Dokumentacja końcowa programu Grafo-Siekacz

Aiman Ghanim, Cyprian Ciesielski

07.05.2025

Cel projektu

Program `gs` ma na celu przeprowadzić podział grafu na określoną liczbę części. Ma on przyjmować wszystkie parametry z linii poleceń, umożliwiając użytkownikowi wskazanie liczby części, na które ma zostać podzielony graf, oraz marginesu procentowego dla różnicy w liczbie wierzchołków w powstałych częściach. W domyślnym ustawieniu (brak wskazania przez użytkownika) graf dzielony jest na 2 części przy marginesie błędu 10%.

Dodatkowo, aplikacja powinna dążyć do minimalizacji liczby przeciętych krawędzi pomiędzy częściami grafu, zapewniając równomierny podział wierzchołków w zadanych częściach.

Argumenty wywołania programu

Program `gs` akceptuje następujące argumenty wywołania:

- `--help / -h` wyświetla pomoc dotyczącą obsługi programu;
- `--output filename / -o [nazwa pliku]` określa plik, do którego zapisany zostanie wynik;
- `--precompute-metrics / -p` oblicza oraz wyświetla statystyki grafu przed podziałem;
- `--postcompute-metrics / -a` oblicza oraz wyświetla statystyki grafu po podziałem;
- `--out-format text|binary / -k` wybiera format pliku wyjściowego (tekstowy, lub binarny);
- `--force / -f` dzieli graf mimo braku spełnienia marginesu podziału;
- `--out-format text|binary / -k` wybiera format pliku wyjściowego (tekstowy, lub binarny);
- `--iterations / -i` ilosc iteracji funkcji cut_optimalization;

- `--statistics / -s` wyświetl szczegółowe statystyki;

Przykładowe wywołania programu:

- `./bin/main 3 20% graf.csrrg --precompute-metrics --output odp.csrrg`
- Efektem będzie podział grafu z pliku `graf.csrrg` na 3 części przy marginesie błedu 20%. Wyświetlą się również statystyki grafu przed podziałem, a grafy wynikowe zostaną zapisane do pliku `odp.csrrg`.

Funkcjonalność programu

Projekt polega na podziale grafu na określoną liczbę części, przy czym liczba wierzchołków w każdej części nie może różnić się o więcej niż ustalony margines procentowy, liczony względem średniej liczby wierzchołków na część.

Główne cele projektu

- Równomierny podział
- Minimalizacja krawędzi między częściami

Podejście do rozwiązania

Ze względu na NP-trudność problemu (czyli brak znanego algorytmu rozwiązującego problem optymalnie), stosuje się metody przybliżone oraz heurystyki, które pozwalają uzyskać satysfakcjonujące rozwiązania w rozsądnym czasie, nawet dla dużych grafów. Kluczowe elementy podejścia to:

- **Analiza danych wejściowych:** Algorytm analizuje strukturę grafu oraz rozkład wierzchołków, co pozwala na określenie średniej liczby wierzchołków na część.
- **Balansowanie względem średniej:** Ustalony margines (np. 20%) definiuje zakres akceptowalnych wielkości części. Jeśli średnia wynosi 5 wierzchołków, to każda część musi zawierać od 4 do 6 wierzchołków.
- **Minimalizacja przecięć:** Równocześnie algorytm dąży do zminimalizowania liczby krawędzi łączących różne części.

Komunikaty błędów

Program `gs` stara się kontynuować pracę, chodź w niektórych przypadkach jest to niemożliwe jak np. błędne dane, lub wymagania niemoliwe do spenienia. Program na bieżąco informuje użytkownika o błędach.

Algorytm partycjonowania grafów z zachowaniem spójności

Cel algorytmu

Celem algorytmu jest podział grafu na dokładnie K spójnych, zbalansowanych podgrafów przy minimalizacji liczby krawędzi przecinających granice między regionami. Podział musi spełniać dwa kluczowe warunki:

- **Spójność:** każda partycja stanowi spójny podgraf.
- **Zrównoważenie:** rozmiary partycji mogą różnić się maksymalnie o $g\%$ od średniej liczby wierzchołków przypadających na partycję.

Hybrydowe podejście dwufazowe

Algorytm składa się z dwóch komplementarnych faz:

- **Faza 1: Region Growing** – generowanie początkowego podziału z zachowaniem spójności i równowagi.
- **Faza 2: Zmodyfikowany algorytm Fiduccia–Mattheyses (FM)** – iteracyjna optymalizacja w celu redukcji liczby krawędzi przecinających partycje.

