Analiza Algorytmów - Projekt

Stawczyk Przemysław 293153

Contents

1 Opis Zagadnienia

1.1 Zadane polecenie

Detektyw Karczoch poszukuje nieuczciwych kont na instagramie. Do tego celu analizuje komentarze dotyczące zamieszczanych zdjęć — wie, że takie konta często walczą między sobą, kupując zwolenników i malkontentów. W komentarzach do takich kont, ludzie z jednej grupy piszą tylko do ludzi z drugiej i odwrotnie. Detektyw Karczoch wie, że zwolennik nigdy nie rozmawia z innym zwolennikiem, a malkontent z malkontentem - to dla nich strata czasu.

Zadanie : *Przygotować program, który na podstawie zamieszczonych komentarzy oceni, czy dane konto jest nieuczciwe.*

1.2 Przykładowe dane wejściowe

Uczciwe :
#Jan: Piękne zdjęcie.
#Ola: @Jan Masz rację.
#Ania: @Ola Nie ma racji!
#Jan: @Ania właśnie, że mam!
#Ola: @Ania sama nie masz!

Nieuczciwe :
#Jan: Piękne zdjęcie.
#Ola: @Jan Nieprawda.
#Jan: @Ola czemu tak twierdzisz?
#Ania: @Ola no właśnie?
#Tomek: @Ania twierdzi tak, bo ma rację!!!

1.3 Założenia

- Zakładam, że treść komentarza nie ma znaczenia więc dla opisanego celu liczy się jedynie adresat komentarza.
- Zakładam, że pojedynczy komentarz może nie mieć adresata lub mieć dokładnie jednego adresata.
- Zakładam, że istnienie pojedynczych komentarzy od osób które nie adresują do nikogo komentarzy ani nie są adresatami żadnego komentarza nie ma znaczenia dla rozstrzygnięcia uczciwości konta

2 Koncepcja Algorytmu

2.1 Używane pojęcia

Przy opisie algorytmu stosuje następujące odwzorowanie:

- Każdy komentujący jest reprezentowany przez wierzchołek grafu.

- Komentarz adresowany do innego użytkownika jest reprezentowany przez krawędź w grafie.
- Wiele komentarzy pomiędzy dwoma użytkownikami nie jest odwzorowane jako wielokrotne krawędzie lub krawędzie ważone.
  Istotny jest fakt komunikacji 2 komentujących, a nie jej objętość
- Komentarz jest odwzorowany jako krawędź nieskierowana.
  Istotny jest fakt komunikacji 2 komentujących, a nie który z użytkowników napisał do którego

2.2 Zarys działania

Opis :

Proponowany algorytm opiera się na założeniu, że istnienie 2 grup *[zwolenników i malkontentów]*, które nie piszą komentarzy wewnątrz swoich grup jest równoznaczne z możliwością podzielenia wierzchołków grafu na 2 części w których nie ma krawędzi *[dwudzielność grafu]*. Pojedyncze wierzchołki*[użytkownicy którzy nie piszą do nikogo, ani nie są adresatami wiadomości]* nie mają znaczenia dla rozstrzygnięcia dwudzielności grafu.

Szkic kroków

1 Wczytanie listy komentarzy ze standardowego wejścia lub pliku tworząc graf identyfikowany nazwami użytkowników

2 Usunięcie pętli, wierzchołków nieposiadających oraz powtarzających się krawędzi [np poprzez niedodawanie ich do grafu wynikowego]

3 Przejście po wierzchołkach "kolorując" je na przeciwne kolory zaczynając od dowolnego niepokolorowanego. Następne wierzchołki brane z kolejki.

   I Koloruj wierzchołki sąsiednie na przeciwny kolor. Umieść niepokolorowane wcześniej wierzchołki w kolejce do odwiedzenia.

   II Gdy wierzchołek po drugiej stronie krawędzi jest w tym samym kolorze:
   STOP –> *Konto jest uczciwe.*

4 Usuń pokolorowane wierzchołki.

5 Jeśli graf niepusty: idź do 3 –> *Przetwarzanie kolejnego spójnego podgrafu*,
jeśli pusty: KONIEC –> *Konto nie jest uczciwe*

2.3 Złożoność Algorytmu

2.3.1 Analiza Złożoności Algorytmu

Algorytm iteruje po wszystkich wierzchołkach, a w ramach każdego z wierzchołków po wszystkich jego krawędziach.
Z tego wynika, że złożoność pesymistyczna algorytmu powinna być klasy *O(V+E)*, gdzie *V* to ilość wierzchołków, a *E* ilość krawędzi w grafie.

