Teoria Algorytmów i Obliczeń

Dokumentacja

Aplikacja znajdująca maksymalną część wspólną dwóch spójnych grafów

Jakub Karolak
Konrad Kurach
Aleksy Miśtal

4 listopad 2018

1. Opis biznesowy

Celem projektu było stworzenie aplikacji znajdującej maksymalny, spójny podgraf dwóch, także spójnych, grafów. Użytkownikiem docelowym jest osoba sprawdzająca zadanie. Projekt został stworzony na potrzeby przedmiotu Teoria Algorytmów i Obliczeń.

2. Wymagania funkcjonalne

Aplikacja wczytuje dwa grafy z pliku, następnie wylicza maksymalną część wspólną tych grafów algorytmem wybranym przez użytkownika. Użytkownik może wybrać jeden z trzech algorytmów. Jeden dokładny i dwa aproksymacyjne. Algorytmy aproksymacyjne muszą mieć złożoność rzędu wielomianowego. Aplikacja zwraca maksymalną część wspólną oraz czas wykonywania obliczeń podany w milisekundach.

3. Wymagania niefunkcjonalne

Aplikacja przyjmuje jako wejście dwa grafy reprezentowane jako macierze sąsiedztwa. Grafy te zapisane są w formacie .csv i przechowywane są w oddzielnych plikach nazwanych ‘graph1.csv’ oraz ‘graph2.csv’. Aplikacja uruchamiana jest z konsoli poprzez wywołanie pliku startowego .exe. Po uruchomieniu i prawidłowym wczytaniu grafów użytkownik widzi tekstowe menu służące do wyboru odpowiedniego algorytmu. Użytkownik może nacisnąć klawisze ‘1’, ‘2’ lub ‘3’, odpowiadające kolejno algorytmowi dokładnemu oraz dwóm algorytmom aproksymacyjnym. Wybór dowolnego innego klawisza powoduje wywołanie komunikatu błędu. Po naciśnięciu jednego z poprawnych klawiszy, aplikacja wykonuje obliczenia. Jako wynik zwracany jest plik z maksymalną częścią wspólną obu grafów, nazwany ‘result[n].csv’, gdzie [n] jest numerem wybranego algorytmu oraz czas wykonania podany w milisekundach, drukowany do konsoli. Zawartość pliku ‘result[n].csv’ będzie mapowaniem wierzchołków obu grafów w postaci dwóch ciągów w ten sposób, aby grafy indukowane były identyczne - kolejne wierzchołki z pierwszego ciągu (pochodzące z pierwszego grafu) odpowiadają kolejnym wierzchołkom z drugiego ciągu (pochodzącym z drugiego grafu). Naciśnięcie klawisza ‘q’ kończy pracę aplikacji.

