Generacja Grafów R-Mat

Marcin Klimek

25 marca 2023

Politechnika Krakowska
Wydział Inżynierii
Elektrycznej i Komputerowej

1 Wstęp

W sprawozdaniu będzie omówiony algorytm generacji grafów R-Mat[1], pozwalającego na rekurencyjne generowanie macierzy 2^n reprezentujące grafy o n wierzchołkach. Pozwala on na generację macierzy zarówno dla grafów skierowanych, jak i nieskierowanych. Kod implementujący ten algorytm został napisany w języku Rust, i na jego przykładzie zostanie omówiona implementacja algorytmu.

2 Działanie algorytmu

Zadaniem algorytmu jest wygenerowanie reprezentacji grafu w postaci macierzy o wielkości 2^n. Użytkownik w celu generacji grafu podaje następujące argumenty: `directed` - czy graf ma być skierowany, `self_connections_allowed` - czy w grafie mogą występować pętle, `n` - ilość wierzchołków w grafie, `propabilites` prawdopodobieństwa na wybranie ćwiartki macierzy, `dest_density` - docelowa gęstość grafu.

Algorytm generuje macierz w sposób rekurencyjny, dzieląc macierz na mniejsze macierze o rozmiarach 2^{n-1}. W każdym kroku algorytmu generowany jest losowy podgraf o rozmiarze 2^{n-1}, a następnie wykorzystywany jest algorytm generowania grafów R-Mat na tym podgrafie. W ten sposób generowany jest graf o rozmiarze 2^n, z którego następnie program jest w stanie wygenerować statystyki, takie jak faktyczna gęstość grafu, liczba krawędzi czy wierzchołki z największym stopniem.

Przykładowe wyjście programu dla grafu skierowanego, bez pętli, o 4 wierzchołkach, prawdopodobieństwach na wybranie ćwiartki macierzy [0.3, 0.15, 0.40, 0.15] i docelowej gęstości 0.4:

```
Total fillable: 240
Filled 97 fields
0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0
1 0 0 0 1 0 1 0 1 1 1 0 0 1 0 0
1 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0
1 1 1 0 0 0 1 0 0 0 0 0 1 0 0 0
1 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0
1 1 1 0 1 0 0 0 1 0 1 1 0 0 0 0
1 1 1 0 1 0 1 0 0 0 0 0 0 0 0 0
1 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0
1 1 1 0 1 0 0 0 1 1 0 0 0 0 0 0
1 1 1 0 0 1 1 0 1 1 0 0 1 0 0 1
1 1 1 1 0 1 0 0 1 0 0 0 0 1 1 0
1 1 1 1 0 1 0 0 1 1 0 0 1 0 1 0
1 1 1 1 1 1 1 0 0 0 1 0 0 0 0 1
1 1 0 1 0 1 1 1 0 0 0 1 1 0 0 0
Density: 40.4167%
Edges: 97
Vertices: 16
Vertex 14 has the highest degree with 9
```

Przykładowe wyjście programu dla grafu nieskierowanego, z pętlami, o 4 wierzchołkach, prawdopodobieństwach na wybranie ćwiartki macierzy [0.15, 0.2, 0.20, 0.45] i docelowej gęstości 0.4:

```
Total fillable: 136
Filled 54 fields
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 0 1 0 0 0 0 0 0 0 0 0
0 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0
0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 1 1 1 0 1 1 1 0 0 0
0 0 0 0 1 0 0 1 0 1 1 1 1 1 0 0
0 1 1 0 1 0 1 1 0 0 1 1 1 1 1 0
1 0 0 1 1 1 1 1 0 1 1 1 1 1 1 1
Density: 39.7059%
Edges: 54
Vertices: 16
Vertex 14 has the highest degree with 25
```

Wyjście programu dla argumentu `--help`:

```
Program do generacji grafów

Usage: rmat --directed --self-connections-allowed -n <N>
            --propabilities <PROPABILITIES> --dest-density <DEST_DENSITY>

Options:
  -d, --directed                       Czy graf ma być skierowany
  -s, --self-connections-allowed       Czy graf może mieć pętle
  -n <N>                               Liczba wierzchołków
  -p, --propabilities <PROPABILITIES>  Prawdopodobieństwo wybrania ćwiartki
                                        (format: [0.15, 0.2, 0.20, 0.45])
  -g, --dest-density <DEST_DENSITY>    Docelowa gęstość grafu
  -h, --help                           Print help
  -V, --version                        Print version
```

3 Implementacja

Kod programu znajduje się w dwóch plikach: `main.rs` oraz `graph.rs`. W pliku `main.rs` znajduje się kod odpowiedzialny za wywoływanie funkcji z pliku `graph.rs` wraz z argumentami podanymi przez użytkownika. W pliku `graph.rs` znajduje się kod zawierający obiekt `Graph` wraz z funckjami, które pozwalają na generowanie grafu, wyliczanie statystyk oraz wiele innych.

