Relatório do projeto 1 de Tópicos Avançados em Algoritmos

Professores Paulo Fonseca e Nivan Ferreira

Equipe:

- Gabriela Vilela Heimer (gvh)
- José Gabriel Silva Pereira (jgsp2)

1. Identificação

A equipe é composta por:

- Gabriela Vilela Heimer (gvh)
- José Gabriel Silva Pereira (jgsp2)

Os algoritmos escolhidos foram CMin e KMV, onde Gabriela foi responsável por implementar o KMV e José o CMin.

2. Implementação

Como mencionado anteriormente, os algoritmos implementados foram o CMin e KMV. Abaixo vamos entrar em mais detalhes sobre a implementação de cada um deles

CMin

O Count Min tem como objetivo estimar para estimar pontualmente os "pesos totais" de uma stream de dados. O sketch é suposto fornecer aproximações tais que $P[|\hat{W}x - Wx| \geq \varepsilon W] \leq \delta$, onde:

- **Wx** → indica o peso total associado a x na stream.
- **$\hat{W}x$** → indica o peso total estimado associado a x pelo algoritmo.
- **ε** → indica o limite do erro relativo aceitável.
- **W** → indica a soma de todos os pesos na stream.
- **δ** → indica o limite superior para a probabilidade de o erro calculado ser menor do que o erro aceitável (**εW**)

Os parâmetros **ε** e **δ** fazem parte das variáveis definidas pelo usuário (eps e delta).

Para a implementação deste algoritmo, foi necessária a utilização de estruturas bidimensionais, tanto para armazenar as estimativas quanto para armazenar as variáveis que definem as funções hash e estruturas unidimensionais dinâmicas para armazenar a sequência de ids a serem estimados ao final. Para tal, foi preferida a utilização de *vectors* (arrays dinâmicos na linguagem C++), que são simples de usar e aproximadamente tão eficientes quanto arrays estáticos.

Com essa estrutura, foi possível inicializar cada sketch com complexidade de tempo e espaço iguais a O(**t*k**), onde **t** e **k** são calculados a partir dos parâmetros dados da seguinte forma:

- **$t = 2/\varepsilon$**
- **$k = \lceil \log(1/\delta) \rceil$**

KMV

O KMV tem como objetivo estimar a quantidade de elementos distintos em uma stream de dados, fornecendo uma aproximação tal que $P[|\hat{W}x - Wx| \geq \varepsilon W] \leq \delta$.

Ao rodar o comando do KMV são passados 3 parâmetros para o código:

- **target** → indica a coluna do csv a ser usada como stream de dados. O valor default para esse parâmetro é 0.
- **eps** → indica a margem de erro para a estimativa do KMV. O valor default para esse parâmetro é 0.05.
- **delta** → indica a probabilidade de que o resultado obtido esteja fora da margem de erro. O valor default para esse parâmetro é 0.01.

Nesta implementação foram usadas as seguintes variáveis e estruturas de dados (fora as mencionadas acima):

- **numSketches** → indica a quantidade de vezes que o KMV será rodado para que ao fim de tudo nós possamos fazer o truque da mediana. Ele é calculado com base do **delta** passado como parâmetro.
- **k** → indica o tamanho da estrutura que vai conter os menores valores da stream. Ele é calculado com base no **eps** passado como parâmetro.
- **P** → número primo usado para tirar o módulo na função de hash. O valor dele sempre é $2^{63} - 1$. (também é o **R** do KMV, ou seja, o número máximo do universo numérico em que os números são mapeados.)
- **a**, **b** → parâmetros da função de hash. Cada instância de sketch que é gerada tem os valores de **a** e **b** aleatorizados, de forma que 0 <= **a**,**b** <= **P**.
- **hashValues** → set usado para guardar os **k** menores valores hasheados da stream. A escolha dessa estrutura de dados foi porque com ela é possível manter os valores ordenados e também checar a ocorrência de um valor no set em O(logn) tempo.

Coisas em comum

Para ambos os algoritmos, foram utilizados 3 classes extras principais:

- **Hasher** → responsável por gerar funções de hash do tipo requisitado nos algoritmos (**h(x) =** (**a** * x + **b**) % **P**). Para tal foi acordado entre os membros que seria utilizado o primo de mersenne $P = 2^{61} - 1$, pois ele é grande o suficiente para que os requisitos probabilísticos dos algoritmos fossem executados. Vale ressaltar que a função de multiplicação modular usada foi copiada de uma lib de programação competitiva e foi usada para evitar overflow, já que os valores de **a** e **b** tendem a ser muito grandes. Essa classe também foi responsável por transformar os ids (pegos no dataset como strings) para números inteiros, através do algoritmo de Rabin Karp.

