania próbek z rozkładu wykładniczego. Konkretnie j-ta pozycja w zanurzeniu jest obliczana jako:

$$S_j = \underset{i \in \{1,2,\ldots,D\}}{\arg\min} \frac{-\log h_j(i)}{V_i},$$

gdzie $V = (V_1, V_2, \ldots, V_D)$ to wektor sąsiedztwa. Łatwo zauważyć, że jest to bardzo podobny pomysł do tego, na którym opierają się algorytmy ExpSketch i FastExpSketch. Podstawowa różnica polega na tym, że w przypadku NodeSketch, w szkicu przechowywane będą indeksy wierzchołków, podczas gdy w przypadku ExpSketch - rzeczywista wartość wygenerowana z rozkładu wykładniczego. Pozwala to między innymi na lepsze uwzględnienie wag krawędzi w zanurzeniach. Dlatego też, naturalnym krokiem wydaje się zastąpienie tej części algorytmu NodeSketch szkicowaniem wierzchołków z wykorzystaniem FastExpSketch.

Przedstawiona modyfikacja powinna zmniejszyć również średnią liczbę wykonywanych operacji, ze względu na bardziej efektywne podejście do szkicowania. To założenie zostało zweryfikowane eksperymentalnie w Rozdziale 5.6. Działanie zmodyfikowanej wersji algorytmu, którą określać będziemy dalej jako EdgeSketch, przedstawia Algorytm 5. W każdej iteracji głównej pętli rozważamy jeden wiersz w macierzy sąsiedztwa, a więc jeden wierzchołek. W wewnętrznej pętli wybieramy jego sąsiadów oraz wagi połączeń. Etykieta krawędzi powstaje

przez połączenie etykiet wierzchołków w porządku leksykograficznym. Wynika to z faktu, że rozważamy grafy nieskierowane i krawędź (a, b) powinna być nierozróżnialna od krawędzi (b, a). Następnie, krawędzie te są przekazywane do metody FastExpSketch, która generuje na ich podstawie właściwe szkice.

Algorytm 5: EdgeSketch($\tilde{A}, m$)

```
foreach rząd r w Ã do
    ns ← [ ] ;                                          // lista sąsiadów
    foreach i ∈ {1, ..., |V|} do
        if Ã[r, i] ≠ 0 then
            ns ← ns ∪ {((min(i, r)|| max(i, r)), Ã[r, i])};
    S^r ← FastExpSketch(ns, m);
return S
```

Co ciekawe, w tak zmodyfikowanym algorytmie można pominąć parametr k przy tworzeniu zanurzeń. Zamiast tego, sąsiedztwo wyższych rzędów może zostać uzyskane poprzez odpowiednie operacje na szkicu. Jako przykład, a zarazem ciekawe zastosowanie algorytmu, będziemy rozważać zadanie rekonstrukcji grafu. Konkretnie, zależeć nam będzie na wyznaczeniu na podstawie szkicu t najbardziej prawdopodobnych krawędzi. W tym celu wykorzystane zostanie podobieństwo Jaccarda między zanurzaniami wierzchołków. Procedura polega na obliczeniu macierzy podobieństwa $simM$ między każdymi dwoma wierzchołkami w grafie, co ilustruje Algorytm 6. W przypadku oryginalnego algorytmu NodeSketch, operacja ta jest wykonywana na podstawie zanurzeń k-tego rzędu. Z kolei w przypadku EdgeSketcha, macierze podobieństwa mogą być obliczane oddzielnie dla różnych stopni sąsiedztwa. Na przykład, dla $k = 3$, do reprezentacji wierzchołka używana jest suma teoriomnogościowa zanurzeń tego wierzchołka i jego sąsiadów, dla $k = 4$ wszystkich wierzchołków odległych o co najwyżej 2 i tak dalej. Ostateczny rezultat powstaje przez połączenie wszystkich macierzy podobieństwa. Parametr rozkładu wykładniczego α decyduje o tym, jakie wagi nadawane są macierzom wyższych rzędów, według wzoru:

$$simM = \sum_{k=2}^{K} \alpha^{k-2} simM_k. \tag{4.1}$$

Powyższa formuła, jak również ogólna idea za nią stojąca, pochodzi z nieopublikowanej jeszcze pracy dr inż. Jakuba Lemiesza oraz prof. Philippe'a Cudré-Mauroux. Spośród obliczonych wartości prawdopodobieństw, wybierane jest t największych. Wyznaczają one krawędzie w zrekonstruowanym grafie.

4.2.1. Złożoność obliczeniowa

Dla uproszczenia, będziemy rozważać złożoność algorytmów dla grafów nieważonych. Rozważmy działanie algorytmu NodeSketch dla $k = 2$. Przetwarza on po kolei $|V|$ wierzchołków, dla każdego z nich generując szkic o m elementach.Każdy z tych elementów powstaje poprzez generowanie próbek z rozkładu wykładniczego, po jednej dla każdego z pozostałych wierzchołków w grafie. Daje to złożoność $O(m|V|)$ dla jednej iteracji i łącznie $O(m(|V|)^2)$ dla całej procedury. Algorytm wykonuje stałą liczbę $k - 2$ wywołań rekurencyjnych. W przypadku wyższych k, oprócz generowania próbek, konstruowany jest także wektor sąsiedztwa wyższego rzędu. W oryginalnej wersji Algorytmu 1 krok ten wymaga

Algorytm 6: ComputeSimilarityMatrix($embeddings, n, m$)

```
1 simM ← [n, n] ;                                   // pusta macierz n × n
2 foreach i ∈ {1, ..., n} do
3     foreach j ∈ {i, ..., n} do
4         simCount ← 0;
5         foreach l ∈ {1, ..., m} do
6             if embeddings[i, l] = embeddings[j, l] then
7                 simCount ← simCount + 1;
8         simM[i, j] = simM[j, i] ← simCount/m
9 return simM
```

$O(m|V|^2)$ operacji dla każdego wierzchołka, ponieważ zakłada przejście po każdej pozycji w m-elementowym szkicu $|V|^2$ razy. Łatwo jednak zauważyć, że zamiast obliczać

$$\sum_{j=1}^{m} \mathbb{1}_{[S_j^n(k-1)=i]}$$

oddzielnie dla każdego $i \in \{1, 2, \ldots, |V|\}$, można podliczać wystąpienia różnych i i zapamiętać wyniki w strukturze takiej jak słownik. Pozwala to na zredukowanie złożoności tworzenia wektora sąsiedztwa do $O(m|V|)$. Stąd cały algorytm ma złożoność $O(m(|V|)^2)$.

EdgeSketch również przetwarza po kolei wierzchołki. Dla każdego z nich wyszukiwani są sąsiedzi w czasie liniowym. Z kolei czas potrzebny na stworzenie zanurzeń dla danego wierzchołka zależy od liczby krawędzi z niego wychodzących. Liczba ta nie przekroczy jednak oczywiście liczby wierzchołków. Dlatego możemy przyjąć, że liczba elementów przekazywanych do procedury `FastExpSketch` jest rzędu $O(|V|)$. Pesymistyczna liczba operacji wykonywanych przez tą procedurę jest rzędu $O(m|V|)$, stąd łączną złożoność całego algorytmu ExpSketch można wyrazić jako $O(m(|V|)^2)$, tak samo jak w algorytmie `NodeSketch`.

Niemniej jednak, warto rozważyć także złożoność w średnim przypadku. Możemy przyjąć średnią złożoność `FastExpSketch` dla elementów o równej wadze. Przypomnijmy, że wynosi ona w takim przypadku $O(mH_mH_{|V|}) = O(m\ln(m)\ln(|V|))$. Daje to łączną złożoność algorytmu $O((|V|)^2 + |V|(m\ln(m)\ln(|V|)))$ lub, traktując m jako stałą, $O((|V|)^2 + |V|(\ln(|V|)))$. Pierwszy składnik sumy odpowiada wyszukiwaniu sąsiadów każdego wierzchołka. Z kolei drugi związany jest z wywołaniami procedury `FastExpSketch`. Choć oczywiście pierwszy składnik sumy jest asymptotycznie dominujący, w praktyce dla wielu grafów drugi składnik może okazać się kluczowy, głównie ze względu na stosunkowo wysoki koszt obliczania funkcji haszującej. Warto też zauważyć, że obliczenia wykonywane w każdej iteracji pętli są niezależne od siebie, co pozwala na łatwe zrównoleglenie obliczeń.

Wynikiem działania obu algorytmów są szkice o rozmiarze $m \times |V|$. NodeSketch konstruuje także wektory sąsiedztwa wyższych rzędów, z których każdy ma $O(|V|)$ elementów, ale ponieważ są one generowane oddzielnie, po jednym w każdej iteracji, nie zwiększa to asymptotycznej złożoności obliczeniowej. Z kolei EdgeSketch alokuje dodatkową pamięć na przechowywanie listy sąsiadów w każdej iteracji pętli, ale jest ona także rzędu $O(|V|)$, więc ogólna złożoność pamięciowa obu algorytmów wynosi $O(m|V|)$.

