Opis Implementacji

W implementacji trzymamy następujące struktury globalne:
(postawiłem zostawić je globalne dla szybkości dostępu. Nie bojąc się o rozspójnienie zależności w programie dlatego że te są zdefiniowane przez algorytm)

vector<uint32_t> originalNodesIds; gdzie originalNodesIds[i] to wczytana wartość wierzchołka który po znormalizowaniu posiada indeks i

queue<uint32_t> Q; kolejka wierzchołków do równoległego przetwarzania
unordered_set<uint32_t> verticesToRepeat; wierzchołki którym w trakcie przetwarzania Q została anulowana choć jedna propozycja bądź nie posiadają już odpowiednio dużych krawędzi (muszą poczekać na kolejne przetworzenie)

std::mutex **lock_Q**;
std::mutex **lock_verticesToRepeat**; ochrona współdzielonych kolejek

vector< vector<Edge> > adjList; lista sąsiedztwa grafu

```
struct Edge
{
   uint32_t neighbour;  indeks po znormalizowaniu
   uint32_t originalNeighbour; index przed normalizacją do zapewnienia ostrego porządku
   uint32_t size;
};
```

vector<uint32_t> posInAdjList; gdy lista sąsiedztwa jest posortowana malejąca to wystarczy ją przejść dla danego generatora <= b-limit najwyżej 1 raz, ponieważ jeśli propozycja dla danego wierzchołka została anulowana albo nie udało się zostać adoratorem to znaczy że już zawsze jego najmniejsza oferta jest lepsze od naszej.

vector<SuitorsQueue> suitors; **suitors**[v] to kolejka wierzchołków które adorują v
vector<uint32_t> sendProposals; **sendProposals**[v] to ilość adorowanych sąsiadów przez wierzchołek v
vector<uint32_t> bValue; obliczone dla każdego wierzchołka b(v) (zgodnie z STL kontenery tylko do odczytu są bezpieczne wielowątkowo)

vector<std::mutex> suitorsProtection; **suitorsProtection**[v] chroni **suitors**[v]
vector<std::mutex> sendProposalsProtection; **sendProposalsProtection**[v] chroni **sendProposals**[v]

vector<uint32_t> partialSortUpperBound; lista sąsiedztwa w danej danej chwili jest posortowana na indeksach [0, partialSortUpperBound[v]) (element ulepszenia "częściowego sortowania")

vector<uint32_t> pValue; pValue[v] to ilość wierzchołków które są dosortowywane po dojściu w liście sąsiedztwa do indeksu partialSortUpperBound[v]

findValueOfbMatching znajduje dla każdego wierzchołka sumę wag do wierzchołków adorowanych i zwraca ich sumę.

Łączy ona najpierw wierzchołki których krawędzie są co najmniej rozmiaru **heavyEdgesPivotVal** (*)
w kolejnych przebiegach użyte są wszystkie pozostałe krawędzie (**)

```
uint32_t findValueOfbMatching(uint32_t generator, uint32_t heavyEdgesPivotVal) {
   setBPvalues(generator); oblicza pValue oraz bValue
   clearSuitorsAssociatedSetsAndQueues();
   moveAllVerticesTo_Q();
   process_Q(heavyEdgesPivotVal); *
   while(!Q.empty()) {
      process_Q(USE_LIGHT_EDGES); **
   }
   return sumOfAllSuitorsEdges();
}
```

Wykonanie **process_Q(param):**
wykonywane przez wątek główny dopóki nie wyczerpał limitu wątków zleca wykonanie znalezienia sąsiadów do adorowania dla wierzchołka **nextVertex** (processVertex(nextVertex);) innym wątkom. Gdy wyczerpie limit wykona go sam.

Gdy wątek znajdzie wierzchołki do adoracji próboje wziąć kolejny wierzchołek z Q dzięki czemu główny proces nie musi zajmować się zlecaniem prac.

```
void processVertex(uint32_t vertex) {
   bool haveNextVertex = true;
   while(haveNextVertex) {
      makeProposes(vertex);
      vertex = getNextVertex(haveNextVertex);
   }
}
```

Ostatnia już funkcja **makeProposes(uint32_t vertex)** dla danego wierzchołka znajduje mu wierzchołki do adoracji.
Znajdujemy więc kandydata v do adoracji **(1)**

blokujemy jego kolejkę adoratorów **(2)**

Jeśli wciąż jest odpowiedni do adorujemy go **(3)**

opuszczamy zamek na adoratorach v **(4)**

zwiększamy licznik naszych propozycji **(5)** (robimy to bez potrzeby blokowania jednocześnie kolejki adoratorów v, gdyż nawet jeśli inne wierzchołki zdołają zmniejszyć liczbę naszych propozycji to może ona spaść najwyżej do -1, a po podniesieniu znowu wyniesie 0)

