Politechnika Poznańska
Wydział Informatyki
Instytut Informatyki

Praca dyplomowa inżynierska

Implementacja algorytmu eksploracji danych z użyciem CUDA API

Marcin Jabłoński
Łukasz Kosiak
Piotr Kurzawa
Marek Rydlewski

Promotor:
dr inż. Witold Andrzejewski

Poznań, 2017 r.

„Coś się popsuło“
Zbigniew Stonoga

Spis treści

1 Wstęp

1.1 Wprowadzenie

Informatyzacja życia codziennego, jaka dokonała się w ostatnich latach sprawiła, że każdego dnia konsumenci często nieświadomie zostawiają po sobie wiele informacji na swój temat. Nawet z pozoru niewinne dane o ludzkich przyzwyczajeniach typu "z której półki bierzemy bułki w sklepie" są zapisywane w systemach informatycznych. Należy do tego oczywiście dodać inne usługi, które wybierane są przez użytkowników świadomie np. zapisywanie lokalizacji przez prywatny telefon komórkowy.

Wbrew pozorom, taka błaha na pierwszy rzut oka informacja może mieć jednak istotne znaczenie dla funkcjonowania przemysłu piekarskiego. Nic nie stoi na przeszkodzie, aby spróbować z tych danych odczytać preferencje bądź przyzwyczajenia przeciętnego Kowalskiego na temat jego codziennych zakupów, które mogą w przyszłości zaprocentować - zarówno dla właściciela, jak i klienta. Jest to oczywiście tylko przykład, ale oddaje doskonale fakt przydatności z pozoru nie mających znaczenia prostych czynności człowieka, jakie często przypadkiem rejestrują działające wokół konsumentów systemy.

Pozostaje jednak problem przetworzenia takich danych w celu otrzymania interesującej informacji, która byłaby potencjalnie użyteczna. Trzeba pamiętać, że rozmiar takich danych nierzadko sięga terabajtów i w praktyce skuteczna analiza takich danych przez człowieka nie jest możliwa. Musi on zatem w tym celu skorzystać z dobrodziejstw, jakie przynosi mu współczesna technologia.

Problem efektywnego przetwarzania zdążył urosnąć do rangi oddzielnego działu w informatyce. W pracy [1] zasugerowano utworzenie nowej dyscypliny mającej na celu opracowanie technik obliczeniowych rozwiązujących takie problemy, zwanej odkrywaniem wiedzy w bazach danych (ang. KDD – *Knowledge Discovery in Databases*). Techniki te mają na celu odnajdywanie prawidłowych, nietrywialnych i potencjalnie użytecznych wzorców w dużych zbiorach danych.

Wspominane wyżej techniki w dużej mierze zależą od rodzaju bazy, a ściślej mówiąc - charakteru danych w niej występujących. W przypadku danych zawierających informację o położeniu zazwyczaj mowa jest o odkrywaniu wiedzy w bazach danych przestrzennych (ang. *spatial data mining*). Takie systemy mogą zawierać atrybut lokalizacji obiektu w danym obszarze, jego opis w formie geometrycznej (np. w postaci wielokątów), a także inne atrybuty nieprzestrzenne. Okazuje się, że tradycyjne metody analizy danych przestrzennych zazwyczaj nie radzą sobie z nimi na tyle efektywnie, by było opłacalne ich użycie w praktyce [3], dlatego też zaczęto szukać nowych sposobów na odkrywanie wiedzy w takich bazach.

W pracy [4] zaproponowano odkrywanie *wzorców kolokacji przestrzennych* (lub krócej: *kolokacji*), czyli zbioru cech przestrzennych występujących w niewielkiej odległości od siebie. Łatwo to można sobie wyobrazić na przykładzie przyrody, gdzie osobniki (gatunki) o podobnych cechach zazwyczaj trzymają się razem. Rozumowanie to działa również dla bliższych współczesnemu człowiekowi cech przestrzennych, np. punktach o podobnej funkcji - stacje, kina, piekarnie, itd. Wraz z rosnącą popularnością obliczeń na kartach graficznych (w dużej mierze spowodowana wprowadzeniem technologii *CUDA* autorstwa firmy NVIDIA) pojawiło się wiele gotowych rozwiązań, pozwalających na efektywne wyszukiwanie kolokacji nawet w bardzo rozbudowanych bazach danych. Przegląd niektórych z nich można znaleźć w pracy [5].

Ostatni rok przyniósł kolejną metodę efektywnego przeszukiwania baz danych w

celu odnalezienia kolokacji [6]. Wykorzystuje ona autorski algorytm wyszukiwania maksymalnych klik w grafie rzadkim oraz skondensowane drzewa instancji przechowywaczce kliki instancji dla każdego kandydata do kolokacji (patrz Rozdział 2) w celu zmniejszenia czasu obliczeń oraz ograniczenia wymagań co do pamięci operacyjnej. Algorytm ten jest przedmiotem badań niniejszej pracy zbiorowej.

1.2 Cel i zakres pracy

Celem niniejszej pracy jest analiza wydajności zaproponowanych w pracy [6] rozwiązań z zakresu odkrywania kolokacji przestrzennych dla GPU i CPU.

Zakres pracy obejmuje następujące zadania szczegółowe:

1. **Zapoznanie się z literaturą.** Zapoznanie się z podstawowymi pojęciami dotyczącymi odkrywania danych w bazach danych przestrzennych oraz wyszukiwania wzorców kolokacji przestrzennych jest niezbędne do stworzenia działającej implementacji powyższego algorytmu. Dodatkowo należy zwrócić uwagę na dodatkowe zagadnienia związane z teorią grafów.

2. **Opracowanie wersji równoległej algorytmu eksploracji danych.** Konieczne jest przemyślenie wykorzystania algorytmów pomocniczych dla poszczególnych kroków całego rozwiązania oraz zaproponowanie możliwie najkorzystniejszego rozwiązania biorąc pod uwagę dostępną pamięć operacyjną, czas przetwarzania i przesyłania danych między pamięcią operacyjną a pamięcią karty graficznej.

3. **Implementacja wersji sekwencyjnej i równoległej ww. algorytmu.** Rozwiązanie podane w punkcie drugim powinno zostać zaimplementowane w technologii NVIDIA CUDA dla wersji GPU oraz biblioteki PPL w przypadku odmiany dla CPU.

4. **Przeprowadzenie eksperymentów wydajnościowych.** Analiza wyników testów wydajnościowych implementacji z punktu 3 jest głównym celem tej pracy. Należy zbadać efektywność obu rozwiązań pod względem czasu wykonywania oraz zapotrzebowania na dostępną pamięć.

1.3 Charakterystyka źródeł

Jak już wspomniano, niniejsza praca w dużej mierze opiera się o algorytm zaprezentowany w dokumencie [6]. Do jej opracowania była wymagana wiedza zawarta w innych źródłach, często również o charakterze naukowym.

Głównym źródłem wiedzy na temat kolokacji przestrzennych była rozprawda doktorska dr inż. Pawła Boińskiego [5], która w dużym przekroju omawia ideę kolokacji zaprezentowaną przez Shekhara i Huanga w pracy [4], a także prezentuje najpopularniejsze techniki ich odkrywania (metody *Co-location Miner*, *iCPI-tree*). Część rozwiązań wykorzystanych w tych technikach została wykorzystana w trakcie realizacji algorytmu.

Oddzielną kwestią jest literatura książkowa, wykorzystana do zapoznania się z technologią CUDA oraz przyjęcia dobrych praktyk optymalizacyjnych i programistycznych. Tutaj szczególnie należy wymienić popularną pozycję *CUDA w przykła-*

dach autorstwa Shane'a Cooke'a [7], a także *Professional CUDA C Programming* [8] będącą również podstawą do wstępu teoretycznego w rozdziale drugim.

1.4 Struktura pracy

W pracy przedstawiono główne pojęcia związane z wyszukiwaniem kolokacji przestrzennych oraz programowaniem równoległym na procesory graficzne i zawarto je w rozdziale 2. Rozdział 3 poświęcony jest algorytmowi będącemu głównym tematem pracy. Rozdział 4 opisuje implementację tego algorytmu w technologii CUDA, natomiast rozdział 5 prezentuje wyniki przeprowadzonych testów.

W tym miejscu zasadniczo będzie można napisać więcej, jeżeli już te rozdziały zostaną ustalone bądź wstępnie uzupełnione. Poza tym należy ustalić, czy w ogóle potrzebujemy takiego działu dla tak małej pracy. Z drugiej strony, zawsze to jednak te pół strony więcej spamu - borewicz. Dawaj ten spam - rydel

1.5 Podział pracy

Marcin Jabłoński w ramach niniejszej pracy wykonał projekt tego i tego, opracował .....

Łukasz Kosiak wykonał ....., itd.

2 Podstawy teoretyczne

2.1 Charakterystyka danych przestrzennych

2.1.1 Modelowanie danych przestrzennych

Sposób reprezentacji danych przestrzennej w dużej mierze zależy od zastosowań, niemniej najczęściej przybiera jedną z następujących form:

- *model pól* - ma formę funkcji, której dziedzina należy do modelowanej przestrzeni, a jego wynikiem jest cecha przestrzenna;
- *model obiektowy* - dla każdego zjawiska jest tworzony nowy obiekt z odpowiednimi właściwościami (etykietami, atrybutami przestrzennymi i nieprzestrzennymi).

W praktyce model pól używany jest przede wszystkim w metodach opartych na dokonywaniu pomiarów z powietrza - takie dane mają wtedy charakter rastrowy (reprezentacja w postaci pikseli). Model obiektowy stosowany jest natomiast w przypadkach, gdzie występuje duża liczba dodatkowych atrybutów nieprzestrzennych.

2.1.2 Źródła danych przestrzennych

Najogólniej źródła danych przestrzennych można podzielić ze względu na ich format.

Pierwotne źródła danych są opracowane w jednym ze standardowych formatów źródeł (najczęściej dla konkretnego systemu) i nie wymagają jakichkolwiek transformacji. Mają one zazwyczaj postać cyfrową i pochodzą z automatycznych pomiarów dokonanych przez specjalizowane systemy wyposażone w odbiorniki GPS czy tachimetry.

Wtórne dane źródłowe nie zostały zebrane z myślą o wykorzystaniu w systemach typu GIS i dlatego wymagają one odpowiedniej transformacji oraz cyfryzacji (jeżeli są one analogowe). Procedury te są one obarczone pewnym ryzykiem, ponieważ istnieje możliwość wystąpienia błędów w trakcie konwersji i w konsekwencji przekłamaniami w danych wynikowych, które należy ręcznie poprawić.

2.1.3 Relacje

Określenie zachodzących relacji między obiektami w źródłach danych przestrzennych jest ważnym elementem przetwarzania danych przestrzennych. Sposób ich określenia zależy od zastosowanego modelu danych.