Rozdział 5

Analiza wyników

W niniejszym rozdziale zweryfikowano eksperymentalnie działanie algorytmu `EdgeSketch` oraz porównano go z algorytmem `NodeSketch`. Przeprowadzono szereg testów na różnych grafach i zbadano wpływ parametrów na jakość uzyskiwanych wyników. Sprawdzono także średnią liczbę operacji wykonywanych przez algorytmy w zależności od liczby krawędzi w grafie.

5.1. Architektura eksperymentów

Na działanie algorytmów `NodeSketch` i `EdgeSketch` wpływ mają trzy główne parametry. Są to:

- k - rząd sąsiedztwa. Przyjmujemy, że przy $k = 2$ rozpatrujemy tylko bezpośrednie połączenia, przy $k = 3$ także ścieżki długości 2 itd.
- m - rozmiar szkicu. Jest to liczba elementów w wektorze opisującym pojedynczy wierzchołek w grafie.
- α - parametr rozkładu wykładniczego. Decyduje on, jakie wagi nadawane są sąsiedztwom wyższych rzędów.

Jednak uzyskiwane wyniki zależą też w naturalny sposób od grafu, na którym przeprowadzane są eksperymenty. W związku z tym, testy przeprowadzono na różnorodnych grafach i z różnymi parametrami. Główną badaną statystyką była precyzja, czyli stosunek liczby poprawnie odgadniętych krawędzi do wartości t, a więc wielkości próbki. Rozważano próbki wielkości $t \in \{100, 1000, 10000, |E|\}$, gdzie $|E|$ to liczba krawędzi w grafie. Algorytmy zostały zaimplementowane w języku Julia[39].

5.2. Badanie uzyskiwanych wyników w zależności od struktury grafów

Model Erdosa-Renyiego

Model Erdosa-Renyiego jest jednym z najpowszechniejszych modeli grafów losowych. W modelu tym, każda para wierzchołków jest połączona krawędzią z jednakowym prawdopodobieństwem p. Steruje on gęstością grafu, a co za tym idzie, średnim stopniem wierzchołków. Grafy w tym modelu charakteryzują się dość jednorodną strukturą, bez wyraźnych klastrów, czy wierzchołków o dysproporcjonalnie wysokich stopniach.

Eksperyment przeprowadzono dla różnych prawdopodobieństw wystąpienia krawędzi p oraz stopni sąsiedztwa $k \in \{2, 3, 4\}$. W każdym przypadku liczba wierzchołków w grafie wynosiła

10000. Rozmiar szkicu wyniósł $m = 10$, a parametr rozkładu wykładniczego $\alpha = 0.3$. Wyniki przedstawiono w Tabeli 5.1.

		NodeSketch				EdgeSketch			
p	**k**	**t = 100**	**t = 1000**	**t = 10000**	**t = |E|**	**t = 100**	**t = 1000**	**t = 10000**	**t = |E|**
0.0005	2	0.94	0.762	0.4745	0.3688	1	1	1	0.5562
	3	0.86	0.732	0.3947	0.2515	1	1	0.9446	0.6179
	4	0.55	0.164	0.0409	0.023	1	0.985	0.8983	0.6072
0.001	2	0.58	0.433	0.2779	0.2048	1	1	1	0.3721
	3	0.38	0.074	0.0185	0.007	1	1	0.9748	0.4285
	4	0	0.004	0.0012	0.0017	1	1	0.9665	0.4301
0.005	2	0.26	0.144	0.0901	0.0497	1	1	1	0.0981
	3	0.11	0.033	0.0137	0.0062	1	0.936	0.3025	0.0557
	4	0.01	0.006	0.0038	0.005	1	0.936	0.3025	0.0557
0.01	2	0.05	0.079	0.0595	0.033	1	1	0.9998	0.0576
	3	0.05	0.026	0.0157	0.0104	1	1	1	0.05
	4	0	0.006	0.0091	0.0102	1	1	1	0.05

Tab. 5.1: Model Erdosa-Renyiego

Stochastyczny model blokowy

Stochastyczny model blokowy (ang. *stochastic block model*) jest modelem, w którym wierzchołki grafu są podzielone losowo na b bloków. Prawdopodobieństwo wystąpienia krawędzi między dwoma wierzchołkami w tym samym bloku wynosi p, a pomiędzy wierzchołkami z różnych bloków q. Zazwyczaj przyjmuje się $p \gg q$. W modelu tym występują wyraźne klastry, ale stopnie poszczególnych wierzchołków są do siebie dość zbliżone. Sprawdza się on w symulowaniu zbiorów danych, w których można wyróżnić grupy podobnych do siebie punktów, jak np. grupy użytkowników o podobnych zainteresowaniach, albo artykuły z podobnymi słowami kluczowymi.

W przeprowadzonym eksperymencie przyjęto rozmiar grafu $n = 1000$, prawdopodobieństwo wystąpienia krawędzi wewnątrz bloku $p = 0.5$ oraz między blokami $q = 0.001$. Zbadano także różne liczby bloków $b \in \{2, 4, 8\}$. Rozmiar szkicu m ustalono na 10, a parametr rozkładu wykładniczego wyniósł $\alpha = 0.3$. Wyniki przedstawiono w Tabeli 5.2.

		NodeSketch				EdgeSketch			
b	**k**	**t = 100**	**t = 1000**	**t = 10000**	**t = |E|**	**t = 100**	**t = 1000**	**t = 10000**	**t = |E|**
2	2	0.57	0.525	0.5136	0.5072	1	1	0.4391	0.2616
	3	0.47	0.527	0.5089	0.5016	1	1	0.4545	0.3426
	4	0.44	0.523	0.5079	0.5022	1	1	0.4545	0.3426
4	2	0.5	0.542	0.5343	0.5131	1	1	0.3352	0.1547
	3	0.53	0.483	0.4991	0.5003	1	1	0.5342	0.4087
	4	0.46	0.499	0.4983	0.5008	1	1	0.5342	0.4087
8	2	0.66	0.594	0.5521	0.5234	1	1	0.2831	0.1289
	3	0.51	0.528	0.5158	0.5063	1	0.884	0.5825	0.4762
	4	0.5	0.496	0.5002	0.5029	1	0.884	0.5821	0.4755

Tab. 5.2: Stochastyczny model blokowy

Model Barabasiego-Alberta

W modelu Barabasiego-Alberta (ang. *Barabasi-Albert Model*) wierzchołki dodawane są do grafu sekwencyjnie. Nowy wierzchołek łączy się z istniejącymi wierzchołkami z prawdopodobieństwem proporcjonalnym do ich stopnia. Bardziej formalnie, konstrukcja grafu rozpoczyna się od wyboru małego zbioru m_0 początkowych wierzchołków i poprowadzenia wśród nich krawędzi w taki sposób, aby każdy wierzchołek był połączony z co najmniej jednym innym. Następnie, w każdym kroku dodawany jest nowy wierzchołek wraz z m_{ba} krawędziami łączącymi go z innymi z prawdopodobieństwem tym większym, im większy jest stopień danego wierzchołka. Konkretnie, prawdopodobieństwo istnienia krawędzi do wierzchołka i wynosi

$$p_i = \frac{k_i}{\sum_{j \in V^*} k_j},$$

gdzie k_i to stopień wierzchołka i, a V^* to zbiór aktualnie występujących w grafie wierzchołków. W powstałych w ten sposób grafach można zaobserwować duże zróżnicowanie stopni wierzchołków, z wyraźnie odznaczającymi się węzłami centralnymi (ang. *hubs*).

W przeprowadzonym eksperymencie przyjęto liczbę wierzchołków $n = 1000$ i stopień wierzchołków $m_{ba} \in \{2, 4, 8\}$, a rozmiar początkowego zbioru wierzchołków ustalono na $m_0 = m_{ba}$. Wyniki przedstawiono w Tabeli 5.3. W przypadku $m_{ba} = 2$ pominięto wyniki dla $t = 10000$, ponieważ liczba wierzchołków w grafie wynosiła tylko 3994.

		NodeSketch				EdgeSketch			
m_{ba}	**k**	**t = 100**	**t = 1000**	**t = 10000**	**t = |E|**	**t = 100**	**t = 1000**	**t = 10000**	**t = |E|**
2	2	0.52	0.269	X	0.2163	1	0.974	X	0.4877
	3	0.32	0.152	X	0.1232	1	0.213	X	0.1567
	4	0.56	0.289	X	0.2153	1	0.134	X	0.0941
8	2	0.19	0.14	0.0709	0.0892	1	1	0.178	0.2241
	3	0.12	0.063	0.0406	0.0511	1	0.921	0.1765	0.2222
	4	0.08	0.07	0.0313	0.0394	1	0.921	0.1765	0.2222
16	2	0.12	0.079	0.0939	0.0939	1	1	0.2105	0.1424
	3	0.1	0.107	0.0656	0.0595	1	1	0.2197	0.1578
	4	0.06	0.028	0.0359	0.0376	1	1	0.2197	0.1578

Tab. 5.3: Model Barabasiego-Alberta

Grafy ważone

Jedną z motywacji stojących za algorytmem `EdgeSketch` była potrzeba lepszego wykorzystania własności grafów ważonych. Przeprowadzono eksperyment mający na celu zweryfikować rzeczywistą skuteczność tego rozwiązania. Testy przeprowadzono na grafach w modelu Erdosa-Renyiego o 10000 wierzchołków. Wagi krawędzi były losowane z rozkładu jednostajnego na zbiorze $\{1, 2, \ldots, 10\}$. Podobnie jak w poprzednich eksperymentach, rozmiar szkicu wyniósł $m = 10$, a parametr rozkładu wykładniczego $\alpha = 0.3$. Wyniki przedstawiono w Tabeli 5.4.