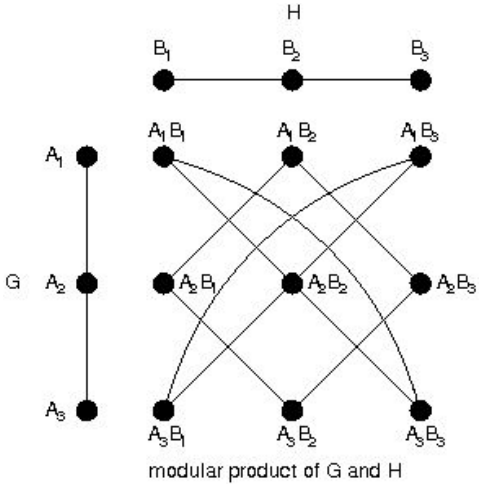
4. Zasada działania

Problem zadany treścią projektu jest bez wątpienia problemem NP-zupełnym, co oznacza, że na chwile obecną (zakładając P ≠ NP) nie istnieje żaden algorytm dokładny rozwiązujący go w czasie wielomianowym.
Opisane poniżej sprowadzenie problemu do szukania kliki w odpowiednio stworzonym grafie jest zabiegiem pozwalającym zastosować szerzej znane algorytmy wyszukiwania kliki maksymalnej w celu jego rozwiązania, jednak w żaden sposób nie wpływają na złożoność algorytmu, ponieważ problem znajdowania maksymalnej kliki jest również NP-zupełny.
W istocie, problem, który rozpatrujemy jest NP-trudny, co oznacza, że nie istnieje (zakładając P ≠ NP) żaden algorytm aproksymacyjny, który w czasie wielomianowym dla n wierzchołków grafu zawsze znajdzie rozwiązanie zachowujące współczynnik $n^{1-\varepsilon}$ rozwiązania optymalnego, gdzie $\varepsilon > 0$[7].

We wszystkich zastosowanych przez nas algorytmach wykorzystaliśmy fakt, iż zadany problem można z powodzeniem sprowadzić do poszukiwania maksymalnej kliki w specjalnie skonstruowanym na potrzeby zadania, tzw. grafie modularnym (*ang. modular product graph*). Zbiór wierzchołków takiego grafu to iloczyn kartezjański wierzchołków dwóch grafów dla których badamy problem (V1 x V2), a krawędź w grafie modularnym istnieje wtedy i tylko wtedy, kiedy spełnione są następujące reguły:

$$(ui, uj) \in E(G1) \wedge (vi, vj) \in E(G2)$$
lub
$$(ui, uj) \notin E(G1) \wedge (vi, vj) \notin E(G2)$$

gdzie (ui, uj) to krawędzie w grafie pierwszym, a (vi, vj) to krawędzie w grafie drugim.



rys.1. Przykładowy graf modularny dla dwóch prostych grafów

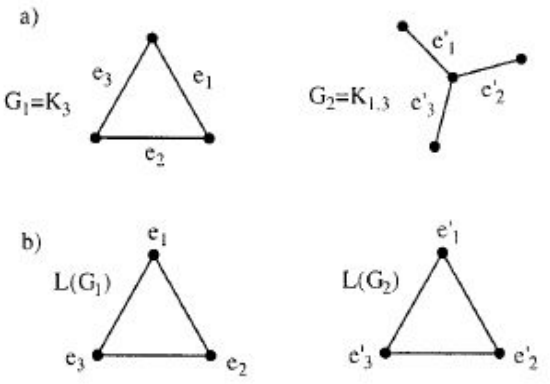
Nie trudno zauważyć, że w tak stworzonym grafie (*rys. 1*) maksymalna klika odpowiada maksymalnemu wspólnemu grafowi indukowanemu przez wierzchołki, które należą do tej kliki (każdy wierzchołek w grafie modularnym to dwa odpowiadające mu wierzchołki w grafach wejściowych). Podamy teraz szkic dowodu, iż w grafie modularnym jest klika wtedy i tylko wtedy, kiedy tworzy ona dwa grafy izomorficzne w grafach, z których powstał ten graf modularny:

1. /⇐/ Nie wprost. Załóżmy, że podgraf grafu modularnego nie jest kliką, ale grafy odpowiadające temu podgrafowi w grafach wejściowych są izomorficzne. Jeśli podgraf nie jest kliką to musi istnieć zależność w grafach G1 oraz G2, że wierzchołki jednego grafu łączą się ze sobą, a drugiego nie (gdyż nie są one połączone w grafie modularnym w myśl przedstawionych wyżej reguł). W takim razie nie mogą być one izomorficzne. Sprzeczność.
2. /⇒/ Nie wprost. Załóżmy, że wspólne podgrafy nie są izomorficzne, ale w grafie modularnym istnieje klika im odpowiadająca. Jeśli nie są izomorficzne to w grafie modularnym musi nie istnieć analogicznie krawędź pomiędzy którąś z par wierzchołków, gdyż w pierwszym grafie będą one połączone, a w drugim nie. Stąd odpowiadający podgraf grafu modularnego nie może być kliką. Sprzeczność.

