Universidade Federal de Pernambuco

Graduação em Ciências da Computação

Centro de Informática

2015.1

Agrupamento de Instâncias no Processo de Identificação de Dados Duplicados

Recife, Julho de 2015

Trabalho de Graduação

Augusto Juvenal F. G. Costa

Agrupamento de Instâncias no Processo de Identificação de Dados Duplicados

Trabalho de Graduação

Trabalho de Graduação apresentado à graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco para obtenção do grau de Bacharel em Ciência da Computação.

Orientadora – Prof.ª Ana Carolina Salgado (acs@cin.ufpe.br)

Recife
2015

Augusto Juvenal F. G. Costa

Agrupamento de Instâncias no Processo de Identificação de Dados Duplicados

Trabalho de Graduação

Trabalho de Graduação apresentado à graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco para obtenção do grau de Bacharel em Ciência da Computação.

Recife, ____ de Julho de 2015.

BANCA EXAMINADORA

_____

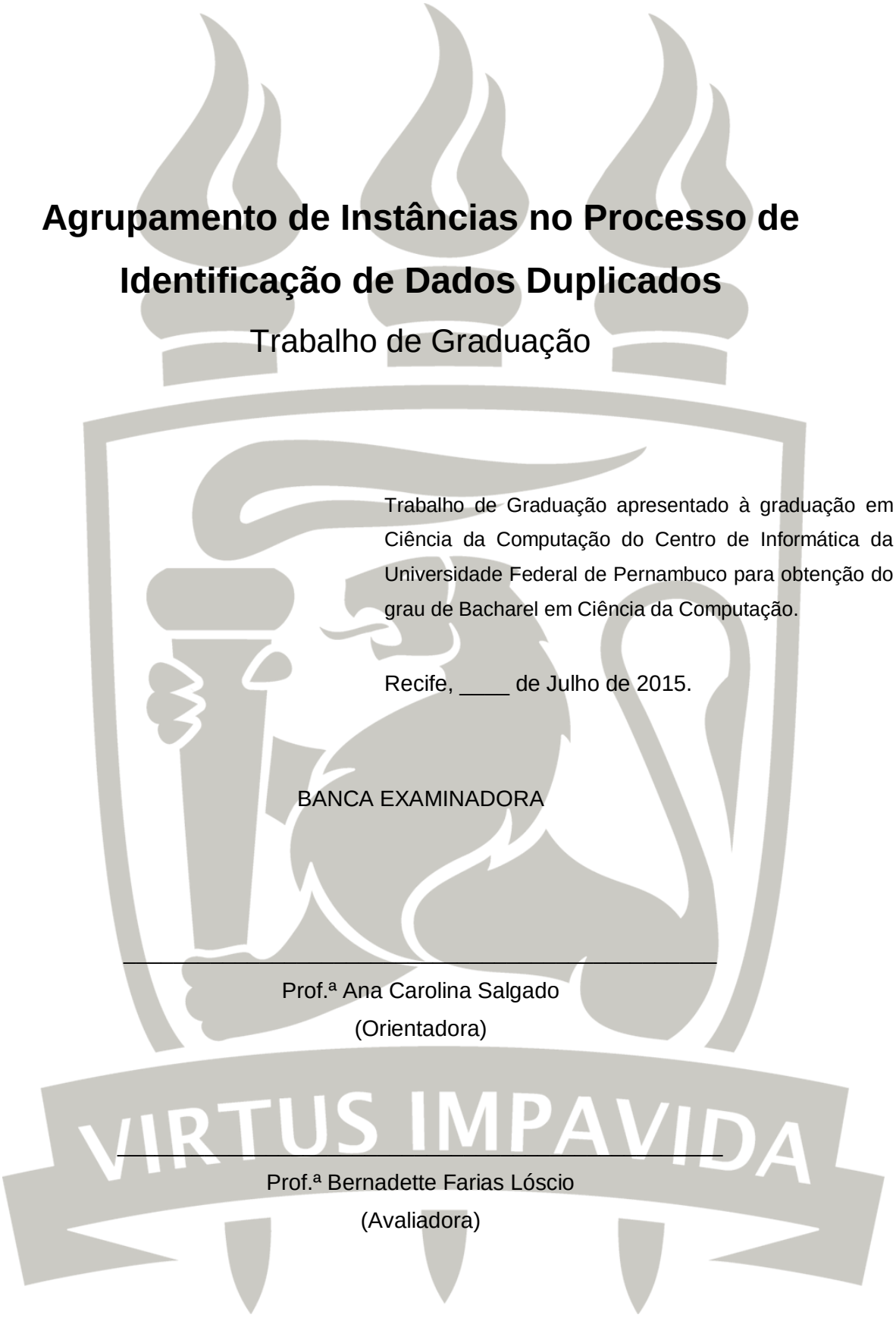
Prof.ª Ana Carolina Salgado

(Orientadora)

_____

Prof.ª Bernadette Farias Lóscio

(Avaliadora)



Agradecimentos

Agradeço a todos que de forma direta ou indiretamente contribuíram para a realização deste trabalho e de toda a minha graduação. Em especial, meus agradecimentos a:

A orientadora, professora Ana Carolina Salgado, pela confiança, disponibilidade e tempo dedicado.

A Pollyanna Alves, minha namorada, que durante os momentos de dificuldade, desde o inicio da graduação, esteve do meu lado, me dando todo e qualquer apoio necessário.

Ao meu pai, Clay Giles, que foi peça fundamental para o meu desenvolvimento pessoal e profissional e minha mãe, Regina Célia, por provê a melhor base que pôde inclusive em minha graduação.

Aos meus familiares, em especial minha avó Esmelinda Giles Costa e meu avô Augusto Costa, por sempre acreditarem em meus esforços, por toda educação, carinho e por me fazerem ter certeza de que a família sempre estará do meu lado.

À Universidade Federal de Pernambuco, especialmente ao Centro de Informática, por oferecer toda a infraestrutura necessária para minha formação.

A todos os colegas de profissão que pude estar junto nesses anos por me ajudarem, mesmo que de forma indireta, em alguma etapa do meu aprendizado. Sem vocês não poderia estar escrevendo este trabalho.

A todos que fizeram parte das turmas em disciplinas CIn/UFPE por proporcionarem momentos de alegria, compartilhar nossos aprendizados e descontração numa fase tão importante da vida.

Por fim, a todos os meus amigos e colegas ainda não citados, que sempre estiveram comigo nos melhores momentos da minha vida. Muito obrigado.

Resumo

A partir da década de 2000 houve uma crescente exponencial de dados, a qual preocupa analistas por uma futura falta de espaço de armazenamento. Segundo a IBM, 90% dos dados virtuais foram produzidos nos últimos dois anos, decorrente da inserção de grandes empresas à internet, como também da criação de várias redes sociais, dados de dispositivos móveis, GPS, entre outros. As grandes organizações agora buscam utilizar este grande volume de dados, voláteis ou não, em seu benefício. O desafio vem na forma de identificar sentimentos semelhantes (reclamações, elogios, críticas,...) dentro deste grande volume, como também a integração desses dados. Um dos desafios da integração de dados é identificar dados que pertencem a uma mesma entidade (pessoa, empresa, semáforo, ônibus,...). Este problema é chamado de Resolução de Entidades. Este trabalho tem como objetivo detalhar o passo a passo das métricas utilizadas para a etapa de pré-processamento necessária para dar início ao algoritmo agrupamento de dados semelhantes e apresentar a implementação do algoritmo de resolução de entidades proposto por [GRUENHEID; DONG and SRIVASTAVA 14].

Abstract

Since the decade of 2000 has an exponential growing of records, which concerns the analysts about a lack of storage space. 90% of virtual records were produced in the last two years, because of companies that were integrated to online storages, as well as the beginning of the social networks, mobiles data, like GPS. The biggest companies now seek to use this large volume of data, volatile or not, for their benefit. The challenge comes in the form of identifying similar sentiments (complaints, praise, criticism ...) inside this large volume, as well as the integration of such data. One of the challenges of record linkage is to identify record that belongs to the same entity (person, business, traffic lights, buses ...). This problem is called Entity Resolution. This paper aims to detail the pre-processing used to generate the entry required to begin the other implementation presented in this work, which is the entity resolution algorithm proposed by [Gruenheid; DONG and Srivastava 14].

Sumário

Lista de Figuras

1 – Introdução

A era Big Data levanta dois desafios para agrupamento de dados semelhantes: o volume de informação é gigante e a velocidade de atualização de dados é muitas vezes elevada, tornando resultados anteriores obsoletos.

No cenário de pesquisas para a Internet das coisas [BHATTACHARYA; GETOOR and LICAMELE 06], é viável prever que cada sistema de banco de dados crie seus próprios identificadores, sua estrutura e semântica que mais convém, dificultando ainda mais o processo de integração de diferentes fontes de dados. Nesse contexto, o gerenciamento de dados diante de um ambiente heterogêneo torna-se desafiador a fim de encontrar informações de interesse do usuário, localizando-os através de diversas fontes de dados, decidindo, dentre as chaves de identificação definidas, se as instâncias pertencem ou não a uma mesma entidade. Este processo é chamado de Resolução de Entidades [BHATTACHARYA; GETOOR and LICAMELE 06].

O agrupamento de dados tem função importante no processo de integração para agrupar dados que se assemelhem e representem a mesma entidade. Através de algoritmos de similaridade, é possível calcular uma margem de semelhança entre cada entidade, sendo capaz de agrupar ou identificar diferentes instâncias de uma mesma entidade em fontes de dados diferentes. Esta etapa é crucial para a identificação e deduplicação de dados [BHATTACHARYA; GETOOR and LICAMELE 06].

Cada *cluster* corresponde a um objeto distinto, onde levando em consideração a era de um grande volume de informações (estruturados ou não), a implementação de algoritmos incrementais é necessária. Um pensamento natural sobre uma abordagem incremental é que para cada informação inserida, a informação é comparada com informações que estão armazenadas em *clusters* já existentes e nova informação é inserida no *cluster* onde as informações mais se assemelhem à mesma. Caso a semelhança entre o dado inserido e os dados existentes não for o suficiente (abaixo de um limiar de semelhança), um novo *cluster* é criado.