- **ArgsReader** → responsável por atualizar os parâmetros de cada algoritmo utilizando os argumentos recebidos pela linha de comando. Para esta classe, foram utilizadas estruturas *set* e *queue* de C++. A utilização dessas estruturas adicionais não influenciou na eficiência de tempo e espaço dos algoritmos, pois a quantidade de elementos armazenados nessas estruturas foram somente tão grandes quanto a quantidade de argumentos passados no comando executado.
- **CSVReader** → responsável por ler arquivos .csv em forma de stream e retornar os dados requisitados pelo algoritmo principal.

3. Testes e Resultados

3.1 Descrição do ambiente de testes:

Count Min: Testado em ambiente com processador Intel® Core™ i7-8750H CPU @ 2.20GHz × 12, e 16GB RAM

KMV: Testado em ambiente com processador Intel® Core™ i7-8565U CPU @ 1.80GHz × 8 e 8GB RAM

3.2 Descrição dos experimentos realizados

Count Min: Como experimento para o Count Min, foi realizado uma série de execuções do algoritmo em cima do dataset *network_flows.csv* utilizando a primeira coluna (*Flow.ID*) como parâmetro para identificação e a quarta coluna (*Protocol)* como parâmetro para os pesos. Essas execuções foram feitas de forma a obter as estimativas dos pesos dos Flow.ID identificados pelos números de 1 a 5, e com os parâmetros **ε** e **δ** variando nos intervalos [0.01, 0.2] e [0.01, 0.1] respectivamente. Com esse experimento foi possível relacionar a qualidade das estimativas com os parâmetros **ε** e **δ.**
A planilha com os resultados das execuções pode ser visualizada aqui.

KMV: Como experimento foram realizadas diversas execuções do algoritmo, totalizando 900 execuções, em cima do dataset network_flows.csv. Para todas as colunas do csv, o algoritmo foi rodado com os parâmetros **eps** e **delta** variando de 0.05 a 0.50, aumentando de 0.05 em 0.05. Para cada uma dessas execuções foram coletados os seguintes dados

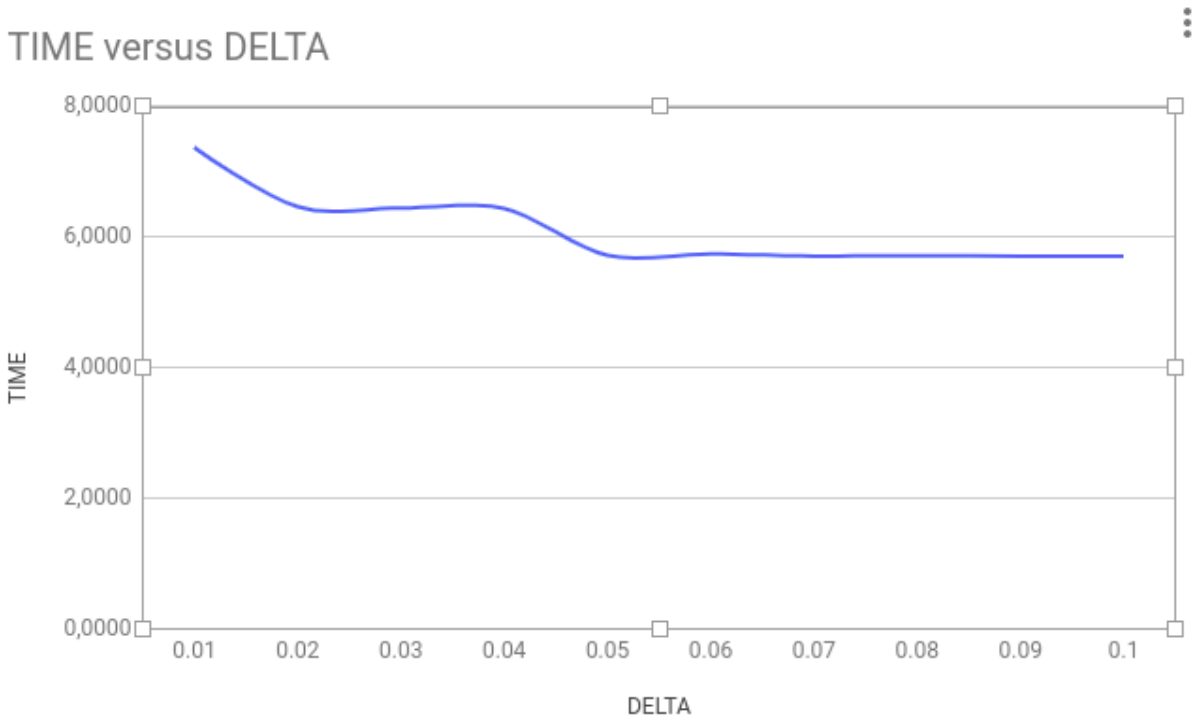
- A estimativa do algoritmo
- O valor real da quantidade de elementos únicos na stream
- O menor e maior valor que poderia ser estimado dentro da margem de erro e se a estimativa estava entre esses valores (true or false)
- A quantidade de sketches gerado para fazer o truque da mediana
- O valor de **k** escolhido
- O tempo de execução do algoritmo