Faza 1: Region Growing – wstępny podział

1. **Strategiczny wybór nasion:**
   - Wybieramy K wierzchołków jako punkty startowe (nasiona).
   - Generujemy wielu losowych kandydatów i stosujemy heurystykę odległości: preferujemy wierzchołki słabo połączone z już wybranymi nasionami.
   - Z losowych kandydatów wybieramy te, które minimalizują połączenia między sobą.
2. **Równoległy rozrost partycji:**
   - Każda partycja utrzymuje własny *frontier* – listę kandydatów do przyłączenia.
   - W każdej iteracji rozszerzamy najmniejszą partycję o jeden wierzchołek spośród jej frontier, zapewniając spójność.
   - Po przyłączeniu wierzchołka aktualizujemy frontier o jego nieprzypisanych sąsiadów.
3. **Zarządzanie zbalansowaniem:**

- Monitorujemy rozmiar partycji, nie pozwalając przekroczyć limitu $(1 + \frac{g}{100}) \times \frac{N}{K}$.
- Partycje, które osiągnęły maksymalny rozmiar, są wyłączane z rozrostu.
- Utrzymujemy listę *aktywnych* partycji dla efektywnej selekcji następnych ruchów.

4. **Obsługa wierzchołków nieprzydzielonych:**
   - Po zakończeniu głównego rozrostu przypisujemy pozostałe wierzchołki do sąsiednich partycji, zachowując spójność.
   - W ostateczności wierzchołki doliczamy do partycji o najmniejszym rozmiarze.

Faza 2: Zmodyfikowany algorytm Fiduccia–Mattheyses (FM) – optymalizacja

1. **Identyfikacja wierzchołków granicznych:** Zbieramy wszystkie wierzchołki posiadające sąsiada w innej partycji.
2. **Obliczanie zysku (*gain*):** Dla wierzchołka v i docelowej partycji P':

   gdzie $P(v)$ to obecna partycja v.
3. **Weryfikacja integralności strukturalnej:**
   - Sprawdzamy, czy usunięcie v nie rozspójni jego obecnej partycji (BFS pomijający v).
   - Upewniamy się, że v ma krawędź łączącą z docelową partycją P'.
4. **Utrzymanie balansu rozmiarów:** Przed ruchem weryfikujemy, czy nowe rozmiary obu partycji mieszczą się w granicach
5. **Mechanizm blokowania wierzchołków:**
   - Po przeniesieniu v blokujemy go na bieżącą iterację.
   - Wierzchołki, których ruch naruszałby spójność lub balans, oznaczamy jako trwałe „nieprzenoszalne”.
6. **Strategia wyboru ruchów:**
   - W każdej iteracji wybieramy wierzchołek z największym dodatnim *gain*.
   - Przenosimy tylko wierzchołki o gain > 0.
   - Po każdym ruchu weryfikujemy spójność wszystkich partycji.

Podsumowanie:

Przedstawiony hybrydowy algorytm partycjonowania grafu składa się z dwóch faz i łączy zalety szybkiego podziału konstrukcyjnego z iteracyjną optymalizacją. Dzięki wbudowanym mechanizmom weryfikującym spójność oraz utrzymującym zrównoważenie, algorytm:

- generuje spójny i zrównoważony wstępny podział metodą *Region Growing*,
- iteracyjnie optymalizuje partycje zmodyfikowanym algorytmem Fiduccia–Mattheyses, redukując liczbę krawędzi przecinających granice,
- sprawdza integralność strukturalną każdej partycji oraz dba o to, by rozmiary nie przekraczały ustalonego marginesu różnicy,
- blokuje już przeniesione wierzchołki i oznacza te, których ruch naruszyłby warunki spójności lub balansu.

Efektem działania jest podział grafu na dokładnie K spójnych, zbalansowanych podgrafów przy minimalnej liczbie krawędzi między regionami, co stanowi kompromis między jakością rozwiązania a szybkością działania algorytmu.

Format pliku wejściowego

Plik opisujący graf może składać się z wielu sekcji:

1. Maksymalna możliwa liczba węzłów w wierszu.
2. Indeksy węzłów w poszczególnych wierszach.
3. Wskaźniki na pierwsze indeksy w liście wierszy.
4. Grupy węzłów połączone krawędziami.
5. Wskaźniki na pierwsze węzły w każdej grupie (sekcja może się powtarzać przy wielu grafach).

Format pliku wejściowego

Plik opisujący graf może składać się zwielu linii. Poniżej opisane jest znaczenie poszczególnych wierszy:

1. Maksymalna możliwa liczba węzłów w wierszu (w grafie nie musi znajdować się wiersz o takiej liczbie węzłów)
2. Indeksy węzłów w poszczególnych wierszach - liczba wszystkich indeksów odpowiada liczbie węzłów grafu
3. Wskaźniki na pierwsze indeksy węzłów w liście wierszy z punktu 2

4. Grupy węzłów połączone przy pomocy krawędzi
5. Wskaźniki na pierwsze węzły w poszczególnych grupach z punktu 4. Ta sekcja może występować w pliku wielokrotnie, co oznacza, że plik zawiera więcej niż jeden graf.