W myśl przytoczonego wyżej szkicu dowodu dosyć łatwo przejść do stwierdzenia, iż znalezienie maksymalnej kliki w grafie modularnym oznacza znalezienie maksymalnego, wspólnego grafu indukowanego przez odpowiadające wierzchołki w grafach wejściowych. Bliższy opis przedstawionych zależności można znaleźć w pracy[4] *s. 3,4*.

Jednak naszym zadaniem było znalezienie maksymalnych wspólnych grafów, które nie będą indukowane przez zbiór wierzchołków, ale posiadały po pierwsze maksymalną ilość wierzchołków, a po drugie, co ważniejsze, były maksymalne pod kątem sumy wierzchołków oraz krawędzi. Okazuje się jednak, że nie jest to wcale aż tak duży problem dla przedstawionych założeń, które już obraliśmy.

Na podstawie badań Whitney'a[5], który udowodnił, że możliwa jest rekonstrukcja grafu z jego krawędziowej wersji i *vice versa* możemy próbować stworzyć algorytm, który operuje na grafach krawędziowych stworzonych z pierwotnych grafów wejściowych, następnie tworzy z nich graf modularny, znajduje w nim maksymalną klikę, a następnie tak znaleziony maksymalny podgraf krawędziowy tłumaczy ponownie na wersję oryginalną. Istotnie, jest to prawidłowe podejście z jednym wyjątkiem, kiedy to rekonstrukcja z grafu krawędziowego daje błędny rezultat - tzw. wymiany Δ-Y, kiedy to dwa grafy krawędziowe są izomorficzne, pomimo tego, iż pierwotne grafy nie są (*rys. 2*).



rys.2. Wymiana Δ-Y

Zjawisko wymiany Δ-Y prowadzi do błędnych wyników podczas stosowania przedstawionego algorytmu, jednak natura wymiany Δ-Y powoduje, iż jest ona stosunkowo prosta w wykryciu- grafy wynikowe generują inne sekwencje stopni wierzchołków (nie są w rzeczywistości izomorficzne). Zastosowanie prostego *post-processingu* może z powodzeniem wykryć i skorygować błędy powstałe w wyniku występowania wymiany Δ-Y.

Opisane wyżej przekształcenia wymuszają na nas przyjęcie odpowiedniego rozmiaru problemu. Jako, że będziemy działać na grafie modularnym, który jest iloczynem kartezjańskim wierzchołków grafów, z których powstał, a grafy te są grafami krawędziowymi grafów pierwotnych to za rozmiar problemu postanowiliśmy przyjąć n = |E(G1)|*|E(G2)|.

4.1. Algorytm dokładny

Aby znaleźć maksymalną część wspólną dwóch grafów, jak już zostało to zasygnalizowane powyżej, rozpatrzymy problem znajdowania maksymalnej kliki w grafie modularnym stworzonym z grafów krawędziowych grafów pierwotnych. Przedstawiony algorytm znajduje jednocześnie graf o maksymalnej ilości wierzchołków, jak i maksymalnej sumie wierzchołków oraz krawędzi, gdyż wejściowe grafy są spójne. Do rozwiązania problemu został użyty algorytm Brona-Kerboscha. Jest to jeden z najszerzej wykorzystywanych algorytmów rekurencyjnych do znajdywania maksymalnej kliki w grafie, który w podstawowej wersji znajduje wszystkie maksymalne kliki zapamiętując już wykorzystane wierzchołki, w ten sposób unikając niepotrzebnych powrotów - w odróżnieniu od wersji podstawowych.
Jako, że dla opisywanego problemu potrzebna jest jedynie klika największa to algorytm został odpowiednio przystosowany do tego aby zwracał jedynie ją.

Należy dodatkowo podkreślić, iż aby ograniczyć wywołania rekurencyjne zastosowane zostało jeszcze jedno usprawnienie, polegające na znajdowaniu wierzchołka zwanego *piwotem*, który to lokalnie posiada najwięcej sąsiadów podczas działania algorytmu.