W modelu pól relacje determinowane są przez operacje pól (ang. *field operations*, [13]), mogące przybierać różne formy w zależności od zastosowań, natomiast w modelu obiektowym rodzaje relacji przestrzennych zależą od definicji przestrzeni. Według standardu OGC istnieją trzy najpopularniejsze rodzaje związków przestrzennych między obiektami:

- *Relacje metryczne* - wyrażane w postaci predykatów typu "w odległości nie większej niż 10 metrów", oparte na odległości;

- *Relacje kierunkowe* - położenie określone jest względem globalnych kierunków dla przestrzeni (np. na północ, na południe - są to relacje bezwzględne) lub względem innego obiektu/obserwatora (nazywamy takie relacjami względnymi);

- *Relacje topologiczne* - najbardziej skomplikowane, wyrażone przez zależności typu pokrywanie, zawieranie, styczność.

W systemach typu GIS stosuje się głównie relacje topologiczne. Mają one postać predykatów przestrzennych dla operacji filtrowania i połączenia przestrzennego w językach zapytań działających na danych przestrzennych. Najczęściej wykorzystuje się je w tzw. *modelu dziewięciu przecięć* [14], za pomocą którego określa się możliwe relacje zachodzącą dla pary obiektów.

Dla każdego obiektu wyznacza się jego wnętrze, granicę i zewnętrze. Następnie, dokonuje się operacji przecięcia dla danej pary obiektów dla każdej z możliwych kombinacji elementów tego obiektu (np. granica pierwszego obiektu z wnętrzem drugiego). Takich relacji w dwuwymiarowej relacji można wyznaczyć osiem, należą do nich np. rozłączność, styczność, częściowe i całkowite pokrycie itd.

Istnieje również rozszerzenie modelu dziewięciu przecięć, zwanym DE-9IM (ang. *Dimensionally Extended nine-Intersection Model*, [15]), które rozróżnia rodzaj obiektu uzyskanego w wyniku przecięcia (mogą być puste, bezwymiarowe, jednowymiarowe i dwuwymiarowe).

2.2 Metody eksploracji danych przestrzennych

Specyfika danych przestrzennych, a w szczególności fakt, że własności obiektu w danych przestrzennych mogą zależeć od cech jego sąsiadów, powoduje, że stosowanie klasycznych metod eksploracji danych może doprowadzić do nieprawidłowych wyników [16] [17] - stąd też istnieje konieczność korzystania z metod eksploracji dedykowanych dla danych przestrzennych. Wiele z nich jest tak naprawdę rozwinięciem metod opracowanych dla klasycznych zbiorów danych.

2.2.1 Grupowanie przestrzenne

Metoda grupowania przestrzennego (ang. *spatial clustering*) zakłada istnienie przestrzeni m-wymiarowej, w której znajdują się punkty odpowiadające obiektom. Przestrzeń ta ma rozkład niejednorodny, a każdy z obiektów jest opisany przez m atrybutów. Celem grupowania jest poszukiwanie gęstych obszarów punktów używając miary Euklidesowej jako funkcji podobieństwa.

Grupowanie przestrzenne w największym stopniu spośród wszystkich metod eksploracji danych przestrzennych jest podobna do swojego klasycznego odpowiednika - wiele algorytmów grupowania opracowanych dla klasycznych zbiorów danych zadziała również dla danych przestrzennych.

2.2.2 Klasyfikacja przestrzenna

Klasyfikacja przestrzenna (ang. *spatial classification*) działa podobnie jak jego odmiana dla danych klasycznych - przewiduje klasy nowych obiektów w oparciu o tzw. zbiór uczący, składający się ze wcześniejszych obserwacji.

W celu dostosowania klasyfikacji dla danych przestrzennych zaproponowano [18] wykorzystanie *grafu sąsiedztwa* będącego reprezentacją relacji przestrzennych między obiektami, w którym wierzchołki stanowią obiekty przestrzenne, a relacje są krawędziami. Następnie w takim grafie wyznaczane są wszystkie ścieżki, których początkiem jest analizowany obiekt. Dalej analiza przebiega zgodnie z algorytmem *ID3* [19].

Z pojęciem klasyfikacji przestrzennej wiąże się także predykcja położenia (ang. *location prediction*), czyli przewidywanie zdarzeń we wskazanym miejscu w przestrzeni, przy uwzględnieniu autokorelacji przestrzennej. Przydaje się ono np. w określaniu regionów o wysokim ryzyku wystąpienia klęsk żywiołowych czy awarii.

2.2.3 Odkrywanie trendów przestrzennych

Trend przestrzenny definuje się [20] jako regularną zmianę co najmniej jednego atrybutu nieprzestrzennego obiektów wraz z oddalaniem się od innego obiektu. Odkrywanie trendów sprowadza się zazwyczaj do analizy regresji, gdzie odległość od danego obiektu jest zmienną niezależną, natomiast różnica wartości atrybutów do obserwacji - zmienną zależną.

Trendy dzieli się na globalne i lokalne. Pierwsze wskazują na zwiększanie (bądź zmniejszanie) wartości obserwowanych atrybutów przy rozpatrywaniu wszystkich obiektów znajdujących się na ścieżkach wychodzących z punktu początkowego. Typowym przykładem jest wzrost bezrobocia wraz z oddalaniem się od centrum miast. Trend lokalny jest reprezentowany przez pojedyncze ścieżki wykazujące inny kierunek zmian na danym atrybucie niż na sąsiednich ścieżkach.

2.2.4 Przypadki osobliwe w danych przestrzennych

Czasem w danych przestrzennych można znaleźć obiekty, których atrybuty nieprzestrzenne są niespójne z innymi obserwacjami dokonanymi w ich otoczeniu. Noszą one miano *przypadków osobliwych* [21].

Wyszukiwanie takich zjawisk jest trudne, szczególnie gdy istnieje więcej atrybutów nieprzestrzennych - odwzorowanie ich w n-wymiarowej przestrzeni może skutkować *przekleństwem wielowymiarowości* (ang. *curse of dimensionality*) [22], utrudnionym rozróżnianiem obiektów podobnych do siebie.

2.2.5 Asocjacje przestrzenne

Problem odkrywania asocjacji został pierwszy raz zdefiniowany w pracy [23] i w ogólności polega na analizie dostępnych transakcji (zbiorów obiektów, np. koszyka zakupów) oraz wykryciu występujących w nich regularności występowania elementów (typu: klient, który kupując bułki wybrał także masło).

Najczęściej reguły charakteryzuje się miarą *wsparcia* (ang. *support*) i *ufności* (ang. *confidence*). Pierwsza z nich wyraża stosunek występowania transakcji zawierającą lewą i prawą stronę reguły do ilości wszystkich transakcji. Ufność z kolei wskazuje na procentowy udział transakcji zawierających lewą i prawą stronę reguły we wszystkich transakcjach, które zawierają jego lewą stronę (jest to tzw. prawdopodobieństwo warunkowe). W celu ograniczenia ilości wykrytych wzorców wprowadzono także pojęcie *zbioru częstego* (ang. *frequent itemsets*) - zbioru elementów, dla

których wyznaczone wsparcie przekracza pewien ustalony przez użytkownika próg minimalnego wsparcia.

Z pracy [23] pochodzi także popularny algorytm wykorzystywany w wielu metodach odkrywania asocjacji i kolokacji, czyli metoda *Apriori*. Wykorzystuje on ważną cechę miary wsparcia, jaką jest *antymonotoniczność*. Wynika z niej, że zbiór może być zbiorem częstym tylko w przypadku, kiedy jego podzbiory są również zbiorami częstymi.

Na początku algorytmu generowane są jednoelementowe zbiory częste. Następnie iteracyjnie wykonywane są następujące kroki:

- tworzenie zbiorów częstych $(i+1)$-elementowych na podstawie zbiorów o długości i,
- filtrowanie zbiorów kandydujących w oparciu o miarę wsparcia,
- dodanie kandydatów do zbioru wynikowego.

Generowanie kandydatów polega na łączeniu wszystkich par zbiorów częstych o identycznych elementach początkowych, a następnie usuwaniu tych, które nie są zbiorami częstymi w oparciu o własność antymonotoniczności. Algorytm kończy się, gdy zbiór kandydatów będzie pusty.

W celu dostosowania metody odkrywania asocjacji do danych przestrzennych wprowadzono pojęcie *przestrzennej reguły asocjacyjnej* [24]. Zakłada ona istnienie predykatów przestrzennych (mogących wyrażać informacje o odległości czy kierunku), które mogą występować zarówno w części warunkującej (poprzedniku), jak w warunkowanej (następniku). Następnie podczas procesu odkrywania przestrzennych reguł asocjacyjnych dane umieszczone w ciągłej przestrzeni są zamieniane na zbiór transakcji. Metoda ta jest zaliczana do modelu zorientowanego na cechę referencyjną [5]. Istnieje też inne podejście, zwane *odkrywaniem zbiorów częstych klas sąsiadów*, opisane w pracy [25].

2.2.6 Kolokacje przestrzenne

Przedstawiony w pracy [4] problem *odkrywania przestrzennych reguł kolokacyjnych* powstał w odpowiedzi na niedoskonałości asocjacji (w szczególności konieczność wyboru cechy referencyjnej) i zakłada istnienie równorzędnych cech przestrzennych.

Praca wprowadza pojęcie *wzorca kolokacji przestrzennej* (zwanego także kolokacją przestrzenną lub krócej - kolokacją), zbioru cech przestrzennych, których instancje często występują we wzajemnym sąsiedztwie [5]. Stanowi on swego rodzaju odpowiednik zbiorów częstych w asocjacjach przestrzennych. Również miara wsparcia została zastąpiona przez *miarę powszechności* (ang. prevalence), które eliminują wymaganie wiedzy o transakcjach.

Kolokacje przestrzenne są przykładem modelu zorientowanego na zdarzenie (ang. *event-centric model*).

2.3 Odkrywanie kolokacji przestrzennych

Niniejszy rozdział zawiera opisy i definicje pojęć niezbędnych do zrozumienia algorytmu zawartego w rozdziale 3.

2.3.1 Cecha przestrzenna

Kluczową kwestią w procesie odkrywania kolokacji jest odpowiednia klasyfikacja obiektów występujących w bazie danych. Każdy zbiór danych przestrzennych, oprócz informacji o lokalizacji obiektu i opisujących go danych nieprzestrzennych powinien zawierać także właściwość pozwalającą na sklasyfikowanie danego obiektu do określonej klasy. Takie przypisanie nazywane jest cechą przestrzenną (ang. spatial feature) lub rzadziej klasą obiektu (ang. object class).

Jako typowy przykład cechy przestrzennej można podać etykietę przypisaną do obiektu na mapie (np. kościół, szkoła, strzelnica). Pozwala ona na jednoznaczne określenie własności przestrzeni w punkcie, gdzie znajduje się obiekt.