5.2.1. Wnioski

Algorytm `EdgeSketch` uzyskuje wyższą precyzję niż `NodeSketch` dla większości zbiorów, na których przeprowadzono eksperymenty. W szczególności dla małych wartości t, uzyskiwana przez `EdgeSketch` precyzja jest bliska 1, co oznacza że wszystkie lub niemal wszystkie krawędzie zostały przewidziane poprawnie. `NodeSketch` uzyskuje dość zróżnicowane wyniki,

		NodeSketch				EdgeSketch			
p	**k**	**t = 100**	**t = 1000**	**t = 10000**	**t = |E|**	**t = 100**	**t = 1000**	**t = 10000**	**t = |E|**
0.0005	2	0	0	0.0012	0.0031	1	1	1	0.5505
	3	0	0	0.007	0.0248	1	1	0.94	0.6125
	4	0	0.006	0.0122	0.012	1	1	0.9207	0.6073
0.001	2	0	0	0.0016	0.0055	1	1	1	0.3721
	3	0.01	0.017	0.012	0.006	1	1	0.9671	0.4235
	4	0	0.003	0.0084	0.0099	1	1	0.9714	0.4275
0.005	2	0.01	0.008	0.0054	0.0065	1	1	1	0.0989
	3	0	0.007	0.0086	0.0063	1	0.941	0.3344	0.0581
	4	0	0.004	0.0064	0.0088	1	0.941	0.3344	0.0581
0.01	2	0	0.01	0.0086	0.0107	1	1	1	0.0583
	3	0.03	0.012	0.0101	0.0117	1	1	1	0.05
	4	0.01	0.011	0.0151	0.0138	1	1	1	0.05

Tab. 5.4: Grafy ważone

od zbliżonych, lecz nieco niższych, jak w przypadku grafu w modelu Erdosa-Renyiego z parametrem $p = 0.0005$, po bliskie 0, jak przy gęstszych grafach w tym samym modelu. Dla obu algorytmów wyniki pogarszają się wraz ze wzrostem t, ale `EdgeSketch` osiąga nadal lepsze wyniki niż `NodeSketch` na wszystkich grafach poza tymi w stochastycznym modelu blokowym.

W przypadku algorytmu `NodeSketch`, wybór większej wartości parametru k zazwyczaj albo nie zmienia znacząco precyzji albo ją pogarsza, niezależnie od t. Inaczej jest w przypadku `EdgeSketch`, gdzie dobór optymalnego k zależy od konkretnego zbioru oraz wartości t. O ile w przypadku małych próbek, np. $t = 100$, najlepiej sprawdza się $k = 2$, o tyle dla $t = |E|$, dobór większego k może poprawić wyniki, co widać szczególnie wyraźnie w stochastycznym modelu blokowym oraz w rzadkich grafach w modelu Erdosa-Renyiego.

W przypadku grafów ważonych, różnica pomiędzy algorytmami jest szczególnie wyraźna. `EdgeSketch` uzyskuje wyniki bardzo zbliżone do tych osiąganych dla grafów nieważonych o podobnej strukturze. Pozostaje on więc skuteczny w przewidywaniu krawędzi. W przypadku `NodeSketcha`, wyniki są natomiast znacznie gorsze, często bliskie zeru. Wynikać to może z faktu, że algorytm ten tworzy szkic, w którym przechowywane są indeksy wierzchołków, a nie próbki z rozkładu wykładniczego, jak w przypadku `EdgeSketcha`.

JL:Trochę dziwne, że aż tak kiepsko w NodeSketch - oni też używają rozkładu wykładniczego do wyboru indeksów, więc wagi powinny być uwzględniane przy odpowiedniej implementacji? W sumie to nie rozumiem tego ostatniego zdania.

5.3. Wykorzystanie stopnia wierzchołków przy rekonstrukcji

Jak wspomniano w rozdziale 3.1.2, szkice generowane przez algorytm `ExpSketch` pozwalają na zdefiniowanie nieobciążonego estymatora sumy wag elementów w strumieniu. W przypadku algorytmu `EdgeSketch`, `ExpSketch` jest wywoływany dla pojedynczego wierzchołka, a w strumieniu znajdują się wychodzące z niego krawędzie. Jeśli graf jest nieważony, to suma wag krawędzi jest po prostu stopniem wierzchołka. Przeprowadzono eksperyment mający na celu sprawdzenie, czy ta właściwość może być wykorzystana do poprawy jakości rekonstrukcji. Konkretnie podczas wyboru najbardziej podobnych wierzchołków, śledzono liczbę krawędzi wychodzących z każdego z nich. Jeśli wierzchołek v osiągał swój estymowany stopień, to był on pomijany w dalszym procesie.

Testy przeprowadzono na trzech zbiorach zawierających rzeczywiste dane. Pierwszy z nich to *HomoSapiens*, zbiór danych zawierający relacje między białkami dla gatunku ludzkiego. *Blogcatalog* to zbiór danych zawierający relacje między blogerami, a *dblp* to fragment

zbioru autorów artykułów naukowych, gdzie krawędzie reprezentują współautorstwo. W eksperymencie przyjęto $\alpha = 0.3$ oraz $m = 10$. Wyniki przedstawiono w Tabeli 5.5.

		EdgeSketch				EdgeSketch z oszacowaniem stopnia			
zbiór danych	**k**	**t = 100**	**t = 1000**	**t = 10000**	**t = |E|**	**t = 100**	**t = 1000**	**t = 10000**	**t = |E|**
HomoSapiens	2	1	1	0.4459	0.1202	0.99	0.998	0.4413	0.1669
	3	1	1	0.3036	0.1153	1	0.999	0.2923	0.1576
	4	1	1	0.3035	0.1163	1	1	0.2921	0.1573
blogcatalog	2	1	1	0.7757	0.0343	1	1	0.7755	0.1943
	3	1	1	0.7649	0.0377	1	1	0.7647	0.1988
	4	1	1	0.7649	0.0377	1	1	0.7647	0.1988
dblp	2	1	1	1	0.4453	0.99	0.994	0.9489	0.426
	3	1	1	0.7684	0.3214	1	0.995	0.731	0.449
	4	1	1	0.6666	0.2927	1	0.997	0.6317	0.4303

Tab. 5.5: Wyniki z wykorzystaniem stopnia wierzchołków przy rekonstrukcji

5.3.1. Wnioski

Wyniki pokazują, że wykorzystanie stopnia wierzchołków przy rekonstrukcji rzeczywiście może wpływać na poprawę precyzji. Jest to szczególnie widoczne dla $t = |E|$. Choć zazwyczaj różnice nie były duże, to np. w przypadku zbioru *blogcatalog* precyzja wzrosła z niecałych 4% do niemal 20%, co jest znaczącą poprawą. Nieco gorzej było w przypadku małych t, gdzie nieco lepsze wyniki uzyskał oryginalny algorytm.

5.4. Wpływ rozmiaru szkicu na uzyskiwane wyniki

Jednym z kluczowych parametrów wpływających na działanie algorytmów `NodeSketch` i `EdgeSketch` jest rozmiar szkicu. Wybór odpowiedniej wartości m może nie być oczywisty. Z jednej strony, większy szkic jest w stanie przechować więcej informacji, lecz z drugiej strony, zwiększa to czas obliczeń oraz ilość używanej pamięci. Poniższe eksperymenty miały na celu weryfikację wpływu rozmiaru szkicu na uzyskiwane wyniki.

Pierwszy z nich przeprowadzono na zbiorach danych blogcatalog oraz dblp. W obu przypadkach rozważano różne wartości parametru $m \in \{8, 16, 32, 64, 128, 256\}$. W eksperymencie przyjęto $\alpha = 0.3$ oraz $k = 2$. Wyniki przedstawiono w Tabeli 5.6.