1.1 – Objetivo

O objetivo geral deste Trabalho de Graduação é a descrição, implementação e análise do algoritmo incremental de identificação de instâncias duplicadas (agrupamento) proposto por [GRUENHEID; DONG and SRIVASTAVA 14]. Estas instâncias são o resultado de consultas realizadas a múltiplas fontes de dados em sistemas de integração de dados. Será abordado todo o pré-processamento, que é utilizado para construir a entrada padrão para a análise dos dados, com a finalidade de:

- Realizar estudos sobre agrupamento de instâncias duplicadas em um ambiente com alto volume de atualizações;
- Implementar todo o processo, desde o pré-processamento até o algoritmo proposto e descrito por [GRUENHEID; DONG and SRIVASTAVA 14].
- Gerar resultados de agrupamento de instâncias em diferentes de base de dados, que serão processadas, assim como gráficos de desempenho do tempo de resposta e precisão.

1.2 – Estrutura do Documento

Este trabalho está organizado em mais quatro capítulos.

No segundo capítulo será descrita a ideia de identificação de dados duplicados, definiremos o problema, serão descritos os algoritmos de diversos métodos de agrupamento de dados e será indicado o algoritmo que foi decidido implementar e, posteriormente, será explicada sua implementação.

No terceiro capítulo será descrito o algoritmo selecionado dentre as propostas apresentadas no capítulo anterior, em mais detalhes, precedendo os detalhes de implementação.

No quarto capítulo serão apresentados os detalhes da implementação, seguidos dos resultados e análise dos resultados gerados.

Por fim, no quinto e último capítulo, será trazida a conclusão gerada no trabalho seguido de propostas de trabalhos futuros.

2 – Técnicas para o Agrupamento de Dados Duplicados

O problema de identificação de dados duplicados surge do contexto de *data cleaning* [ELMAGARMID; IPEIROTIS and VERYKIOS 07], o qual permite a análise de similaridade entre dois objetos com minimização de ruído em várias bases de dados. Tais bases frequentemente contêm campos duplicados e informações que se referem ao mesmo objeto, mas não necessariamente idênticas, como podemos ver no exemplo a seguir.

Exemplo 2.1

Um hospital possui uma base com milhares de pacientes. Todo ano, novos pacientes são recebidos de outras fontes, como UPAs, SUS ou até mesmo outro hospital que não faça parte de sua rede. No entanto, corriqueiramente, um mesmo tipo de informação é representada com valores diferentes. Um paciente de nome Bruno Gomes Silva, pode ser representado como “Bruno Gomes Silva”, “Bruno G. Silva”, “B. Silva”, “Bruno Silva” entre outras possibilidades possíveis de se imaginar. A ideia é associar informações potencialmente semelhantes de diferentes bases de dados, isolando informações que não pertençam ao mesmo conjunto.

2.1 – Definição do problema

Dado um conjunto de objetos, a relação de similaridade entre eles é essencial ao problema de agrupamento, onde cada *cluster* contém dados que se assemelham a uma única e distinta entidade. Denotamos por **D** o conjunto de informações e por $\mathbf{L_D}$ o agrupamento em **D** como resultado da similaridade de seus dados. Seja **F** o conjunto de operações realizadas no método de agrupamento pelo qual $\mathbf{L_D}$ é gerado a partir de **D**, denotamos $\mathbf{F(D) = L_D}$.

Consideramos três tipos de operação: inserção de uma nova informação; deleção de uma informação já existente; e modificação de um ou mais atributos de uma informação existente. Estas operações são denominadas pelo [GRUENHEID; DONG and SRIVASTAVA 14] como operações de *Update*, que são feitas incrementalmente e denotadas por ΔD. O resultado da aplicação de ΔD em D é definido por $D + \Delta D$. Note que ΔD pode conter deleções e modificações. Logo, o número de informações pode ser menor que a soma do número de dados originais e o número de dados incrementais

|(D + ∆D| ≤ |D| + |∆D|). Nesta monografia, tomamos como base a definição de *Incremental Record Linkage* ou Agrupamento de Informações de Forma Incremental proposta por [GRUENHEID; DONG and SRIVASTAVA 14].

2.3 – Algoritmos Existentes

Dado um conjunto de pontos em um espaço arbitrário, o problema de agrupamento é dividi-los em partes, tal que cada parte possua apenas pontos similares. Os algoritmos de agrupamento de dados são divididos basicamente em dois métodos, hierárquicos e não hierárquicos. Outros métodos são encontrados nas literaturas, porém são apenas variações de um dos citados.

2.3.1 – Métodos Hierárquicos

Dado um conjunto de *n* indivíduos, o ponto de partida para os métodos de classificação hierárquicos em geral será, uma matriz *n x n* cujo elemento genérico (*i, j*) é uma medida de semelhança (ou dissemelhança) entre *i* e *j*. Em cada etapa, fusões divisões nos agrupamentos são realizadas e o número de agrupamentos tende a diminuir.

2.3.2 – Métodos Não-Hierárquicos

Nos métodos não-hierárquicos, é frequente pré-definir o número *k* de classes que se pretende criar. O objetivo é determinar a classificação dos *n* objetos em *k* classes onde cada grupo otimize, em seu interior ou em seu exterior, algum critério pelo qual se deseja agrupar.

Há diferentes abordagens para algoritmos de clusterização não-hierárquica hierárquica como *k-means* e *k-medians*, e hierárquica como o baseado em densidade, baseado em modelos, *fuzzy*, correlacional, e incremental. Algumas destas abordagens requerem um método que defina uma distância, dada pela média da similaridade dos objetos dentro de cada grupo, entre os *clusters* formados. Uma abordagem completa considera a distância mínima entre os elementos mais distantes de dois *clusters* diferentes, onde são considerados apenas os elementos mais próximos. A seguir, descrevemos alguns algoritmos baseados nos métodos citados acima.

2.3.3 – Algoritmo Connected Component

Seja G = (V,E) denotando um grafo não direcionado com um conjunto de nós V {1, 2, …, n} e um conjunto de arestas E que pode ser um conjunto ordenado ou não de pares de vértices V x V [HAN and WAGNER]. Para um grafo não direcionado, há uma aresta conectando o nó i e j se somente se (i, j) ou (j, i) pertence a E. G é conectado se somente se para cada par i, j, existe um caminho entre i e j. Um Componente Conectado de G é o subgrafo máximo de G que é conectado, onde é possivel descrever um caminho de quaisquer dois pontos deste subgrafo.

Porém, com um grafo direcionado, é possível que haja caminhos não alcançáveis entre dois vértices. Deve-se assumir que o grafo é representado por um *array* de arestas, com todas as arestas de um mesmo nó, guardadas sequencialmente no *array*. Também é assumido que todas as arestas (i, i), 1 ≤ i ≤ n, estão nas listas de entrada, assim, todo nó isolado é representado. Esta lista de arestas é inicialmente inserida no *array* E. A Figura 1 a seguir mostra o grafo e sua representação.

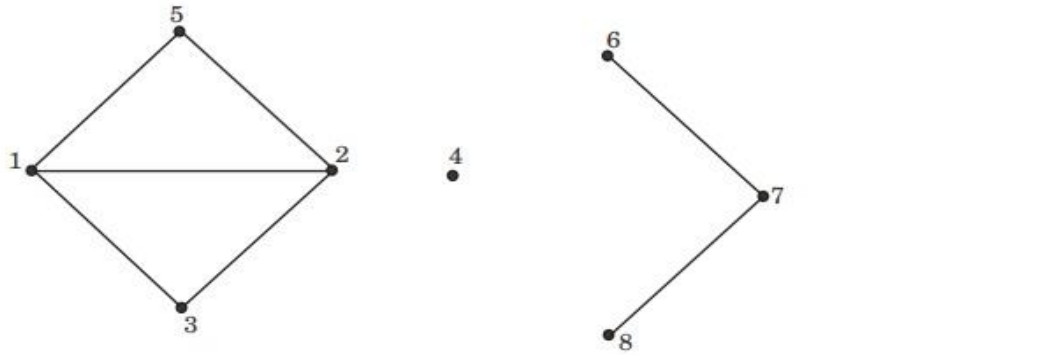


Figura 1 - Exemplo de representação de um grafo

Também é necessária a noção de multigrafos. Um multigrafo M = (S,G) consiste em um grafo G e um conjunto de nós S denominado na literatura de “super nó”. G, como definido anteriormente, é um grafo não-direcionado (V, E). S é um subconjunto de V. A Figura 2 representa o multigrafo. Uma aresta (i, j) é uma aresta interna se ambos, i e j, estão no mesmo super nó. Caso contrário, a aresta é uma aresta externa. Um super nó é isolado se não há arestas externas.

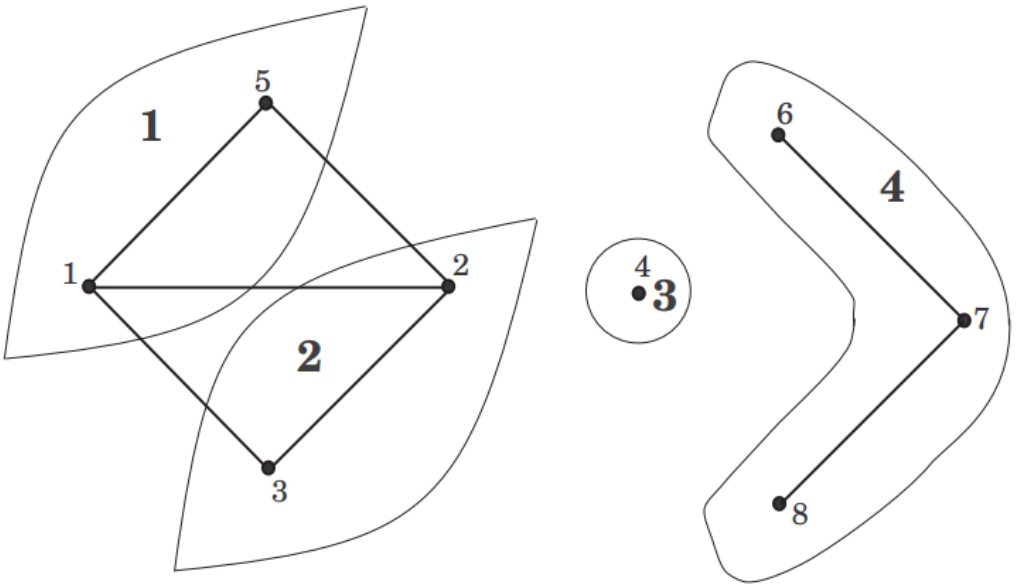


Figura 2 - Representação de Subgrafo

O trabalho de [GRUENHEID; DONG and SRIVASTAVA 14] define *Connected Component* como:

Seja G um grafo de similaridade e ΔG o incremento de G. O fecho transitivo de um nó é definido como o subgrafo conectado em G + ΔG onde o próprio nó está incluso. O fecho transitivo de uma aresta é definido como o *connected component* de ΔG, denotado por T(ΔG), contém a união dos fechos transitivos para cara nó ou aresta inserida, removida ou modificada.