A planilha com os resultados das execuções pode ser visualizada aqui.

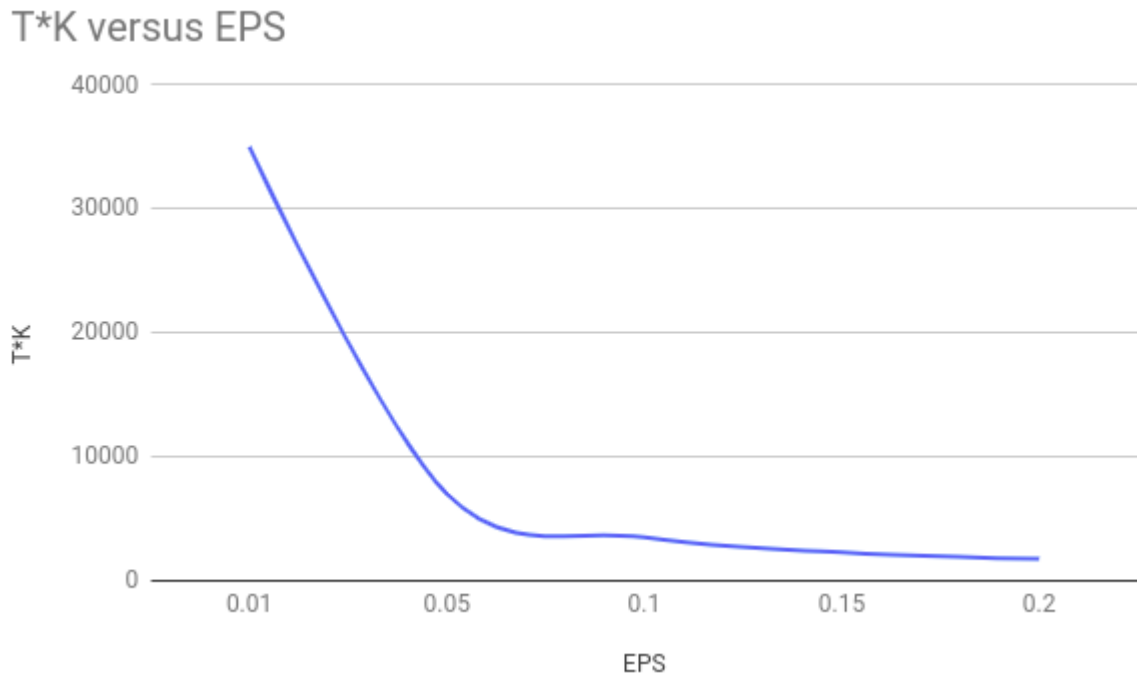
3.3 Dados e resultados obtidos

Count Min:

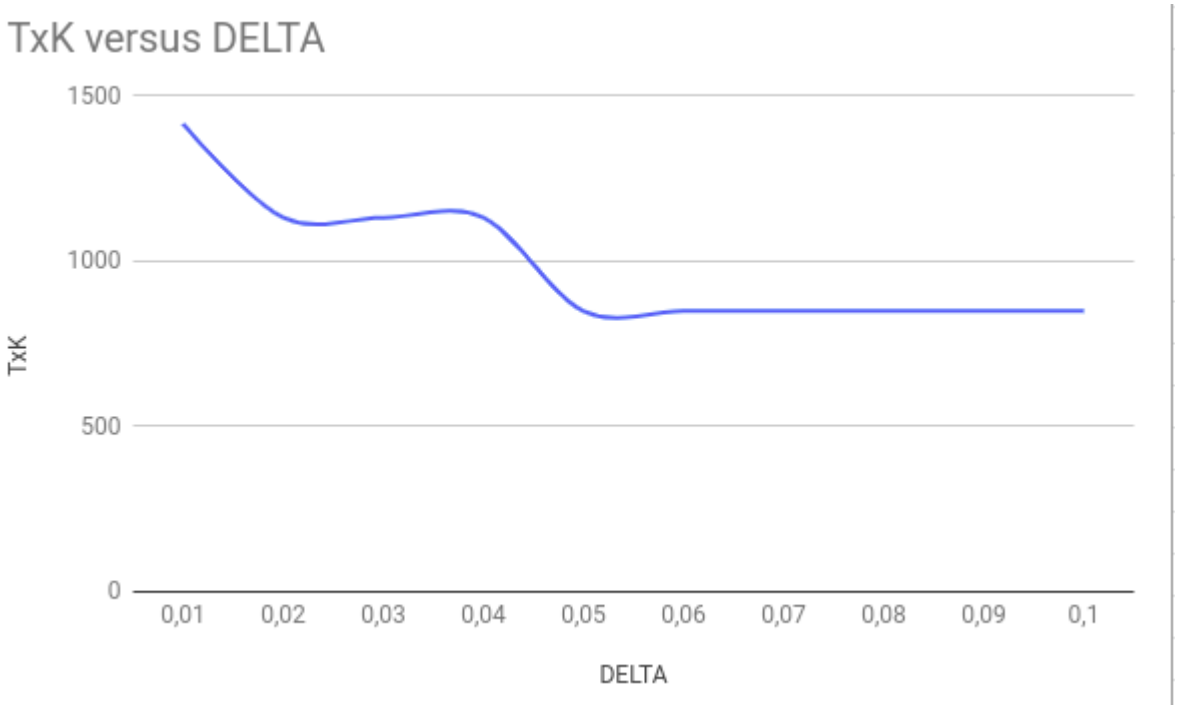
1. Gráfico do tempo de execução de acordo com a variação de delta



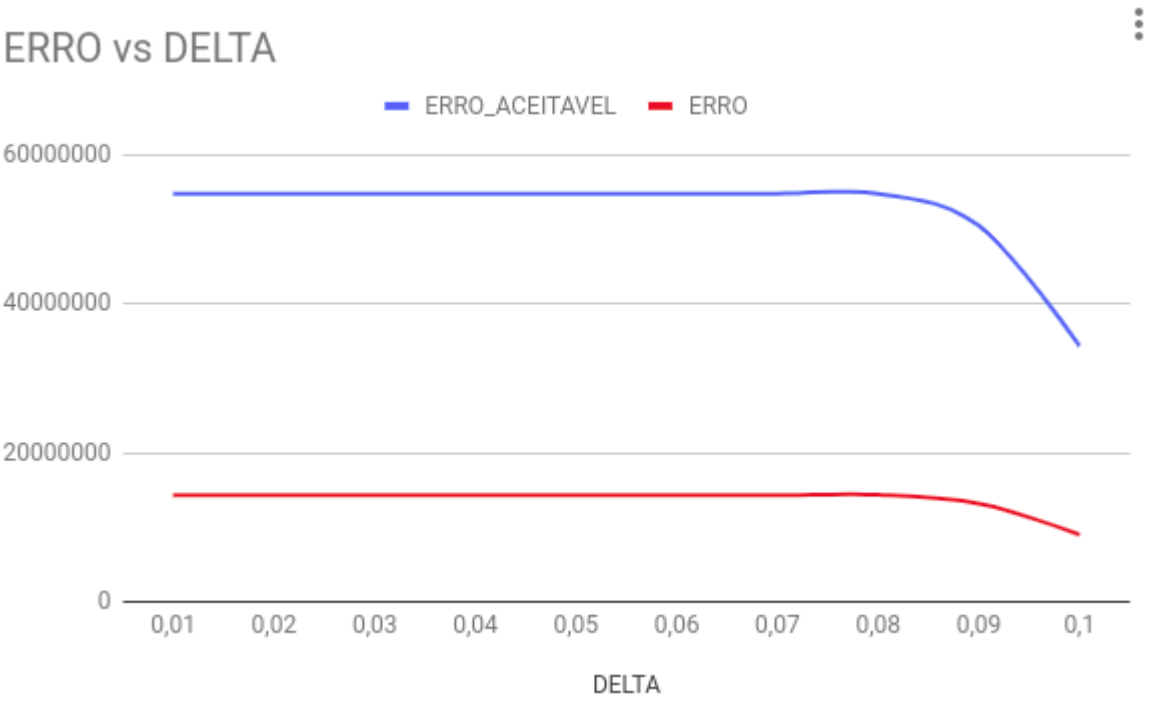
2. Gráfico relacionando a complexidade de espaço com o eps