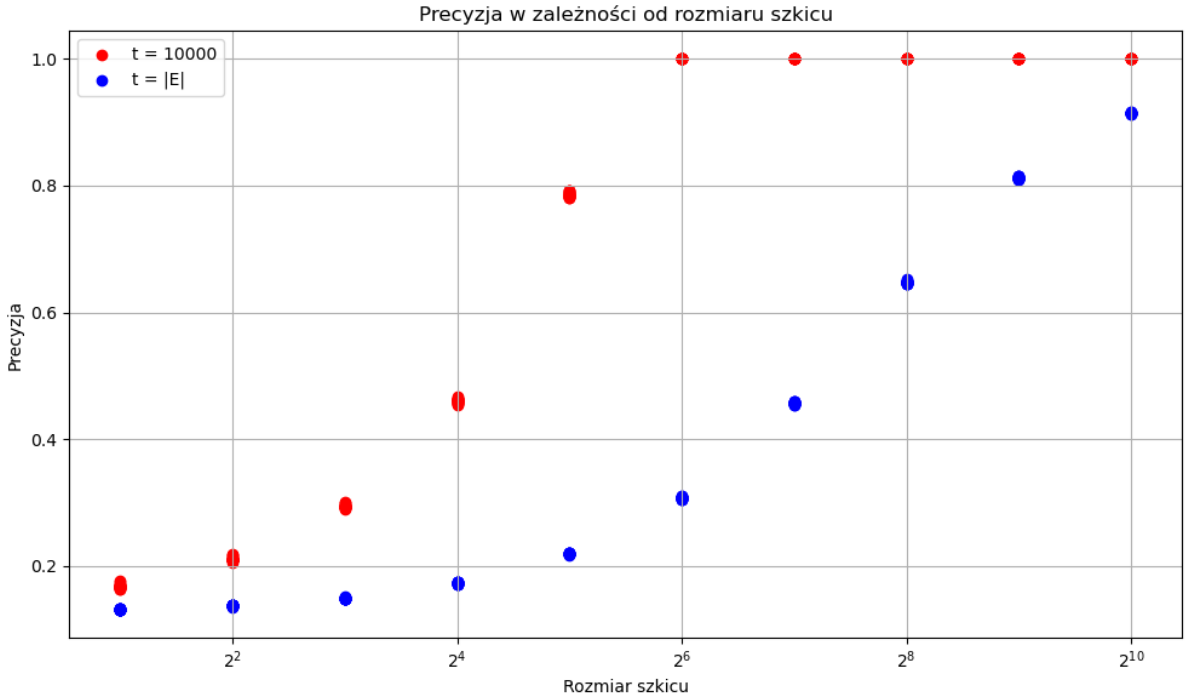
Drugi eksperyment przeprowadzono jedynie dla algorytmu `EdgeSketch` z tymi samymi parametrami. Został on wykonany na 20 grafach w stochastycznym modelu blokowym, z których każdy miał 1000 wierzchołków. Zbadano zmiany precyzji dla $t = 10000$ oraz $t = |E|$ wraz z rozmiarem szkicu. Rezultaty ilustruje Rysunek 5.1.

5.4.1. Wnioski

Zgodnie z intuicją, zwiększenie rozmiaru szkicu ma korzystny wpływ na precyzję. Dla algorytmu `EdgeSketch` jest to szczególnie widoczne w przypadku dużych wartości t. Warto jednak zaznaczyć, że im większy szkic, tym dalsze jego zwiększanie ma mniejszy wpływ na precyzję. Co ciekawe, dla małych t, `EdgeSketch` jest w stanie uzyskać wysoką precyzję nawet przy małym rozmiarze szkicu. Przykładowo, dla $t = 10000$, szkic o rozmiarze 64 jest wystarczający, aby w każdym z badanych przypadków uzyskać precyzję 1, a w dla zbioru dblp, wystarczający jest nawet szkic o 8 elementach. W przypadku algorytmu

		NodeSketch			EdgeSketch		
zbiór danych	**m**	**t = 1000**	**t = 10000**	**t = |E|**	**t = 1000**	**t = 10000**	**t = |E|**
blogcatalog	8	0.031	0.0298	0.0277	1	0.6228	0.0298
	16	0.023	0.0276	0.0292	1	0.9998	0.0474
	32	0.009	0.0224	0.034	1	0.9998	0.0805
	64	0.016	0.0224	0.0312	1	1	0.138
	128	0.01	0.0185	0.0466	1	1	0.2302
	256	0.018	0.0257	0.0406	1	1	0.3609
dblp	8	0.996	0.8975	0.5495	1	1	0.3839
	16	1	0.9381	0.5892	1	1	0.5737
	32	1	0.9264	0.5974	1	1	0.7481
	64	1	0.9333	0.589	1	1	0.87
	128	1	0.9389	0.6017	1	1	0.9421
	256	1	0.9415	0.6051	1	1	0.9757

Tab. 5.6: Wyniki dla różnych rozmiarów szkicu



Rys. 5.1: Precyzja rekonstrukcji w zależności od rozmiaru szkicu dla algorytmu `EdgeSketch`.

`NodeSketch`, zyski występują, ale są niewielkie w porównaniu do `EdgeSketcha`. Warto zaznaczyć, że większy rozmiar szkicu implikuje większą liczbę wykonywanych operacji oraz większe zapotrzebowanie na pamięć, dlatego w praktycznych zastosowaniach konieczne może być znaleznienie kompromisu pomiędzy jakością uzyskiwanych wyników a kosztem obliczeniowym.

5.5. Dobór parametru alpha

Kolejnym parametrem, mającym znaczenie dla obu algorytmów jest α. Jego wartość decyduje, jak szybko maleje wpływ kolejnych, coraz bardziej odległych sąsiadów na generowane szkice, lub, w przypadku `EdgeSketcha`, obliczane podobieństwa Jaccarda. Grafy wykorzystane w eksperymencie zostały wygenerowane w modelu Erdosa-Renyiego i miały 5000 wierzchołków.

Badano różne wartości $\alpha \in \{0, 0.15, 0.3, 0.45\}$, przy czym $\alpha = 0$ oznacza, że sąsiedztwa wyższych rzędów nie były brane pod uwagę. Rozmiar szkicu m ustalono na 10, a parametr k na 4. Rezultaty przedstawiono w Tabeli 5.7.

		NodeSketch				EdgeSketch			
p	α	**t = 100**	**t = 1000**	**t = 10000**	**t = |E|**	**t = 100**	**t = 1000**	**t = 10000**	**t = |E|**
0.0005	0	0.97	0.826	0.3424	0.5415	1	1	0.4462	0.7057
	0.15	0.9	0.798	0.2917	0.4613	1	1	0.4678	0.7398
	0.3	0.95	0.782	0.2863	0.4528	1	1	0.4363	0.69
	0.45	0.9	0.788	0.2743	0.4338	1	0.785	0.3954	0.6253
0.001	0	0.91	0.69	0.4114	0.3773	1	1	0.6996	0.5548
	0.15	0.15	0.056	0.0168	0.0146	1	1	0.7469	0.6264
	0.3	0.16	0.047	0.0185	0.0164	1	0.996	0.7014	0.6161
	0.45	0.19	0.069	0.0198	0.0173	0.97	0.939	0.6041	0.5373
0.005	0	0.38	0.258	0.144	0.0972	1	1	1	0.1814
	0.15	0	0.007	0.0049	0.0053	1	1	0.9276	0.1871
	0.3	0	0.004	0.0042	0.005	1	0.932	0.3196	0.1062
	0.45	0	0.006	0.0044	0.0052	0.95	0.381	0.1227	0.0559
0.01	0	0.15	0.185	0.1091	0.0563	1	1	1	0.1011
	0.15	0.01	0.011	0.0101	0.0094	1	1	0.9231	0.1022
	0.3	0.01	0.009	0.0095	0.0095	1	1	0.3234	0.0647
	0.45	0.01	0.009	0.0094	0.0095	1	0.63	0.1928	0.0428

Tab. 5.7: Wyniki dla różnych wartości parametru α

5.5.1. Wnioski

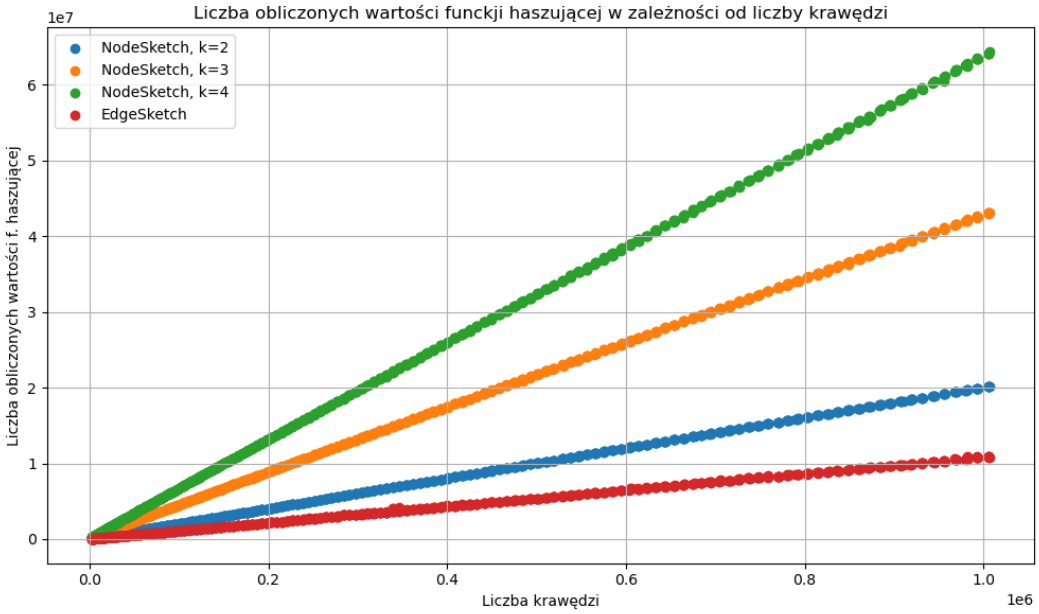
Odpowiedni dobór parametru rozkładu wykładniczego zależy od konkretnego grafu. W badanym modelu dla algorytmu `NodeSketch` najlepsze wyniki uzyskano ignorując sąsiedztwa wyższych rzędów. Inaczej było w przypadku `EdgeSketcha`, gdzie niewielkie wartości parametru, zwłaszcza $\alpha = 0.15$ lub $\alpha = 0.3$ wpływały korzystnie na uzyskiwane wyniki. Pogarszały się one jednak przy wyborze większych współczynników. Większa α pozwala na ujęcie większej ilości informacji o dalszym sąsiedztwie, ale może również przykrywać informacje o najbliższych sąsiadach wierzchołka, co później zaburza proces rekonstrukcji krawędzi. W większości przypadków wybór małych wartości α wydaje się więc najskuteczniejszą strategią dla algorytmu `EdgeSketch`.