Este algoritmo é utilizado geralmente em problemas de clusterização correlacional. Um método correlacional é utilizado em um cenário onde relacionamentos entre os objetos são conhecidos. Dado um grafo G = (V, E) onde o peso das arestas indica se dois nós são similares (pesos de valor positivo) ou diferentes (pesos possuem valores negativos). Diferente de outros métodos de clusterização, os métodos correlacionais não predefinem uma quantidade de *clusters* a serem criados, pois seu objetivo é maximizar o número de acordos (*agreements*) ou minimizar o número de desacordos (*disagreements*) dentro e entre *clusters*.

2.3.4 – Algoritmos Sequenciais

O algoritmo apresentado nesta seção [BERKHIN 07] é muito utilizado no método de clusterização hierárquica. Método de análise que busca construir uma hierarquia ou uma árvore entre os *clusters*, também conhecida como dendograma. Este método pode ser categorizado em dois tipos:

- **Aglomerativo:** Uma abordagem *bottom up* onde cada observação inicia em seu próprio *cluster*, e pares de *clusters* mesclados sobem na hierarquia.
- **Divisivo:** Uma abordagem *top down* onde todas as observações iniciam em um *cluster* e são aplicados processos de divisão recursivamente, movendo-o para baixo na hierarquia.

O algoritmo em resumo pode ser descrito como segue:

1. Determina e guarda a distância entre cada par de *clusters*. Inicialmente cada ponto é considerado um *cluster* e para cada *cluster*, determina-se seu vizinho mais próximo.
2. Determina o par de *clusters* com a menor distância entre eles e os aglomera.
3. Atualiza a distância entre os pares e os novos vizinhos mais próximos.
4. Se mais de um *cluster* ainda existir, vai para o passo 2.

O método de clusterização hierárquica tem como vantagens:

- Facilidade de lidar com qualquer tipo de semelhança ou dissemelhança entre características dos objetos;
- Aplicabilidade a quaisquer tipos de atributos;

Em contrapartida possui as seguintes desvantagens:

- Indefinição de critérios de parada;
- A maioria dos algoritmos hierárquicos não revisitam *clusters* (intermediários), uma vez construído.

2.3.5 – Algoritmo Fuzzy *c-means*

Este algoritmo é um dos mais utilizados dentre os pertencentes ao método de clusterização *fuzzy* [ZHANG an CHEN 03]. Nos algoritmos do método *fuzzy*, em geral, cada membro de um *cluster* pode pertencer a outros *clusters* (não exclusivo a um único

cluster). Em agrupamento de dados, este tipo de algoritmo é utilizado para calcular um agrupamento probabilístico. Muitas fórmulas são utilizadas para descrever critérios de parada e etapas do algoritmo. Como a ideia não é aprofundar, e sim mostrar algoritmos típicos de alguns métodos de clusterização, não será entrado em detalhes dos cálculos e da ideia do algoritmo.

2.3.6 – Algoritmo *k-means*

Clusterização baseada em *k-means* é muito relacionado ao problema de localização [KANUNGO 00]. O objetivo é minimizar a soma das distâncias das informações mais próximas ao centro e ao mesmo tempo maximizar a distância máxima entre todos os pontos aos pontos mais próximos do centro, ou seja, aproximar os mais próximos e distanciar os mais distantes.

Um algoritmo *k-means* consiste em dado um conjunto de k centros Z, para cada centro $z \in Z$, Seja V(z) a denotação de seus vizinhos mais próximos, como descrito acima, cada elemento $\in$ V(z) é aproximado ao máximo de seu centro z e aqueles que não pertencem ao conjunto V(z) (para cada z) se distancia dos k centros.

2.3.7 – Algoritmo DBSCAN

Clusterização de dados espaciais é uma das técnicas promissoras, onde conjuntos de objetos são agrupados, de acordo com algum critério de agrupamento, em classes ou *clusters*. DBSCAN tenta reconhecer os *clusters* tomando como vantagem o fato que cada *cluster* contém uma densidade consideravelmente maior de objetos do que fora de qualquer *cluster* [BORAH and BHATTACHARYYA 04].

DBSCAN (vêm de *Density-Based Spatial Clustering of Applications with Noise*) é um algoritmo de clusterização baseado em densidade. A ideia básica de uma clusterização envolve as definições apresentadas a seguir:

- A vizinhança dentro de um raio ε de um determinado objeto é chamado o ε-vizinhança do objeto.
- Se a ε-vizinhança do objeto contém pelo menos um número mínimo de objetos, *MinPts*, então o objeto é chamado de objeto *core*

- Dado um conjunto de objetos D, dizemos que um objeto P é *directly-density-reachable* [BORAH and BHATTACHARYYA 04] por um objeto Q se P está em ε-vizinhança de Q, e Q é objeto *core*.
- Um objeto P é *density-reachable* por um objeto Q a relação de ε e *MinPts* em um conjunto de objetos D, se existe uma cadeia de objetos P_1, P_2, …, Pn, onde P_1 = Q e P_n = P tal que P_{i+1} é *directly-density-reachable* por P_i, com relação a ε e *MinPts*, para $1 \leq i \leq n$, $P_i \in D$.
- Um objeto P é *density-connected* a um objeto Q com relação a ε e *MinPts*, em um conjunto de objetos D, se existe um objeto $O \in D$ tal que ambos, P e Q são *density-reachable* por O com relação a ε e *MinPts.*
- *Cluster* baseado em densidade é um conjunto de objetos *density-connected* que são máximos com respeito a *density-reachability*. Todo objeto não contido em qualquer *cluster* é considerado ruído.

2.3.8 – Algoritmo Guloso

Este algoritmo é uma proposta incremental de [GRUENHEID; DONG and SRIVASTAVA 14] que promete resultados idênticos ou muito similares aos resultados quando é aplicado um algoritmo de carga, porém muito mais rápido. Como colocado no Capítulo 1, é um pensamento natural do agrupamento que cada objeto inserido, seja comparado com os *clusters* já existentes e então seja inserido em um *cluster* já existente. O que este método acrescenta é a oportunidade de recuperar erros que possam ser cometidos, colocando uma etapa de atualização para identificar e alterar em cada nova iteração.

2.3.9 – O Problema de Resolução de Entidades

Resolução de Entidades [BHATTACHARYA; GETOOR and LICAMELE 06] consiste na aproximação objetos que são possivelmente semelhantes. Mas quase sempre não é possível se não for levado em conta uma padronização entre os objetos envolvidos. Independente do algoritmo de agrupamento utilizado é necessário realizar essa aproximação.

Esta etapa é responsável pela geração de dados que serão fornecidos para que o agrupamento seja realizado com maior eficiência. Neste processo é realizada a

minimização de ruídos em cada objeto da base, a fim de realizar uma comparação, ou um cálculo de similaridade (se necessário para o algoritmo escolhido) entre dois objetos. Também nesta etapa, são executadas heurísticas (se for preciso no processo) para definir constantes; onde pode ser gerado grafo, ou uma lista de objetos com menos ruídos, dependendo da necessidade do método de agrupamento.

Em resumo, nesta etapa é gerado o que é necessário para o início do algoritmo de agrupamento escolhido. No Capítulo 4 será explicado, na seção de Pré-Processamento, as etapas utilizadas para que uma entrada com menos ruídos seja utilizada na etapa de agrupamento, tornando os resultados mais confiáveis.

2.4 – Considerações

Decidimos pela escolha do algoritmo relatado pelo autor [GRUENHEID; DONG and SRIVASTAVA 14] devido a resultados quando comparados com dois outros algoritmos. Além disso, o Algoritmo *Greedy* descrito pelo autor, mas que será chamado de Guloso neste trabalho, destaca-se em três aspectos:

1. O algoritmo assume apenas tempo polinomial;
2. É um algoritmo incremental;
3. Além de incremental, sua maior característica é ser ajustável, corrigindo possíveis erros de iterações anteriores.

Os aspectos 1 e 2 são de muito valor quando se trata de um grande volume de dados, ou seja, o algoritmo deve ter um bom tempo de resposta e reaproveitável em tempo hábil. Além disso, deve ser confiável, o que nos leva ao aspecto 3, onde seus ajustes aumentam a robustez e a correção dos *clusters* já formados. Para que o algoritmo não assuma um custo operacional muito alto, explicaremos os procedimentos seguidos por [GRUENHEID; DONG and SRIVASTAVA 14] para manter a complexidade em tempo polinomial no capítulo seguinte.

3 – Algoritmo de Clusterização Selecionado

Segundo [ELMAGARMID; IPEIROTIS and VERYKIOS 07], o processo de identificação de dados duplicados ou similares é precedido da etapa de *preparação de dados*, onde cada entrada dentre todas as possíveis, em bancos de dados diferentes, são capturadas e armazenadas de maneira uniforme. Também deste autor utilizamos a etapa de preparação *transformação de dados*.

3.1 – Transformação de dados e Seleção de Atributos

O passo de transformação, descrito por [ELMAGARMID; IPEIROTIS and VERYKIOS 07], se refere a transformações de campos de elementos para um único tipo, isto é, transformar todos os tipos dos campos em um tipo *String* por exemplo, para obter uma padronização também na manipulação dos dados. Em alguns casos se aplica renomear campos para padronizar o nome de campos que são definidos de forma diferente pelos bancos de dados. Retomando o exemplo do hospital (Exemplo 2.1), poderíamos ter instâncias onde os nomes dos atributos relacionados ao nome do médico poderiam ser “nome_medico”, “nome_doutor”, “nome”, entre outros possíveis.

Ainda preparando os dados, foi visto como necessária a seleção dos campos a serem comparados, pois [CHEN et al 12] verificou que campos que se repetem em várias entradas, ou campos que contêm muitas vezes valores nulos, não são indicados para serem utilizados como entrada. Foi decidido, neste trabalho, selecionar manualmente os atributos a serem utilizados para o algoritmo de agrupamento. Após a transformação dos dados e a seleção de atributos, é possível gerar uma palavra não-única que será utilizada para o cálculo da similaridade entre os objetos que as possuem.