2.3.2 Przyjęta koncepcja pomiarów

Algorytm w przypadku konta uczciwego może zakończyć się wcześniej, więc aby sprawdzać złożoność algorytmu należy mierzyć czasy wykonania analizy nieuczciwych kont. Aby profilować sam algorytm,

mierzony powinien być jedynie czas spędzony na obliczeniach więc będzie on mierzony dopiero po wczytaniu całego grafu z pliku do pamięci.

Dla wybranych wielkości grafu mierzony byłby czas dla kilku danych testowych. Analiza byłaby wykonana osobno ze względu na ilość krawędzi *[przy stałej ilości wierzchołków]*, oraz na ilość wierzchołków *[przy stałej ilości krawędzi, jednak większej od liczby wierzchołków]*.

3 Założenia Implementacyjne

3.1 Technologie planowane do wykorzystania

- *C++* jako język do implementacji struktury grafu i algorytmu.
- Program do generowania plików z komentarzami zintegrowany z głównym programem
- *boost::chrono* w celu pomiaru czasu spędzonego przez algorytm na procesorze.
- *CMake* jako narzędzie do zarządzania projektem i jego budowaniem
- *doxygen* jako narzędzie do generowania dokumentacji kodu źródłowego

3.2 Struktury danych

- "Node" - *jako klasa zawierająca nazwę użytkownika, pole określające kolor, identyfikowana przez numer pozycji w dynamicznej tablicy*

  W Node : vector<int> - *przechowuje wierzchołki sąsiednie.*

  W Node : enum "Kolor" - *określa czy wierzchołek był odwiedzony oraz kolor odwiedzonego wierzchołka.*
- map<String, int> - *zapewniająca odwzorowanie nazwy użytkownika na pozycję w tablicy*
  Pomocniczo przy wczytywaniu grafu
- vector<Node> - *kontener przechowujący wierzchołki*
  Zapewnia dostęp do wierzchołka poprzez numer
- list<int> - *przechowuje listę wierzchołków do odwiedzenia, umieszczane są w niej kolejni nieodwiedzeni sąsiedzi przetwarzanego wierzchołka*

4 Generowanie danych wejściowych

4.1 Parametry

- liczebność wierzchołków w grafie lub liczebność obu grup z osobna
- liczebność krawędzi w grafie
- uczciwość konta
- nazwa pliku wyjściowego

4.2 Zarys działania

Przy generowaniu danych w pierwszym kroku tworzone są 2 grupy wierzchołków o podanych licznościach lub wedle losowego podziału jednej liczby na wejściu. Następnie tworzone są krawędzie *[komentarze]* poprzez losowanie po jednym wierzchołku z każdej grupy. W przypadku zadania parametrem uczciwości konta dodawane są w małej liczbie krawędzie w obrębie grup stworzonych wcześniej. Jako ostatni etap występuje zapis danych do pliku gdzie dla każdej krawędzi podstawiane są nazwy użytkowników oraz format linii *[dla uwiarygodnienia danych wyjściowych]*.

Struktury danych

- String - *przechowuje linię komentarza zakończoną znakiem nowej linii lub login użytkownika*
- vector<String> - *kontener przechowujący loginy użytkowników*
- vector<String> - *kontener przechowujący komentarze użytkowników*