Obiekt `Graph` zawiera następujące pola:

```rust
pub struct Propabilities([f64; 4]);
impl Propabilities {
    // w funkcji new weryfikowane jest, czy prawdopodobieństwa
    // sumują się do 1
    pub fn new(vals: [f64; 4]) -> Propabilities { ... }
    pub fn get_random_quarter(&self) -> u8 { ... }
}

pub struct GraphStats {
    ...
}

pub struct Graph {
    directed: bool,
    self_connections_allowed: bool,
    connections: Vec<Vec<bool>>,
    n: u32,
    propabilities: Propabilities,
    dest_density: f64,
}
impl Graph {
    pub fn new(directed: bool, self_connections_allowed: bool, n: u32,
        propabilities: Propabilities, dest_density: f64) -> Graph { ... }
    pub fn fill(&mut self) { ... }
    pub fn poke(&mut self) -> bool { ... }
    pub fn poke_range(&mut self, start_x: usize, start_y: usize,
        size: usize, ignore_b: bool) -> bool { ... }
    pub fn print(&self) { ... }
    pub fn get_stats(&self) -> GraphStats { ... }
    pub fn save(&self, filename: &str) { ... }
}
```

W celach zwięzłości omówione zostaną jedynie funkcje odpowiadające za generowanie grafu. Całość kodu wraz z funckjami pomocniczymi jest dostępnd w repozytorium na GitHubie `https://github.com/klmkyo/rmat`.

Za generowanie grafu odpowiadają funckje `new`, `poke`, `poke_range` oraz `fill`. Funkcja `new` inicjalizuje obiekt `Graph` z podanymi parametrami. `poke` znajduje losowy element (na podstawie podanych prawdopodobieństw) `false` (oznaczający brak połączenia), oraz ustawia go na `true`. Wywołuje ona `poke_range`, które początkowo działa na całej macierzy, jednak następnie ta sama funckja będzie rekursywnie wywoływała samą siebie, jedynie na zakresie zmniejszonym do wylosowanej ćwiartki. Funkcja `fill` wywołuje funkcję `poke` , dopóki nie zostanie osiągnięta gęstość grafu sprecyzowana przez użytkownika.

3.1 Funkcja `graph.new`

```rust
pub fn new(directed: bool, self_connections_allowed: bool, n: u32,
    propabilities: Propabilities, dest_density: f64) -> Graph {

    let connections = vec![vec![false; 2_usize.pow(n)]; 2_usize.pow(n)];
    Graph { directed, self_connections_allowed, connections,
        propabilities, n, dest_density }
```

```
}
```

Odpowiada za stworzenie obiektu `Graph` z podanymi parametrami. Inicjalizuje również tablicę o wielkości 2^n, w której przechowywane będą relacje między wierzchołkami. Wartość `false` oznacza brak połączenia, a `true` - połączenie.

3.2 Funkcja `graph.poke_range`

```
pub fn poke_range(&mut self, start_x: usize, start_y: usize,
    size: usize, ignore_b: bool) -> bool {

    if size == 1 {
        if !self.self_connections_allowed && start_x == start_y {
            return false;
        }

        if self.connections[start_y][start_x] {
            return false;
        } else {
            self.connections[start_y][start_x] = true;
            return true;
        }
    }

    let mut searched = [false; 4];

    if ignore_b {
        searched[1] = true;
    }

    loop {
        let mut quarter = self.propabilities.get_random_quarter();

        if ignore_b && (quarter == 1) {
            quarter = 2;
        }

        if searched[quarter as usize] {
            continue;
        } else {
            searched[quarter as usize] = true;
        }
        let size = size / 2;
        let mut start_x = start_x;
        let mut start_y = start_y;

        if quarter % 2 != 0 {
            start_x += size;
        }
        if quarter > 1 {
            start_y += size;
```

```
        }

        let ignore_b = match ignore_b {
            true => quarter != 2,
            false => false,
        };

        if self.poke_range(start_x, start_y, size, ignore_b) {
            return true;
        }

        if searched.iter().all(|&x| x == true) {
            return false;
        }
    }
}
```