3.2 – *Sorted Neighborhood*

Na aplicação do cálculo de similaridade, utilizamos um método para otimizar e diminuir o tempo de processamento deste passo, conhecido como blocagem [GRUENHEID; DONG and SRIVASTAVA 14]. O método de *Sorted Neighborhood* (SNM) primeiro ordena todas as entradas usando um atributo chave pré-selecionado. Então, uma janela de tamanho fixo, um limitante que indique com quantos atributos de

uma comparação será executada, desliza do primeiro até o ultimo elemento da lista [YAN *et al.* 07].

A cada movimento da janela, o primeiro elemento que está localizado nela é comparado com o resto dos componentes que se encontram na janela. O processo se repete até alcançar o fim da lista. Assim, a janela deslizante funciona como um bloqueio com a premissa de que os elementos internos são lexicograficamente semelhantes para que fiquem localizados dentro da mesma janela. A Figura 3 simula um movimento do SNM sobre *n* elementos com uma janela de tamanho *w*.

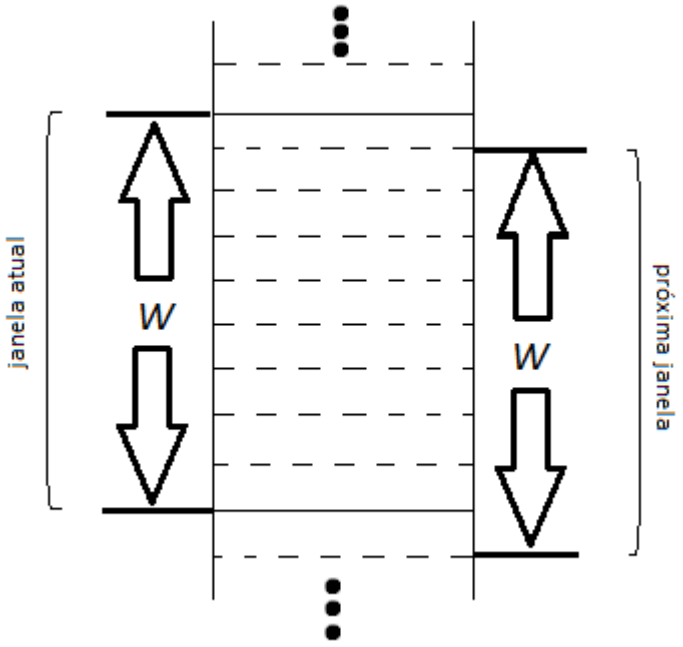


Figura 3 - Simulação da movimentação da janela no algoritmo SNM

3.3 – Algoritmo Implementado

Seguindo as considerações do fim do Capítulo 2, foi escolhido implementar o algoritmo Guloso por ser um algoritmo incremental, dado que este trabalho propõe trabalhar com grande volume de dados em tempo real, além de atualizar e recuperar possíveis análises errôneas feitas em iterações anteriores. Nesta seção procuramos descrever o algoritmo proposto por [GRUENHEID; DONG and SRIVASTAVA 14] que servirá como fundamentação para a implementação realizada.

Dado que possuímos um grafo não direcionado de similaridade entre os objetos, aplicamos o algoritmo levando em conta algumas notações. É definida uma fila de

trabalho Q^C que guarda grupos de objetos similares (*clusters*). Cada vez que um *cluster* C da fila Q^C é examinado, consideramos três possíveis operações que podem ser aplicadas no *cluster* (1) *merge* com outro *cluster*, (2) *split* em dois ou mais *clusters* e (3) mover alguns dos nós de C para outros *clusters* e vice-versa. Elucidamos as operações a serem consideradas a seguir:

Merge: Dado um *cluster* C da fila Q^C, consideramos que fundi-lo com outro *cluster* pode gerar um *cluster* melhor (um subgrafo conectado que possua média de similaridade acima da média atual). Para finalizar a exploração em tempo polinomial, consideramos fundir apenas pares de *clusters*. O algoritmo MERGE é como segue:

1. Para cada vizinho C' de C, avalie se fundir C com C' gera um *cluster* melhor.
2. Ao encontrar o melhor agrupamento, (1) fundir C com C', (2) adicionar C U C' na fila Q^C, e (3) remover C' de Q^C se C' pertence a Q^C.

Split: Dado um *cluster* C que pertence a Q^C, consideramos que ao dividi-lo em um ou mais *clusters*, podemos gerar um *cluster* melhor. Para restringir o algoritmo ao tempo polinomial, consideramos apenas dividir *clusters* em dois e examinarmos um nó por vez como segue.

1. Para cada nó v de um *cluster* C, avalie se tirando v, um *cluster* melhor é gerado.

2. Ao encontrar um nó v que satisfaça o passo 1, crie um *cluster* novo C' = {v} e vá para os passos 3 e 4.

3. Para cada nó remanescente v' que pertença a C, avalie se movendo v' para c', um *cluster* melhor é obtido. Se for, mova v' para C' e repita o passo 3.

4. Adicione C e C1 a Q^C se eles são conectados a outro *cluster*.

Move**:** Dado um *cluster* C pertencente a Q^C, consideramos que se movermos alguns dos nós para outro *cluster* ou mover nós de outros *clusters* para C, poderíamos gerar um *cluster* melhor. Novamente, consideramos nós se movendo entre dois *clusters* para

que o algoritmo termine em tempo polinomial. O algoritmo MOVE é descrito como segue:

1. Para cada vizinho C' do *cluster* C, faça os passos 2-3.
2. Para cada nó v pertencente a C que é conectado com C' e para cada v pertencente a C' conectado a C, avalie se movendo v para outro *cluster* gera um *cluster* melhor. Ao encontrar um nó v que satisfaça, mova-o para o outro *cluster*.
3. Repita 2 até que não haja mais nós para serem movidos. Então (1) adicione os dois novos *clusters* a fila Q^c, e (2) tire da fila C' se C' estiver na fila.

```
/*Grafo(nó, aresta): grafo de similaridade original;
  ΔG: incremento; (possiveis nós adicionados ao longo
  da iteração por novas inserções);
  LG: clustering do grafo original, onde clusters C, inicialmente em Qc,
  são colocados após não haver mais operações a serem aplicadas
*/

AlgoritmoGuloso(){
QC = vazio
G' = T(ΔG) //grafo pós-incremento
QC= G'

    while QC != null{
        C = QC. remove;
        changed = false;
        changed = Merge(C,(G+ΔG), LG, QC);
        if(!changed){
            changed = Split(C,(G+ΔG), LG, QC);
        }if(!changed){
            changed = Move(C,(G+ΔG), LG, QC);
        }
    }
    return LG;
}
```

Figura 4 - Overview do Algoritmo Guloso

3.4 – Considerações

Este capítulo apresentou o algoritmo de agrupamento de dados escolhido para a implementação, citado inicialmente na Seção 2.2.6, a fim de descrever melhor todo o estudo no qual foi baseada a implementação do mesmo. Iniciamos o capítulo com a preparação dos dados a serem agrupados até o algoritmo em si. No Capítulo 4 descreveremos os passos da implementação.

4 – Implementação

Neste capítulo será descrito todo o passo a passo que foi utilizado para o agrupamento de dados semelhantes. Todo o processo de preparação dos dados está incluso, pois um bom pré-processamento destes dados, significa que o algoritmo de agrupamento retorne resultados mais críveis.

A implementação do algoritmo foi feita em Java, utilizando consultas a banco de dados MySQL com duas instâncias de cerca de 1000 entradas em cada. As instâncias correspondem a bases de dados reais retiradas da internet que corresponde a artigos, monografias e livros publicados. O grafo gerado fica armazenado em arquivo *.csv* e os dados, após serem agrupados, são mapeados em uma planilha *.xls* a nível de visualização dos resultados apresentados. A etapa foi iniciada com a preparação dos dados, passo que é necessário para toda a construção do grafo inicial e das interações seguintes.

[CHEATHAM and HITZLER 12] cita técnicas que são utilizadas para pré-processamento, das quais algumas se aplicam ao nosso tipo de problema como:

- *Tokenization* - Será descrito mais a frente neste capítulo como foi aplicada a geração de *tokens*, mas basicamente, como descreve o autor, pedaços de *strings* da informação são selecionados compondo uma nova *string* (*token*).
- Normalização - Eliminação de estilo fonte, letras maíusculas, pontuações, ordenações e caracteres que não estejam no alfabeto.
- Remoção de *stop words* - neste caso excluímos as abreviações e/ou artigos definidos ou indefinidos (*an, a, the,* os, as, o, a, um, uma, uns, umas, etc)

A partir destas técnicas geramos os *tokens* e executamos os procedimentos descritos nas próximas seções com a finalidade de não interferir negativamente no cálculo de similaridade entre os objetos. As etapas de implementação realizada neste trabalho podem ser descritas como:

- Limpeza de dados: Os objetos a serem comparados passam por um processo de redução de ruídos a fim de otimizar o processo de cálculo de similaridade.
- Geração de *tokens*: Após minimizar os ruídos, são gerados novas palavras a partir de partes dos atributos de cada objeto. Estes *tokens* são utilizados para comparar o quão similar dois objetos são.

- Gerando Similaridade: Dado uma lista de *tokens*, é gerada uma lista de valores de similaridade *s*, onde cada *s* é referente a dois tokens.
- Grafo de Similaridade: Por fim, ao possuir uma lista de valores de similaridade, podemos construir um grafo no contexto de quão parecidos dois objetos são. Grafos são compostos por um conjunto de vértices e um conjunto de arestas. Neste caso, os vértices aos objetos localizados nos bancos de dados, podendo ser referenciado pelo seu *token*, e as arestas correspondem aos valores de similaridade.
- Algoritmo Guloso: Ao obter um grafo, já é possível executar o algoritmo proposto por [GRUENHEID; DONG and SRIVASTAVA 14]. Podemos considerar apenas um vértice como um grafo, pois é esperado tanto que o algoritmo possa iniciar com fontes de dados vazias, quanto com fontes bastante volumosas.

Nas próximas seções serão detalhadas as etapas descritas acima da forma a qual foi realizada para gerar os resultados que serão mostrados no Capítulo 5.