2.3.2 Podstawowe definicje

Definicja 1 (Instancja cechy przestrzennej) *Niech f będzie cechą przestrzenną. Mówimy, że obiekt x jest instancją cechy przestrzennej f, wtedy i tylko wtedy, gdy obiekt x jest typu f oraz jest opisany przez lokalizację i identyfikator.*

Definicja 2 (Wzorzec i instancja kolokacji) *Załóżmy F jako zbiór cech przestrzennych $F = \{f_1, f_2, ..., f_m\}$, a $FI = FI^{f_1} \cup FI^{f_2} \cup ... \cup FI^{f_m}$ niech będzie zbiorem ich instancji. Niech $>_F$ oznacza dowolną relację porządku zdefiniowaną dla zbioru F. Niech f_i oznacza i-tą cechę przestrzenną (ze względu na relację $>_F$), zatem $\forall i, j \in 1, ..., m \; f_i <_F f_j \Leftrightarrow i < j \wedge f_i, f_j \in F$. Mając daną relację sąsiedztwa R (zwrotną i przechodnią) mówimy, że wzorzec kolokacji przestrzennej (w skrócie "kolokacja") jest podzbiorem cech przestrzennych $c \subseteq F$, których instancje $I \subseteq FI$ tworzą klikę ze względu na relację R. Zbiór wszystkich instancji kolokacji przestrzennej c jest oznaczany przez CI^c. Przez długość kolokacji należy rozumieć liczbę elementów w zbiorze cech przestrzennych, który tworzy tę kolokację.*

Definicja 3 (Sąsiedztwo) *Mające daną zwrotną i symetryczną relację sąsiedztwa R, sąsiedztwem lokalizacji l nazywamy zbiór lokalizacji $L = \{l_1, l_2, ..., l_n\}$, gdzie l_i jest sąsiadem l, tzn. zachodzi $R(l, l_i)$ $\forall i \in 1, ..., n$.*

Przykład TODO (oprzeć na przykładzie chińczyków?) nk te przykłady

2.3.3 Miary kolokacji

Definicja 4 (Współczynnik uczestnictwa) *Współczynnik uczestnictwa (ang. participation ratio) cechy f i w kolokacji c jest równy procentowemu udziałowi wszystkich instancji cechy f i w instancjach kolokacji c:*

$$pr(f_i, c) = \frac{|\pi^{f_i}(CI^c)|}{FI^{f_i}} \tag{1}$$

gdzie $\pi^{f_i}(CI^c)$ oznacza projekcję relacyjną zbioru instancji CI^c względem cechy f_i (z usuwaniem duplikatów).

Definicja 5 (Indeks uczestnictwa) *Indeks uczestnictwa (ang. participation index) kolokacji c jest równy najmniejszemu ze współczynników uczestnictwa wyznaczonych dla każdej cechy przestrzennej $f_i \in c$:*

$$pi(c) = min_{f_i \in c} pr(f_i, c) \tag{2}$$

Indeks uczestnictwa najczęściej określany jest w literaturze mianem miary powszechności lub krótko powszechnością kolokacji.

Definicja 6 (Maksymalny wzorzec kolokacji przestrzennej) *Niech będzie dana wartość min_prev oznaczająca pewien minimalny próg powszechności. Jeżeli* $c = \{f_1, ..., f_m\}$ *jest kolokacją powszechną (tzn.* $pi(c) \geqslant min\_prev$*) i nie istnieje żaden nadzbiór c taki, że powszechność dla tego nadzbioru jest równa co najmniej min_prev, kolokacja c nazywana jest kolokacją maksymalną.*

2.3.4 Problem

Definicja 7 (Reguła kolokacyjna) *Reguła kolokacyjna to reguła postaci* $c_1 \rightarrow c_2(p, cp)$*, gdzie* $c_1 \subseteq F$*,* $c_2 \subseteq F$ *i* $c_1 \cup c_2 = \emptyset$*. Potencjalna użyteczność reguły może być mierzona przy pomocy jej powszechności p oraz prawdopodobieństwa warunkowego cp.*

Definicja 8 (Prawdopodobieństwo warunkowe) *Prawdopodobieństwem warunkowym* $cp(c_1, c_2)$ *reguły kolokacyjnej* $c_1 \rightarrow c_2$ *nazywamy stosunek liczby instancji wzorca c 1 w sąsiedztwie instancji wzorca* c_2 *do liczby wszystkich instancji wzorca* c_1*:*

$$cp(c_1, c_2) = \frac{|\pi^{c_1}(CI^{c_1 \cup c_2})|}{CI^{c_1}} \tag{3}$$

gdzie $\pi^{c_1}(CI^{c_1 \cup c_2})$ *oznacza projekcję relacyjną instancji wzorca* $CI^{c_1 \cup c_2}$ *względem wzorca* c_1 *(z usuwaniem duplikatów).*

Definicja 9 (Problem odkrywania kolokacji) *Problem odkrywania kolokacji przestrzennych jest zdefiniowany w następujący sposób. Mając dane:*

- *zbiór cech przestrzennych* $F = \{f_1, f_2, ..., f_m\}$
- *zbiór obiektów* $FI = FI^{f_1} \cup FI^{f_2} \cup ... \cup FI^{f_m}$ *, gdzie* $FI^{f_i}, (0 < i \leqslant m)$ *jest zbiorem instancji cechy* f_i*, przy czym każda instancja jest opisana przez lokalizację i identyfikator,*
- *symetryczną i zwrotną relację sąsiedztwa R,*
- *próg minimalnej powszechności min_prev oraz próg minimalnego prawdopodobieństwa warunkowego min_cond,*

znajdź wszystkie poprawne reguły kolokacyjne z powszechnością nie mniejszą niż min_prev i prawdopodobieństwem warunkowym nie mniejszym niż min_cond.

2.4 Przegląd algorytmów odkrywania wzorców kolokacji przestrzennych

W tym podrozdziale zostaną zaprezentowane skrótowo najważniejsze algorytmy odkrywania kolokacji przestrzennych.

2.4.1 Co-location Miner

Wraz z wprowadzeniem pojęcia kolokacji autorzy pracy [4] zaprezentowali także podstawowy obecnie algorytm rozwiązujący problem odkrywania wzorców kolokacji przestrzennych, zwany *Co-location Miner*. W algorytmie tym wyróżnia się następujące fazy:

- generowanie kandydatów na kolokacje przestrzenne (o długości i),
- wyznaczanie instancji dla wygenerowanych kandydatów,
- usuwanie kandydatów, których powszechność wynosi mniej niż przyjęty próg minimalnej powszechności.

Pozostali kandydaci trafiają do zbioru wynikowego, a następnie na ich podstawie są tworzone reguły kolokacyjne. Same reguły również podlegają filtracji - usuwane są te reguły, których prawdopodobieństwo warunkowe jest poniżej określonego progu.

W następnej iteracji algorytm wykonuje dokładnie te same kroki, przy czym generowani kandydaci są o długości o jeden większej. Całość kończy się, gdy nie jest możliwe już wygenerowanie nowych kandydatów.

2.4.2 Multiresolution Co-location Miner

Korzystanie z oryginalnego algorytmu *Co-location Miner* wiąże się niestety z dużymi kosztami obliczeniowymi, głównie ze względu na pracochłonny krok generowania kandydatów na kolokacje. Dlatego też niedługo później w pracy [9] autorzy zaproponowali drobną modyfikację oryginalnego algorytmu, dodając dodatkowy krok filtrowania w opraciu o przybliżoną reprezentację zbioru wejściowego.

W algorytmie *Multiresolution Co-location Miner* zbiór wejściowy zostaje podzielony na obszary (mniejsze fragmenty). Zanim rozpocznie się faza wyznaczania instancji dla wygenerowanych kandydatów, nastepuje szacowanie ich powszechności na podstawie sąsiadujących instancji cech przestrzennych w ramach obszarów. W przypadku zbyt niskiej wartości szacowanej powszechności kandydata, można go wykluczyć z dalszego przetwarzania i tym samym oszczędzić zasoby niezbędne na wyznaczenie jego instancji.

Dalsze kroki przebiegają identycznie jak w przypadku oryginalnego *Co-location Miner*.

2.4.3 Joinless

Celem autorów pracy [10] było stworzenie algorytmu, który omijałby konieczność tworzenia kosztownych połączeń przestrzennych na etapie wyznaczania instancji kandydatów na kolokacje (tak jak np. w przypadku rodziny algorytmów *Co-location Miner*). Nosi on nazwę algorytmu bezpołączeniowego (ang. *joinless*).

Główną różnicą w porównaniu do wcześniejszych algorytmów jest sposób generowania instancji kolokacji. Są one generowane na podstawie sąsiedztw typu gwiazda - zbiorów obiektów, w którego skład wchodzi rozpatrywany obiekt oraz jego sąsiedzi posiadający większą cechę przestrzenną. Wyznacza się je na podstawie oddzielnych algorytmów (np. *plane sweep*), lub korzysta z specjalnych struktur ułatwiających wykrywanie sąsiadów typu *R-drzewo*.

Wygenerowane instancje muszą zostać dodatkowo zweryfikowane (poprawne instancje powinny być kliką, czego nie gwarantuje sąsiedztwo typu gwiazda), a następnie - podobnie jak w algorytmie *Multiresolution Co-location Miner* - dokonuje się ich wstępnego filtrowania pod kątem progu minimalnej powszechności.

2.4.4 iCPI-tree

Drzewo iCPI (*improved Co-location Pattern Instance*, [12]) stanowi zmodyfikowaną odmianę drzewa CPI zawartego w pracy [11]. Struktura ta zawiera informacje o wszystkich zachodzących relacjach sąsiedztwa.

iCPI-tree posiada następującą strukturę:

- Poziom 1 - korzeń drzewa (oznaczony etykietą *NULL*),
- Poziom 2 - cechy elementów centralnych, czyli cechy przestrzenne obiektów centralnych *sąsiedztw typu gwiazda*;
- Poziom 3 - instancje elementów centralnych, dla których ma zostać przechowana informacja o sąsiadach;
- Poziom 4 - cechy sąsiadów,
- Poziom 5 - instancje sąsiadów.

Sąsiedzi uporządkowani są według rosnącej cechy przestrzennej, a w przypadku instancji tej samej cechy - zgodnie z rosnącym identyfikatorem. Takie uporządkowanie nosi nazwę *uporządkowanego zbioru sąsiadów*.

Powyższa struktura drzewiasta jest wykorzystana w algorytmie w celu generowania instancji coraz dłuższych kandydatów w kolejnych iteracjach. Dokonuje się tego poprzez systematyczną ich rozbudowę o kolejne elementy. Na początku wszystkie instancje są jednoelementowe, a w kolejnych iteracjach są one rozbudowywane poprzez wyszukiwanie sąsiadów z odpowiednią cechą i weryfikowane (należy sprawdzić, czy nowo dodany obiekt do instancji jest sąsiadem każdego z obiektów należących do tej instancji).