5 Testy i weryfikacja działania

5.1 Metodyka

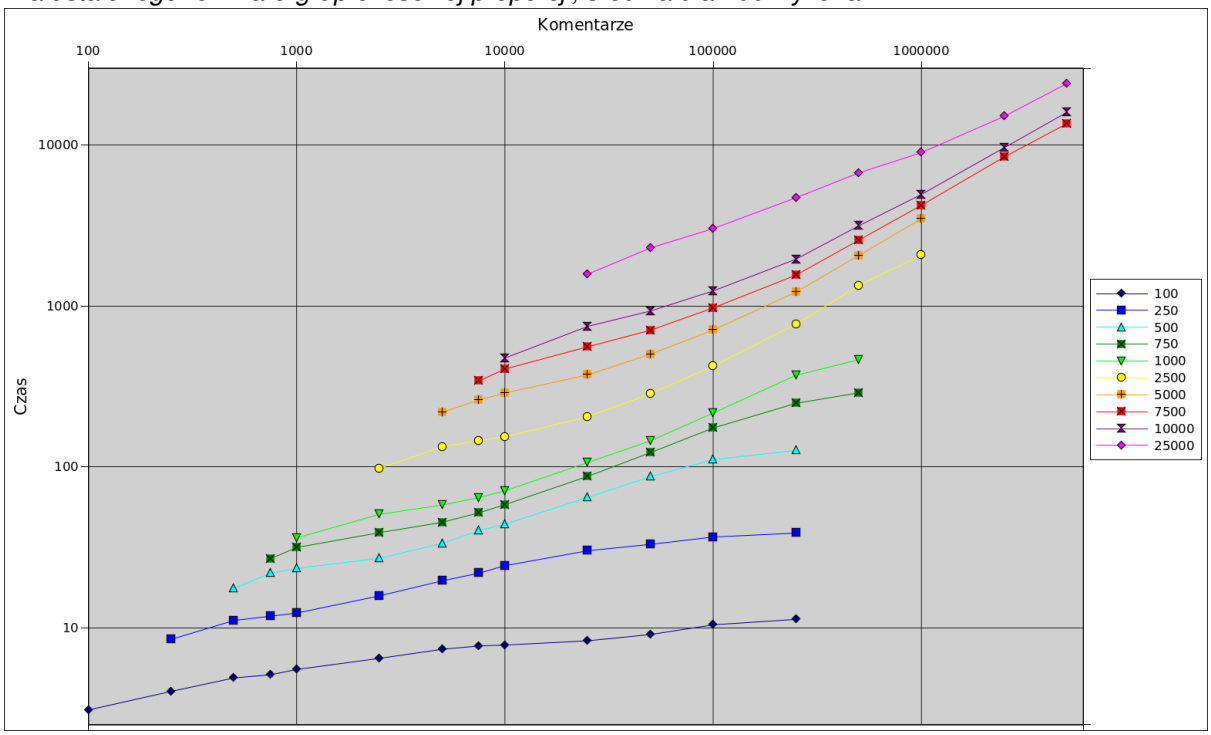
Czas wykonani algorytmu był mierzony po wczytaniu oraz przygotowaniu grafu a pomocą biblioteki *boost::chrono* obliczając odległość pomiędzy początkiem a końcem algorytmu. Zgodnie z wcześniej przyjętym założeniem mierzone były jedynie problemy dające wynik nieuczciwy by badać pesymistyczną złożoność algorytmu. Testy zostały przeprowadzone poprzez skorzystanie z trybu profilowania stworzonego programu, dla każdej pary: *ilości użytkowników i komentarzy* dokonano 100 wykonań algorytmu przyjmując do dalszej analizy wartość średnią dzięki czemu wszelkie wstępne opóźnienia pierwszych wykonań zmniejszają swój wpływ na wynik. Pomiarów dokonano na nieobciążonej innymi zadaniami maszynie stacjonarnej obciążając 1 z 6 dostępnych rdzeni procesora by wpływ działającego systemu był minimalny.