4.1 – Gerando os *tokens*

Dada a lista de objetos que se deseja agrupar por semelhança, são gerados dois tipos de *tokens*, um para identificação única do objeto e um para cálculo de semelhança entre os objetos. Para o *token* de identificação, apenas é feita uma varredura entre os atributos, agrupando-os de forma que:

- Campos numéricos, são adicionados sem cortes;
- Campo de ano ou data, caso seja apenas o ano, deixamos apenas os campos da dezena e unidade, caso seja uma data completa (YYYY-MM-DD) preservamos o mês e o dia. O caractere que separa os campos da data são eliminados e como definido acima, o campo do ano não é levado em conta. Por fim, o campo da data é agrupado numa única *string* (por exemplo YYYY-MM-DD torna-se YYMMDD);
- Para campos de textos (título de livro, nome, sobrenome, endereço) é feita a varredura no texto, selecionando até 3 palavras iniciais de cada subdivisão do texto que são agrupadas em uma única *string* (Ana Carolina torna-se AnaCar).

Ao término, todas as *strings* geradas dos atributos do objeto são agrupadas em uma só (nesse caso YYMMDDAnaCar gerada de YYMMDD e AnaCar) e é feita uma varredura final eliminando qualquer pontuação (parêntese, vírgula, ponto e vírgula, etc.) que possa ser colocada em uma possível abreviatura de campos textos, ou até mesmo de forma errônea em algum campo. Para o *token* de comparação, foram considerados os atributos mais relevantes para identificação dos objetos. Após isso, o mesmo procedimento descrito acima é realizado para os campos com números, datas e textos.

Pelas conclusões chegadas por [GONDIM 06] de que o algoritmo Smith Waterman obteve resultados próximos a média para comparações de *strings* formadas por um único *token*, utilizamos tal métrica para a similaridade no pré-processamento com a finalidade de se construir as arestas que compõem o grafo. Ainda para a aresta, verificamos na literatura [NAVARRO 01], [BHATTACHARYA; GETOOR and LICAMELE 06] e [CHEATHAM and HITZLER 12] que os grafos compostos apenas por arestas onde a criação de uma nova aresta é levada em conta se o cálculo da similaridade assume apenas um valor $\delta \geq 0.75$.

Tal valor, segundo os autores, está dentro de um intervalo (que vai de 0,7 a 0,8) de ótimos resultados de agrupamentos por similaridade e menor custo de processamento para alcançar o resultado final. Ou seja, utilizando um valor δ fora deste intervalo, resultados semelhantes de agrupamento de dados por similaridade podem até ser alcançados, mas custará mais tempo.

4.2 – Pré-Processamento

Antes da primeira iteração do algoritmo é preciso obter a estrutura que será utilizada como entrada para o Algoritmo Guloso, definindo o grafo que o representa. Os nós do grafo são os elementos que queremos agrupar e as arestas são os pesos dados da comparação. Para calcularmos a similaridade entre dois vértices, utilizamos o Algoritmo Smith Watterman [SMITE and WATERMAN 81] entre os *tokens* de comparações gerados para cada tupla. Dado que a comparação retorne um valor acima do *threshold,* definido de acordo com estudos realizados entre a similaridade de *strings* [CHEATHAM and HITZLER 12], a aresta é formada. Caso contrário, ambos

serão comparados com outros *tokens* de comparação para verificar a possibilidade de novas arestas. O autor mostra que definir um *threshhold* entre sete e oito décimos é uma forma simples e não-leiga de se impor um limiar de similaridade relativamente eficiente, dado que a proposta de [GRUENHEID; DONG and SRIVASTAVA 14] também não impõe um método específico.

4.3 – Blocagem

Assim como os outros procedimentos iniciais, [GRUENHEID; DONG and SRIVASTAVA 14], não impõem um método de blocagem para ser utilizado na formação do grafo de similaridade utilizado em seu algoritmo. Foi então escolhido o método de blocagem *Sorted Neighborhood*, descrito na Seção 3.2, devido a estudos realizados em [BHATTACHARYA; GETOOR and LICAMELE 06] que indicam a sua completude, rapidez e eficiência na taxa de redução definida pelo autor como *reduction ratio* (RR) onde “RR é a redução relativa no número de pares de informações a serem comparados”.

Ainda com relação à [BHATTACHARYA; GETOOR and LICAMELE 06], é definido o número da janela W a ser utilizada para melhores resultados (W = 10), evitando comparações exaustivas no passo descrito na seção anterior, a fim de evitar comparação de palavras que estão a uma distância muito grande quando ordenadas.

4.3.1 – Algoritmo de Smith Waterman

O algoritmo leva o nome dos seus propositores, Temple F. Smith e Michael S. Waterman, e é um algoritmo de programação dinâmica, evitando recálculos de subproblemas contidos no problema principal. Tem como propriedade que é garantido encontrar o alinhamento ótimo entre duas palavras dadas como entrada. Dado duas palavras *m* e *n* como entrada do algoritmo, o processo se resume a montar uma matriz de tamanho |*m*|x|*n*|, definir valores para o *match* e punição (valores fixos predefinidos) para aplicar cada resultado da comparação na posição (i, j) da matriz, o qual corresponde à comparação do caractere na posição i da palavra *m* com o caractere na posição i da palavra *n*.

“São atribuídos escores diferentes para cada operação possível: *match* (casamento, igualdade dos caracteres); *mismatches* (substituições); inserções,

remoções. Todas as possibilidades são avaliadas para se chegar ao maior escore" [GONDIM 06]. O algoritmo Smith Waterman é definido como:

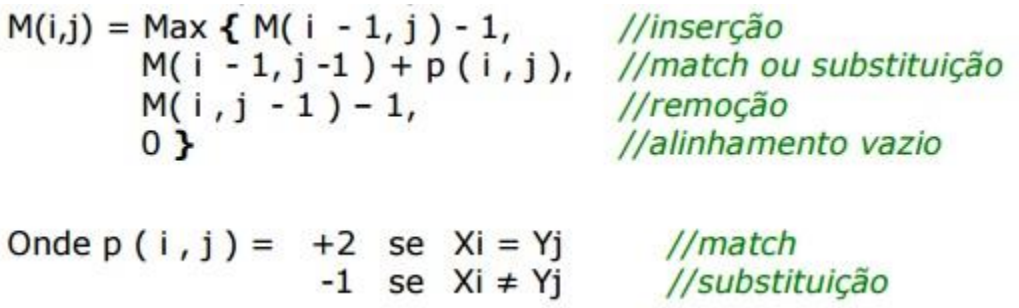


Figura 5 - Algoritmo Smith Waterman

M é a matriz onde M(0,0) = 0 (as palavras são alinhadas a partir da posição (1,1) da matriz). A função p(i,j) é utilizada para determinar se houve igualdade entre os termos comparados (*match*) ou não (substituição). X e Y são as *strings* que estão sendo comparadas, *i* e *j* são respectivamente as posições dos caracteres das duas palavras.

4.4 – Gerando Similaridade

A blocagem descrita na seção anterior é utilizada para gerar a similaridade par a par dentro da mesma janela, verificando se o valor δ gerado está contido dentro do valor aceitável para gerar uma aresta, como descrito na Seção 4.1.

4.5 – Grafo de Similaridade

Uma vez que é sabido a similaridade entre cada par de objetos, é possível construir um grafo G(*V, E*) de similaridade com o universo representado por todos os dados contidos nas instâncias dos bancos de dados. Cada nó $v_r \in V$ representa um objeto e cada aresta $(v_r, v_r') \in E$ representa a similaridade δ calculada entre os objetos e $\delta \geq 0.75$ (*threshold*), simplificando e otimizando o grafo para a proposta de agrupamento por similaridade. Em paralelo com a etapa da geração de similaridade, descrita na Seção 4.4, o grafo é criado ou incrementado, dependendo de qual etapa da iteração o processo se encontra. A seguir, serão descritos os passos do algoritmo durante a primeira iteração, as iterações incrementais e a continuidade do processo por meio de recuperação do grafo salvo em arquivo.

4.5.1 – Grafo Inicial

Essa etapa é realizada caso o algoritmo não possua nenhum *backup* de execuções anteriores, um possível erro geral e queda do serviço ou se é a primeira vez que o processo será executado. É formado o primeiro grafo de todo o processo, onde é passível de receber grande volume de dados salvos em diferentes bases de dados. É bastante razoável de se imaginar em um ambiente real, onde uma empresa que possua um grande volume de dados que lhe interessa identificá-los, que ao implementar e executar o algoritmo, a primeira execução demandará um nível relevante de tempo. Formado o grafo inicial contido em todas as instâncias de dados em um tempo *t*, que ainda não tem nenhum agrupamento, é feito o agrupamento do grafo formado no tempo *t,* e todas as inserções, deleções e atualizações feitas nos objetos feitas no tempo $t' \geq t + 1$ são tratadas no passo a seguir.

4.5.2 – Grafo Incremental

Essa etapa do grafo é gerada a cada nova iteração do algoritmo, onde é possível que novos objetos sejam inseridos em qualquer instância, alguns objetos sejam removidos ou atualizados. Todas essas modificações são tratadas como atualizações do grafo, pois independente de qual dos três cenários aconteça, a forma do grafo será alterada. Tais atualizações podem ser divididas e detalhadas como:

- Inserção: Inserir um objeto é équivalente a adicionar um novo nó e novas arestas. Voltamos à etapa da blocagem (Seção 4.2) onde será criado um *token* de comparação relacionado ao novo objeto, inserido na lista de blocagem, e é calculada a similaridade entre os $w - 1$ *tokens*, anteriores e posteriores ao novo *token*, e o *token* recém-inserido, onde w é o tamanho da janela fixa definida para o algoritmo de Smith Waterman (Seção 4.3.1).
- Remoção: Remover um objeto é equivalente a remover um nó e as arestas que o ligam aos outros nós. Basta apenas localizar o nó a ser removido e removê-lo do grafo, excluindo também todas as arestas que o contenha.
- Modificação: Modificar um objeto é equivalente a fazer uma remoção, seguida de uma inserção.

4.5.3 – Recuperação do Grafo

A recuperação do grafo foi implementada com o objetivo de obter resultados previamente calculados, antes de uma queda do sistema, sem que um retrabalho seja feito pelo sistema. Dado o contexto que a monografia está inserida, e o algoritmo escolhido com o intuito de uma proposta incremental tomando vantagem de agrupamentos feitos anteriormente, corrigindo-os ou modificando-os. Foi decido implementar um *backup* do grafo afim de diminuir o tempo de recuperação de algum problema de indisponibilidade.

