Generating Realistic Optical Transport Network Topologies

João Gehlen, Roberto Delavi, Watson V. C. Junior, Winder Dias[1]

[1]Universidade Federal da Fronteira Sul (UFFS)

{roberto.delavy, joaogehlen91, watsonmaster, winderdias}@gmail.com

Abstract. *Esse trabalho tem como objetivo a implementação de um algoritmo Generating Realistic Optical Transport Network Topologies. Para isso, foram realizados estudos sobre as características desse tipo de rede [citepavan], de modo que, o modelo gerado siga um formato mais parecido com o modelo de redes óticas reais. De posse dessas informações, o modelo de topologia proposto, basicamente, distribui os nós em R regiões do plano, conecta os nós dentro cada região e termina interligando as regiões.*

1. Criação de divisão do plano

As redes óticas de comunicação aumentaram em tamanho e complexidade, ajustar a arquitetura e a implementação para reduzir custos, simplificar o gerenciamento, melhorar o tempo de resposta e otimizar a utilização dos recursos se tornou crucial. Nesse cenário, os algoritmos geradores de topologias são importantes para a criação e simulação de diferentes tipos de redes, visando estudar o comportamento das redes, antes mesmo de serem executadas na prática. Essa implementação baseia-se no artigo Generating Realistic Optical Transport Network Topologies, que propõe um algoritmo gerador de topologias mais parecido com o modelo real de redes óticas, ou seja, o que é mais usado na prática. Para a construção do algoritmo, realizou-se um levantamento de dados sobre um conjunto de redes reais e verificou-se os parâmetros relevantes para a criação do algoritmo.

Funcionamento do algoritmo:

N, representando o número de nodos;
A, representando a área do plano ((xmin, xmax), (ymin, ymax));
R, representando o numero de regiões;
d, representando a distancia mínima entre dois nodos;
d(G)min, representando o mínimo grau medio;
d(G)max, representando o maximo grau medio;

O primeiro passo do algoritmo é dividir o plano em *R* regiões de tamanho igual, cada região recebe um conjunto *N* de nós, cuja posição dentro de cada região é determinada aleatoriamente. As regiões são criadas em forma de linhas e colunas, que são definidas através de duas variáveis, p1 e p2. Sendo p1 número de colunas de cada região e p2 o número de linhas de cada região. O cálculo destas duas variáveis é realizado a partir de *R* (valor inserido na ntrada da simulação). Para encontrar p1, você precisa encontrar o maior número primo, tal que R é

divisível, assim p1 será o resultado da divisão de R por seu divisor. E p2 será o valor do divisor de *R*.

```
for(int i = (R-1); i >= 1; i--){
   if(((R % i) == 0) && isPrime(i)){
      p1 = (R/i);
      p2 = i;
      break;
   }
}
```

Antes das inserções dos nodos são realizados os sorteios para definir o numero de nodos que serão inseridos em cada regiao *R*. Para melhorar esse sorteio e utilizada uma variável auxiliar i para evitar que muitos nodos fiquem na primeira regiao do plano, e assim que as últimas regioes nao fiquem sem nodos. Sorteio obedecendo o limite de nodos por região, que será a capacidade ja calculada anteriormente, conforme código abaixo:

```
if(N == 1){
    nodosForReg = 1;
    }else{
    nodosForReg = r.nextInt(Math.min((i == 0 ? N/2 : N), (int)cap));
    }

public boolean isPrime(int n){
   for(int i=2; i<n; i++){
      if(n % i == 0)
         return false;
   }
   return true;
}
```

Após obter os valores das variaveis p1 e p2, consegue-se calcular as áreas das regiões R, áreas que serão iguais para todas as regiões do plano. Com o cálculo dado pela largura do plano dividida por p1 e multiplicada pela altura do plano dividido por p2, conforme fórmula abaixo:

$$\frac{xmax - xmin}{p1} \times \frac{ymax - ymin}{p2}$$

Para definir o número máximo de nodos que podem ser inseridos em cada região do plano, obedecendo a regra de que o nodo não pode ser sobreposto sobre outro nodo ou sobre posições reservadas de adjacencias de outros nodos, conforme a distancia informada. Para calcular essa capacidade usa-se a seguinte função:

```
int cap = (int)(areaReg/(d*d));
```

Com esses valores já calculados, é gerado uma matriz para simular o plano, salvando as posiçoes em que iniciam e terminam cada região do plano. Nessa matriz tambem terá indicado as posiçoes que já possuem nodos, posições adjacentes aos nodos e as posições livres.

```
int indiceRegiao[ ][ ] = new int[R][4];
int nreg = 0;
for(int i = 0; i < lin; i++){
   for(int j = 0; j < col; j++){
      indiceRegiao[nreg][0] = sidex * i;
      indiceRegiao[nreg][1] = sidey * j;
      indiceRegiao[nreg][2] = sidex * (i+1);
      indiceRegiao[nreg][3] = sidey * (j+1);
      nreg++;
   }
}
```

