Aplicação do Algoritmo de Dijkstra e do de Eppstein: Prevendo Cadeias Mais Prováveis do Comércio de Madeira

Gabrielle Scherer Mascarelo
Gabriel Macri Mello Cardoso Glioche

Novembro 2024

Conteúdo

1 Introdução

Além da mineiração e de fenômenos naturais, um fator de peso para a degradação das florestas brasileiras é o desmatamento ilegal. Como medida preventiva, o governo do Brasil implementou mecanismos de regularização do transporte da madeira no país, nos quais todo produto advindo da obra-prima deve entrar nos sistemas de controle antes de ser transportado. Assim, toda movimentação fica registrada em bases de dados extensas para análise de irregularidades.

Entretanto, mesmo com os registros feitos, a inspeção continua sendo custosa. Cria-se um lacuna a respeito do caminho exato que a madeira percorre pelas empresas e pessoas nos textos, sendo necessário calcular as possíveis mais prováveis rotas que ela segue para contemplar o panorama completo. Tal cálculo pode ser feito usando algoritmos e aplicando os dados em Teoria dos Grafos, de modo com que empresas fornecedoras e compradoras sejam vértices e as transações sejam arestas dirigidas da origem para o destino tendo o peso do volume de madeira respectivo em um grande grafo possivelmente desconexo. A melhor estimativa é feita usando o racíocinio de que a rota teria a melhor taxa entre maior volume possível transportado no menor caminho possível. Para construir essa previsão é necessário compreender as ferramentas que serão utilizadas: o Algoritmo de Dijkstra [4] e o Algoritmo de Eppstein [5]. Este trabalho se concentra na explicação e no aprofundamento da aplicação dos algoritmos para resolver a questão discutida e agregar na análise das irregularidades do comércio de madeira no Brasil ao abordar as chamadas "Timber Trade Networks", redes de comércio da madeira [6][7]. Ao longo do artigo foram usados códigos em Python que podem ser encontrados no seguinte repositório do GitHub: $https://github.com/g4briwlle/Eppstein_in_Timber_Chains$

2 Algoritmo de Dijkstra

O algoritmo de Dijkstra é uma técnica amplamente utilizada no estudo dos grafos para encontrar o menor caminho em grafos ponderados com arestas de pesos não negativos. Seu objetivo é determinar o menor custo necessário para alcançar todos os outros vértices a partir de um vértice de origem, registrando também os caminhos correspondentes. Por sua eficiência e simplicidade, ele é amplamente aplicado em problemas práticos, como roteamento de redes, sistemas de navegação e análise de cadeias logísticas.

O funcionamento do algoritmo de Dijkstra baseia-se em dois conceitos fundamentais: a propagação progressiva de custos e uma estratégia gananciosa para decidir qual vértice explorar em cada etapa. No grafo G (que possui os subconjuntos "E"das arestas e "V"dos vértices), atribui-se um valor temporário (que em dado momento fica definitivo) para cada elemento de V, que chamaremos de custo $L(x)$ para o vértice x, associado às distâncias entre os vértices e a origem.

Inicialmente, todas as distâncias entre o vértice inicial e o restante são consideradas

infinitas, e para o vértice de origem o custo é definido como zero. Começando pela fixação da origem, a cada iteração o algoritmo seleciona um vértice com a menor distância acumulada entre os que ainda não foram processados, sendo que no empate de custos (como em todo início) a escolha é aleatória e não altera o resultado. Em seguida, ele calcula os custos acumulados para os vizinhos desse vértice e atualiza suas distâncias, caso um caminho mais curto seja encontrado conforme o cálculo feito. O vértice processado é então marcado como visitado e considerado definitivo, sendo garantido que sua distância não será alterada em etapas futuras. Esse processo é repetido até que todos os vértices tenham sido processados ou a fila de prioridade esteja vazia. Abaixo segue apenas um pseudo-código [1] com as etapas da ferramenta. Considere que w é o peso da aresta em questão, a é o vértice inicial, z é o final e que L retorna o custo de um vértice.

```
1.  def dijkstria(w,a,z,L):
2.  L(a) = 0
3.  for all x != a:
4.       L(x) = infinite
5.  T = set of all vertices
6.  while (z in T) :
7.      choose v in T with min L(v)
8.      T = T\{v}
9.      for all x in T adjacent to v:
10.         L(x) = min(L(x), L(v)+w(v,x))
```

Após a inicialização já descrita (linha 2 à 4), dentro do conjunto de todos os vértices se escolhe um com o menor custo (linha 6 e 7) - note que a primeira iteração sempre escolhe a, remove-o desse conjunto de modo com que ele não seja mais analisado (linha 8), e atribui a todo vèrtice adjacente o menor valor entre i) o custo que ele já tem; e ii) o custo que o vértice em análise na iteração tem somado com o peso a aresta que liga ele com o vizinho em questão (linha 10). Na realização do algortimo, pode-se armazenar o histórico de iterações em tabelas para uma visualização completa dos trajetos.