5.6. Złożoność obliczeniowa

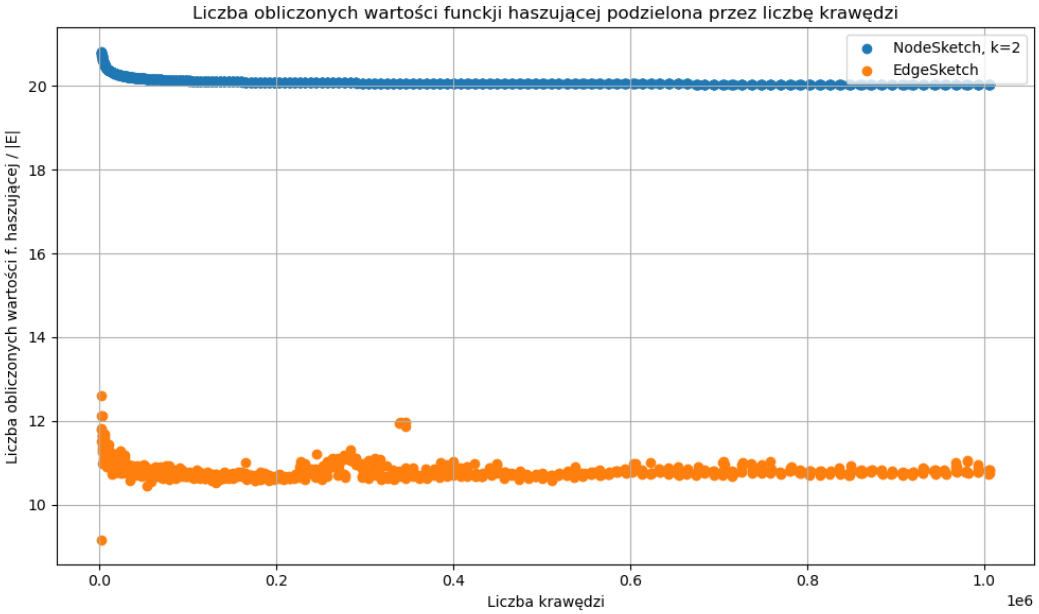
5.6.1. Liczba operacji w zależności od liczby krawędzi w grafie

Teoretyczna złożoność obliczeniowa algorytmów `NodeSketch` i `EdgeSketch` została skomentowana w rozdziale 4.2.1. Analiza ta obejmowała wszystkie kroki wykonywane w trakcie działania algorytmów. Jednak w praktyce, pewne operacje mogą mieć o wiele istotniejszy wpływ na ostateczny czas działania niż inne. W szczególności, przy dużej liczbie wywołań, koszt obliczenia wartości funkcji haszującej ma znaczący wpływ na czas działania algorytmu. W poniższym eksperymencie zbadano średnią liczbę obliczonych wartości funkcji haszujących w zależności od liczby krawędzi w grafie. W tym celu wygenerowano grafy w stochastycznym modelu blokowym o różnych rozmiarach, od 200 do 8000 wierzchołków. Wszystkie z nich miały 4 bloki, a prawdopodobieństwa krawędzi wewnątrz bloku i pomiędzy blokami wynosiły

odpowiednio 0.5 i 0.001. Podobnie jak w innych eksperymentach, rozmiar szkicu m wynosił 10 i przyjęto $\alpha = 0.3$. Wyniki przedstawiono na Rysunku 5.2. Dodatkowo, Rysunek 5.3 przedstawia te same wyniki, ale znormalizowane względem liczby krawędzi w grafie. Warto zaznaczyć, że w implementacji algorytmu `NodeSketch` zastosowano podobną optymalizację jak w algorytmie `EdgeSketch`, polegającą na obliczaniu wartości funkcji haszującej tylko dla niezerowych elementów w macierzy SLA.



Rys. 5.2: Liczba obliczonych wartości funkcji haszującej w zależności od liczby krawędzi w grafie



Rys. 5.3: Liczba obliczonych wartości funkcji haszującej znormalizowana względem liczby krawędzi w grafie

Wnioski

W przypadku obu algorytmów, liczba wywołań funkcji haszującej rośnie liniowo wraz z liczbą krawędzi w grafie. Jest to spodziewany wynik, ponieważ każda krawędź może

być wykorzystywana tylko przy generowaniu zanurzeń dla dwóch wierzchołków, które ją wyznaczają. Jest to także zgodne z teoretyczną złożonością obliczeniową algorytmu `EdgeSketch`. Zauważmy, że procedura `FastExpSketch` jest wywoływana $|V|$ razy, a średnia liczba przekazywanych do niej elementów to iloraz liczby krawędzi i liczby wierzchołków, a więc gęstość grafu

$$\rho = \frac{|E|}{|V|},$$

która w rozważanym modelu jest stała. Ostatecznie otrzymujemy więc:

$$O(|V|\frac{|E|}{|V|}\ln(\rho)) = O(|E|\ln(\rho)) = O(|E|)$$

JL:Wyjaśnić dokładniej skąd się bierze powyższa formuła

Wraz ze wzrostem parametru k, `NodeSketch` wykonuje więcej operacji ze względu na rekurencyjną procedurę szkicowania. Oczywiście `EdgeSketch` generuje zanurzenia tylko raz, co daje mu przewagę przy rozważaniu wyższych stopni sąsiedztwa. Jednak nawet dla $k = 2$, `EdgeSketch` wykonuje wyraźnie mniej wywołań funkcji haszującej niż `NodeSketch`.

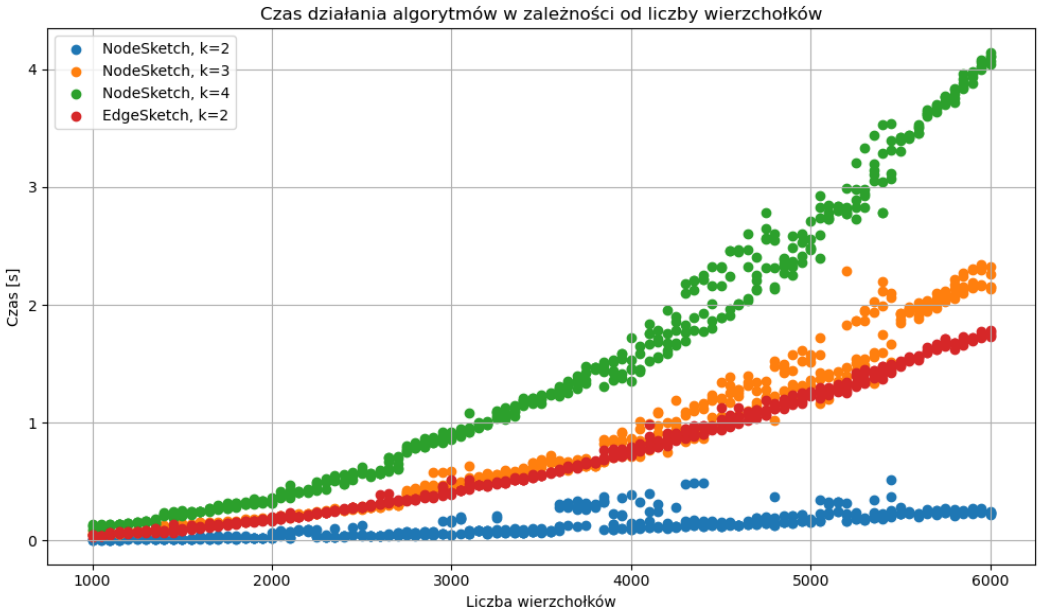
Na drugim wykresie widać ukazującym liczbę wywołań podzieloną przez liczbę krawędzi, widać, że stosunek ten jest rzeczywiście liniowy. Jedynie dla małych grafów odznaczają się nieco większe wartości. Wynika to z faktu, że algorytmy wykorzystują macierz SLA, a nie zwykłą macierz sąsiedztwa, a więc biorą pod uwagę sztuczną pętle od wierzchołka do siebie samego, nie ujętą w zaprezentowanej liczbie krawędzi. Dla algorytmu `NodeSketch` stosunek ten stabilizuję się na ok 20. Wynika to z faktu, że szkic ma rozmiar 10, a każdą z krawędzi rozważamy dla dokładnie dwóch wierzchołków. Wykorzystanie procedury `FastExpSketch` pozwala zredukować tę wartość prawie dwukrotnie, choć jest to okupione większą wariancją.