A proposta é relevante para um ambiente onde é esperado trabalhar com dados que sejam atualizados ou inseridos em um curto espaço de tempo que necessite se recuperar de uma indisponibilidade rapidamente, porém não do zero. Mesmo que com essas atualizações, parte deste arquivo recuperado esteja obsoleto, espera-se que a grande maioria das variáveis estejam mantidas. O arquivo gerado para este tipo de recuperação é um arquivo *.csv* onde são guardados os vértices referentes a cada objeto de cada instância, seguidos das arestas listadas.

Os vértices guardam o seu identificador único, criado pelo algoritmo implementado, os objetos e seus respectivos atributos, assim como a instância de dado, seguido do *token* gerado pelo algoritmo descrito na Seção 4.5. As arestas guardam seu identificador único, gerado também pelo algoritmo da implementação, o identificador dos vértices, seguido do resultado do cálculo da similaridade entre os *tokens* dos dois vértices. Podemos ver um exemplo do dos resultados nas Figuras 6, 7 e 8 a seguir.

Vértice ID	Tupla	Token ID
0	M Ahlskog J Paloheimo H Stubb P Dyreklev M Fahlman O 76 Inganas and MR 1994 893	mahlsjpalohstubpdyremfahlo76inmr1994893ahlskog1994a
1	M Ahlskog J Paloheimo H Stubb P Dyreklev M Fahlman O Inganas and MR Andersson 76 1994 883 ahlskog199	mahlsjpalohstubpdyremfahloingamrande761994893ahlskog1994a
2	M Ahlskog J Paloheimo H Stubb P Dyreklev M Fahlman O </au Inganas and MR Andersson 76 1994 883 ahlsko	mahlsjpalohstubpdyremfahloauingamrande761994893ahlskog1994a
3	C Ray Asfahl Robots and Manufacturing Automation New York 1992 second edition asfahl1992a	crayasfaromaau1992asfahl1992a
4	Bettina Buth KarlHeinz Buth Martin Franzle Burghard von Karger Yassine Lakhneche buth1992a	bettbuthkarlbuthmartfranburgvonkargyasslakhhanslangmarkmullprcocodeim1992buth199
5	B Buth KH Buth M Franzle B v Karger Y Lakhneche H Langmaack and M MullerOlm buth1992a	bbuthkhbuthmfranbvkargylakhhlangmmullprcocodeim1992vbuth1992a5

Token de Comparação
mahljpalhstupdyrmfahoingmrahlskog1994a
mahljpalhstupdyrmfahoingmrandahlskog1994a
mahljpalhstupdyrmfahoauingmrandahlskog1994a
crayasfrobmanautasfahl1992a
betbutkarbutmarfraburvonkaryaslakhanlanmarmulprocorcomdevimpbuth1992a
bbutkhbutmfrabvkarylakhlanmmulprocorcomdevimpbuth1992a5

Figura 6 - Exemplo do Armazenamento dos Vértices para Recuperação

Aresta ID	Vértice1	Vértice 2	Similaridade
0	4	5	0.8035
1	0	1	0.95
2	0	2	0.904
3	1	2	0.952

Figura 7 - Exemplo de Arestas Armazenadas no Arquivo de Recuperação

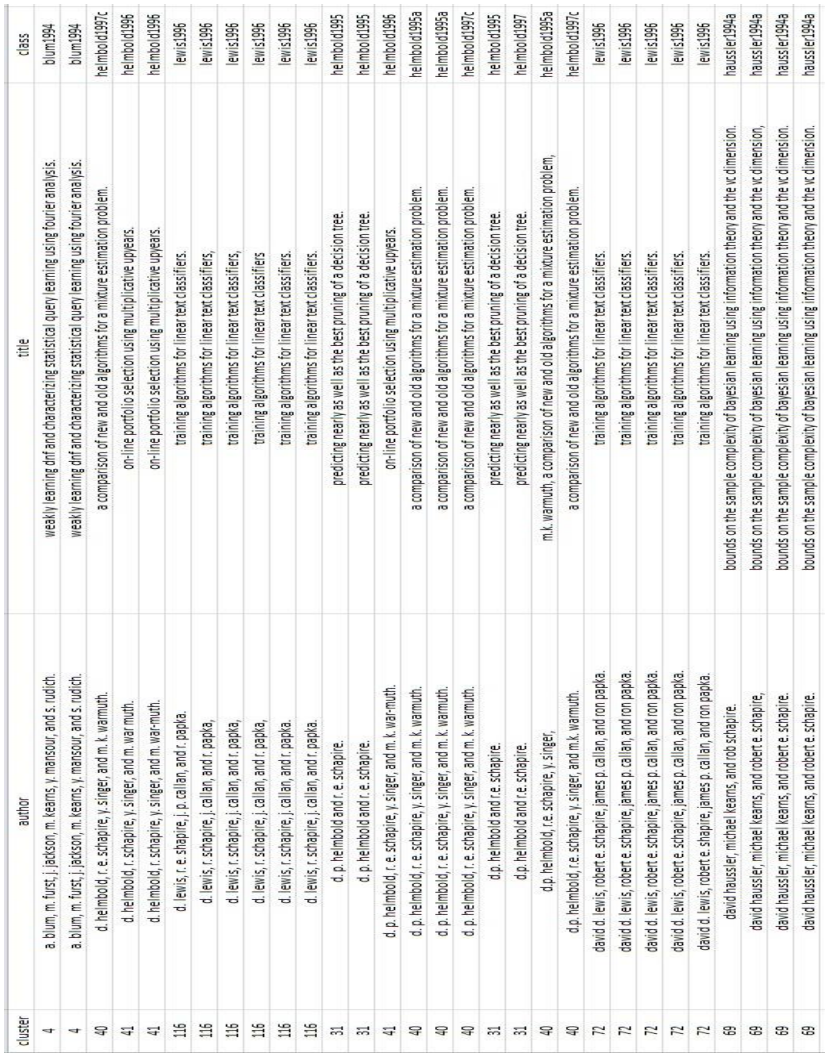
Cluster ID	<Vertice ID> (Lista de vertices)
0	[0,1,2]
1	[4,5]
2	[3]

Figura 8 - Exemplo de Clusters Armazenados no Arquivo de Recuperação

4.6 – Considerações Finais

Neste capítulo foram descritos todos os detalhes de implementação do algoritmo escolhido. Detalhes do código encontram-se no Apêndice A. Todo o pré-processamento dos dados também é discutido para que outras propostas possam se basear nesse trabalho. O capítulo seguinte trará uma discussão geral sobre o os assuntos que foram abordados durante o trabalho além de contribuições e possíveis trabalhos futuros.

5 – Experimento



cluster	author	title	class
4	a. blum, m. furst, j. jackson, m. kearns, y. mansour, and s. rudich.	weakly learning dnf and characterizing statistical query learning using fourier analysis.	blum1994
4	a. blum, m. furst, j. jackson, m. kearns, y. mansour, and s. rudich.	weakly learning dnf and characterizing statistical query learning using fourier analysis.	blum1994
40	d. helmbold, r. e. schapire, y. singer, and m. k. warmuth.	a comparison of new and old algorithms for a mixture estimation problem.	helmbold1997c
41	d. helmbold, r. schapire, y. singer, and m. war muth.	on-line portfolio selection using multiplicative upyears.	helmbold1996
41	d. helmbold, r. schapire, y. singer, and m. war-muth.	on-line portfolio selection using multiplicative upyears.	helmbold1996
116	d. lewis, r. e. shapire, j. p. callan, and r. papka.	training algorithms for linear text classifiers.	lewis1996
116	d. lewis, r. schapire, j. callan, and r. papka,	training algorithms for linear text classifiers,	lewis1996
116	d. lewis, r. schapire, j. callan, and r. papka,	training algorithms for linear text classifiers,	lewis1996
116	d. lewis, r. schapire, j. callan, and r. papka,	training algorithms for linear text classifiers	lewis1996
116	d. lewis, r. schapire, j. callan, and r. papka.	training algorithms for linear text classifiers.	lewis1996
116	d. lewis, r. schapire, j. callan, and r. papka.	training algorithms for linear text classifiers.	lewis1996
31	d. p. helmbold and r. e. schapire.	predicting nearly as well as the best pruning of a decision tree.	helmbold1995
31	d. p. helmbold and r. e. schapire.	predicting nearly as well as the best pruning of a decision tree.	helmbold1995
41	d. p. helmbold, r. e. schapire, y. singer, and m. k. war-muth.	on-line portfolio selection using multiplicative upyears.	helmbold1996
40	d. p. helmbold, r. e. schapire, y. singer, and m. k. warmuth.	a comparison of new and old algorithms for a mixture estimation problem.	helmbold1995a
40	d. p. helmbold, r. e. schapire, y. singer, and m. k. warmuth.	a comparison of new and old algorithms for a mixture estimation problem.	helmbold1995a
40	d. p. helmbold, r. e. schapire, y. singer, and m. k. warmuth.	a comparison of new and old algorithms for a mixture estimation problem.	helmbold1997c
31	d.p. helmbold and r.e. schapire.	predicting nearly as well as the best pruning of a decision tree.	helmbold1995
31	d.p. helmbold and r.e. schapire.	predicting nearly as well as the best pruning of a decision tree.	helmbold1997
40	d.p. helmbold, r.e. schapire, y. singer,	m.k. warmuth, a comparison of new and old algorithms for a mixture estimation problem,	helmbold1995a
40	d.p. helmbold, r.e. schapire, y. singer, and m.k. warmuth.	a comparison of new and old algorithms for a mixture estimation problem.	helmbold1997c
72	david d. lewis, robert e. schapire, james p. callan, and ron papka.	training algorithms for linear text classifiers.	lewis1996
72	david d. lewis, robert e. schapire, james p. callan, and ron papka.	training algorithms for linear text classifiers.	lewis1996
72	david d. lewis, robert e. schapire, james p. callan, and ron papka.	training algorithms for linear text classifiers.	lewis1996
72	david d. lewis, robert e. schapire, james p. callan, and ron papka.	training algorithms for linear text classifiers.	lewis1996
72	david d. lewis, robert e. schapire, james p. callan, and ron papka.	training algorithms for linear text classifiers.	lewis1996
69	david haussler, michael kearns, and rob schapire.	bounds on the sample complexity of bayesian learning using information theory and the vc dimension.	haussler1994a
69	david haussler, michael kearns, and robert e. schapire,	bounds on the sample complexity of bayesian learning using information theory and the vc dimension,	haussler1994a
69	david haussler, michael kearns, and robert e. schapire.	bounds on the sample complexity of bayesian learning using information theory and the vc dimension.	haussler1994a
69	david haussler, michael kearns, and robert e. schapire.	bounds on the sample complexity of bayesian learning using information theory and the vc dimension.	haussler1994a

Figura 9 - Ilustração após execução do Algoritmo Guloso

Foram utilizados dados retirados de fontes da internet referentes a livros e artigos científicos publicados. Após a execução do Algoritmo Guloso com 1500 objetos inseridos em duas fontes distintas de dados. Não há uma quantidade de fontes limite, assim como a quantidade objetos. A Figura 9 Ilustra o resultado da associação de tuplas, onde são mostrados os atributos utilizados para a geração do *token* de comparação e o cluster onde cada tupla se encontra.