Format standardowego pliku wyjściowego

Standardowy plik będzie działał na takich samych zasadach jak plik wejściowy. W czwartek linijce są wypisane wszystkie wierzchołki oraz ich sąsiedzi po kolei z każdej części. w koljenych linijkach są wskażniki na tą czwartą linijke tworząc nowe grafy.

Format pliku binarnego

Plik binarny zostanie wykorzystany do przechowywania podzielonego grafu. Jego struktura odpowiada formatowi pliku wejściowego, jednak dane są zapisane w postaci surowych bajtów.

Pierwsza wartość w pliku to liczba wierzchołków (np. 1582), zapisana jako liczba zakodowana za pomocą vByte Encoding.

Każdy wiersz z pliku `.csrrg` odpowiada ciągowi liczb binarnych reprezentujących sąsiadów danego wierzchołka. Dane dla jednego wiersza są oddzielone od kolejnego specjalnym znacznikiem:

```
0xDEADBEEFCAFEBABE
```

Zapisywany jako 8 bajtów w porządku **little-endian**, tj. najmniej znaczący bajt (LSB) zapisywany jest jako pierwszy:

```
BE BA FE CA EF BE AD DE
```

Do zapisu danych wykorzystane zostaną funkcje języka C: `fopen()` z trybem `"wb"` do otwarcia pliku w trybie binarnym, `fwrite()` do zapisania przetworzonych danych oraz `fclose()` w celu prawidłowego zamknięcia pliku po zakończeniu operacji.

vByte Encoding

Każda liczba jest zapisywana w grupach 7-bitowych. Najmniej znaczące bity (LSB) zapisywane są jako pierwsze — kodowanie odbywa się w porządku little-endian.

Najbardziej znaczący bit (MSB) każdego bajtu pełni rolę flagi:

- MSB = 1 → oznacza, że to **nie jest** ostatni bajt danej liczby, należy czytać dalej.
- MSB = 0 → oznacza, że to **ostatni bajt** liczby.

Każdy bajt zawiera więc 7 bitów wartości i 1 bit kontrolny.

Przykład kodowania liczby $300_{10} = 0x012C$**:**

$$300 = 0x012C = \texttt{10101100 00000010}$$

W bajtach: `0xAC 0x02`, gdzie:

- `0xAC` = `10101100` → MSB = 1, wartość = `00101100`
- `0x02` = `00000010` → MSB = 0, ostatni bajt, wartość = `00000010`

Wartość jest obliczana jako:

$$300 = (0x02 \ll 7) + 0x2C = 256 + 44$$

Przykładowy wygląd pliku binarnego (hex view)

```
[liczba_wierzcholkow] => 1582 => AC 0C
[lista 1] => 43;113;136 -> delta: 43, 70, 23 -> vByte
          => 2B 46 17
[separator] => BE BA FE CA EF BE AD DE
[lista 2] => ...
```

Zapis i odczyt

Każda sekcja pliku jest kodowana w kolejności:

1. Sortowanie sąsiadów rosnąco
2. Zastosowanie kodowania vByte (LSB-first)
3. Dopisanie 8-bajtowego separatora

Przy odczycie każda liczba jest dekodowana z vByte, a następnie odbudowywana na podstawie sumy kolejnych delt.

Uwaga dotycząca przenośności

Wszystkie dane są kodowane jako bajty (uint8), a wszystkie liczby są vByte-encoded, co uniezależnia format od różnic pomiędzy platformami (np. różnej długości typów takich jak `short` w języku Java na systemie Windows). Ważne jest jedynie, aby zachować zgodność z porządkiem little-endian oraz interpretacją LSB i MSB w vByte.

Struktura plików i folderów

Główne foldery projektu:

- **data/** – pliki wejściowe i wyjściowe (`.csrrg`).
- **docs/** – dokumentacja (`.pdf`).
- **include/** – pliki nagłówkowe (`.h`):
  - `fm_optimization.h` – algorytm FM,
  - `graph.h` – operacje na grafie,
  - `partition.h` – integracja podziału,
  - `region_growing.h` – Region Growing,
  - `stats.h` – statystyki,
  - `file_reader.h` – odczyt plików.
  - `file.h` – zapis plików wyjściowych.
- **src/** – implementacje (`.c`):
  - `fm_optimization.c` – algorytm FM,
  - `graph.c` – operacje na grafie,
  - `main.c` – punkt wejścia,
  - `partition.c` – integracja metod,
  - `region_growing.c` – Region Growing,
  - `stats.c` – statystyki.
  - `file_reader.c` – odczyt plików.
  - `file.c` – zapis plików wyjściowych.
- **tests/** – testy jednostkowe i integracyjne.
- **Makefile** – plik kompilacji.
- **README.md** – podstawowe informacje o projekcie.