5.2 Analiza wyników

5.3 Wizualizacja wyników pod kątem złożoności algorytmu

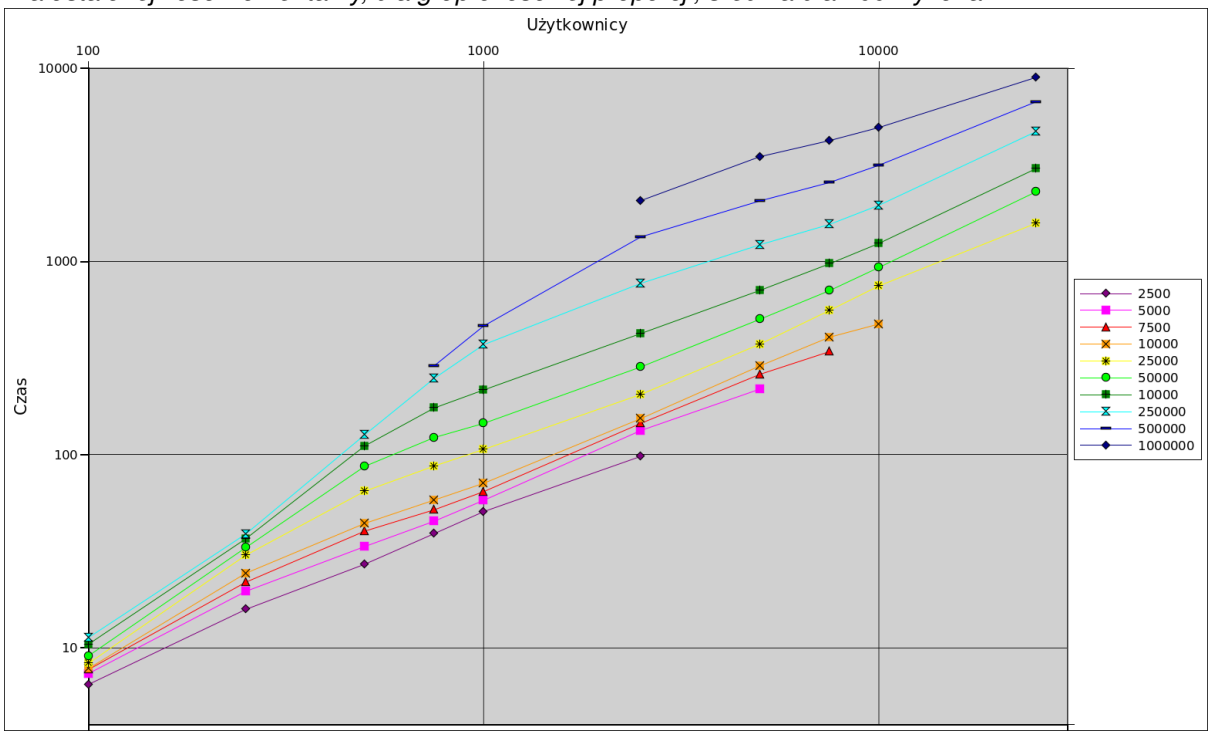
5.3.1 Czas od ilości komentarzy

Dla ustalonego rozmiaru grup o losowej proporcji, średnia dla 100 wykonań



5.3.2 Czas od ilości użytkowników

Dla ustalonej ilości komentarzy, dla grup o losowej proporcji, średnia dla 100 wykonań



6 Dokumentacja kodu źródłowego - *English*

6.1 Class Index

6.1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

6.2 Class Documentation

6.2.1 Generator Class Reference

Generator class for problem instances generation.

`#include <Generator.h>`

Public Member Functions

- Generator (std::initializer_list< string > names_list=NAMES__ALL)

  Constructor for the generator.
- std::stringstream generate_instance_output (bool fair, uint64_t users_count, uint64_t l_group_count, uint64_t comments_count)

  Generates problem instance according to parameters as a stream.
- std::stringstream generate_instance_output (vector< string > &problem_instance)

  Creates printable stream from a vector of lines.
- vector< string > generate_instance (bool fair, uint64_t users_count, uint64_t l_group_count, uint64_t comments_count)

  Generates problem instance according to parameters as a vector of lines.

6.2.2 Detailed Description

Generator class for problem instances generation.

6.2.3 Constructor & Destructor Documentation

Generator() `Generator::Generator (`
`std::initializer_list< string > names_list = NAMES__ALL )`

Constructor for the generator.

Wraps random number generator and basic list of names for login creation. Stateless excluding RNG state.

Parameters

names_list	list of names to be used, default is list of polish names from wikipedia.org

6.2.4 Member Function Documentation

generate_instance() `vector< string > Generator::generate_instance (`
`bool fair,`
`uint64_t users_count,`
`uint64_t l_group_count,`
`uint64_t comments_count )`

Generates problem instance according to parameters as a vector of lines.

Parameters

fair	determine if account should be fair or not
users_count	count of all users
l_group_count	count of one of the "groups" commenting
comments_count	number of comments

Returns

problem instance in a vector

generate_instance_output() `[1/2] std::stringstream Generator::generate_instance_output (`
`bool fair,`
`uint64_t users_count,`
`uint64_t l_group_count,`
`uint64_t comments_count )`

Generates problem instance according to parameters as a stream.

Parameters

fair	determine if account should be fair or not
users_count	count of all users
l_group_count	count of one of the "groups" commenting
comments_count	number of comments

Returns

problem instance in a stream

generate_instance_output() `[2/2]` `std::stringstream Generator::generate_instance_output ( vector< string > & problem_instance )`

Creates printable stream from a vector of lines.

Parameters

problem_instance	problem instance

Returns

problem instance in a stream

6.2.5 Solver Class Reference

class created to solve problem instances

`#include <Solver.h>`

Public Member Functions

- bool is_fair (std::vector< std::string > &problem_instance, uint64_t &time)

  loads problem instance to the object and runs check
- bool is_fair (std::stringstream &problem_instance, uint64_t &time)

  loads problem instance to the object and runs check

6.2.6 Detailed Description

class created to solve problem instances

6.2.7 Member Function Documentation

is_fair() `[1/2]` `bool Solver::is_fair ( std::vector< std::string > & problem_instance, uint64_t & time )`

loads problem instance to the object and runs check

Parameters

problem_instance	problem instance as a vector of lines
time	reference to which time elapsed will be added

Returns

result : if account is fair

is_fair() [2/2] `bool Solver::is_fair (`
`std::stringstream & problem_instance,`
`uint64_t & time )`

loads problem instance to the object and runs check

Parameters

problem_instance	problem instance as a stream containing lines
time	reference to which time elapsed will be added

Returns

result : if account is fair