Pozostałe kroki algorytmu (generowanie kandydatów i reguł, filtrowanie według powszechności) są podobne jak w metodach *Co-location Miner* i *joinless*.

3 Przetwarzanie równoległe na kartach graficznych

Dziedzina informatyki, jaką są obliczenia ogólnego przeznaczenia na układach GPU (ang. *general-purpose computing on graphics processing units*, w skrócie *GPGPU*) należy do stosunkowo świeżych technik programowania - jej właściwy początek można datować na okolice 2009 roku. Jeszcze do niedawna wśród programistów panowało przekonanie, że karty graficzne powinny być odpowiedzialne tylko za rysowanie obrazu, a cała odpowiedzialność za niezbędne obliczenia powinna padać na CPU, jako "serce" komputera. Były to czasy, kiedy pojęcie równoległego wykonywania zadań nie było szeroko rozpowszechnione - rdzenie procesora były jedynie nowinką, a cały rozwój procesorów CPU szedł w zwiększanie częstotliwości taktowania.

Szybko się okazało, że nie da się tego robić w nieskończoność. Podnoszenie częstotliwości spowodowało, że procesory zaczęły pobierać spore ilości energii, a także wydzielać ogromne ilości ciepła, którego nie dało się okiełznać bez korzystania z specjalnego chłodzenia. Rozwiązaniem okazało się skorzystanie z wielordzeniowości - wielu procesorów połączonych ze sobą specjalnymi magistralami, zamkniętymi w jednej obudowie. Wymagało to także zmiany podejścia do programowania. Równoległość daje olbrzymie możliwości, o ile potrafi się z nich skorzystać w odpowiedni sposób - bez tego programista nie uzyska zauważalnego wzrostu szybkości obliczeń, jakie dają współczesne układy wielordzeniowe.

Wielu osobom w trakcie tej "rewolucji" umknął jeden drobny fakt - kiedy swoje triumfy świecił słynny procesor *Pentium 4* firmy *Intel* o kosmicznej wtedy częstotliwości taktowania rzędu 3 GHz, na rynku istniały już rozwiązania równoległe, na których można było uzyskać znacznie lepsze wyniki. Były to oczywiście karty graficzne, które od dawna działały w sposób równoległy.

Dopóki nie powstały pierwsze specyfikacje bibliotek wspierających obliczenia na kartach graficznych (pierwsza była *CUDA* firmy *NVIDIA*), programiści próbowali wykorzystywać potencjał brzmiący w procesorach graficznych w oparciu o rendering 3D. Było to dosyć karkołomne zadanie i nie każdy algorytm można było rozwiązać w ten sposób. Karty graficzne musiały odczekać jeszcze parę lat, aby dało się wykorzystać w pełni ich możliwości obliczeniowe.

3.1 CPU a GPU

Jak już wspomiano, procesory graficzne, mające w pierwotnym założeniu wyłącznie wspomagać generowanie obrazu, od początku były projektowane jako układy przetwarzajace dane w sposób równoległy. Karty graficzne posiadają znacznie więcej rdzeni niż procesory GPU, mają one jednak mniejsze taktowanie, a także są gorzej wyposażone (w szczególności brakuje im rozszerzeń typowych dla CPU typu SSE2 czy MMX). W związku z tym, nie wszystkie algorytmy nadają się dobrze do implementacji w środowisku GPU. Jeżeli jednak istnieje możliwość zrównoleglenia jakiegoś problemu, to wyniki osiągnięte na GPU będą lepsze niż te uzyskane na procesorze CPU (niezależnie od tego, czy jest napisany sekwencyjnie czy równolegle). Obecne procesory CPU z kolei potrafią działać sekwencyjnie, jak i równolegle, za pośrednictwem wielu mechanizmów wspierających równoległość (np. wątki, mutexy, wsparcie dla specjalnych struktur).

Oddzielną kwestią jest przydział tranzystorów tworzących poszczególne układy do pełnienia poszczególnych funkcji. W przypadku CPU większość tranzystorów sta-

nowi pamięć podręczną oraz układy sterowania - za wykonywanie obliczeń jest odpowiedzialna ok. 1/4 tranzystorów wbudowanych w procesor. Natomiast w przypadku GPU gros tranzystorów wykorzystanych jest do budowy jednostek arytmetyczno-logicznych oraz zmiennoprzecinkowych. Z tego powodu duża część zadań związana z przepływem danych, uruchamianiem wątków czy podziałem zadań na GPU spoczywa na programiście, co dosyć mocno komplikuje kod.

3.2 Wprowadzenie do technologii CUDA

Spośród wszystkich technologii umożliwiających przeprowadzanie obliczeń na kartach graficznych zdecydowanie największą popularność osiągnęła pierwsza z nich, czyli CUDA (ang. *Compute Unified Device Architecture*). Została ona zaprezentowana po raz pierwszy w 2007 roku przez firmę NVIDIA i jest dedykowana wyłącznie produktom tej firmy (w porównaniu do konkurencyjnej technologii OpenCL). Do jej największych zalet z pewnością należy bardzo dobra dokumentacja oraz dostępność na wszystkie najważniejsze platformy systemowe (Windows, Linux, Mac OS X).

3.2.1 Architektura

Procesor graficzny kompatybilny z CUDA zbudowany jest z wielu multiprocesorów strumieniowych (ang. *streaming multiprocessors*, w skrócie SM). Na każdy z nich składa się określona ilość rdzeni CUDA, które mogą przetwarzać dane w sposób równoległy - w każdym z nim jest umieszczona dodatkowo jednostka arytmetyczno-logiczna (ALU, umożliwia obliczenia na liczbach całkowitych) oraz jednostka zmiennoprzecinkowa (FPU). Do tego w każdym *SM* znajdują się jednostki funkcji specjalnych - w nich wyznaczane są wartości funkcji matematycznych typu pierwiastkowanie czy funkcje trygonometryczne. Liczba powyższych elementów nie jest stała - zależą one od miary *potencjału obliczeniowego CUDA* (ang. *CUDA Compute Capability*), który obsługuje dany sprzęt.

Każdy *SM* zawiera 16 jednostek wejścia i wyjścia, mających na celu pośredniczenie w przekazywaniu danych między różnymi typami pamięci. Przez nie odbywa się transfer danych do pamięci RAM karty graficznej (zlokalizowanej poza procesorem graficznym). W multiprocesorze są dostępne następujące typy pamięci podręcznej:

- pamięć rejestrów - najszybsza pamięć, w której wątek może przechowywać swoje zmienne,
- pamięć wspólna (ang. *shared memory*) - pamięć, poprzez którą może odbywać się wymiana danych między wątkami;
- pamięć podręczna pierwszego poziomu (ang. *L1 cache*).

Oprócz tego multiprocesor wyposażony jest w układy sterujące wywołaniem instrukcji (ang. *Instruction Dispatch Unit*) odpowiadające za wykonywanie obliczeń na zgrupowanych rdzeniach.

3.2.2 Programowanie wielowątkowe w języku CUDA C

Środowisko NVIDIA CUDA umożliwia programowanie w języku CUDA C - stanowi on podzbiór języka C++ i jest z nim w dużym stopniu kompatybilny. Kod

w języku CUDA C da się łączyć z standardowym kodem C++ kompilowanym pod procesory CPU - w tym przypadku fragmenty kodu wywoływane po stronie karty graficznej są kompilowane przez specjalny kompilator NVCC dołączany do środowiska CUDA.

Funkcje wywoływane przez hosta, a wykonywane przez procesor graficzny w sposób równoległy przez wiele wątków są nazywane funkcjami jądra (ang. kernel). Wszystkie wątki uruchomione poprzez wywołanie jądra wykonują identyczny kod. Oczywiście da się je odróżnić poprzez unikalny identyfikator dostępny w specjalnej zmiennej. Zgodnie z modelem przetwarzania SIMD, na jednym multiprocesorze uruchomione wątki są grupowane w tzw. *warpy* (ang. *warps*) - zbiory 32 wątków wykonujących tą samą instrukcję.

Wątki można organizować w większe grupy. Podstawową grupą wątków jest *blok*, w ramach którego można uformować wątki w jednym, dwóch, a nawet trzech wymiarach. Bloki grupowane są z kolei w siatkę obliczeniową (ang. *computation grid*), która również może składać się z trzech wymiarów. Przy wywołaniu kernela programista określa liczbę równoległych bloków oraz ilość wątków na blok, za pomocą których urządzenie będzie przetwarzać równolegle kernel. Odpowiednie ustawienie tych wartości jest kluczowe dla wydajności przetwarzania - zbyt mała/wielka ilość przyznanych bloków może spowodować odwrotny skutek do zamierzonego. Maksymalne ilości wątków na blok i bloków na siatkę określa miara *potencjału obliczeniowego CUDA*.

3.2.3 Model pamięci

Karty graficzne obsługujące technologię CUDA posiadają następujące typy pamięci, dostępne dla programisty:

- *rejestry* - zdecydowanie najszybsza i najwygodniejsza pamięć, umieszczona w procesorze graficznym, mała, ograniczona żywotność do czasu życia wątku;
- *pamięć lokalna* - umiejscowiona w pamięci karty graficznej, wolna, będąca alternatywą do rejestrów;
- *pamięć wspólna* - dostęp do niej możliwy jest ze wszystkich wątków pracujących w ramach jednego bloku, mała (umiejscowiona w chipie GPU), lecz o szybkim czasie dostępu;
- *pamięć globalna* - duża (rzędu paru gigabajtów), wolna, o dużym czasie dostępu, dostęp z poziomu każdego wątku oraz hosta;
- *pamięć stała* - dostępna dla wszystkich wątków w trybie tylko do odczytu, buforowana, posiada unikalną możliwość rozgłaszania danych (ang. *broadcast*);
- *pamięć tekstur* - również tylko do odczytu, przeznaczona z myślą o przechowywaniu danych w postaci macierzy.

3.2.4 *Thrust*

Biblioteka *Thrust* stanowi odpowiednik popularnej *Standard Template Library* dla języka CUDA C. Od wersji 1.4.0 jego wersja produkcyjna jest dołączana wraz z środowiskiem *CUDA Toolkit 4.0*.

Podobnie jak jego odpowiednik dla języka C++, Thrust zawiera bogatą kolekcję wbudowanych algorytmów i operacji dostosowanych do równoległego środowiska CUDA. W szczególności dostarcza równoległe operacje skanowania, sortowania czy redukcji, zwalniając programistę od implementowania własnych, potencjalnie niewydajnych rozwiązań.