A propagação progressiva descreve como o algoritmo constrói as distâncias acumuladas de maneira incremental, partindo do vértice de origem e estendendo os caminhos gradualmente para os vizinhos. À medida que o grafo é explorado, as distâncias acumuladas podem apenas aumentar ou permanecer as mesmas, graças à ausência de pesos negativos. Isso garante que cada vértice, ao ser visitado, já contenha o menor caminho possível para ele. Esse comportamento se alinha diretamente com o princípio de otimalidade dos caminhos mais curtos: o menor caminho de A para Z, por exemplo, deve conter o menor caminho de A para M e M para Z, se M for um ponto intermediário.[2]

Já a estratégia gananciosa do Dijkstra aproveita o fato de que, ao escolher o vértice com a menor distância acumulada no momento, esse vértice representa um ótimo local (custo localmente mínimo). Graças à propagação progressiva, esse ótimo local também reflete o ótimo global já que a distância acumulada de um vértice não pode ser reduzida

após ele ser processado.[3] Essa propriedade confere eficiência, dispensando a necessidade de revisitar vértices já processados.

A combinação entre a propagação progressiva e a estratégia gananciosa é o que habilita o algoritmo de Dijkstra resolver certos problemas de caminhos mais curtos. Entretanto, o algoritmo não é naturalmente adequado para grafos com arestas de pesos negativos, pois sua estratégia de tornar imutável o custo de vértices visitados impede que distâncias já registradas sejam revisadas, não lidando com ciclos negativos, no qual o custo de um caminho pode ser reduzido indefinidamente. Entretanto, o raciocínio que será usado posteriormente implica a necessidade de passar por essa configuração, tal problema será discutido e resolvido quando for invocado.

A seguir, é exemplificado o funcionamento do algoritmo de Dijkstra no contexto de um grafo representando uma Timber Trade Network (TNN).

Um grafo direcionado, cujos vértices são categorizados da seguinte forma: florestas licenciadas = [‘F1’, ‘F2’], companhias = [‘C1’, ‘C2’, ‘C3’] e consumidores finais = [‘E1’, ‘E2’]. As arestas do grafo, juntamente com seus respectivos pesos, são: (‘F1’, ‘C1’, 5), (‘F1’, ‘C2’, 3), (‘F2’, ‘C2’, 7), (‘C1’, ‘C3’, 2), (‘C2’, ‘E1’, 6), (‘C3’, ‘E2’, 1) e (‘C2’, ‘E2’, 6).

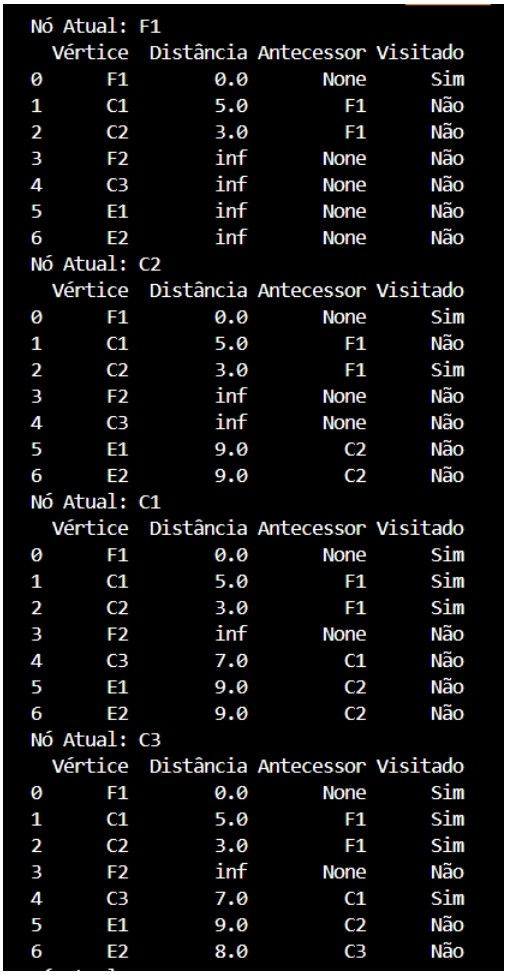
Note que as florestas licenciadas possuem apenas evasão de madeira, enquanto os consumidores finais apenas recebem madeira.

O algoritmo foi replicado em Python para ilustrar seu funcionamento. As Figuras 2 e 3 representam as saídas geradas pelo código, demonstrando o passo a passo do cálculo das menores distâncias a partir do vértice ‘F1’ para os demais vértices do grafo. Como esperado, a distância de ‘F1’ para ‘F2’ é infinita, uma vez que não existe um caminho viável entre florestas no grafo.