Opis działania podstawowej wersji algorytmu Brona-Kerbosha w języku naturalnym:

Przy pierwszym wywołaniu algorytmu zbiory R i X są puste, a zbiór P zawiera wszystkie wierzchołki grafu. Zasada działania algorytmu jest następująca:

Najpierw sprawdzamy w algorytmie, czy zbiory P i X są puste. Jeśli tak, to zbiór R zawiera maksymalną klikę. Wypisujemy zawartość zbioru R i kończymy.

Jeśli zbiór P nie jest pusty, to wybieramy z niego kolejne wierzchołki *v*. Dla każdego z tych wierzchołków tworzymy następujące zbiory tymczasowe:

- N – zbiór wszystkich sąsiadów wierzchołka *v*
- R' – zbiór R z dodanym wierzchołkiem *v*
- P' – iloczyn zbiorów P i N (z P' zostaną usunięte wszystkie wierzchołki, które nie są sąsiadami wierzchołka *v*)
- X' – iloczyn zbiorów X i N (z X' zostaną usunięte wszystkie wierzchołki, które nie są sąsiadami wierzchołka *v*)

Wywołujemy rekurencyjnie algorytm ze zbiorami P', R' i X'.

Następnie ze zbioru P usuwamy wierzchołek *v*, a dodajemy go do zbioru X i kontynuujemy pętlę.

Algorytm Brona-Kerboscha w wersji podstawowej, napisany przy użyciu pseudokodu:

```
G = wejściowy graf
R = zbiór wierzchołków, które są częściowym wynikiem znajdowania kliki
P = zbiór wierzchołków, które są kandydatami do rozważenia
X = zbiór wierzchołków pominiętych
C = zbiór wyjściowych klik maksymalnych

BK_Basic(G, R, P, X):
begin
      if P is empty and X is empty then
            add R to C ;      // znaleziona klika maksymalna
      end
      else
            for vertex v ∈ P do
                  // jako kolejne argumenty - zbiory R', P', X'
                  BK_Basic(R ∪ v, P ∩ N(v), X ∩ N(v));
```

```
                P = P \ v;
                X = X ∪ v;
            end
        end
end
```

Przytoczone już dalsze usprawnienie algorytmu, polegało na zauważeniu faktu, iż algorytm w wersji podstawowej słabo sprawdza się dla grafów, które posiadają dużą ilość klik nie będących maksymalnymi (wywołanie rekurencyjne odbywa się bez względu na wielkość kliki). Aby zaoszczędzić czas i pozwolić algorytmowi na szybsze powroty z gałęzi poszukiwań, które nie zawierają kliki maksymalnej wykorzystany został właśnie wierzchołek zwany piwotem, który wybierany jest ze zbioru P ∪ X i posiada najwięcej sąsiadów w P. W ten sposób tylko piwot oraz niesąsiadujące z nim wierzchołki muszą zostać zbadane dla wierzchołka v dodawanego do zbioru R. . Zbiór P \ N(u) zawiera mniej wierzchołków, zatem liczba wywołań rekurencyjnych spadnie.

Uwzględniając zastosowanie piwotu oraz to, że poszukujemy jedynie klikę maksymalną algorytm Brona-Kerbosha prezentuje się następująco (pseudokod):

```
G = wejściowy graf
R = zbiór wierzchołków, które są częściowym wynikiem znajdowania kliki
P = zbiór wierzchołków, które są kandydatami do rozważenia
X = zbiór wierzchołków pominiętych
C = klika największa


BK(G, R, P, X):
begin
      if P is empty and X is empty then
            if R > C then     // sprawdzamy czy R jest większe od C
                  C = R ;
            end
      end
      else
            choose a pivot vertex u in P ∪ X // u - najwięcej sąsiadów w P
            for vertex v ∈ P \ N(u) do
                  // jako kolejne argumenty - zbiory R', P', X'
                  BK(R ∪ v, P ∩ N(v), X ∩ N(v));
                  P = P \ v;
                  X = X ∪ v;
            end
      end
end
```

W końcowej fazie działania algorytmu należy przetłumaczyć wierzchołki znalezionej kliki na wierzchołki grafów krawędziowych, a te z kolei na wierzchołki grafów wejściowych pamiętając o korygowaniu błędów zachodzących podczas wystąpienia wymiany Δ-Y. Tak znalezione podgrafy będą rozwiązaniem naszego problemu.