Funkcja `poke_range` jest rekurencyjną funkcją, która wywołuje samą siebie, jedynie na mniejszym zakresie. Funkcja ta odpowiada za wybór ćwiartki macierzy, w której zostanie wylosowane połączenie. Działa ona na podstawie parametrów `start_x`, `start_y`, `size` oraz `ignore_b`. Parametr `start_x` i `start_y` określają współrzędne lewego górnego wierzchołka, a `size` - wielkość boku kwadratu, w którym znajduje się ćwiartka. Ostatecznie parametr `ignore_b` określa, czy ćwiartka `b` ma być pomijana oraz zamieniana na `c` (co jest przydatne w przypadku grafów nieskierowanych). Zwracane jest `true` w przypadku, gdy udało się wylosować i ustawić połączenie, a `false` w przypadku, gdy nie udało się wylosować żadnego połączenia.

```
if size == 1 {
    if !self.self_connections_allowed && start_x == start_y {
        return false;
    }

    if self.connections[start_y][start_x] {
        return false;
    } else {
        self.connections[start_y][start_x] = true;
        return true;
    }
}
```

Powyższy kod sprawdza, czy funckja operuje na jednym elemencie macierzy. W przypadku gdy w grafie nie mogą wystepować pętle pomijane są elementy na ukośnej macierzy (czyli takie, które mają takie same współrzędne `x` i `y`). Jeśli wylosowany element jest już połączony, to zwracane jest `false`. W przeciwnym wypadku ustawiane jest połączenie na `true` i zwracane jest `true`, co sygnalizuje powodzenie operacji.

```
let mut searched = [false; 4];

if ignore_b {
    searched[1] = true;
}
```

Jeśli funkcja nie operuje na jednym elemencie, to powinna znaleźć kolejne, mniejsze ćwiartki. W celu ułatwienia tego procesu inicjalizowana jest zmienna `searched` w której znajdują się informaja, czy

dana ćwiartka została już przeszukana (`a`: 0, `b`: 1, etc.). Jeśli funkcja została poproszona o ignorowanie ćwiartki `b`, jest ona zaznaczona jako już przeszukana, co powstrzymuje algorytm przed wybraniem jej.

W dalszej części funkcji wywoływana jest pętla, w której wybierana jest losowa ćwiartka:

```
loop {
    let mut quarter = self.propabilities.get_random_quarter();

    if ignore_b && (quarter == 1) {
        quarter = 2;
    }

    if searched[quarter as usize] {
        continue;
    } else {
        searched[quarter as usize] = true;
    }
```

Wpierw losowana jest ćwiartka na podstawie podanych prawdopodobieństw. Następnie, jeśli wybrana ćwiartka jest równa `b` a funkcja została poproszona o jej ignorowanie, to ćwiartka zostaje zamieniona na `c`. Następnie sprawdzane jest, czy ćwiartka została już wcześniej przeszukana. Jeśli tak, to w celu poszukiwania nowej ćwiartki pętla zostaje zrestartowana. Jeśli nie, to ćwiartka zostaje zaznaczona jako przeszukana, a funkcja jest dalej wykonywana.

```
    let size = size / 2;
    let mut start_x = start_x;
    let mut start_y = start_y;

    if quarter % 2 != 0 {
        start_x += size;
    }
    if quarter > 1 {
        start_y += size;
    }
```

Ten fragment kodu odpowiada za znalezienie odpowiedniego zakresu macierzy dla danej ćwiartki. W pierwszym kroku zmienne podane w parametrach funkcji są cieniowane, by nie naruszyć ich wartości dla funkcji wyżej na stosie. Następnie w opytmalny sposób zmienanie są zmienne `start_x` i `start_y`. Patrząc na układ ćwiartek można zauważyć, że koordynat `start_x` powinny być zmienone tylko wtedy, gdy wybrana ćwiartka to `b` lub `d`, a koordynat `start_y` tylko wtedy, gdy wybrana to `c` lub `d`. Na podstawie tych obserwacji można dojść do wniosku, że po zamianie liter na odpowiadające im liczby (`a`: 0, `b`: 1, `c`: 2, `d`: 3) `start_x` należy zwiekszyć o `size`, jeśli wybrana ćwiartka jest nieparzysta, a `start_y` należy zwiekszyć o `size`, jeśli wybrana ćwiartka jest większa od 1. W ten sposób zakres szukania zostaje optymalnie zmniejszony.

```
    let ignore_b = match ignore_b {
        true => quarter != 2, // c = 2
        false => false,
    };
```