O vértice inicial foi escolhido como ‘F1’. A partir desse vértice, são calculadas as distâncias para os vértices adjacentes ainda não visitados, registrando também o antecessor necessário para alcançá-los (neste caso, o próprio ‘F1’). Após essa etapa, o vértice ‘F1’ é marcado como visitado.

Na etapa seguinte, o algoritmo seleciona o próximo vértice a ser processado: aquele que não foi visitado e possui o menor caminho acumulado. Aqui, o vértice escolhido é ‘C2’, com um custo acumulado de 3. A partir de ‘C2’, são calculadas as distâncias acumuladas para os vértices adjacentes ainda não visitados (‘E1’ e ‘E2’, ambas com custo 9), registrando ‘C2’ como antecessor de ambos. Após essa etapa, ‘C2’ é marcado como visitado.

O algoritmo segue iterativamente, verificando e atualizando os menores caminhos para os vértices não visitados. Sempre que encontrar um caminho mais curto do que o previamente registrado para um vértice, essa atualização é feita. Esse processo continua até que todos os vértices do grafo sejam processados.



```
Nó Atual: F1
  Vértice  Distância Antecessor Visitado
0      F1        0.0       None      Sim
1      C1        5.0         F1      Não
2      C2        3.0         F1      Não
3      F2        inf       None      Não
4      C3        inf       None      Não
5      E1        inf       None      Não
6      E2        inf       None      Não
Nó Atual: C2
  Vértice  Distância Antecessor Visitado
0      F1        0.0       None      Sim
1      C1        5.0         F1      Não
2      C2        3.0         F1      Sim
3      F2        inf       None      Não
4      C3        inf       None      Não
5      E1        9.0         C2      Não
6      E2        9.0         C2      Não
Nó Atual: C1
  Vértice  Distância Antecessor Visitado
0      F1        0.0       None      Sim
1      C1        5.0         F1      Sim
2      C2        3.0         F1      Sim
3      F2        inf       None      Não
4      C3        7.0         C1      Não
5      E1        9.0         C2      Não
6      E2        9.0         C2      Não
Nó Atual: C3
  Vértice  Distância Antecessor Visitado
0      F1        0.0       None      Sim
1      C1        5.0         F1      Sim
2      C2        3.0         F1      Sim
3      F2        inf       None      Não
4      C3        7.0         C1      Sim
5      E1        9.0         C2      Não
6      E2        8.0         C3      Não
```

(a) Quatro primeiras etapas.

```
Nó Atual: E2
  Vértice  Distância Antecessor Visitado
0      F1        0.0       None      Sim
1      C1        5.0         F1      Sim
2      C2        3.0         F1      Sim
3      F2        inf       None      Não
4      C3        7.0         C1      Sim
5      E1        9.0         C2      Não
6      E2        8.0         C3      Sim
Nó Atual: E1
  Vértice  Distância Antecessor Visitado
0      F1        0.0       None      Sim
1      C1        5.0         F1      Sim
2      C2        3.0         F1      Sim
3      F2        inf       None      Não
4      C3        7.0         C1      Sim
5      E1        9.0         C2      Sim
6      E2        8.0         C3      Sim
Nó Atual: F2
  Vértice  Distância Antecessor Visitado
0      F1        0.0       None      Sim
1      C1        5.0         F1      Sim
2      C2        3.0         F1      Sim
3      F2        inf       None      Sim
4      C3        7.0         C1      Sim
5      E1        9.0         C2      Sim
6      E2        8.0         C3      Sim
```

(b) Três últimas etapas

Figura 1: Demonstração completa do passo a passo do algoritmo de Dijkstra.

3 Algoritmo de Eppstein

O algoritmo de Eppstein é uma técnica eficiente para encontrar os k-menores caminhos em um grafo dirigido com pesos não negativos. Ele foi projetado para explorar alternativas ao menor caminho, construindo múltiplas soluções ordenadas por custo de maneira incremental, sem recalcular subcaminhos já processados. Essa abordagem combina a utilização de uma estrutura de dados eficiente, como a fila de prioridade, com a exploração de desvios estratégicos a partir de um caminho base, geralmente o menor caminho obtido pela árvore de caminho mais curto.

O primeiro passo do algoritmo de Eppstein é calcular a árvore de caminho mais curto (SPT, do inglês Shortest Path Tree) do destino t até todos os outros vértices no grafo.

Essa árvore é construída utilizando o algoritmo de Dijkstra reverso, onde as arestas são analisadas no sentido contrário, permitindo determinar as distâncias mínimas de cada vértice ao destino. Essa etapa não apenas fornece o menor caminho de s para t, mas também cria a base para explorar caminhos alternativos.

Após construir a SPT, identificam-se as arestas de desvio, ou seja, as arestas que não pertencem à árvore. Essas arestas representam as possibilidades de desviar do menor caminho em busca de alternativas. Cada aresta de desvio é analisada em termos de seu custo relativo, que é calculado considerando o peso da aresta e as distâncias acumuladas nos vértices que ela conecta. Esse custo adicional é essencial para determinar quais desvios são mais promissores.