Thrust zawiera struktury znane z biblioteki STL, takie jak wektory (w wersji dla hosta oraz karty graficznej), iteratory oraz wiele innych typów generycznych ułatwiających korzystanie z możliwości dostarczanych przez środowisko CUDA C. Użycie ich jest analogiczne jak w "tradycyjnym" STL-u.

Projekt będący wynikiem tej pracy wykorzystuje CUDA Toolkit 8.0 wraz z biblioteką Thrust 1.8.

4 Algorytm

Niniejszy rozdział ma za zadanie przybliżenie algorytmu będącego tematem tej pracy - metody odkrywania maksymalnych kolokacji przestrzennych w oparciu o graf rzadki i skondensowane drzewo instancji (ang. *sparse-graph and condensed tree-based maximal co-location algorithm*) przedstawionej w pracy [6].

Poszczególne kroki pierwotnego algorytmu *SGCT* zostaną opisane w kolejnych podrozdziałach. Szczegóły implementacji wraz z zaproponowanymi usprawieniami znajdują się w Rozdziale 4.

4.1 Generowanie tabeli instancji kolokacji o rozmiarze 2

Pierwszy krok algorytmu jest podobny do metody *Co-location Miner* i polega na wygenerowaniu 2-elementowych kandydatów na kolokacje.

Kolokacje o rozmiarze 2 tworzone są na podstawie wygenerowanych w oparciu o cechy przestrzenne jednoelementowych kolokacji. Nie jest do tego wykorzystywana jednak metoda *Apriori*, ponieważ udowodniono w pracy [4], że dla kandydatów dwuelementowych lepszą wydajność można uzyskać w oparciu o algorytm *spatial join*. Wykorzytany został zatem algorytm *sweeping-based spatial join* [26] z dodatkową modyfikacją, usuwającą pary instancji o tej samej cesze przestrzennej.

Przykład 1 *Przykładowe zapytanie tworzące kandydatów na kolokacje o rozmiarze 2 przestawia się następująco:*
select p', p''
from $\{p_1, ..., p_{12}\}p'$, $\{p_1, ..., p_{12}\}p''$
where $p'.feature \neq p''.feature$, p'‘p'', $(p', p'') \in R$

Na podstawie wygenerowanych kolokacji oraz tzw. *progu odległości* (ang. *distance threshold*) tworzona jest dwuwymiarowa tablica z haszowaniem (ang. *hash table*). Jest ona indeksowana cechami przestrzennymi. Każdy element tablicy zawiera wskaźnik do listy zawierającej instancje kandydatów o odpowiadających indeksom cechach przestrzennych. Instancja zostanie dodana do tej listy tylko wtedy, gdy odległość między instancjami nie przekracza dopuszczalny próg odległości między nimi.

Przykład 2 *Zapytanie* $InsTable_2(A, B)$ *zwróci listę kandydatów* (A_2, B_2), (A_3, B_1), *itd. Nie znajdziemy na tej liście pary* (A_2, B_{100000}), *gdyż odległość między tymi instancjami przekracza dopuszczalny próg odległości.*

4.2 Obliczanie miary powszechności

Również krok obliczania powszechności dla kandydatów nie różni się od tego znanego z *Co-location Miner*.

Dla każdego kandydata w tabeli wyliczany jest współczynnik uczestnictwa (ang. *participation index*). Dokonuje się tego poprzez wybieranie wszystkich unikalnych instancji cech przestrzennych, która są ujęte w danej kolokacji. Następnie zgodnie z definicją miary powszechności z tabeli instancji są usuwani kandydaci, dla których obliczona miara powszechności jest mniejsza niż zadany próg minimalnej powszechności *min_prev* [4].

4.3 Generowanie kandydatów na kolokacje maksymalne

Krok ten wprowadza nową strukturę, zwaną *grafem kolokacji o rozmiarze 2* (ang. *size-2 co-location graph*). Jego definicja brzmi następująco:

Definicja 10 (Graf kolokacji o rozmiarze 2) *Jeżeli przyjąć relacje sąsiedztwa między kolokacjami o rozmiarze 2 jako krawędzie $E = \{e_1, ..., e_u\}$, a cechy przestrzenne występujące w kolokacjach jako wierzchołki $V = \{v_1, ..., v_\lambda\}$, gdzie u i λ są odpowiednio liczbą krawędzi i liczbą wierzchołków, to graf kolokacji o rozmiarze 2 można zamodelować jako graf nieskierowany $G = (V, E)$, przechowywany w listowej strukturze danych uporządkowanej rosnąco. Zbiór N jest zbiorem sąsiedztw wierzchołka i definiuje się go następująco:*

$$N(v_i) = \{W | v_i, w \in E\} \tag{4}$$

Zadaniem tego kroku jest wyszukanie w takim grafie maksymalnych klik, określanych jako *kandydaci na maksymalne kolokacje* (ang. *candidate maximal co-location*).

Definicja 11 (Kandydat na maksymalną kolokację) *Kandydat na maksymalną kolokację C_m składa się z uporządkowanych cech przestrzennych o następujących właściwościach: każda para cech w C_m jest ze sobą połączona krawędzią, a żadne dodatkowe cechy nie mogą być dodane do C_m bez zachowania ich kompletnego połączenia.*

Autorzy pracy [6] udowodnili, że graf kolokacji o rozmiarze 2 można traktować jako graf rzadki. Umożliwia to efektywne korzystanie z algorytmu Brona-Kerboscha [28] do wyszukiwania maksymalnych klik w grafie nieskierowanym. Wprowadzone zostały do niego pewne modyfikacje uwzględniające rozproszenie grafu oraz wybieranie *pivotu* w celu usprawienia wyszukiwania kandydatów na kolokacje.

Wejście: $G = (E, V)$
Wyjście: CP_m
1 $CP_m \leftarrow \emptyset; X \leftarrow \emptyset; P \leftarrow \emptyset;$
2 **foreach** v_i^* *in degeneracy ordering* $v_1^*, v_2^*, ..., v_\lambda^*$ **do**
3 $\quad P \leftarrow N(V_i^*) \cup \{v_{i+1}^*, ..., v_\lambda^*\};$
4 $\quad X \leftarrow N(V_i^*) \cup \{v_1^*, ..., v_{i-1}^*\};$
5 $\quad BK\_Pivot(P, \{v_i^*\}, X);$
6 **end**
7 **Procedure** `BK_Pivot(M,K,T)`
8 $\quad$ **if** $M \cup T = \emptyset$ **then** $\{CP_m \leftarrow CP_m \cup K\};$
9 $\quad$ wybór punktu pivot $u \in M \cup T$; % do maksymalizacji $|M \cap N(u)|$;
10 $\quad$ **foreach** $v_i \in M \backslash N(u)$ **do**
11 $\quad\quad BK\_Pivot(M \cap N(V_i), K \cup \{v_i\}, T \cap N(V_i));$
12 $\quad\quad M \leftarrow M \backslash \{v_i\};$
13 $\quad\quad T \leftarrow T \cup \{v_i\};$
14 $\quad$ **end**

Algorithm 1: Generowanie maksymalnych kandydatów na kolokacje

Pierwsza z modyfikacji oryginalnego algorytmu dodaje mechanizm *pivoting selection* opisany pierwotnie w pracy [29] (linia 9). Jak wykazano w pracy [30] wybranie

wierzchołka zmniejszającego liczbę rekurencyjnych wywołań algortytmu znacząco zmniejsza ogólny czas wykonania. Wybiera on wierzchołek będący osią podziału zbioru (tzw. *pivot* w oparciu o rozmiar unii sąsiadów tego wierzchołka i wierzchołków kandydatów. Każda maksymalna klika musi zawierać albo wierzchołek u, albo niesąsiadujące z nim wierzchołki - jeżeli nie zawiera, zostanie on dodany (linie 11-13). W związku z tym, tylko wierzchołek u i jego nie-sąsiedzi muszą być przetestowani (linia 10).

Druga modyfikacja opiera się o pojęcie rozproszenia grafu. Opisuje się je miarą *degeneracji grafu* [32]:

Definicja 12 (Degeneracja grafu) *Degeneracja grafu G jest najmniejszą wartością k, taką, że każdy niepusty podgraf G zawiera wierzchołki o stopniu co najwyżej k. Oznacza to, że wielkość maksymalnej kliki nie może przekroczyć* $k + 1$.

Definicja 13 (Uporządkowanie według miary degeneracji) *Uporządkowanie według miary degeneracji wierzchołków grafu G to takie uporządkowanie, które minimalizuje stopień degeneracji grafu. Taka uporządkowanie gwarantuje m.in. optymalną kolejność kolorowania w problemie kolorowania wierzchołków.*

Wierzchołki w zewnętrznej rekurencji są uporządkowane według stopnia degeneracji (linia 2). W ciele rekurencji (linie 3-5) liczba wierzchołków czekających na weryfikację nie przekroczy k. Tym sposobem ograniczono liczbę zewnętrznych rekurencji. Dla grafów o małym stopniu degeneracji obserwuje się duży wzrost wydajności [27].

4.4 Proces odcinania

W ostatnim kroku w oparciu o tzw. *prunning framework* [31] nastepuje otrzymywanie końcowych maksymalnych kolokacji spośród kandydatów wyznaczonym w poprzednim procesie. Na początku dla każdego z nich uruchamiany jest algorytm wyszukiwania klik instancji. Do jego zrozumienia niezbędne jest wprowadzenie poniższych definicji.

Definicja 14 (Uporządkowana klika instancji) *Dany jest kandydat na maksymalną kolokację* C_m. *Jego uporządkowana klika instancji* $InsC_m$ *jest grupą instancji przestrzennych spełniających następujące warunki:*

- *rozmiar* $InsC_m$ *jest równy rozmiarowi* C_m, *a cechy w odpowiadających sobie instancjach w* $InsC_m$ *i* C_m *są takie same;*
- *instancje każdej pary instancji w* $InsC_m$ *sąsiadują ze sobą w przestrzeni i można je znaleźć w tabeli instancji 2-elementowych* $InsTable_2$.

Definicja 15 (Skondensowane drzewo instancji) *Dany jest kandydat na maksymalną kolokację* C_m. *Skondensowane drzewo instancji* $CInsTree$ *jest konstrukcją kompresującą wszystkie uporządkowane kliki instancji* C_m.

Algorytm ma charakter iteracyjny. Zmienna i jest zmienną sterującą pętli. Na początku zostaje zainicjalizowane drzewo $CInsTree$ i tworzony jest jego korzeń. Następnie uruchamiany jest proces konstrukcji drzewa, podzielony na dwa etapy.