wrzucamy go do kolejki **verticesToRepeat (6)**

```
void makeProposes(uint32_t vertex) {
   uint32_t proposes = currentProposalsNumber(vertex);
   Edge* partner;
   uint32_t annuledVertex;
   bool proposalAnnuled;
   bool createdProposal;

   while (proposes < bValue[vertex] && !neighboursExhausted(vertex)) {
      createdProposal = false;
      partner = findEligiblePartner(vertex); (1)
      if (partner != EXAUSTED) {
         suitorsProtection[partner->neighbour].lock(); (2)
         if(isEligiblePartner(vertex, *partner, !LOCK_SUITORS_QUEUE)) {
            proposalAnnuled = makeUsuitorOfP(annuledVertex, vertex,
partner->size, partner->neighbour); (3)
            createdProposal = true;
            proposes++;
         }
         suitorsProtection[partner->neighbour].unlock(); (4)
         if (createdProposal) {
            addToProposals(vertex, partner->neighbour); (5)
            updateAnnuledVertex(proposalAnnuled, annuledVertex,
partner->neighbour); (6)
         }
      }
   }
}
```

Zastosowane ulepszenia:

PARTIAL SORTING
Wykorzystywany jest PA (partial sorting) który na raz dosortowywuje kolejne **pValue**[v] wierzchołków. Wartość **pValue**[v] to 9*bValue[i]; (zaczerpnięte z Khan gdyż kazuje się najlepsze dla większości grafów). Ulepszenie to mocno traci dla rosnącego b-limit.

ORDER OF PROCESSING
Pierwsze wykonanie dla Q szuka kandydatów tylko wśród sąsiadów do których krawędź s spełnia **s >= heavyEdgesPivotVal** co zmniejsza liczbę anulowanych propozycji. ponownie Khan zaleca znalezienie B = P $\sum v \in V$ b(v) i użycia wartości PivotS = k*B-tej krawędzi (w kolejności rosnącej) jako elementu dzielącego dla małych k = 1,2,3,4,5. Jednak dla dużego b-limit wyszukiwanie za każdym razem
nth_element (edges.begin(), universalPivot, edges.end());
jest zbyt dużym kosztem. Dlatego postanowiłem dobrać uniwersalną wartość na początku programu. Dającą wartości zbliżone do PivotS , a dla grafów o nie dużym rozrzucie wag np ze zbioru {1,2,3,4,5,6} często lepsze.

```
uint32_t findUniversalEdgePivot() {
    uint32_t k = 5;
    uint32_t d = 5;
    uint32_t c = 2;
    uint32_t B = (uint32_t)adjList.size()*k;
    auto universalPivot = edges.begin()+B*k;
    nth_element(edges.begin(), universalPivot, edges.end());
    while(k > 0) {
        if(universalPivot < edges.end()) {
            nth_element (edges.begin(), universalPivot, edges.end());
            return *(universalPivot)+findEdgeAverage()/d+c;
        }
        k--;
        universalPivot = edges.begin()+B*k;
    }
    return 0;
}
```

Dodatkowo wierzchołek odrzucony nie szuka dla siebie adoratora od razu ale zrobi to w kolejnym przejściu Q. (Gdy również dopiero użyte zostaną wszystkie wagi krawędzi)

Złożoność algorytmu dla b-limit = 0 (jedno wykonanie algorytmu):
m - liczba krawędzi grafu
n - liczba wierzchołków
β - $\max v \in V\ b(v)$
p - $\max v \in V\ p(v)$ gdzie p(v) liczba sąsiadów v
c - maksymalna liczba partii do posortowania dla danego wierzchołka w jego liście sąsiedztwa
Wtedy średni koszt znalezienia findUniversalEdgePivot() to **O(m + m)** obliczenie średniej wagi i std::nth elementu, w najgorszym **O(m + m^2)**

znalezienie adoratorów: **O(ncp log p + m log β)** (dla każdego wierzchołka wykonane zostanie maksymalnie c razy partial sorting o koszcie log p oraz m log β będziemy poprawiać kolejkę adoratorów wierzchołka)

Czasy działania

Procesor:
ID producenta: GenuineIntel
Rodzina CPU: 6
Model: 42
Model name: Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz
Wersja: 7

Czasy działania dla http://snap.stanford.edu/data/as-skitter.html uproszczonego do 80MB.
dla
b-limit = 30
przy średnim czasie sekwencyjnego wczytywania 0m4.000s +- 0.05s

p_m	1	2	3	4	5	6	7	8	48	100
real (czas działania)	0m37.605s	0m38.666s	0m34.970s	35.866s	0m35.328s	0m35.382s	0m34.944s	0m34.224s	0m33.794s	0m33.470s
user	0m37.060s	1m0.280s	1m6.068s	1m17.284s	1m15.612s	1m13.912s	1m12.108s	1m10.068s	1m8.844s	1m8.052s
sys	0m0.292s	0m7.084s	0m16.664s	0m30.736s	0m33.204s	0m33.908s	0m35.788s	0m38.032s	0m39.400s	0m39.868s
p_1/p_m	1	0.972559	1,0753503	1,0484860	1,0644531	1,0628285	1,0761504	1,0987903	1,1127714	1,1235434

wyniki po wykorzystaniu optymalizacji -03. Doświadczalnie okazało się że optymalizacja znacznie zmniejszyła stosunek p_1/p_m.