O próximo passo é organizar os caminhos em uma fila de prioridade, também conhecida como min-heap. Inicialmente, o menor caminho obtido pela SPT é adicionado à fila. Em seguida, os desvios identificados são utilizados para gerar novos caminhos alternativos, que são inseridos na fila com base no custo total acumulado. A fila de prioridade garante que, a cada iteração, o próximo caminho a ser processado será aquele com o menor custo.

Ao processar um caminho extraído da fila, o algoritmo verifica todos os vértices desse caminho para identificar novos desvios possíveis. Esses desvios, se existirem, geram novos caminhos alternativos, que são adicionados de volta à fila. É importante ressaltar que os desvios de um caminho só são explorados quando o próprio caminho é extraído da fila. Isso evita redundâncias e garante que os caminhos sejam processados na ordem correta.

O processo continua até que os k-menores caminhos sejam identificados. A eficiência do algoritmo reside no fato de que ele evita recalcular caminhos já processados, aproveitando a estrutura da SPT para gerenciar subcaminhos de forma eficiente. Para ilustrar o funcionamento do algoritmo passo a passo, utilizamos o grafo direcionado abaixo. Nesse grafo, os vértices são categorizados da seguinte forma: florestas licenciadas = [‘F1’, ‘F2’, ‘F3’], companhias = [‘C1’, ‘C2’, ‘C3’] e consumidores finais = [‘E1’, ‘E2’, ‘E3’, ‘E4’]. As arestas do grafo, juntamente com seus respectivos pesos, são definidas como: (‘F1’, ‘C1’, 1), (‘F1’, ‘C2’, 2), (‘F2’, ‘C2’, 1), (‘F3’, ‘C2’, 3), (‘F3’, ‘C3’, 2), (‘C1’, ‘E1’, 3), (‘C1’, ‘E3’, 4), (‘C2’, ‘C1’, 2), (‘C2’, ‘C3’, 1), (‘C2’, ’E3’, 7), (‘C3’, ’E2’, 4), (‘C3’, ‘E4’, 3) e (‘C3’, ‘E3’, 2).

Definimos para este exemplo a origem sendo ‘F1’ e o destino ‘E3’. Almejamos encontrar os k=3 menores caminhos entre ‘F1’ e ‘E3’.

A primeira etapa do algoritmo consiste em inverter as arestas do grafo original, gerando um novo grafo direcionado com as conexões invertidas. Após realizar essa operação, o grafo invertido é definido pelas seguintes arestas, juntamente com seus respectivos pesos: (‘C1’, ‘F1’, 1), (‘C2’, ‘F1’, 2), (‘C2’, ‘F2’, 1), (‘C2’, ‘F3’, 3), (‘C3’, ‘F3’, 2), (‘E1’, ‘C1’, 3), (‘E3’, ‘C1’, 4), (‘C1’, ‘C2’, 2), (‘C3’, ‘C2’, 1), (‘E3’, ‘C2’, 7), (‘E2’, ‘C3’, 4), (‘E4’, ‘C3’, 3) e (‘E3’, ‘C3’, 2).

Neste grafo, aplicamos o algoritmo de Dijkstra para calcular as menores distâncias de

‘E3’ até os demais vértices. Considerando o grafo original, isso nos permite determinar as menores distâncias dos demais vértices até ‘E3’. O resultado dessa execução é o que se segue:

Vértice	**Distância**	**Antecessor**
C1	4	E3
F1	5	C2
C2	3	C3
F2	4	C2
F3	4	C3
C3	2	E3
E1	∞	None
E3	0	None
E2	∞	None
E4	∞	None

Tabela 1: Resultado do algoritmo de Dijkstra.

A próxima etapa consiste em criarmos uma árvore com os menores caminhos até ‘E3’, calculados com o algoritmo de Dijkstra. Nossa árvore, portanto, é composta pelas arestas: (‘F1’, ‘C2’, 2), (‘F2’, ‘C2’, 1), (‘C2’, ‘C3’, 1), (‘F3’, ‘C3’, 2), (‘C3’, ‘E3’, 2) e (‘C1’, ‘E3’, 4). Nota-se a ausência dos vértices ‘E1’, ‘E2’ e ‘E4’, uma vez que não existe um caminho viável deles até ‘E3’.

Em seguida, identificamos as arestas de desvio, que são aquelas que não estão contidas na árvore de menor caminho: (‘F1’, ‘C1’, 1), (‘C1’, ‘E1’, 3), (‘C2’, ‘E3’, 7), (‘C3’, ‘E2’, 4), (‘C3’, ‘E4’, 3), (‘C2’, ‘C1’, 2) e (‘F3’, ‘C2’, 3). Após a identificação das arestas de desvio, procedemos ao cálculo de seus respectivos **custos de desvio** em relação à árvore de menor caminho. Para isso, utilizamos a seguinte fórmula:

$$\text{Custo de Desvio} = \text{peso } p \text{ da aresta de desvio } (u, v, p) + \text{dist}(v, t) - \text{dist}(u, t)$$

onde $t =$ ‘E3’ neste caso. As distâncias $\text{dist}(v, t)$ e $\text{dist}(u, t)$ foram previamente calculadas pelo algoritmo de Dijkstra.