5.6.2. Rzeczywisty czas działania algorytmów

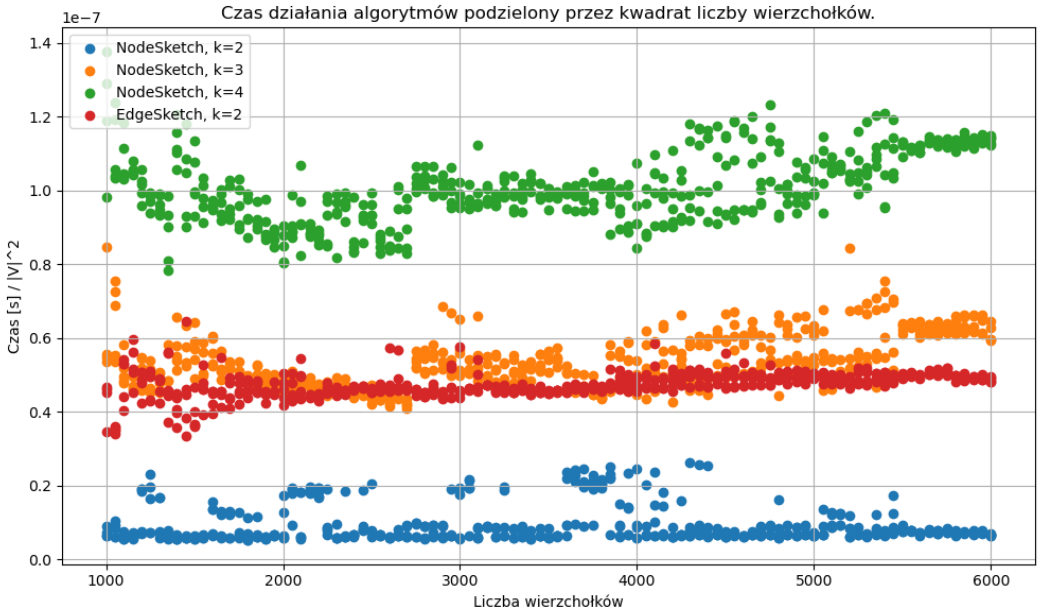
Asymptotyczna analiza złożoności obliczeniowej jest źródłem cennych informacji o efektywności algorytmów. Jednak w calu lepszego uchwycenia charakterystyki ich działania, przydatne jest eksperymentalne zbadanie czasu wykonania na rzeczywistych danych. Oczywiście dokładny czas działania zależy od konkretnej implementacji, a także wybranej platformy testowej. W przypadku niniejszego eksperymentu, testy przeprowadzono na komputerze z procesorem AMD Ryzen 7 5700X oraz 32GB pamięci RAM. Testowane grafy były generowane w modelu Erdosa-Renyiego z parametrem $p = 0.01$. Przeprowadzono dwa eksperymenty. Pierwszy z nich obejmował zasadniczą część obu algorytmów, czyli generowanie zanurzeń wierzchołków. Wyniki zilustrowano na wykresach 5.4 oraz 5.5. Na drugim z nich czas działania został podzielony przez kwadrat liczby wierzchołków, aby lepiej zobrazować złożoność obliczeniową. Drugi eksperyment zakładał dodatkowo obliczenie macierzy podobieństwa dla obu algorytmów, co lepiej odzwierciedla ich rzeczywiste zastosowanie oraz odpowiada przedstawionym w niniejszej pracy eksperymentom. Uzyskane wyniki przedstawiono na wykresach 5.6 oraz 5.7.

Wnioski

Eksperymenty pokazują, że algorytm `EdgeSketch` działa w ogólności wolniej od algorytmu `NodeSketch` dla tych samych wartości k. Może to wynikać ze stosunkowo wysokich kosztów operacji wykonywanych w ramach metody `FastExpSketch`, takich jak tworzenie i łączenie ze sobą łańcuchów znakowych, kopiowanie tablicy, czy praca z generatorem liczb pseudolosowych. W praktyce różnice nie były jednak bardzo znaczące. W przypadku właściwego algorytmu bez obliczania macierzy podobieństwa, `ExpSketch` osiąga podobny czas działania jak `NodeSketch`



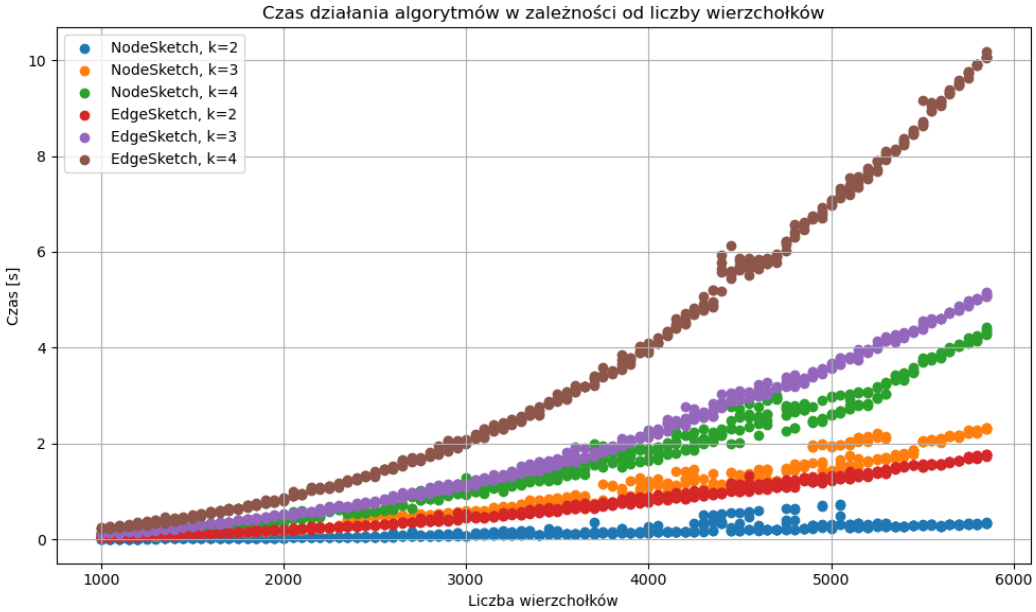
Rys. 5.4: Czas działania w sekundach dla obu algorytmów w zależności od liczby wierzchołków w grafie



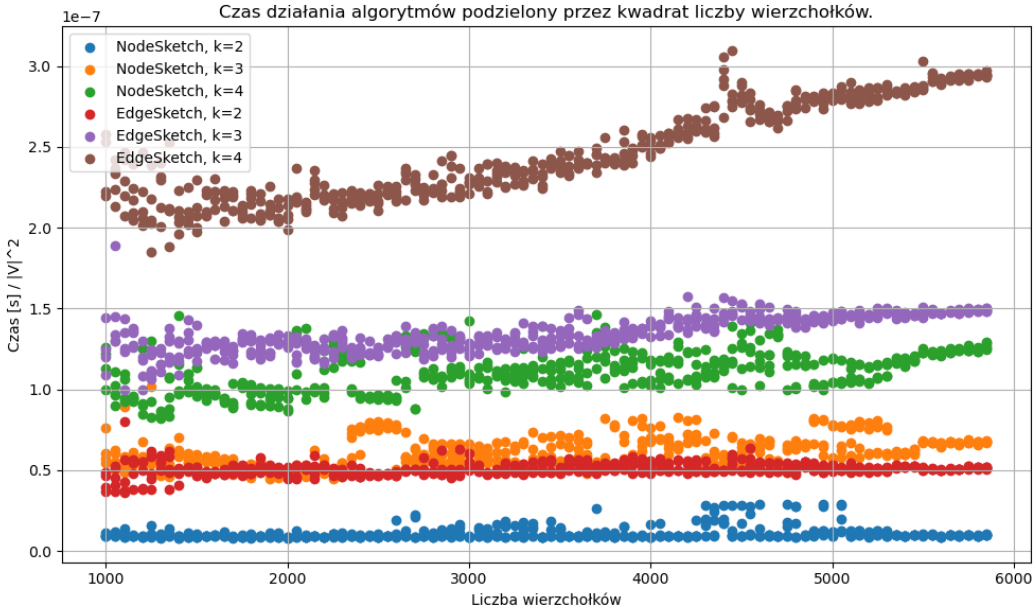
Rys. 5.5: Czas działania algorytmów znormalizowany względem kwadratu liczby wierzchołków w grafie

dla $k = 3$. Dla grafu o 6000 wierzchołków i około 180000 krawędzi, nie przekraczał on 2 sekund. Wykres 5.5 pokazuje, że złożoność dla obu algorytmów jest kwadratowa względem liczby wierzchołków.