6 – Conclusão

Este trabalho descreve a implementação passo a passo do Algoritmo guloso proposto por [GRUENHEID; DONG and SRIVASTAVA 14] além de todo o pré-processamento que foi descrito e realizado a fim de colaborar com a visualização completa, desde a descrição até a implementação do algoritmo, de acordo com as conclusões dos trabalhos de [GONDIM 06], [CHEATHAM and HITZLER 12], [BHATTACHARYA; GETOOR and LICAMELE 06], [BAXTER; CHRISTEN and CHURCHES 14], [COVÕES *et al.* 09].

As referências restantes foram utilizadas para descrever algoritmos existentes e para fundamentar definições. A partir deste passo a passo é possível construir o algoritmo, o qual foi conseguido e se encontra no Apêndice A no fim deste trabalho.

6.1 – Contribuições

A principal contribuição deste trabalho é a implementação do algoritmo baseado no algoritmo de Agrupamento de Dados proposto por [GRUENHEID; DONG and SRIVASTAVA 14]. Ao iniciar o processo de desenvolvimento foi percebido que o pré-processamento, que dá todas as condições para o início e manutenção do processo. Foi possível estudar e filtrar as soluções que, pelos resultados apresentados nas conclusões, contribuiriam de forma eficiente para uma implementação que agrupe objetos em tempo hábil para receber e processar uma grande quantidade de atualizações na próxima iteração.

6.2 – Dificuldades Encontradas

A maior dificuldade de todo o ciclo deste trabalho foi a filtragem de experimentos que contribuíssem com a implementação do pré-processamento dos dados, descrito nas seções 4.1, 4.2 e 4.3 e foram o limitante do tempo de implementação do trabalho.

6.3 – Trabalhos Futuros

Um dos trabalhos a serem realizados é a análise da implementação descrita comparada com outros algoritmos existentes, demonstrando o comportamento, em um mesmo cenário, dos algoritmos e definir a qual a melhor finalidade de cada um.

Referências

[GONDIM 06] GONDIM, Flavio Melo. Algoritmo de Comparação de Strings para Integração de Esquemas de Dados. 2006 UFPE, Recife, 2006. Disponível em: <http://www.cin.ufpe.br/~tg/2005-2/fmg.pdf>

[CHEATHAM and HITZLER 12] CHEATHAM, Michelle; HITZLER, Pascal. String Similarity Metrics for Ontology Alignment. 2012, Kno.e.sis Center, Wright State University, Dayton, 2012. Disponível em: <http://knoesis.cs.wright.edu/faculty/pascal/pub/strings-iswc13.pdf>

[NAVARRO 01] NAVARRO, Gonzalo. A Guided Tour to Approximate String Matching. University of Chile. ACM Computing Surveys, Vol. 33, No. 1, pp. 31-88. 2001.

[GRUENHEID; DONG and SRIVASTAVA 14] GRUENHEID, Anja; DONG, Xin Luna; SRIVASTAVA, Divesh. Incremental Record Linkage. Journal Proceedings Of The Vldb Endowment. p. 697-708. May 2014.

[BHATTACHARYA; GETOOR and LICAMELE 06] BHATTACHARYA, Indrajit ; GETOOR, Lise; LICAMELE, Louis. Query-time entity resolution, Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, August 20-23, 2006, Philadelphia, PA, USA

[ELMAGARMID; IPEIROTIS and VERYKIOS 07] ELMAGARMID, Ahmed K.; IPEIROTIS, Panagiotis G.; VERYKIOS, Vassilios S. Duplicate Record Detection: A Survey. Ieee Transactions On Knowledge And Data Engineering. Nj, p. 1-16. January 2007.

[BAXTER; CHRISTEN and CHURCHES 14] BAXTER, Rohan; CHRISTEN, Peter; CHURCHES, Tim. A Comparison of Blocking Methods for Record Linkage. Ibiza: Springer International Publishing, 2014

[COVÕES *et al.* 09] COVÕES, Thiago F. et al. A cluster-based feature selection approach. Salamanca: Springer International Publishing, June 2009

[CHEN *et al.* 12] CHEN, Jie et al. A Learning Method for Entity Matching., Shanghai Key Laboratory Of Trustworthy Computing, Software Engineering Institute, Shanghai, 2012.

[BORAH and BHATTACHARYYA 04] BORAH, B; BHATTACHARYYA, D K. An Improved Sampling-Based DBSCAN for Large Spatial Databases. Dept. Of Inf. Technol., Tezpur University, Tezpur, 2004.

[YAN *et al.* 07] YAN, Su. et al. Adaptive Sorted Neighborhood Methods for Efficient Record Linkage. The Pennsylvania State University, Vancouver, 2007.

[KANUNGO 00] KANUNGO, T. et al. An Efficient k-Means Clustering Algorithm: Analysis and Implementation. American University, Washington, Dc, San Jose, Ca, 2000.

[BERKHIN 07] BERKHIN, Pavel. A Survey of Clustering Data Mining Techniques., Yahoo!, Inc., Sunnyvale, CA, 2007.

[HAN and WAGNER] HAN, Yijie; WAGNER, Robert A. An Ecient and Fast Parallel Connected Component Algorithm**.** Department Of Computer Science, Duke University, Durham, 1988.

[ZHANG an CHEN 03] ZHANG, Dao-qiang; CHEN, Song-can. Clustering incomplete data using kernel-based fuzzy c-means algorithm. Department Of Computer Science And Engineering, Nanjing University Of Aeronautics And Astronautics, Nanjing, 2003.

[SMITE and WATERMAN 81] SMITE, T. F. and WATERMAN M. S. Identification of Common Molecular Subsequences. USA, 1980 Academic Press Inc. (London) Ltd. Reprinted from J . Mol. Biol. 147, 195-197. 1981

Apêndice A

A.1 – Método Merge

```
/**
 * A etapa de Merge é como descrita:
 * 1. For each node v of C, evaluate whether splitting v out generates a better clustering.
 * 2. Upon finding such a node v, create a new cluster C' = {v} and conduct steps 3-4.
 * 3. For each remaining node v' ∈ C evaluate whether moving  v1 to C' obtains a better clustering.
 *    If so, move v' to C' repeat Step 3.
 * 4. Add C and C' to Qc if they are connected to other clusters.
 * @param clus Cluster a ser dividido
 * @return retorna true se tiver havido alteração em algum Cluster.
 **/
public boolean merge(Cluster clus){
        boolean changed = false;
        if(this.clusterList.size() > 0){
            Map<Integer, VertName> verts;
            Map<Integer, Edge> eds;

            Map<Cluster,Cluster> pai2 = new HashMap<Cluster,Cluster>();
            Map<Float,Cluster> candidatos = new HashMap<Float,Cluster>();
            List<Float> medias = new ArrayList<Float>();

            for (int j = 0; (j < clusterList.size()) ; j++) {
               eds = new HashMap<Integer, Edge>(clus.getArestas());
               verts = new HashMap<Integer, VertName>(clus.getVertices());

               //inicializa o novo cluster já com todos os vértices e arestas do cluster 1
               Cluster newClus = new Cluster(verts, eds, clus.getIdKeys());
               Cluster clus2 = this.clusterList.get(j);

               boolean vizinhos = rNeighbor(clus, clus2);
               // se clusters são vizinhos
               if(vizinhos){
                       //adiciona ao novo cluster todos os vértices do cluster 2
                       newClus.InsertVertices(clus2.getVertices());
                       Map<Integer,Edge> edgstToNew = new HashMap<Integer, Edge>();

                       for (Entry<Integer, VertName> vert : clus2.getVertices().entrySet()){
                               for(Entry<Integer, VertName> vertc : newClus.getVertices().entrySet()){
                                       //se existe aresta entre estes dois vertices, adicionar a lista
                                       if(this.grafo.existEdge(vertc.getValue(), vert.getValue())){
                                               Edge edg = this.grafo.getEdge(vertc.getValue(), vert.getValue());
                                               if(edgstToNew.get(edg.getId()) == null){
                                                       edgstToNew.put(edg.getId(), edg);
                                               }
                                       }

                               }
                       }
```

```
                        /*
                         * lista de novos vertices, que antes estava fora de ambos os clusters (intracluster)
                         * agora é inserida no novo cluster, resultado do merge
                         */
                        newClus.insertNewEdges(edgstToNew);
                        if(newClus.getArestas().size()>0)
                                newClus.setAvgPenalty(clusterAvg(newClus));
                        else
                                newClus.setAvgPenalty(0);

                        if(!candidatos.containsKey(newClus.getAvgPenalty())){
                                candidatos.put(newClus.getAvgPenalty(), newClus);
                                medias.add(newClus.getAvgPenalty());
                                pai2.put(newClus, clus2);
                        }

                }
        }
        if(!medias.isEmpty()){
                float menor = Collections.max(medias);
                Cluster vencedor = candidatos.get(menor);
                Cluster clu2Rmv = pai2.get(vencedor);
                if((vencedor.getAvgPenalty() > clu2Rmv.getAvgPenalty() || clu2Rmv.getAvgPenalty() == 0)){
                        //remover o cluster c2
                        //e adicionar o vencedor
                        System.out.println("clusters antes de remover");
                        for(int j = 0; j < this.clusterList.size();j++){
                                System.out.print(this.clusterList.get(j).getId_cluster()+" ");
                        }
                        System.out.println();

                        this.clusterList.remove(clu2Rmv);
                        changed = true;
                        System.out.println("removido cluster de ID: "+clu2Rmv.getId_cluster());
                        this.clusterList.add(vencedor);

                        System.out.println("adicionando cluster de ID: "+vencedor.getId_cluster());
                        System.out.println("clusters apos adicionar " + vencedor.getId_cluster());
                        for(int j = 0; j < this.clusterList.size();j++){
                                System.out.print(this.clusterList.get(j).getId_cluster()+" ");
                        }
                        System.out.println("end");
                }
        }
}
return changed;
```