Caso uma ou ambas as distâncias envolvidas no cálculo sejam infinitas, a aresta é considerada **inválida** e descartada. Para as demais arestas, o cálculo é realizado normalmente, atribuindo a cada uma o respectivo custo de desvio.

Em nosso exemplo, as arestas válidas, juntamente com seus respectivos custos de desvio já calculados, são: (‘F1’, ‘C1’, 0), (‘C2’, ‘E3’, 4), (‘C2’, ‘C1’, 3), (‘F3’, ‘C2’, 2). A partir dessas informações, construímos um grafo auxiliar que combina as arestas da árvore de menor caminho com as arestas de desvio válidas. Para essas últimas, os pesos originais foram substituídos pelos seus custos de desvio calculados.

Assim, o grafo auxiliar resultante é definido pelas seguintes arestas: (‘C3’, ‘E3’, 2), (‘C1’, ‘E3’, 4), (‘F3’, ‘C3’, 2), (‘C2’, ‘C3’, 1), (‘F1’, ‘C2’, 2), (‘F2’, ‘C2’, 1), (‘F1’, ‘C1’, 0), (‘C2’, ‘E3’, 4), (‘C2’, ‘C1’, 3), (‘F3’, ‘C2’, 2). Destaca-se que as seis primeiras arestas pertencem à árvore de menor caminho, enquanto as quatro últimas correspondem às arestas de desvio, com seus pesos substituídos pelos respectivos custos de desvio.

A etapa final segue o passo a passo:

1. Extraímos o menor caminho (no primeiro loop, o menor caminho é o caminho da árvore).
2. Para cada um dos vértices desse caminho, identificamos (se houver) as arestas de desvio.
3. Para cada uma dessas arestas, calculamos o custo do caminho alternativo.
4. Inserimos esses caminhos na fila de prioridade, que organiza automaticamente em ordem crescente.
5. Extraímos o novo menor caminho...

No nosso cenário, temos:

1. **(Custo, Caminho)** = (5, [‘F1’, ‘C2’, ‘C3’, ‘E3’]).
2. Identificação das arestas de desvio:
   - ‘F1’ → (‘F1’, ‘C1’, 0).
   - ‘C2’ → (‘C2’, ‘E3’, 4) e (‘C2’, ‘C1’, 3).
   - ‘C3’ → não há.
3. Cálculo dos custos de desvio:
   - (‘F1’, ‘C1’, 0) → custo = 5 + 0 = 5, portanto, (5, [‘F1’, ‘C1’, ‘E3’]).
   - (‘C2’, ‘E3’, 4) → custo = 5 + 4 = 9, portanto, (9, [‘F1’, ‘C2’, ‘E3’]).
   - (‘C2’, ‘C1’, 3) → custo = 5 + 3 = 8, portanto, (8, [‘F1’, ‘C2’, ‘C1’, ‘E3]).
4. Fila de prioridade após inserção dos caminhos:
   - (5, [‘F1’, ‘C1’, ‘E3’]).
   - (8, [‘F1’, ‘C2’, ‘C1’, ‘E3]).
   - (9, [‘F1’, ‘C2’, ‘E3’]).
5. Extraímos (5, [‘F1’, ‘C1’, ‘E3’]).
6. Não há desvios extras em ‘F1’, ‘C1’ ou ‘E3’.

7. Nova fila de prioridade:
   - (8, [‘F1’, ‘C2’, ‘C1’, ‘E3’]).
   - (9, [‘F1’, ‘C2’, ‘E3’]).
8. Não precisamos continuar explorando. Já temos os 3 menores caminhos. Qualquer possível caminho oriundo desses terá um custo maior ou igual a 8 e não estará entre os $k = 3$.

   Resposta: (5, [‘F1’, ‘C2’, ‘C3’, ‘E3’]), (5, [‘F1’, ‘C1’, ‘E3’]) e (8, [‘F1’, ‘C2’, ‘C1’, ‘E3’]).

4 Previsão das cadeias

Finalmente, podem-se unir as ferramentas necessárias para fazer a previsão das TTN's. Lembrando que o objetivo é identificar as cadeias mais plausíveis que conectam uma origem (floresta licenciada) a um destino (consumidor final ou intermediário) com base no volume transportado e na eficiência da rota.