Obliczanie macierzy podobieństwa okazuje się tanią operacją dla algorytmu `NodeSketch`, gdyż wykonywane jest tylko raz, w czasie kwadratowym. Podobnie jest w przypadku `EdgeSketcha`, dla $k = 2$. Koszt ten rośnie jednak wyraźnie dla wyższych wartości k. Jak pokazuje wykres 5.7, obliczanie macierzy podobieństwa wyższych rzędów ma w tej implementacji złożoność ponadkwadratową. Wynika to z zastosowanego schematu wyznaczania k-sąsiedztwa, który opiera się na potęgowaniu macierzy, odbywającym się w czasie $O(|V|^3)$. Potencjalna optymalizacja tego procesu może stanowić rozwinięcie nieniniejszej pracy.



Rys. 5.6: Czas działania w sekundach dla obu algorytmów w zależności od liczby wierzchołków w grafie, wliczając obliczanie macierzy podobieństwa



Rys. 5.7: Czas działania algorytmów, wliczając obliczanie macierzy podobieństwa, znormalizowany względem kwadratu liczby wierzchołków w grafie

Podsumowanie

W ramach niniejszej pracy zdefiniowano problem analizy wielkich grafów w modelu strumieniowym oraz omówiono praktyczne zastosowania takiego modelu. Dokonano także przeglądu istniejących algorytmów i struktur danych, które znajdują bądź mogą znaleźć swoje zastosowanie w analizie danych grafowych. Szczególną uwagę poświęcono metodom korzystającym ze szkiców danych oraz tworzącym zanurzenia wierzchołków, czyli ich kompaktowe reprezentacje w niskowymiarowych przestrzeniach wektorowych. Jednym z takich algorytmów jest `NodeSketch`, który szkicuje wierzchołki grafu, rekurencyjnie analizując ich k-sąsiedztwo. Stał się on punktem wyjścia do konstrukcji algorytmu `EdgeSketch`, łączącego główny schemat działania `NodeSketch` z metodą szkicowania wierzchołków znaną z algorytmu `FastExpSketch`. Pozwoliło to, między innymi, na efektywne wykonywanie operacji teoriomnogościowych na szkicach. Przeprowadzono teoretyczną analizę złożoności obliczeniowej, która została następnie zweryfikowana eksperymentalnie. Testy na zróżnicowanych zbiorach danych pokazały, że tak skonstruowany algorytm osiąga w większości wypadków lepszą precyzję przy rekonstrukcji krawędzi grafu niż `NodeSketch`. Wyniki te potwierdzają, że `EdgeSketch` jest skutecznym w praktyce algorytmem i stanowi wartościowy wkład badawczy w dziedzinę analizy dużych grafów.

Potencjalne kierunki rozwoju

Badanie technik generowania szkiców danych grafowych, takich jak te zastosowane do opracowania algorytmu `EdgeSketch`, oraz ich zastosowań jest niezwykle ciekawym i szerokim zagadnieniem, które trudno zamknąć w obrębie jednej pracy. Stąd też istnieje kilka potencjalnych obszarów, które mogą stanowić rozwinięcie niniejszej pracy.

Jednym z podstawowych kierunków rozwoju jest oczywiście dalsze badanie algorytmu `EdgeSketch`. Bardziej szczegółowe testy lub analizy teoretyczne mogłyby pozwolić na jeszcze lepsze zrozumienie, jaki wpływ na uzyskiwane wynika ma charakterystyka grafu i pozwolić na optymalizację doboru parametrów.

Ponadto, rekonstrukcja grafu poprzez obliczanie macierzy podobieństwa nie jest oczywiście jedynym zadaniem, które może służyć do oceny skuteczności algorytmów generujących zanurzenia wierzchołków. Innym ważnym obszarem badań może być zastosowanie stworzonych zanurzeń w uczeniu maszynowym, na przykład w zadaniach klasyfikacji wierzchołków.

Kolejną wartą uwagi kwestią jest możliwość połączenia techniki szkicowania znanej z algorytmu `FastExpSketch` ze schematem działania `SGSketch`. Algorytm ten działa na podobnej zasadzie, jak `NodeSketch`, ale z uwagi na efektywny mechanizm aktualizacji szkiców, może być stosowany do analizy dynamicznych grafów. Zastąpienie wykorzystanego w nim szkicowania przez `FastExpSketch` mogłoby potencjalnie polepszyć precyzję przy rekonstrukcji grafu.

Literatura

[1] K. Ahn, S. Guha, A. McGregor. Analyzing graph structure via linear measurements. Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms, strony 459–467, 01 2012.

[2] N. Alon, Y. Matias, M. Szegedy. The space complexity of approximating the frequency moments. Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96, 1996.

[3] N. Ashrafi-Payaman, M. R. Kangavari, S. Hosseini, A. M. Fander. Gs4: Graph stream summarization based on both the structure and semantics. The Journal of Supercomputing, 77(3):2713–2733, 2020.

[4] S. Cao, W. Lu, Q. Xu. Grarep. Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, Oct 2015.

[5] M. Chen, R. Zhou, H. Chen, H. Jin. Scube: Efficient summarization for skewed graph streams. 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS), 2022.

[6] M. Chen, R. Zhou, H. Chen, J. Xiao, H. Jin, B. Li. Horae: A graph stream summarization structure for efficient temporal range query. 2022 IEEE 38th International Conference on Data Engineering (ICDE), 2022.

[7] G. Cormode, S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. Journal of Algorithms, 55(1):58–75, Apr 2005.

[8] L. Devroye. Non-Uniform Random Variate Generation. Springer New York, 1986.

[9] M. Elkin, C. Trehan. Brief announcement: (1+ϵ)-approximate shortest paths in dynamic streams. Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing, 2022.

[10] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, J. Zhang. On graph problems in a semi-streaming model. Theoretical Computer Science, 348(2–3):207–216, 2005.

[11] I. J. Good. Diversity as a concept and its measurement: Comment. Journal of the American Statistical Association, 77(379):561, Sep 1982.

[12] X. Gou, L. Zou, C. Zhao, T. Yang. Fast and accurate graph stream summarization. 2019 IEEE 35th International Conference on Data Engineering (ICDE), 2019.

[13] A. Grover, J. Leskovec. Node2vec. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Aug 2016.

[14] W. L. Hamilton, R. Ying, J. Leskovec. Inductive representation learning on large graphs. CoRR, abs/1706.02216, 2017.

[15] R. W. Hamming. Error detecting and error correcting codes. Bell System Technical Journal, 29(2):147–160, Apr 1950.

[16] D. K. Hammond, P. Vandergheynst, R. Gribonval. Wavelets on graphs via spectral graph theory. Applied and Computational Harmonic Analysis, 30(2):129–150, Mar 2011.

[17] Z. Jiang, H. Chen, H. Jin. Auxo: A scalable and efficient graph stream summarization structure. Proceedings of the VLDB Endowment, 16(6):1386–1398, 2023.

[18] L. Katz. A new status index derived from sociometric analysis. Psychometrika, 18(1):39–43, Mar 1953.

[19] A. Khan, C. Aggarwal. Query-friendly compression of graph streams. 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), 2016.

[20] T. N. Kipf, M. Welling. Semi-supervised classification with graph convolutional networks. CoRR, abs/1609.02907, 2016.

[21] J. Ko, Y. Kook, K. Shin. Incremental lossless graph summarization. Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, Aug 2020.

[22] J. R. Lee, S. O. Gharan, L. Trevisan. Multiway spectral partitioning and higher-order cheeger inequalities. Journal of the ACM, 61(6):1–30, Dec 2014.

[23] J. Lemiesz. On the algebra of data sketches. Proceedings of the VLDB Endowment, 14(9):1655–1667, May 2021.

[24] J. Lemiesz. Efficient framework for operating on data sketches. Proceedings of the VLDB Endowment, 16(8):1967–1978, 2023.

[25] P. Li. 0-bit consistent weighted sampling. Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '15, strona 665–674, New York, NY, USA, 2015. Association for Computing Machinery.

[26] D. Lian, K. Zheng, V. W. Zheng, Y. Ge, L. Cao, I. W. Tsang, X. Xie. High-order proximity preserving information network hashing. Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, Jul 2018.

[27] Z. Ma, Y. Liu, Z. Yang, J. Yang, K. Li. A parameter-free approach to lossless summarization of fully dynamic graphs. Information Sciences, 589:376–394, Apr 2022.

[28] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski. Pregel. Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, Jun 2010.

[29] G. S. Manku, R. Motwani. Approximate frequency counts over data streams. Proceedings of the VLDB Endowment, 5(12):1699–1699, Aug 2012.

[30] T. Mikolov, K. Chen, G. Corrado, J. Dean. Efficient estimation of word representations in vector space, 2013.

[31] M. Ou, P. Cui, J. Pei, Z. Zhang, W. Zhu. Asymmetric transitivity preserving graph embedding. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Aug 2016.

[32] A. Pacaci, A. Bonifati, M. T. Özsu. Regular path query evaluation on streaming graphs. Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, May 2020.

[33] B. Perozzi, R. Al-Rfou, S. Skiena. Deepwalk. Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, Aug 2014.

[34] Python Software Foundation. Python Documentation, 2023. Dostęp: 2024-06-07.

[35] J. Qiu, Y. Dong, H. Ma, J. Li, K. Wang, J. Tang. Network embedding as matrix factorization. Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining, Feb 2018.