Aby oszacować złożoność pesymistyczną algorytmu Brona-Kerbosha należy zastanowić się nad operacjami dominującymi, które w nim występują. Za operacje dominujące przyjmijmy liczbę wywołań rekurencyjnych (wywołań funkcji BK(G, R, P, X)) Bez wątpienia w najgorszym przypadku, nawet dla wersji z wykorzystaniem piwotu algorytm będzie musiał przejrzeć wszystkie możliwe kliki w danym grafie. Dochodząc do takich wniosków, bardzo pomocna w dalszej analizie jest praca J.W. Moon'a oraz L. Moser'a[6], z której wynika, że maksymalna liczba klik w dowolnym grafie o n wierzchołkach wynosi $3^{n/3}$. Biorąc to pod uwagę dochodzimy do wniosku, iż złożoność zastosowanego na potrzeby zadania algorytmu Brona-Kerbosha jest rzędu $O(3^{n/3})$ [8].

4.2. Algorytm aproksymacyjny 1

Jako pierwszy algorytm aproksymacyjny został zaimplementowany algorytm wykorzystujący zachłanną heurystykę. Jest to algorytm dający dosyć dobre wyniki dla grafów losowych, jednak stosunkowo łatwo jest podać złośliwy przykład grafu, dla którego algorytm zwraca jako wynik klikę składającą się z dwóch wierzchołków pomimo, iż w grafie wejściowym istnieje klika o wiele większa.
Procedura przygotowania grafu modularnego oraz końcowej korekcji możliwych błędów wynikających z wymiany Δ-Y wygląda analogicznie jak w przypadku algorytmu dokładnego.

Algorytm z zachłanną heurystyką napisany za pomocą pseudokodu:

```
G = wejściowy graf

Greedy(G):
if G is empty then return ∅;
else choose a vertex v ∈ G of highest degree;
      return {v} ∪ Greedy(neighborhood(v));
end
```

Algorytm ten rozpoczyna wybierając wierzchołek o najwyższym stopniu. Ten wierzchołek jest dodawany do potencjalnej kliki. Następnie usuwa ten wierzchołek i wszystkie wierzchołki nie będące jego sąsiadami i przechodzi na sąsiada o

najwyższym stopniu. Ten proces jest powtarzany dopóki wszystkie wierzchołki grafu nie zostaną usunięte.

Złożoność pesymistyczna tego algorytmu jest rzędu $O(n^3)$, ponieważ operacją dominującą w tym algorytmie jest znajdowanie wierzchołka o najwyższym stopniu, dla której złożoność czasowa wynosi $O(n^2)$ [2] natomiast funkcja `Greedy(G)` jest wywoływana maksymalnie n razy, bo tyle wierzchołków jest w grafie co daje sumaryczną złożoność rzędu $O(n^3)$.

4.3. Algorytm aproksymacyjny 2

Jako drugi z algorytmów aproksymacyjnych został zaimplementowany algorytm Ramseya. Algorytm Ramseya jest ulepszeniem algorytmu zachłannej heurystyki, więc powinien on w naturalny sposób dawać lepsze wyniki. Usprawnienie to polega w głównej mierze na rozpatrywaniu również części grafu, który algorytm z zachłanną heurystyką by od razu odrzucił (wierzchołki nie będące sąsiadami wierzchołka o najwyższym stopniu). Dzięki temu algorytm Ramseya ma zawsze do wyboru dwie kliki, jedną znalezioną w sąsiedztwie wraz z wierzchołkiem o największym stopniu oraz drugą znalezioną poza sąsiedztwem. Algorytm wybiera zawsze większą z tych dwóch. *Preprocessing* oraz *postprocessing* wygląda również analogicznie jak w dwóch zaprezentowanych już algorytmach.