4.1 Representação Matemática e Estrutura do Problema

Uma TTN para uma espécie de madeira S é representada como um grafo direcionado ponderado $G_S = \{V, E, W\}$, onde:

- V é o conjunto de vértices (nós), dividido em:
  - V_f: Florestas licenciadas (nós de origem, apenas exportam madeira).
  - V_c: Empresas intermediárias (nós com entrada e saída de madeira).
  - V_e: Consumidores finais (nós de destino, apenas recebem madeira).
- E: Conjunto de arestas direcionadas, representando transações de madeira entre os nós.
- W: Matriz de pesos, onde cada entrada w_{ij} representa o volume total de madeira transportado do nó v_i para v_j.

Uma cadeia de suprimento é um *caminho direcionado sem ciclos* ($v_{c0} \to v_{c1} \to \cdots \to v_{cs}$) que:

1. Começa em um nó de floresta licenciada ($v_{c0} \in V_f$).
2. Termina em um nó de consumidor ou empresa ($v_s \in V_c \cup V_e$).

3. Atribui a cada aresta do caminho da madeira o volume total transportado ao longo do caminho, dado por:

$$\sigma(scv) = \sum_{i=1}^{s} w_{c_{i-1}c_i}.$$

As cadeias mais prováveis são definidas conforme a minimização do número de arestas no caminho (cadeiais mais curtas), e a maximização do somatório dos volumes (cadeiais que transportam mais produto). Devido à dificuldade de rastrear precisamente os caminhos reais da madeira é que recorremos a forma heurística de usar os algoritmos de caminhos mínimos e ordenação de volumes.

4.2 Etapas do Algoritmo para Determinar as k-Mais Prováveis Cadeias

Primeiro, as arestas do grafo G_S têm suas direções invertidas e os volumes w_{ij} são transformados em valores negativos ($-w_{ij}$). Embora o algoritmo de Dijkstra não suporte pesos negativos em geral como mencionado acima, ele funciona nesse caso porque respeita estas condições: os pesos eram todos positivos antes da inversão (medidos em volume) e a configuração das direções das arestas não permite ciclos negativos que diminuem indefinidamente os valores. Isso é muito importante já que, ao longo do processo, estamos encontrando as cadeias com "menos"peso (na medida negativa), usando Dijkstra, que tem apenas a capacidade de achar caminhos com mínimos, e no final convertemos os pesos para positivo novamente, obtendo, na verdade, as rotas com mais volume transportado - que sempre foi o objetivo.

Usa-se Dijkstra para encontrar os caminhos mais curtos (em termos de peso, ou volumes negativos) de v (nó inicial) para todas as florestas licenciadas V_f. O resultado é um conjunto C de caminhos, ordenados pela soma de seus volumes negativos: uma classificação em ordem crescente dos volumes negativos, ou seja, do maior para o menor volume transportado na realidade. Caso o número total de caminhos encontrados ($|C|$) seja menor que k, k é ajustado para $|C|$.

Assim, são computados as cadeias de abastecimento da madeira a partir de qualquer vértice v para todos os outros vértices em G, em particular as florestas licenciadas, que estão na mesma componente conexa. Entretanto, múltiplas cadeias de abastecimento podem originar-se de uma determinada floresta licenciada V_f e convergir em v. Os volumes transportados através dessas cadeias podem exceder aqueles das cadeias de fornecimento que se conectam a outras florestas licenciadas, portanto, elas devem ser consideradas. A partir disso surge a necessidade de recorrer ao Algoritmo de Eppstein para calcular as cadeias adicionais que ligam v a V_f. O processo funciona assim:

1. Para cada caminho $C[i]$ em C, considera-se $v_0 = C[i][0]$ (nó inicial) e $v_f = C[i][-1]$ (floresta licenciada correspondente).

2. Usa-se Eppstein para calcular os k caminhos mais curtos adicionais entre v_0 e v_f.

3. Ordenam-se os caminhos encontrados pelo volume total transportado e comparam-se com o segundo menor caminho encontrado anteriormente.

4. Se os novos caminhos tiverem volumes maiores, eles são adicionados a C.

O processo é repetido até que k cadeias sejam identificadas. Caso o volume de novos caminhos não ultrapasse os volumes já encontrados, o algoritmo termina.

Por fim, os volumes negativos dos caminhos identificados são revertidos para valores positivos, representando as k-cadeias mais prováveis com os maiores volumes transportados.

5 Aplicação Prática e Conclusão

Para fazer uma aplicação computacional do algoritmo foi decidido criar um grafo sinteticamente em códigos Python, que podem ser localizados no repositório do GitHub disponibilizado na pasta "Whole Algorithm", no qual foi escolhido o número de vértices e os seus tipos, e foi aleatorizado as transações de madeira. Esse processo foi feito com o cuidado de respeitar o conceito de fluxo de redes estudado na Teoria dos Grafos. Essa atenção possibilita um cenário lógico e real, de maneira que a quantia de madeira que sai das florestas licencias é sempre respeitada ao longo do grafo, ou seja, a soma do volume de madeira dos consumidores finais e das empresas não pode ser maior do que a soma que saiu das florestas.Esse raciocínio também segue localmente: não pode sair uma quantia maior de madeira de uma empresa do que a quantia que entrou. Importante notar que, no código, a função de Dijkstra não aceita valores negativos, tendo sido alterada por uma função com o mesmo papel que aceita, manutenindo o passo a passo do algoritmo usado.