W dalszej kolejności sprawdzane jest, czy do kolejnego wywołania funkcji należy ignorować ćwiartkę `b`. Operuje to na założeniu, że w grafie nieskierowanym chcemy unikać ćwiartki `b` aż do momentu gdy natrafimy na ćwiartkę `c` (jedynie w ćwiartce `c` wszystkie elementy są częścią grafu).

```
    if self.poke_range(start_x, start_y, size, ignore_b) {
        return true;
    }
```

W przedostatniej części pętli wywoływana jest funkcja `poke_range` z nowymi parametrami. Jeśli zwróciła ona `true`, to znaczy, że operacja się powiodła. Ta informacja zostaje następnie propagowana do góry stosu by zakończyć działanie funkcji.

```
    if searched.iter().all(|&x| x == true) {
        return false;
    }
}
```

Ostatecznie sprawdzane jest, czy wszystkie ćwiartki zostały już przeszukane. Jeśli tak, to znaczy, że nie udało się znaleźć żadnego wolnego miejsca w danej ćwiartce, więc funkcja wyżej w stosie zostaje powiadomiona o niepowodzeniu operacji i kontynuuje poszukiwania na innych ćwiartkach.

3.3 Funkcja `graph.poke`

```
pub fn poke(&mut self) -> bool {
    let size = self.connections.len();
    self.poke_range(0, 0, size, !self.directed)
}
```

Działanie funkcji `poke` jest proste - jej zadaniem jest wywołanie `poke_range` na całej macierzy, by wśród niej znaleźć i ustawić element nie połączony z żadnym innym na `true`. Jeśli graf jest nieukierunkowany, to parametr `ignore_b` jest ustawiany na `true`, by uniknąć ćwiartki `b` (patrz opis funkcji `poke_range`).

3.4 Funkcja `graph.fill`

```
pub fn fill(&mut self) {
    // pole macierzy - elementy na przekątnej
    let mut total_fillable = 4_usize.pow(self.n) - 2_usize.pow(self.n);
    if !self.directed {
        total_fillable
    }
    if self.self_connections_allowed {
        total_fillable += 2_usize.pow(self.n);
    }
    println!("Total fillable: {}", total_fillable);
    let to_fill = (total_fillable as f64 * self.dest_density) as usize;

    let mut filled = 0;

    while self.poke() {
        filled += 1;
        if filled >= to_fill {
            break;
        }
    }
    println!("Filled {} fields", filled);
}
```

Funkcja znaduje ilość elementów potrzebnych by wypełnić macierz, po czym wywołuje funkcję `poke` tyle razy, by wypełnić tę ilość elementów. Funkcja bierze pod uwagę, czy graf jest skierowany czy nie, oraz czy graf ma pozwolić na połączenia z samym sobą.

4 Wydajność

Poniżej przedstawione są wyniki testów wydajności programu. Testy przeprowadzono we wszystkich przypadkach dla grafu skierowanego, bez pętli, z docelową gęstością 50 oraz z prawodopodobieństwami połączeń $0.1, 0.2, 0.3, 0.4$. Dla każdej ilości wierzchołków dokonano 10 testów, z czego następnie wyliczono średnią oraz odchylenie standardowe. Testy zostały wykonane na komputerze z procesorem ARM. Program został skompilowany przy użyciu wbudowanego narzędia `cargo` z flagą `--release`. Czas został zmierzony przy użyciu funkcji `std::time::Instant`.

Ilość wierzchołków	Czas średni	Czas min.	Czas max.
4	19.38 µs ± 1.87	16 µs	22 µs
5	187.62 µs ± 17.51	161 µs	214 µs
6	2.28 ms ± 0.19	2.03 ms	2.6 ms
7	32.16 ms ± 1.9	29.59 ms	34.61 ms
8	506.76 ms ± 13.09	487.03 ms	530.11 ms
9	7.83 s ± 0.08	7.73 s	7.99 s
10	126.45 s ± 0.98	125.21 s	128.8 s

Jak można zauważyć w tabeli, z każdym zwiększeniem ilości wierzchołków czas wykonywania programu zwiększa się o 16 razy. Taka złożoność czasowa może wynikać między innymi z racji, że macierz generowana jest o wielkości 2^n, co wpływa znacząco na szukanie elementów w ćwiartkach w rekursywnej funkcji `poke_range`. Dlatego też podczas korzystania z programu należy być świadomym jego potencjalnie długiego czasu wykonywania.

Bibliografia

[1] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *SIAM Proceedings Series*, Apr. 2004, vol. 6. doi: 10.1137/1.9781611972740.43.