A.2 – Método Split

```
/**
 * Como definido no artigo do Srivastava:
 * 1. For each neighbor cluster C' of C, evaluate whether merging them generates a better cluster.
 * 2. Upon finding a better clustering, (1) merge C with C', (2) add C U C' generates a better clustering. to Qc,
 * and (3) remove C' from Qc if C' ∈ Qc.
 * @param clus Cluster a ser verificado com a lista quais são seus vizinhos
 * @return retorna true se tiver havido alteração em algum Cluster.
 **/
public boolean split(Cluster clus){

        boolean changed = false;
        boolean temVizinho = false;
        List<Cluster> copy = new ArrayList<Cluster>(this.getClusterList());

        List<Cluster> news = new ArrayList<Cluster>();

        HashMap<Float, VertName> verticeBymedia = new HashMap<Float, VertName>();
        List<Float> medias = new ArrayList<Float>();

        if(clus.getArestas().size() > 0){ //se possui uma aresta ou mais, o cluster pode ser dividido

                VertName aRetirar = null;
                Map<Integer, VertName> conexs = null;//
                for (Map.Entry<Integer, VertName> vert : clus.getVertices().entrySet()){
                        //calcula-se o valor da media de similaridade do cluster sem o vertice que se deseja remover
                        float valor = clusterAvgWithoutVert(clus,vert.getValue());
                        /*
                         * se o valor da media do cluster com o vertice for menor do que o valor sem o vertice
                         * é procedido os passos para o split.
                         if(valor > clus.getAvgPenalty()){
                                aRetirar = vert.getValue();
                                break;
                         }
                         else{
                                //pegamos o subgrafo conectado ao vertice
                                conexs = getVerticesConectados(vert.getValue(), clus);
                                break;
                         }

                 }

                 if(aRetirar != null){//se temos vertice a ser retirado
                        boolean tem = true;
                        List<VertName> retirar = new LinkedList<VertName>();
                        retirar.add(aRetirar);
                        Cluster newClus = new Cluster(aRetirar); //cria-se um novo cluster
                        clus.remove(aRetirar);

                        Map<Integer,Edge> edgstToNew = new HashMap<Integer, Edge>();

                        //procura e insere vertices que fazem arestas com o vertice aRetirar no novo cluster
                        while(tem){

                                VertName tirar = retirar.remove(0);
                                List<VertName> vrts = this.isVertWith(tirar);
                                for(VertName v : vrts){
```

```
                                boolean containVertice = clus.getVertices().containsKey(v.getId());
                                if(containVertice){

                                        Edge edg = this.grafo.getEdge(v, tirar);
                                        /*
                                         * adicionamos os vertices que fazem aresta com
                                         * o vertice da variavel aRetirar no novo cluster
                                         */
                                        newClus.add(v);
                                        retirar.add(v);
                                        /*
                                         *  removemos os vertices adicionados no
                                         *  novo cluster do seu antigo cluster.
                                         *  Ao remover um vertice de um cluster,
                                         *  o metodo retira todas as arestas
                                         *  que o vertice removido faz parte.
                                         */
                                        clus.remove(v);
                                        edgstToNew.put(edg.getId(), edg);

                                }
                        }
                        if(retirar.isEmpty()){
                                tem = false;
                        }
                }
                /*
                 * arestas que fazem parte do grafo e que o vertice da variavel aRetirar
                 * se encontra, são inseridos no novo cluster
                 */
                newClus.insertNewEdges(edgstToNew);
                updateClusAvg(clus, newClus);
                temVizinho = this.hasNeighbor(newClus);
                /*
                 * se o novo cluster formado possui vizinho, ele é colocado na lista
                 * que será iterada para sofrer novas operações. Caso contrário
                 * colocamos na lista de clusters que sera retornada no fim do algoritmo
                 * de agrupamento
                 */
                if(temVizinho){
                        news.add(newClus);
                }else{
                        this.lg.add(newClus);
                }
                changed = true;
        }

    }

    boolean add = copy.addAll(news);
    if(changed)
            this.setClusterList(copy);
    return changed;
}
```

A.3 – Método Move

```
/**
 * 1. For each neighbor cluster C' of  C, do Steps 2-3.
 * 2. For each node v' ∈ C that is connected to C' and for each v' ∈ C' connected to C,
 *    evaluate whether moving v to the other cluster generates a better clustering.
 *    Upon finding such a node v, move it to the other cluster.
 * 3. Repeat Step 2 until there is no more node to move. Then, (1) add the two new clusters to Qc,
 *    and (2) dequeue C' if C'∈Qc.
 * @param clus Cluster a ser verificado com a lista quais são seus vizinhos.
 * @return retorna true se tiver havido alteração em algum Cluster.
 */
public boolean move(Cluster clus){System.out.println("move");
        boolean changed = false;

        if(clus.getId_cluster() == 3){
                System.out.println("esse da bronca");
        }

        if(this.clusterList.size() > 0){

                Map<Integer, VertName> verts;
                Map<Integer, Edge> eds;
                Map<Integer, VertName> verts2;
                Map<Integer, Edge> eds2;

                Map<Integer,Cluster> pais2 = new HashMap<Integer,Cluster>();
                Map<Float,List<Cluster>> filhos = new HashMap<Float,List<Cluster>>();

                Map<Float,Cluster> candidatos = new HashMap<Float,Cluster>();
                List<Float> medias = new ArrayList<Float>();
                boolean hasChangedc1 = false;

                for (int j = 0; (j < clusterList.size()) ; j++) {
                        List<Cluster> sons = new ArrayList<Cluster>();

                        eds = new HashMap<Integer, Edge>(clus.getArestas());
                        verts = new HashMap<Integer, VertName>(clus.getVertices());
                        Cluster newClus1 = new Cluster(verts, eds, clus.getIdKeys());

                        Cluster clus2 = this.clusterList.get(j);
                        eds2 = new HashMap<Integer, Edge>(this.clusterList.get(j).getArestas());
                        verts2 = new HashMap<Integer, VertName>(this.clusterList.get(j).getVertices());
                        Cluster newClus2 = new Cluster(verts2, eds2, clus2.getIdKeys());

                        if(newClus1.getArestas().size()>0)
                                newClus1.setAvgPenalty(clusterAvg(newClus1));
                        else
                                newClus1.setAvgPenalty(0);

                        if(newClus2.getArestas().size()>0)
                                newClus2.setAvgPenalty(clusterAvg(newClus2));
                        else
                                newClus2.setAvgPenalty(0);

                        boolean vizinhos = rNeighbor(newClus1, newClus2);

                        if(vizinhos){
                                boolean atualizou = false;
                                List<Edge> edges = getArestasBtwClusters(clus, clus2);
```

```
                        if(newClus1.contais(v1) && ((newClus2.getAvgPenalty() == 0)
                        || (edg.getWeight() > newClus1.getAvgPenalty() || edg.getWeight() < newClus2.getAvgPenalty()))){
                        //e ou v1 tem q ser maior que media de newClus1 ou tem q ser menor que media de newClus2

                          newClus2.remove(v2); //remove edg.getElemone() de clus1
                          newClus1.InsertVertice(v2); //add edg.getElemone() de clus2
                          newClus1.insertNewEdge(edg); //adiciona edg ao newCluster2
                          atualizou = true;

                        }else if(newClus1.contais(v2) && ((edg.getWeight() < newClus1.getAvgPenalty()
                                                          || edg.getWeight() > newClus2.getAvgPenalty()))){
                        //e ou v2 tem q ser maior que media de newClus1 ou tem q ser menor que media de newClus2

                          newClus1.remove(v2); //remove edg.getElemtwo() de clus1
                          newClus2.InsertVertice(v2); //adiciona edg.getElemtwo() de clus2
                          newClus2.insertNewEdge(edg); //adiciona edg ao newCluster2
                          atualizou = true;

                        }
                      if(atualizou){
                        if(newClus1.getArestas().size()>0)
                          newClus1.setAvgPenalty(clusterAvg(newClus1));
                        else
                          newClus1.setAvgPenalty(0);
                        if(newClus2.getArestas().size()>0)
                          newClus2.setAvgPenalty(clusterAvg(newClus2));
                        else{
                          newClus2.setAvgPenalty(0);
                          }
                        //atualiza a penalidade dos clusters newClus1, newClus2
                      }
                    }
                   if(atualizou){
                     float media = (newClus1.getAvgPenalty()+newClus2.getAvgPenalty())/2;
                     medias.add(media);

                     sons.add(newClus1);
                     sons.add(newClus2);
                     filhos.put(media, sons);
                     pais2.put(newClus1.getId_cluster(), clus2);
                     // adiciona no map pai2 ambos os clusters
                   }
                  }
            }
            // depois do processo aqueles que tem a menor média é de fato adicionados
            // no this.clusterList e o seu pai2 sera removido junto com o clus1
            if(!medias.isEmpty()){
              System.out.println("clusters antes ");
              for(int j = 0; j < this.clusterList.size();j++){
                System.out.print(this.clusterList.get(j).getId_cluster()+" ");
              }
              System.out.println();

              float menor = Collections.max(medias);
              List<Cluster> sons = filhos.get(menor);
              Cluster clus2 = pais2.get(sons.get(0).getId_cluster());

              this.clusterList.remove(clus2);
              if(sons.get(1).getVertices().size()>0){
                this.clusterList.add(sons.get(1));
              }
              if(sons.get(0).getVertices().size()>0){
                this.clusterList.add(sons.get(0));
              }
              changed = true;
            }
          }
          if(changed){
            System.out.println("clusters depois ");
            for(int j = 0; j < this.clusterList.size();j++){
              System.out.print(this.clusterList.get(j).getId_cluster()+" ");
            }
            System.out.println();
          }
        return changed;
}
```