2. Inserção dos nodos

Antes das inserções dos nodos são realizados os sorteios para definir o numero de nodos que serão inseridos em cada regiao *R*. Para melhorar esse sorteio e utilizada uma variável auxiliar i para evitar que muitos nodos fiquem na primeira regiao do plano, e assim que as últimas regioes nao fiquem sem nodos. Sorteio obedecendo o limite de nodos por região, que será a capacidade ja calculada anteriormente, conforme código abaixo:

```
if(N == 1){
   nodosForReg = 1;
   }else{
      nodosForReg = r.nextInt(Math.min((i == 0 ? N/2 : N), (int)cap));
   }
```

Ao inserir um novo nodo na matriz teria que ser verificado se a posição sorteada estaria válida (vazia), caso nao estivesse, a posição teria que ser sorteada novamente, até encontrar uma posiçao válida, assim podendo haver muitos sorteios desnecessários. Para não ocorrer este problema, foi criado um vetor contendo somente as posiçoes válidas para inserção. Vetor que inicia com todas as posiçoes do plano, quando um nodo é inserido no plano, as posições do nodo e de suas adjacencias sao removidas deste vetor, assim restando somente posições válidas para inserçao. Com o vetor contendo somente posiçoes válidas, ao inserir um novo nodo, a posição de inserção é sorteada neste vetor, desta forma sempre será sorteada uma posição válida. Isso ocorre no codigo abaixo, onde as variaveis pi e pj corresponderão a posição no plano para a inserção do nodo, posição obtida a partir do acesso posicoes.get(posicaoSorteada).getFirst()/Second(), ao vetor. Após o nodo ser inserido no plano, a posição sorteada é removida do vetor, pois não estará mais vazia.

```
List<Long> idNodesReg = new ArrayList<Long>();
for(int k = 0; k < nodosForReg; k++){
   if(posicoes.size() > 0){
      int posicaoSorteada = r.nextInt(posicoes.size());
      int pi = posicoes.get(posicaoSorteada).getFirst();
      int pj = posicoes.get(posicaoSorteada).getSecond();
      //
```

```
      if(plano[pi][pj] == 0){
      // se está livre a posicao, mais um teste para garantir
         idNodesReg.add(netPlan.addNode(pi, pj, k+" "+i, null));
         posicoes.remove(posicaoSorteada);
         plano = insereNodo(pi, pj, plano, x, y, (int) d);
         N--;
      }
   }
}
idNodesRegioes.add(idNodesReg);
if(N==0) break;
```

3. Inserção das Arestas

Para a inserção das arestas, inicialmente é selecionado o primeiro nodo do plano (nodo com o menor ID), e a partir dele é realizado o cálculo para obter a maior distância Euclidiana entre este nodo e os demais da mesma região, tal valor que é usado para realizar o cálculo da maior probabilidade de inserção de aresta da região. No código abaixo é realizado o cálculo da distância máxima.

```
for (long destinationNodeId : reg){
for (long originNodeId : reg){
   if (originNodeId >= destinationNodeId) break;
      double dist = netPlan.getNodePairEuclideanDistance(originNodeId,
      destinationNodeId);
      if (dist > dist_max) dist_max = dist;
      }
}
```

Na abordagem do modelo proposto por Waxman, modelo que segue uma distribuição de Poisson com grau nodal [48, 49], com tendências de redes de transporte óptico do mundo real. Na abordagem de Waxman é utilizada a seguinte fórmula para calcular a probabilidade de existir um link entre os nodos da região:

$$P(i,j) = \beta \exp \frac{-d(i,j)}{\zeta \alpha} \tag{1}$$

Onde d(i, j) é a distancia Euclidiana entre dois nodos i e j. Dzeta é a distância maxima entre dois nodos. Alpha e betha são parâmetros na faixa de (0,1]. Com a distância maxima ja calculada, é hora de utilizar a mesma para calcular a probabilidade (p), calculo realizado e testado com todos os nodos da região, determinando qual é o nodo mais apto para ser ligado com o nodo corrente, que após calcular o nodo N0 mais apto ao nodo N0, N0 é salvo em um vetor auxiliar e então é realizado o mesmo cálculo para o nodo N1 e assim por diante, até o fim da regiao. Este vetor auxiliar, chamado regOrdenada, será utilizado para guardar os nodos ordenadamente, vetor criado para facilitar a inserção das arestas, pois o mesmo terá salvo o ID de cada nodo para linkagem, onde será linkado o nodo atual com o seu próximo e assim por diante, até chegar ao nodo final, nodo que