Sam algorytm składa się z dwóch procedur. Podprocedura `Ramsey(G)` wygląda bardzo podobnie jak w przypadku pierwszego algorytmu aproksymującego, jednak różni się wprowadzeniem zmian opisanych powyżej oraz bardzo istotną dla dalszej części algorytmu modyfikacją polegającą na zwracaniu oprócz kliki także zbioru niezależnego, który to jest ściśle z nią związany (znajdowanie zbioru niezależnego jest problemem dualnym do znajdowania kliki). Znalezione zbiory niezależne są następnie wykorzystywane przez procedure `IS_Removal(G)` (ang. *IS = Independent Set*), która to sukcesywnie pomniejsza graf o znalezione zbiory niezależne (usunięcie zbioru niezależnego może prowadzić do maksymalnie usunięcia jednego wierzchołku z kliki, gdyż klika i zbiór niezależny mogą współdzielić maksymalnie jeden wierzchołek) i w ten sposób usprawniając działanie finalnego algorytmu.

Podprocedura `Ramsey(G)` napisana za pomocą pseudokodu:

```
G = wejściowy graf

Ramsey(G):
```

```
begin
      if G = ∅ then return (∅, ∅);
      choose a vertex v ∈ G of highest degree;
      (C1; I1) = Ramsey(N(v));
      (C2; I2) = Ramsey(N̄(v));
      return (larger of (C1∪{v},C2), larger of (I1; I2∪{v}));
end
```

Procedura IS_Removal(G) napisana za pomocą pseudokodu:

```
G = wejściowy graf

IS_Removal(G):
begin
      i = 1;
      (Ci; Ii) = Ramsey(G);
      while G ≠ ∅:
            G = G \ Ii;
            i = i + 1;
            (Ci; Ii) = Ramsey(G);
      return max_{j≤i} Cj;
end
```

Złożoność pesymistyczna tego algorytmu jest rzędu $O(n^4)$. Wiemy już z analizy poprzedniego algorytmu aproksymacyjnego, że znajdowanie wierzchołka o najwyższym stopniu wymaga $O(n^2)$ operacji. Liczbę wywołań funkcji Ramsey(G) możemy oszacować przez $O(n)$ gdyż każdy z wierzchołków wybierany jest jako maksymalny dokładnie raz i nie bierze udziału w dalszych wywołaniach. Zarówno wybór sąsiadów wierzchołka v: N(v) jak i dodanie wierzchołka do kliki oraz zbioru wierzchołków niezależnych nie przekracza $O(n^2)$ operacji elementarnych, więc nie mają wpływu na rząd wielkości złożoności obliczeniowej. Funkcja IS_Removal(G) wywołuje funkcję Ramsey(G) $O(n)$ razy, co daje sumaryczną złożoność rzędu $O(n^4)$.

Literatura

[1] Edmund Duesbury, *''Applications and Variations of the Maximum Common Subgraph for the Determination of Chemical Similarity''*, 2015

[2] Steven Homer, Marcus Peinado, *"On the Performance of Polynomial-time CLIQUE Approximation Algorithms on Very Large Graphs"*, 2000

[3] Ravi B. Boppana, Magnús M. Halldórsson, *"Approximating maximum independent sets by excluding subgraphs"*, 1992

[4] Andrea Torsello, Marcello Pelillo, Andrea Albarelli, *"Matching Relational Structures using the Edge-Association Graph"*, 2007

[5] Whitney, H, *"Congruent Graphs and the Connectivity of Graphs", 1932*

[6] Moon, J. W., Moser, L., "*On cliques in graphs*", 1965

[7] Zuckerman, D., "*Linear degree extractors and the inapproximability of max clique and chromatic number*", 2006

[8] Akira Tanaka, Haruhisa Takahashi, "*The worst-case time complexity for generating all maximal cliques and computational experiments"*, 2006