Wejście: $C_m, InsTable_2$
Wyjście: $CInsTree$ - skondensowane drzewo instancji C_m

```
1  i ← 1; CInsTree ← ∅; utwórz korzeń drzewa CInsTree;
2  while i < size(C_m) do
3      if i = 1 then
4          foreach para instancji InsPair_k ∈ InsTable_2(C_m(1), C_m(2)) do
5              if InsPair(1) ∈ CInsTree_0.children then
6                  dodaj węzeł podrzędny InsPair_k(2) do
                    CInsTree_1(InsPair_k(1));
7              else
8                  utwórz poddrzewo z InsPair_k(1) jako korzeniem i InsPair_k(2)
                    jako pierwszym dzieckiem;
9                  dołącz to podrzewo do korzenia CInsTree;
10             end
11         end
12     else
13         foreach węzeł instancji ins_k ∈ CInsTree_i do
14             znajdź indeksy elementów równych ins_k od pierwszej kolumny
                InsTable_2(C_m(i), C_m(i + 1));
15             przechowaj drugi element odpowiadającej pary instancji w liście
                El;
16             foreach ei_t ∈ El do
17                 flag ← i − 1;
18                 currIns ← ins_k.parent;
19                 while flag ⩾ 1 do
20                     if (currInt, ei_t) ∈ InsTable_2(C_m(flag), C_m(i + 1)) then
21                         currIns ← currIns.parent;
22                     else
23                         break;
24                     end
25                     flag ← flag − 1;
26                 end
27                 if flag = 0 then dodaj węzeł podrzędny ei_t do
                    CInsTree_i(ins_k);
28             end
29         end
30     end
31 end
```

Algorithm 2: Konstrukcja skondensowanego drzewa instancji C_m

Pierwszy etap wykonywany jest tylko w pierwszej iteracji algorytmu - kolejne iteracje będą wykonywać krok drugi.

W pierwszym kroku (linie 3-11) dla każdej pary instancji pierwszych dwóch cech przestrzennych C_m nastepuje sprawdzenie, czy pierwszy element aktualnie przetwarzanej pary instancji istnieje na danym poziomie drzewa $CInsTree$ (oznaczonego jako $CInsTree_1$). Jeżeli tak, następuje dodanie drugiego elementu jako węzeł podrzędny odpowiadającego mu węzła na poziomie pierwszym. W przeciwnym wypadku, na podstawie obecnej pary instancji tworzone jest poddrzewo, które następnie jest dołączane do korzenia $CInsTree$.

Drugi etap rozpoczyna się konstrukcją listy El zawierającej instancje cechy $C_m(i+1)$. Dokonuje się tego dla każdego węzła instancji na poziomie i-tym drzewa $CInsTree$ popzez skanowanie $InsTable_2(C_m(i), C_m(i+1))$, gdzie $c_m(i)$ jest i-tą cechą C_m. Nastepnie dla każdego elementu tej listy nastepuje sprawdzenie, czy zarówno para instancji składająca się z tego elementu, jak i każdy przodek aktualnego węzła instancji $CInsTree$ znajduje się w tablicy kolokacji o długości 2 - $InsTable_2$. Jeżeli tak, dodaje się ten element jako węzeł podrzędny aktualnego węzła instancji.

Proces działa do czasu, gdy nie będzie żadnego węzła na i-tym poziomie drzewa lub dopóki i nie będzie mniejsze niż $len(C_m) - 1$.

Po zakończeniu algorytmu pozostaje jeszcze wyliczenie indeksów powszechności dla odnalezionych klik instancji. Podobnie jak wcześniej, w przypadku kiedy miara powszechności nie jest mniejsza niż przyjęty na początku próg powszechności, kandydata można przyjąć jako właściwy wzorzec kolokacji [4].

5 Implementacja CPU

W tym rozdziale zostaną opisane implementacje rozwiązania problemu [6] wykorzystujące wyłącznie potencjał procesorów CPU (ang. *Central Procession Unit*), bez udziału karty graficznej w obliczeniach. W szczególności zostaną zaprezentowane techniki optymalizacyjne dedykowane dla poszczególnych podejść do rozwiązania problemu.

Do celów porównawczych zostały zrealizowane dwa rozwiązania: sekwencyjne (bez żadnego wsparcia dla równoległości) oraz równoległe (z wykorzystaniem biblioteki *Parallel Patterns Library* firmy *Microsoft*).

5.1 Wersja sekwencyjna

5.1.1 Generowanie instancji w oparciu o próg odległości

Podobnie jak w pierwotnej wersji algorytmu, generowanie kandydatów na kolokacje o rozmiarze 2 następuje na podstawie *łączenia przestrzennego w oparciu o zamiatanie* (ang. *sweeping-based spatial join*). Algorytm zamiatania (ang. *plane sweep algorithm*) użyty w procesie łączenia został jednak nieznacznie zmodyfikowany.

Na początku wektor będący zbiorem wejściowym jest traktowany algorytmem *sortowania szybkiego* (ang. *quick sort*) dostępnym w bibliotece standardowej C++. Jako kryterium sortowania zostaje wybrana relacja między arbitralnie wybranym wymiarem przestrzeni.

Następnie następuje wyznaczenie progu odległości, na podstawie którego będą dobierani kandydaci na kolokacje. Warto zwrócić w tym miejscu uwagę na fakt, że algorytmowi nie jest potrzebna wiedza o rzeczywistej odległości między dwoma punktami w przestrzeni, a jedynie relacja między tą odległością a zadanym progiem odległości. Daje to możliwość uniknięcia niepotrzebnych obliczeń (pierwiastkowania) przy wyznaczaniu odlegości między punktami. Dlatego też, zamiast bezpośrednio działać w oparciu o zadany próg odległości, wyliczany jest *efektywny próg odległości*, będącym progiem odległości podniesionym do potęgi drugiej.

Pozostało już jedynie filtrowanie obiektów w oparciu o wyliczony efektywny próg odległości. Dla każdego obiektu ze zbioru wejściowego następuje przeszukiwanie we wszystkich kierunkach obiektów sąsiadujących z nim, a nastepnie sprawdzana jest odległość między nimi. Gdy jest ona mniejsza niż ustalony wcześniej próg odległości obiekt umieszczany jest w specjalnej strukturze $insTable$. Jednocześnie tworzony jest wektor $typeIncidenceCounter$, w którym zliczane są wystąpienia kolejnych cech instancji przestrzennych (informacja o tym przyda się przy wyznaczaniu miary powszechności). Dodatkowo, pętla wewnętrzna jest przerywana, jeżeli wartość bezwzględna różnicy posortowanych współprzędnych poszczególnych obiektów będzie większa niż zadany próg odległości. Ma to na celu ograniczenie zbednych porównań w przypadku, kiedy istnieje pewność, że żadne kolejne punkty nie spełnią progu odległości - a zatem nie wejdą do zbioru $insTable$.

Struktura $insTable$ pełni podobną rolę, jak w oryginalnym algorytmie $SGCT$, nie jest ona jednak dwuwymiarową tablicą z haszowaniem. Zamiast tego zastosowano trójwymiarową mapę wskaźników na wektor liczb, w której wymiarami są kolejno: typ elementu A, typ elementu B oraz numer instancji przestrzennej A. W wektorze przechowywane są kolejne numery instancji przestrzennej B. W celu efektywnego odczytywania sąsiadów o danej cesze konkretnej instancji, dane w wektorze

są umieszczane w taki sposób, że zawsze spełniają relację *numer cechy A > numer cechy B*. Dodatkowo, zamiast standardowej mapy (*std::map*) została wykorzystana mapa nieuporządkowana - gwarantuje ona większą wydajność przeglądania (kosztem większego zużycia pamięci).

5.1.2 Filtrowanie sąsiadów w oparciu o próg minimalnej powszechności

Krok ten stanowi zmodyfikowaną wersję algorytmu obliczania miar powszechności znanego z algorytmu *Co-location Miner* (patrz Rozdział 2).

Na początku wykonywana jest funkcja *countUniqueInstances*. Ma ona na celu zliczenie wszystkich wystąpień par instancji bez duplikatów. Podobnie jak w kroku pierwszym zostały tu użyte mapy nieuporządkowane (*std::unordered_map*) w celu przyspieszenia przetwarzania. Do budowy mapy została użyta własna funkcja mieszająca powstała na bazie *hash_combine* z biblioteki *boost*. Ogranicza ona występowanie kolizji w przypadku wstawiania nowych elementów do tablicy z haszowaniem.

Następnie w oparciu o wektor wynikowy tej funkcji oraz utworzony wcześniej wektor *typeIncidenceCounter* (patrz Rozdział 4.1.1) dla każdego kandydata w tabeli *insTable* wyliczany jest współczynnik uczestnictwa. W przypadku, gdy obliczona miara powszechności jest mniejsza od zadanego progu minimalnej powszechności, kandydat jest usuwany z struktury *insTable*, a użyta przez niego pamięć zostaje zwolniona na poczet dalszych obliczeń.

5.1.3 Generowanie kandydatów na kolokacje maksymalne

Tak jak w przypadku oryginalnego algorytmu, kandydaci na kolokacje maksymalne są generowani w oparciu o *graf kolokacji o rozmiarze 2* (patrz Definicja 10). Tworzony jest na bazie struktury *insTable* za pomocą funkcji *createSize2ColocationsGraph*. Krawędź między elementami pary instancji jest tworzona tylko i wyłącznie wtedy, kiedy numer instancji przestrzennej A jest większy od zera.

Po wygenerowaniu grafu liczona jest jego miara degeneracji (patrz Definicja 11). W tym celu został wykorzystany algorytm zaprezentowany przez Matulę i Becka w pracy [32]. Działa on w czasie wielomianowym, co pozwala na odciążenie CPU. Funkcja zwraca zarówno miarę degeneracji, jak uporządkowany według niej wektor wierzchołków wystepujących w grafie.

Dalsze kroki algorytmu wykonywane są dla poszczególnych wierzchołków uporządkowanych według miary degeneracji. Podobnie jak w oryginalnym algorytmie z pracy [29] tworzone są grupy wierzchołków o niższych i wyższych indeksach, a następnie uruchamiana jest funkcja *BK_Pivot*, w której następuje wyszukiwanie maksymalnych klik w oparciu o algorytm Brona-Kerboscha [28].

Aby ograniczyć czas wyszukiwania maksymalnych klik, zostały zastosowane tzw. *wektory sortowane* zamiast zazwyczaj używanych zbiorów (*std::set*) z biblioteki C++. Posiadają one większą wydajność, a do tego pozwalają na skorzystanie z funkcjonalności zarezerwowanej wyłącznie dla zbiorów.

Wyliczone kliki są umieszczane w obiektach *CliqueContainer* i *LapsedCliqueContainer*. Pełnią one rolę pamięci podręcznej dla powyższych algorytmów. Dzięki temu nie ma potrzeby ponownego przeliczania tych samych klik instancji, co prowadzi do kilkunastokrotnego wzrostu wydajności obliczeń.