[36] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, G. Monfardini. The graph neural network model. IEEE Transactions on Neural Networks, 20(1):61–80, 2009.

[37] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, Q. Mei. Line. Proceedings of the 24th International Conference on World Wide Web, May 2015.

[38] N. Tang, Q. Chen, P. Mitra. Graph stream summarization. Proceedings of the 2016 International Conference on Management of Data, 2016.

[39] The Julia Language Development Team. The Julia Language Documentation, 2024. Dostęp: 2024-06-07.

[40] A. Tsitsulin, D. Mottin, P. Karras, E. Müller. Verse. Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW '18, 2018.

[41] D. Wang, P. Cui, W. Zhu. Structural deep network embedding. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Aug 2016.

[42] J. Wang, T. Zhang, J. Song, N. Sebe, H. T. Shen. A survey on learning to hash, 2017.

[43] R. J. Wilson. Introduction to graph theory. Prentice Hall, 2015.

[44] W. Wu, B. Li, L. Chen, C. Zhang. Efficient attributed network embedding via recursive randomized hashing. Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, Jul 2018.

[45] W. Wu, B. Li, C. Luo, W. Nejdl. Hashing-accelerated graph neural networks for link prediction. Proceedings of the Web Conference 2021, Apr 2021.

[46] R. S. Xin, J. E. Gonzalez, M. J. Franklin, I. Stoica. Graphx. First International Workshop on Graph Data Management Experiences and Systems, Jun 2013.

[47] D. Yang, B. Qu, R. Hussein, P. Rosso, P. Cudré-Mauroux, J. Liu. Revisiting embedding based graph analyses: Hyperparameters matter! IEEE Transactions on Knowledge and Data Engineering, 35(11):11830–11845, Nov 2023.

[48] D. Yang, B. Qu, J. Yang, L. Wang, P. Cudre-Mauroux. Streaming graph embeddings via incremental neighborhood sketching. IEEE Transactions on Knowledge and Data Engineering, strona 1–1, 2022.

[49] D. Yang, P. Rosso, B. Li, P. Cudre-Mauroux. Nodesketch. Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, Jul 2019.

[50] J. Zhang, Y. Dong, Y. Wang, J. Tang, M. Ding. Prone: Fast and scalable network representation learning. Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, Aug 2019.

[51] P. Zhao, C. C. Aggarwal, M. Wang. gsketch: On query estimation in graph streams. Proceedings of the VLDB Endowment, 5(3):193–204, 2011.

[52] D. Zhu, P. Cui, D. Wang, W. Zhu. Deep variational network embedding in wasserstein space. Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, Jul 2018.

Dodatek A

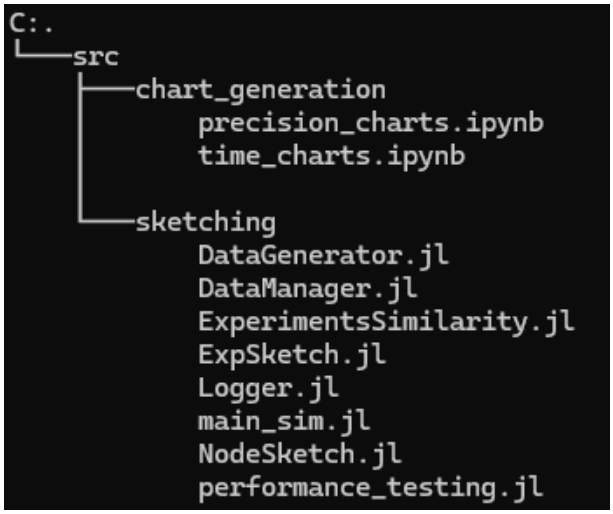
Instrukcja użytkowania biblioteki

Do pracy dołączona jest biblioteka, zawierająca implementacje omawianych algorytmów, jak również dodatkowe pliki umożliwiające odtworzenie przeprowadzonych eksperymentów. Kod został przetestowany na systemie Windows 11, z zainstalowanym językiem `Julia`[39] w wersji `1.10.2`. Dodatkowo, do stworzenia wykresów wykorzystano język `Python`[34] w wersji `3.11.9`.

Struktura plików

Część implementacyjna pracy zawarta jest w folderze `src`. Jego strukturę ukazuje rysunekA.1. Znajdują się w nim dwa główne foldery. Pierwszy z nich, `chart_generation` zawiera pliki `.ipynb` używane do rysowania wykresów na podstawie plików `.csv`. Drugi folder, `sketching` zawiera kod właściwej biblioteki. W jej skład wchodzą następujące pliki:

- `DataGenerator.jl` – zawiera kod do generowania losowych macierzy sąsiedztwa w różnych modelach.
- `DataManager.jl` – znajdujące się w nim metody pozwalają na wczytywanie i zapisywanie macierzy do plików `.mat`.
- `ExperimentsSimilarity.jl` – plik ten zawiera kod umożliwiający testowanie precyzji rekonstrukcji grafów.
- `ExpSketch.jl` – zawiera implementacje algorytmów `ExpSketch` i `FastExpSketch`.
- `Logger.jl` – pozwala na sterowaniem tym, w jaki sposób wypisywane są informacje na temat działania algorytmów.
- `main_sim.jl` – plik ten uruchamia wszystkie eksperymenty związane z badaniem precyzji rekonstrukcji grafu.
- `NodeSketch.jl` – znajduje się w nim właściwa implementacja algorytmów `NodeSketch` i `EdgeSketch`.
- `performance_testing.jl` – kod eksperymentu sprawdzającego czas działania algorytmów.



Rys. A.1: Struktura plików biblioteki.

Przykład użycia

Poniżej przedstawiono przykładowe użycie biblioteki wraz z wyjściem konsoli. Wszystkie komendy zostały w tym przypadku wykonane w interaktywnej konsoli języka `Julia`. Pierwszym krokiem jest wczytanie pliku `DataGenerator.jl`, i wygenerowanie grafu w modelu Erdosa-Renyiego o 6 wierzchołkach, prawdopodobieństwie krawędzi $p = 0.5$ oraz maksymalną wagą krawędzi równą 1.

```
julia> include("DataGenerator.jl")
generate_all (generic function with 1 method)

julia> matrix = generate_erdos_renyi_matrix(6, 0.5, 1)
6x6 Matrix{Float64}:
 0.0  0.0  0.0  0.0  1.0  0.0
 0.0  0.0  1.0  0.0  1.0  1.0
 0.0  1.0  0.0  1.0  1.0  1.0
 0.0  0.0  1.0  0.0  1.0  0.0
 1.0  1.0  1.0  1.0  0.0  1.0
 0.0  1.0  1.0  0.0  1.0  0.0
```

Następnie należy wczytać plik `NodeSketch.jl`. Poniżej ukazano także wywołania algorytmu `NodeSketch` dla wygenerowanej macierzy, z parametrami $k = 2$, $m = 4$ oraz $\alpha = 0.3$.

```
julia> include("NodeSketch.jl")
Main.NodeSketch

julia> NodeSketch.nodesketch(matrix, 2, 4, 0.3)
Main.NodeSketch.Sketch([5.0 2.0 ... 2.0 2.0; 5.0 6.0 ... 6.0 6.0; 5.0
    ↪ 2.0 ... 2.0 2.0; 1.0 6.0 ... 1.0 6.0], [1.0 0.0 ... 0.25 0.0; 0.0
    ↪  1.0 ... 0.75 1.0; ... ; 0.25 0.75 ... 1.0 0.75; 0.0 1.0 ... 0.75
    ↪  1.0], 96)
```

Kolejny przykład ilustruje wywołanie algorytmu `EdgeSketch` z tymi samymi parametrami. Zwraca on obiekt zawierający trzy pola: `embeddings` – macierz zanurzeń wierzchołków, `similarity_matrix` – macierz podobieństwa wierzchołków oraz `calculated_hashes`, czyli liczbę obliczonych wartości funkcji haszującej.

```
julia> res = NodeSketch.edgesketch(matrix, 2, 4, 0.3)
```

```
Main.NodeSketch.Sketch([0.023946932553343094 0.11321003816732249 ...
    ↪ 0.023946932553343094 0.8068069829529673; 0.20572686024856923
    ↪ 0.16361792227247998 ... 0.20572686024856923 0.16361792227247998;
    ↪ 0.2101007056250945 0.0060560405326058765 ...
    ↪ 0.0060560405326058765 1.4828069059435225; 0.10337951871171033
    ↪ 0.32929093744654353 ... 0.08786964958953339 0.3704157329572328],
    ↪ [1.0 0.0 ... 0.5 0.0; 0.0 1.0 ... 0.25 0.25; ... ; 0.5 0.25 ...
    ↪ 1.0 0.0; 0.0 0.25 ... 0.0 1.0], 85)

julia> res.embeddings
4x6 Matrix{Float64}:
 0.0239469   0.11321     0.103898    0.238676    0.0239469   0.806807
 0.205727    0.163618    0.957174    0.34143     0.205727    0.163618
 0.210101    0.00605604  0.145416    0.177179    0.00605604  1.48281
 0.10338     0.329291    0.00293567  0.00293567  0.0878696   0.370416
```