No grafo, as florestas licenciadas são representadas em verde, as empresas intermediárias em azul e os consumidores finais em vermelho.

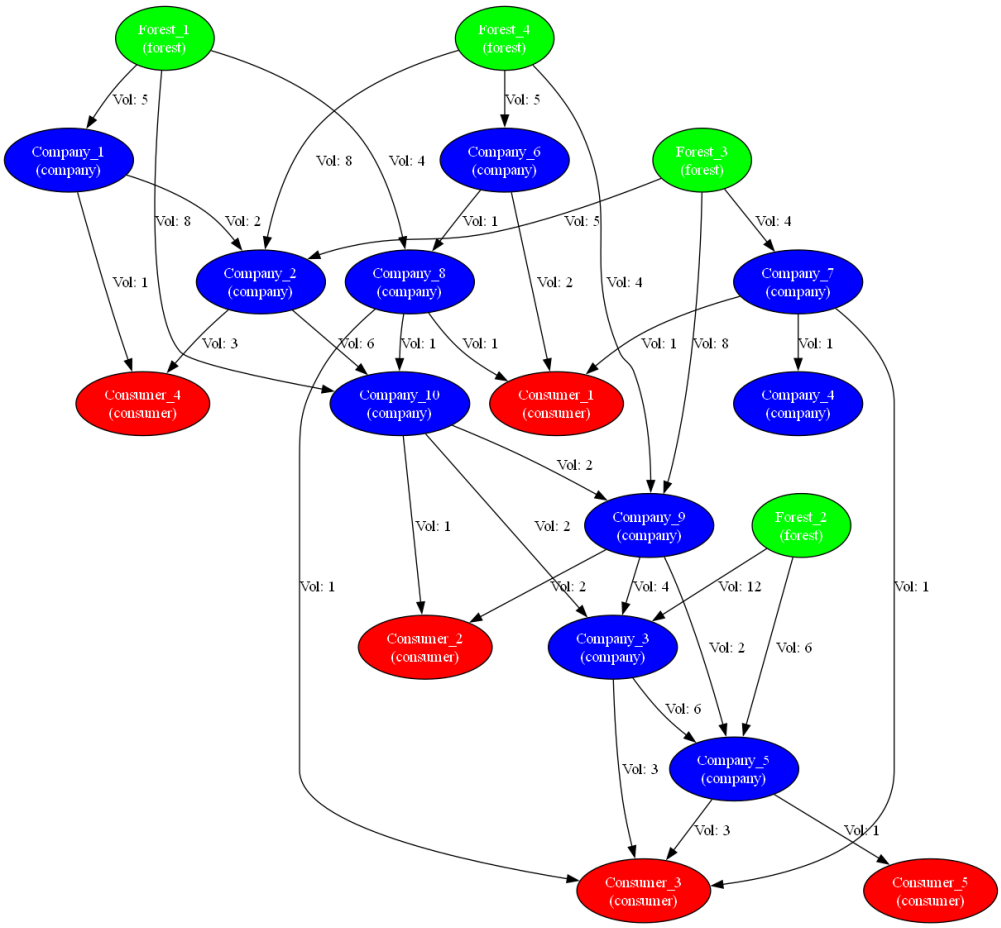


Figura 2: Grafo gerado sintéticamente

A partir disso, foi aplicado o código do algoritmo descrito (que se encontra no módulo "algorithm_supply_chains.py").

```
k_most_likely_supply_chains(G, 'Company_2', 4)
print(" ")
k_most_likely_supply_chains(G, 'Company_1', 3)
print(" ")
k_most_likely_supply_chains(G, 'Company_9', 5)
print(" ")
k_most_likely_supply_chains(G, 'Consumer_5', 5)
print(" ")
k_most_likely_supply_chains(G, 'Consumer_2', 5)
print(" ")
```

Figura 3: Chamada da função que recebe o grafo com os dados, o vértice que se procuram as supply chains com mais volume transportado e por fim quantas se procuram