Wejście: $G = (E, V)$
Wyjście: k (miara degeneracji), L (lista wierzchołków uporządkowanych według miary degeneracji)

```
1  L ← ∅;
2  D ← ∅;
3  foreach v_i ∈ G do
4  |   D(d_v) ← liczba sąsiadów v ∉ L (początkowo równa stopniowi
   |    wierzchołka);
5  end
6  k ← 0;
7  foreach v_i ∈ G do
8  |   znajdź takie i, dla którego D(i) ≠ ∅;
9  |   k ← max(k, i);
10 |   v ← wierzchołek z D(i);
11 |   L ← {v} ∪ L;
12 |   D(i) ← D(i)\{v};
13 |   foreach w ← sąsiedzi v ∉ L do
14 |   |   d'_w ← d_w − 1;
15 |   |   D(d_w) ← D(d_w)\{w};
16 |   |   D(d'_w) ← D(d'_w) ∪ w;
17 |   end
18 end
```

Algorithm 3: Obliczanie miary degeneracji metodą Matuli i Becka (1983)

5.1.4 Generowanie skondensowanych drzew instancji oraz filtrowanie kandydatów na podstawie progu powszechności

Na początku wszystkie rozważane maksymalne kliki z poprzedniego kroku są umieszczane w specjalnej strukturze *cliquesToProcess* będącej trójwymiarowym wektorem, którego indeks stanowi rozmiar kolokacji.

Następnie iteracyjnie przetwarzana jest każda klika, począwszy od największej. Zrezygnowano w tym miejscu z podejścia rekurencyjnego, gdyż prowadziłoby to do niepotrzebnego utrzymywania ogromnego stosu wywołań (ang. *call stack*). W każdej iteracji sprawdzana jest miara powszechności dla kliki instancji - odpowiedzialna jest za to funkcja *isCliquePrevalent*.

Dla klik o długości wiekszej niż 2 tworzone jest skondensowane drzewo instancji (patrz Definicja 14). Implementacja konstrukcji takiego drzewa zasadniczo nie różni się od pierwotnego algorytmu zaprezentowanego w pracy [6], zastosowano w nim jednak pewne metody optymalizacyjne - wskaźniki do rodziców w celu ograniczenia przeszukiwań drzewa w dół, użycie wektora zawierającego wskaźniki do liści znajdujących się na ostatnim poziomie drzewa, wskaźniki typu *unique_ptr* (więcej szczegółów w Dodatku A). Następnie zliczane są wystąpienia instancji w klikach za pomocą odwróconej pętli i na ich podstawie wyliczana jest miara powszechności.

W przypadku, kiedy miara nie jest mniejsza niż przyjęty na początku próg powszechności, kandydat jest dodawany do zbioru rozwiązań. W przeciwnym wypadku do struktury *cliquesToProcess* dodawane są podkliki, które również będą rozważane jako potencjalni kandydaci na kolokacje.

5.2 Wersja wielowątkowa

Jak już wcześniej wspomniano, rozwiązanie równoległe zostało oparte o multiplatformową, wysokopoziomową bibliotekę *Parallel Patterns Library* firmy *Microsoft*. Została ona pierwszy raz zaprezentowane szerzej publiczności wraz z wydaniem środowiska *Visual Studio 2010* i docelowo ma być konkurencją dla popularnej biblioteki *OpenMP*.

Cechą charakterystyczną tej biblioteki jest podobieństwo składni do tej z biblioteki standardowej C++ (ang. *C++ Standard Library*). Wykorzystuje też wszystkie właściwości języka C++, jakie przynosi standard C++11 i późniejsze (w tym funkcje *lambda*).

Biblioteka *PPL* wprowadza zbiór obiektów zwanych *kontenerami o dostępie równoległym* (ang. *concurrent containers*). Są to odpowiedniki kontenerów z biblioteki standardowej C++, które cechują się równoległymi wersjami funkcji operujących na kontenerach (np. operacji wstawiania czy usuwania). W zdecydowanej większości przypadków ich użycie nie różni się od wersji sekwencyjnych dostępnych w bibliotece STD - wymagane jest jedynie umieszczenie ich w odpowiednim bloku, np. *parallel_for*.

W przypadku, kiedy istnieje potrzeba dostarczenia kopii kontenera dla każdego z wątków (np. żeby nie blokować dostępu innym wątkom do współdzielonego obiektu), istnieje także klasa kontenerów typu *combinable*. Kiedy równoległe przetwarzanie zostanie zakończone, prywatne kopie wygenerowane dla każdego z wątków są wtedy przezroczyście łączone w całość. Oczywiście *PPL* zawiera także tradycyjne metody zapewnienia równoległości w programie, takie jak zadania (ang. *tasks*) oraz klasyczne mutexy.

Zasadniczą różnicą między *PPL* a *OpenMP* jest zastosowanie dynamicznego planisty (ang. *dynamic scheduler*), co pozwala na lepszą optymalizację równoległego przetwarzania w zależności od aktualnie dostępnych zasobów w systemie. Na dynamicznym planiście zyskują szczególnie problemy o charakterze rekurencyjnym (np. algorytmy sortujące czy szeroko wykorzystywane w niniejszej pracy przeszukiwanie danych) [33]. Do tego technologia *OpenMP* nie zawiera żadnego mechanizmu anulowania, często wymaganego w algorytmach o charakterze równoległym [34].

Powyższe wady środowiska *OpenMP* były powodem, dla których ostatecznie - pomimo ubogiej dokumentacji oraz niskiej popularności - została wybrana biblioteka *Parallel Patterns Library* jako najbardziej optymalna dla algorytmu *SCGT* będącego tematem tej pracy.

Bibliografia

[1] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From Data Mining to Knowledge Discovery in Databases. AI Magazine, 17:37–54, 1996.

[2] Ł. Stanisławowski. *Bogactwo i nędza narodów.* O'reilly, 2013.

[3] Harvey J. Miller and Jiawei Han. Geographic Data Mining and Knowledge Discovery. Taylor & Francis, Inc., Bristol, PA, USA, 2001

[4] S. Shekhar and Y. Huang. Discovering Spatial Co-location Patterns: A Summary of Results. In SSTD 2001, pages 236–256, 2001.

[5] Przetwarzanie zbiorów przestrzennych zapytan neksploracyjnych w srodowiskachzograniczonym rozmiarem pamiecioperacyjnej

[6] A fast space-saving algorithm for maximal co-location pattern mining

[7] CUDA by Example: An Introduction to General-Purpose GPU Programming, Jason Sanders, Edward Kandrot

[8] Professional CUDA C Programming, John Cheng, Max Grossman, Ty McKerche

[9] Shashi Shekhar and Yan Huang. The Multi-resolution Co-location Miner: A New Algorithm to Find Co-location Patterns in Spatial Dataset. Technical Report 02-019, University of Minnesota, 2002.

[10] Jin Soung Yoo and Shashi Shekhar. A Joinless Approach for Mining Spatial Colocation Patterns. IEEE Transactions on Knowledge and Data Engineering, 18(10):13231337, 2006.

[11] Lizhen Wang, Yuzhen Bao, Joan Lu, and Jim Yip. A New Join-less Approach for Co-location Pattern Mining. In Qiang Wu, Xiangjian He, Quang Vinh Nguyen, Wenjing Jia, and Mao Lin Huang, editors, Proceedings of the 8th IEEE International Conference on Computer and Information Technology (CIT 2008), pages 197–202, Sydney, July 2008. IEEE.

[12] Lizhen Wang, Yuzhen Bao, and Joan Lu. Efficient Discovery of Spatial Co-Location Patterns Using the iCPI-tree. The Open Information Systems Journal, 3(2):69–80,2009.

[13] Christopher Jones and Mark Hall. A Field Based Representation for Vague Areas Defined by Spatial Prepositions. In Proceedings of the Workshop on Methodologies and Resources for Processing Spatial Language at 6th Language Resources and Evaluation Conference (LREC 2008), 2008.

[14] Max J. Egenhofer and Robert Franzosa. Point-set topological spatial relations. International Journal of Geographic Information Systems, 5(2):161–174, 1991.

[15] Eliseo Clementini, Paolino Di Felice, and Peter van Oosterom. A small set of formal topological relationships suitable for end-user interaction. In Proceedings of the 3rd International Symposium on Advances in Spatial Databases (SSD 1993), pages 277-295, London, UK, UK, 1993. Springer-Verlag.

[16] Harvey J. Miller and Jiawei Han. Geographic Data Mining and Knowledge Discovery. Taylor & Francis, Inc., Bristol, PA, USA, 2001.

[17] John F. Roddick and Myra Spiliopoulou. A Bibliography of Temporal, Spatial and Spatio-Temporal data Mining Research. ACM SIGKDD Exploration Newsletter, 1(1):34–38, 1999.

[18] Martin Ester, Hans-Peter Kriegel, and Jörg Sander. Spatial Data Mining: A Database Approach. In Proceedings of the 5th International Symposium on Advances in Spatial Databases (SSD 1997), pages 47–66, London, UK, UK, 1997. Springer-Verlag.

[19] John R. Quinlan. Induction of Decision Trees. Machine Learning, 1(1):81–106, March 1986.

[20] Martin Ester, Alexander Frommelt, Hans-Peter Kriegel, and Jörg Sander. Algorithms for characterization and trend detection in spatial databases. In Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining (KDD 1998), pages 44–50, 1998.

[21] Shashi Shekhar and Sanjay Chawla. Spatial Databases: A Tour. Prentice Hall, 2003

[22] Richard E. Bellman. Adaptive control processes - A guided tour. Princeton University Press, Princeton, New Jersey, U.S.A., 1961.

[23] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In Proceedings of the 20th International Conference on Very Large Data Bases (VLDB 1994), pages 487–499, San Francisco, 1994. Morgan Kaufmann Publishers Inc.

[24] Krzysztof Koperski and Jiawei Han. Discovery of Spatial Association Rules in Geographic Information Databases. In Max J. Egenhofer and John R. Herring, editors, Proceedings of the 4th International Symposium on Advances in Spatial Databases (SSD 1995), volume 951 of Lecture Notes in Computer Science, pages 47–66. Springer Berlin Heidelberg, 1995

[25] Yasuhiko Morimoto. Mining Frequent Neighboring Class Sets in Spatial Databases. In Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2001), pages 353–358, New York, NY, USA, 2001. ACM.

[26] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. Vitter. Scalable Sweeping- Based Spatial Join. In Proc. of the Int'l Conference on Very Large Databases, 1998.

[27] Eppstein, D. , Löffler, M. , i Strash, D. (2010). Listing all maximal cliques in sparsegraphs in near-optimal time. In O. Cheong, K. Y. Chwa, & K. Park (Eds.), 21st international symposium on algorithms and computation (pp. 403–414). Berlin, Germany: Springer-Verlag.