será linkado com o primeiro. Por exemplo:
regOrdenada: N0, N1, N2, N3, N4, N5, N6, ... , Nn-1;
Onde:
N0 <-> N1, N1 <-> N2, N2 <-> N3, N3 <-> N4, N4 <-> N5, N5 <-> N6 e Nn-1 <-> N0.
Conforme código abaixo:

```
List<Long> regOrdenada = new ArrayList<Long>();
if(!reg.isEmpty()) regOrdenada.add(reg.get(i));
while(reg.size()>1){
   int nexti = 0;
   long originNodeId = reg.get(i), idD = 0;
   double pmax = 0, dist = 0;
   for (int j = 0; j < reg.size(); j++) {
   long destinationNodeId = reg.get(j);
   if (originNodeId == destinationNodeId) continue;
      dist = netPlan.getNodePairEuclideanDistance(originNodeId,
      destinationNodeId);
      double p = alpha * Math.exp(-dist / (beta * dist_max));
   if(p > pmax) {
      pmax = p;
      idD = destinationNodeId;
      nexti = j;
   }
}
regOrdenada.add(idD);
reg.remove(i);
if(i > nexti){
   i = nexti;
   }else{
      i = (nexti -1);
   }
}
```

Com o vetor preenchido ordenadamente com os ID's dos nodos a serem linkados, são adicionadas as arestas a partir dos mesmo, conforme código abaixo:

```
if(regOrdenada.size()==2){
   netPlan.addLink(regOrdenada.get(0),
   regOrdenada.get(regOrdenada.size()-1), linkCapacities, 10, null);
   netPlan.addLink(regOrdenada.get(regOrdenada.size()-1),
   regOrdenada.get(0), linkCapacities, 10, null);
   }else{
      for (int j = 1; j < regOrdenada.size(); j++) {
         long origin = regOrdenada.get(j-1), destination = regOrdenada.get(j);
         if (origin == destination) continue;
         netPlan.addLink(origin, destination, linkCapacities, 10, null);
         netPlan.addLink(destination, origin, linkCapacities, 10, null);
   }
```

```
   if(regOrdenada.size()>2){
      netPlan.addLink(regOrdenada.get(0),
      regOrdenada.get(regOrdenada.size()-1), linkCapacities, 10, null);
      netPlan.addLink(regOrdenada.get(regOrdenada.size()-1),
      regOrdenada.get(0), linkCapacities, 10, null);
   }
}
```

Após linkar todos os nodos das regiões do plano, basta apenas interligar as regiões, onde será realizada basicamente a mesma metodologia realizada para linkar os nodos de cada região. Sendo calculada a distancia Euclidiana máxima para cada nodo, calculado a probabilidade máxima e inserido os nodos ordenadamente no vetor auxiliar, porém, varrendo todos os nodos do plano. Tambe sera criada uma lista de lista, onde cada lista contém os nodos de cada regiao do plano, onde é varrido as duas listas e encontrado os dois nodos a serem linkados, é realizada a linkagem entre os dois nodos no plano e em seguida removido estes dois nodos das listas de controle, para que as mesmas não sejam selecionadas novamente para receber um link no próximo passo. conforme código abaixo:

```
int indexNoOrigem = 0, indexNoDestino = 0, indexRegOrigem = 0, indexRegDestino =
for (int j = 0; j < 2; j++) {

for (List<Long> reg1 : idNodesRegioesAux) {
   long noOrigem = 0, noDestino = 0;
   for (Long no1 : reg1) {
      double pmax = 0;
      for (List<Long> reg2 : idNodesRegioesAux) {
         if(reg1 == reg2 ) continue;
         for (Long no2 : reg2) {
            double dist = netPlan.getNodePairEuclideanDistance
            (no1, no2);
            double p = alpha * Math.exp(-dist / (beta * dist_max));
            if(p < pmax) {
               pmax = p;
               noOrigem = no1;
               noDestino = no2;
               indexRegOrigem = idNodesRegioesAux.indexOf(reg1);
               indexRegDestino = idNodesRegioesAux.indexOf(reg2);
               indexNoOrigem = reg1.indexOf(noOrigem);
               indexNoDestino = reg2.indexOf(noDestino);
            }
       }
   }
}

if (noDestino != noOrigem) {
   netPlan.addLink(noOrigem, noDestino, linkCapacities, 10, null);
```

```
   netPlan.addLink(noDestino, noOrigem, linkCapacities, 10, null);
      if (idNodesRegioesAux.get(indexRegOrigem).size() > 1 || j == 1) {
         idNodesRegioesAux.get(indexRegOrigem).remove(indexNoOrigem);
         }

        if (idNodesRegioesAux.get(indexRegDestino).size() > 1 || j == 1){
        idNodesRegioesAux.get(indexRegDestino).remove(indexNoDestino);
      }
   }
}
```