Saída do Código

- Exibindo as 4 supply chains mais prováveis de Company_2 em DiGraph with 19 nodes and 33 edges: [(['Company_2', 'Forest_4'], 8), (['Company_2', 'Company_1', 'Forest_1'], 7), (['Company_2', 'Forest_3'], 5)].
- Exibindo as 2 supply chains mais prováveis de Company_1 em DiGraph with 19 nodes and 33 edges: [(['Company_1', 'Forest_1'], 5)].
- Exibindo as 5 supply chains mais prováveis de Company_9 em DiGraph with 19 nodes and 33 edges: [(['Company_9', 'Company_10', 'Company_2', 'Forest_4'], 16), ['Company_9', 'Company_10', 'Company_2', 'Company_1', 'Forest_1'], 15), (['Company_9', 'Company_10', 'Company_2', 'Forest_3'], 13)]
- Exibindo as 5 supply chains mais prováveis de Consumer_5 em DiGraph with 19 nodes and 33 edges: [(['Consumer_5', 'Company_5', 'Company_3', 'Company_9', 'Company_10', 'Company_2', 'Forest_4'], 27), (['Consumer_5', 'Company_5', 'Company_3', 'Company_9', 'Company_10', 'Company_2', 'Company_1', 'Forest_1'], 26), (['Consumer_5', 'Company_5', 'Company_3', 'Company_9', 'Company_10', 'Company_2', 'Forest_3'], 24), (['Consumer_5', 'Company_5', 'Company_3', 'Forest_2'], 19)]
- Exibindo as 5 supply chains mais prováveis de Consumer_2 em DiGraph with 19 nodes and 33 edges: [(['Consumer_2', 'Company_9', 'Company_10', 'Company_2', 'Forest_4'], 18), (['Consumer_2', 'Company_9', 'Company_10', 'Company_2', 'Company_1', 'Forest_1'], 17), (['Consumer_2', 'Company_9', 'Company_10', 'Company_2', 'Forest_3'], 15)]

5.1 Resultado Final

O algoritmo retorna:

- Um conjunto C com as k cadeias mais prováveis para cada vértice inicial solicitado.
- Cada cadeia consiste em uma sequência de nós que conecta um consumidor ou empresa às florestas licenciadas.
- O volume total transportado por cada cadeia.

Com esse resultado se obtem as estimativas de cadeias e redes de abastecimento maís prováveis que tiveram cada empresa intermediária ou consumidor final solicitado. Também se aplica à estimativa o raciocínio de que as melhores cadeias com mais volume teriam o menor caminho, em termos de comprimento de transporte. Note que o valor k foi ajustado em cada cálculo ao limite de cadeiais existentes.

5.2 Considerações Finais

Este trabalho demonstrou como os algoritmos de Dijkstra e Eppstein, aliados com os raciocínios empregados, podem ser aplicados de maneira integrada para analisar e prever as cadeias mais prováveis no comércio de madeira, utilizando redes direcionadas e ponderadas para modelar Timber Trade Networks (TTNs). A abordagem proposta não apenas identifica rotas de transporte eficientes, mas também fornece uma estimativa quantitativa do volume de madeira movimentado em cada caminho, facilitando a análise de possíveis irregularidades e a implementação de políticas de fiscalização. Essa metodologia, portanto, pode ser um recurso valioso para órgãos reguladores, pesquisadores e empresas interessadas em otimizar a gestão de cadeias de suprimento e combater o desmatamento ilegal. Para finalizar, citamos as complexidades computacionais dos algoritmos e ferramentas usadas:

- **Dijkstra**: $O(m + n \log n)$, onde m é o número de arestas e n o número de nós.
- **Eppstein**: $O(m + n \log n + k)$, eficiente para grandes grafos.
- **Ordenação**: $O(k \log k)$.

Referências

[1] Maria Soledad Aronna. *Notas de aula da disciplina de Ciência de Dados*. Aulas ministradas na graduação em Ciência de Dados, FGV-Emap. 2024.

[2] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it. This optimal-substructure property is a hallmark of the applicability of both dynamic programing and the greedy method. (Chapter 24, p. 581). MIT Press, 2001.

[3] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. A greedy algorithm always makes the choice that looks best at the moment. That is, it makes locally optimal choice in the hope that this choice will lead to a globally optimal solution. [...] Later chapters will present many algorithms that can be viewed as applications of the greedy method, including [...] Dijkstra's algorithm for shortest paths from a single source. (Chapter 16, p. 370). MIT Press, 2001.

[4] E. W. Dijkstra. ≪A Note on Two Problems in Connexion with Graphs.≫ Em: *Numerische Mathematik* 1 (1959), pp. 269–271. DOI: 10.1007/BF01386390.

[5] David Eppstein. ≪Finding the k Shortest Paths≫. Em: *SIAM Journal on Computing* 28.2 (1998), pp. 652–673. DOI: 10.1137/S0097539795290477.

[6] Luis Nonato et al. ≪Analyzing Timber Trade in Brazil: assessing timber networks and supply chains≫. Em: *Preprint* (jul. de 2024). License CC BY 4.0. DOI: 10.21203/rs.3.rs-4580916/v1.

[7] Victor Russo et al. *Supplementary Material: Analyzing Timber Trade in Brazil: assessing timber networks and supply chains*. Supplementary material accompanying the article "Analyzing Timber Trade in Brazil". 2024. URL: https://doi.org/10.21203/rs.3.rs-4580916/v1.