[28] Bron, C., & Kerbosch, J. (1973). Algorithm 457: Finding all cliques of an undirected graph. Communications of the ACM, 16 , 575–577.

[29] Tomita, E., Tanaka, A., & Takahashi, H. (2006). The worst-case time complexity for generating all maximal cliques and computational experiments. Theoretical Computer Science, 363 , 28–42.

[30] Cazals, F.; Karande, C. (2008), "A note on the problem of reporting maximal cliques" , Theoretical Computer Science, 407 (1): 564–568.

[31] Wang, L., Zhou, L., Lu, J., & Yip, J. (2009). An order-clique-based approach for mining maximal co-locations. Information Sciences, 179 , 3370–3382.

[32] Matula, D. W.; Beck, L. L. (1983), "Smallest-last ordering and clustering and graph coloring algorithms", Journal of the ACM, 30 (3): 417–427, doi:10.1145/2402.322385, MR 0709826.

[33] https://msdn.microsoft.com/en-us/library/dd998048.aspx#openmp

[34] http://stackoverflow.com/questions/9700088/microsoft-parallel-patterns-library-ppl-vs-openmp

A Dodatek A - Opis klas i struktur pomocniczych

A.1 Opis architektury wersji CPU

Podczas implementacji stworzono dwie główne klasy: CpuMiningAlgorithmSeq oraz CpuMiningAlgorithmParallel dziedziczące po klasie abstrakcyjnej CpuMiningAlgorithmBase. CpuMiningAlgorithmBase zawiera implementacje metod pomocniczych oraz cztery metody czysto wirtualne (ang. *pure virtual functions*) odpowiadające za główne kroki algorytmu:

- Filtrowanie sąsiadów na podstawie progu odległości:

```
virtual void filterByDistance(float threshold) = 0;
```

- Filtrowanie sąsiadów na podstawie progu minimalnej powszechności:

```
virtual void filterByPrevalence(float prevalence) = 0;
```

- Generowanie kandydatów na kolokacje maksymalne:

```
virtual void constructMaximalCliques() = 0;
```

- Generowanie skondensowanych drzew instancji kandydatów na kolokacje maksymalne i ich filtrowanie na podstawie progu minimalnej powszechności:

```
virtual std::vector<std::vector<unsigned short>>
    filterMaximalCliques(float prevalence) = 0;
```

Klasy CpuMiningAlgorithmSeq oraz CpuMiningAlgorithmParallel zawierają implementację owych metod odpowiednio w wersji sekwencyjnej jak i równoległej.

W kolejnych podrozdziałach znajduje się dokładny opis tych metod oraz implementacji która za nimi stoi.

A.2 Opis struktur & klas pomocniczych i ich implementacji

- CinsTree - Klasa zawierająca implementację skondensowanego drzewa instancji (ang. *condensed instance tree*). Zawiera wskaźnik do korzenia drzewa `std::unique_ptr<CinsNode>root`) oraz wektor `std::vector<CinsNode*>lastLevelChildren` zawierający wskaźniki do liści znajdujących się na ostatnim poziomie (wynikającym z rozmiaru aktualnie budowanej kliki - w przypadku gdy nie udało się rozbudować drzewa o kolejny poziom lista ta jest pusta) owego drzewa. Takie podejście umożliwia szybki dostęp do finalnych instancji poszczególnych kandydatów na kolokacje.

- CinsNode - Klasa zawierająca implementację pojedynczego węzła skondensowanego drzewa instancji *CinsTree*. Obiekty tej klasy zawierają

informacje o cesze `unsigned short type` i numerze `unsigned short instanceId` instancji przestrzennej, wektor wskaźników potomków `std::vector<std::unique_ptr<CinsNode>>children` oraz wskaźnik pokazujący na rodzica danego węzła `CinsNode* parent`. Klasa zawiera metody umożliwiające m.in.:

  - dodanie potomka.
  - zwrócenie potomka o danej cesze i numerze instancji.
  - zwrócenie listy wszystkich przodków.

- Graph - Klasa implementująca graf nieskierowany, bazująca na macierzy sąsiedztwa (ang. *adjacency matrix*). Oprócz podstawowych metod umożliwiających działanie i budowanie grafu, klasa zawiera również metody pozwalające na:

  - obliczenie maksymalnych klik w grafie za pomocą zmodyfikowanego algorytmu Brona-Kerboscha [6].
  - obliczenie optymalnego *pivotu* [29] dla algorytmu Brona-Kerboscha.
  - obliczenie stopnia degeneracji grafu (ang. *degeneracy*) i uporządkowanie wierzchołków według miary degeneracji (ang.*degeneracy ordering*) [27].

- SubcliquesContainer - Klasa umożliwiająca przechowywanie przetworzonych już kandydatów na kliki maksymalne. W łatwy i wydajny sposób ułatwia sprawdzenie czy dana klika lub klika będąca nadzbiorem danej kliki została już przetworzona - dzięki temu unika się przeprowadzenia wtórnych obliczeń i ewentualnych duplikatów w rozwiązaniu. Główną ideą jest stworzenie mapy wektorów `std::map<short, std::vector<unsigned short>>typesMap`, której kluczami są numery cech występujące w poszczególnych klikach a wartościami wektory kolejnych wartości licznika `unsigned int cliquesCounter` którego bieżąca wartość służy do oznaczania kolejnych klik.

  Weryfikację umożliwia algorytm:

```
bool SubcliquesContainer::checkCliqueExistence(
    std::vector<unsigned short>& clique)
{
    assert(clique.size() >= 2);

    std::vector<bool> types(cliquesCounter, false);
    std::vector<bool> typesNew(cliquesCounter, false);

    for (auto type : typesMap[clique[0]])
    {
        types[type] = true;
    }

    for (auto i = 1; i < clique.size(); ++i)
    {
        for (auto id : typesMap[clique[i]])
        {
            if (types[id]) typesNew[id] = true;
        }
        types = typesNew;
        std::fill(typesNew.begin(), typesNew.end(), false);
    }

    if (std::find(types.begin(), types.end(), true) != types.end()
        ())
        return true;

    return false;
}
```

Listing A.1: Kod metody checkCliqueExistence klasy SubcliquesContainer

- ParallelSubcliquesContainer - Klasa odpowiedzialną za tą samą funkcjonalność co klasa *SubcliquesContainer* jednakże zapewniająca bezpieczeństwo przetwarzania wielowątkowego. Cel ten osiągnięto za pomocą skorzystania z sekcji krytycznych w przypadku inkrementowania licznika jak i posłużenia się współbieżnymi wektorami `concurrency::concurrent_vector<unsigned short>` z biblioteki PPL, co przełożyło się również na zwiększenie efektywności przetwarzania.
- CliquesContainer - Klasa zapewniająca dwie funkcjonalności:
  - Sprawdzenia czy dokładnie taka klika jest już przechowywana, za co odpowiada funkcja `bool checkCliqueExistence(std::vector<unsigned short>& clique)`
  - Sprawdzenie czy taka klika lub jej dowolna podklika jest już przechowywana, odpowiada za to funkcja:

```
bool CliquesContainer::checkSubcliqueExistence(
    std::vector<unsigned short>& clique)
{
    bool isSubclique;
    for (auto& c : cliques)
    {
        if (clique.size() < c.size()) continue;
        auto it = clique.begin();
        isSubclique = true;
        for (auto id : c)
        {
            it = std::find(it, clique.end(), id);
            if (it == clique.end()) {
                isSubclique = false;
                break;
            }
        }
        if (isSubclique) return true;
    }
    return false;
}
```

Listing A.2: Kod metody checkSublicqueExistence klasy CliquesContainer

- ParallelCliquesContainer - Klasa odpowiedzialną za tą samą funkcjonalność co klasa *CliquesContainer* zapewniająca w tym samym czasie bezpieczeństwo przetwarzania wielowątkowego.
- RandomDataProvider - Klasa zapewniająca losowy generator danych z parametryzowaną liczbą cech, liczbą instancji a także granicami danych przestrzennych.
- SimulatedRealDataProvider - Klasa wykorzystująca gotowe, przygotowane wcześniej dane wczytywane z plików mające symulować dane rzeczywiste.
- pair_hash - Struktura zapewniająca generyczną implementację funkcji hashującej (ang, textithash function dla pary `std::pair<T1, T2>` - w przypadku typu zdefiniowanego przez użytkownika konieczne jest własnoręczne przeładowania operatora `()`. Aby zapewnić odpowiednią wydajność, należy zadbać o właściwą funkcję hashująca tzn. taką która generuje możliwie mało kolizji. Popularną metodą jest skorzystania z funkcji XOR i zastosowanie jej do dających się pojedynczo hashować elementów pary. Okazało się jednak, że funkcja ta generuje niezadowalająco dużą ilość kolizji, dla tego stworzono bardziej zaawansowany hasher korzystający z funkcji `hash_combine` z biblioteki *boost*:

```cpp
struct pair_hash {
    template <class T1, class T2>
    std::size_t operator () (const std::pair<T1, T2> &p) const {
        std::size_t seed1(0);
        ::hash_combine(seed1, p.first);
        ::hash_combine(seed1, p.second);

        std::size_t seed2(0);
        ::hash_combine(seed2, p.second);
        ::hash_combine(seed2, p.first);

        return std::min(seed1, seed2);
    }
};
```

Listing A.3: Kod struktury pair_hash

Funkcja hash_combine:

```cpp
template<typename T>
void hash_combine(std::size_t &seed, T const &key) {
    std::hash<T> hasher;
    seed ^= hasher(key) + 0x9e3779b9 + (seed << 6) +
        (seed >> 2);
};
```

Listing A.4: Kod funkcji hash_combine

- vector_hash - Struktura umożliwiająca kodowanie mieszające (ang. *hashing*) dla wektorów dowolnych typów dla których istnieje implementacja funkcji hashującej. Również w tym przypadku skorzystano z funkcji `hash_combine`.

```cpp
struct vector_hash {
    template <class T>
        std::size_t operator()(std::vector<T> const& vec) const
        {
            std::size_t seed = vec.size();
            for (auto& i : vec) {
                ::hash_combine(seed, i);
            }
            return seed;
        }
};
```

Listing A.5: Kod struktury vector_hash

- Timer - Generyczna klasa umożliwiająca pomiar czasu dla dowolnej funkcji lub funktora z dowolną liczbą argumentów. Zapewnia ustawienie dowolnego typu zegara i dokładności pomiaru. Odpowiada za sprawdzenie czasu wykonywania poszczególnych kroków algorytmu.
- Benchmark - Klasa umożliwiająca przeprowadzenie parametryzowanych testów wydajnościowych całego algorytmu, zapewniając m.in. serializację do pliku. Wyeksportowane dane mogą zostać zwizualizowane za pomocą

wykresów do których tworzenia wykorzystano odpowiedni skrypt w języku Python.

B Dodatek B