3. Gráfico relacionando a complexidade de espaço com o delta



4. Gráfico relacionando os erros com o delta

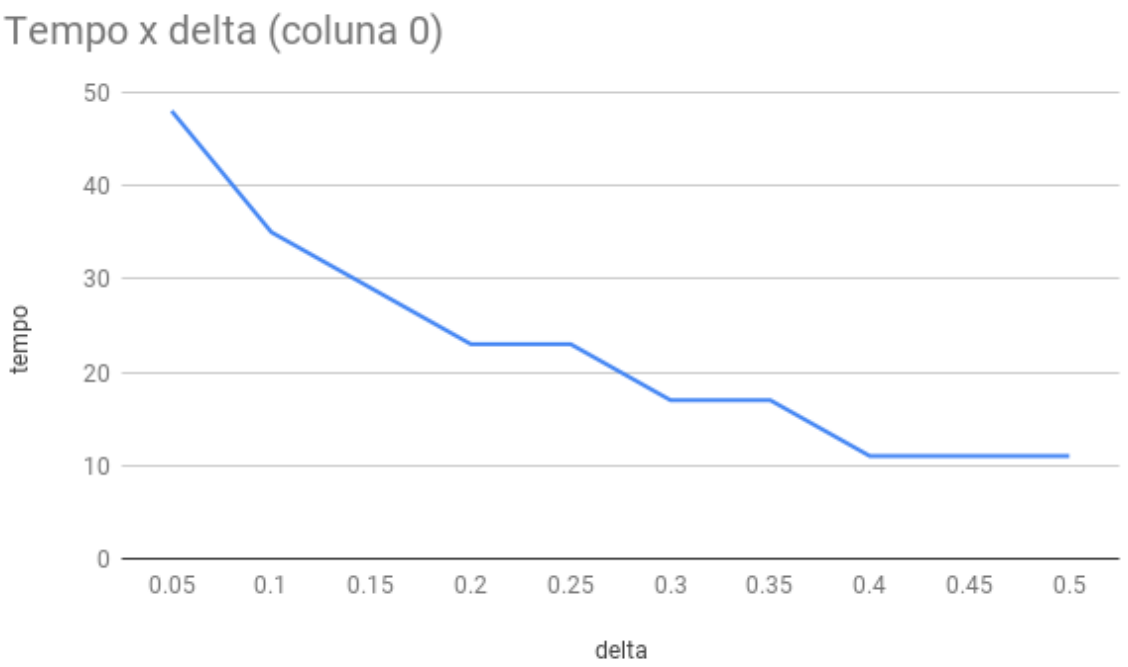


KMV:

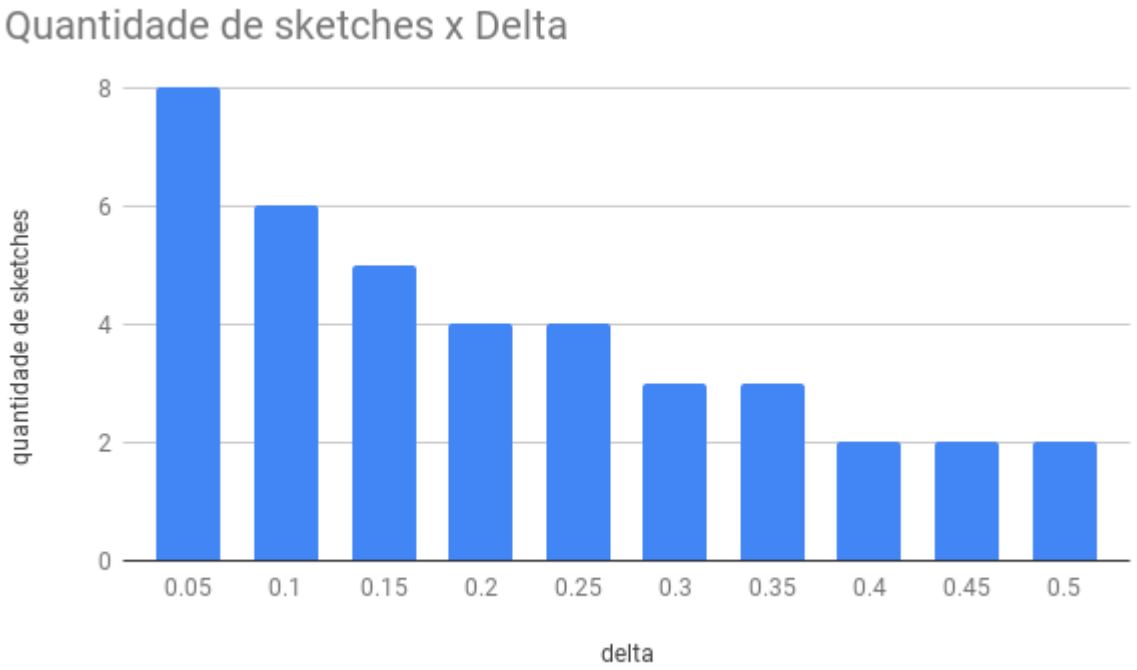
1. Tabela da frequência de execuções com a estimativa dentro e fora do esperado

good estimate	1
0	3
1	897

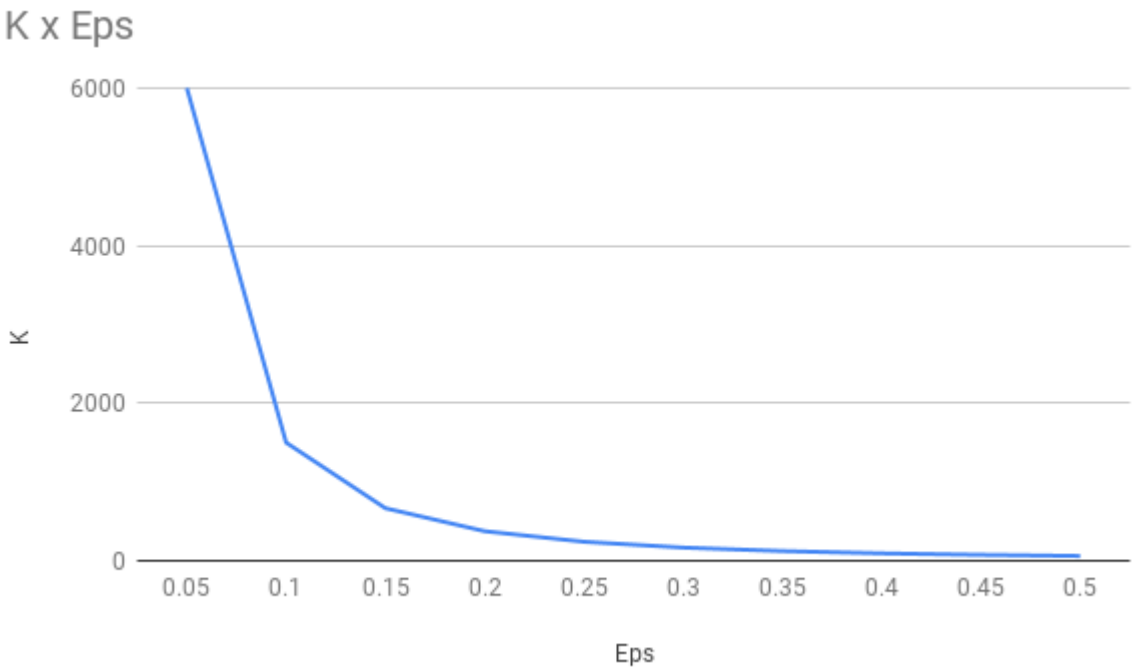
2. Gráfico do tempo de acordo com os valores do delta, para a coluna 0



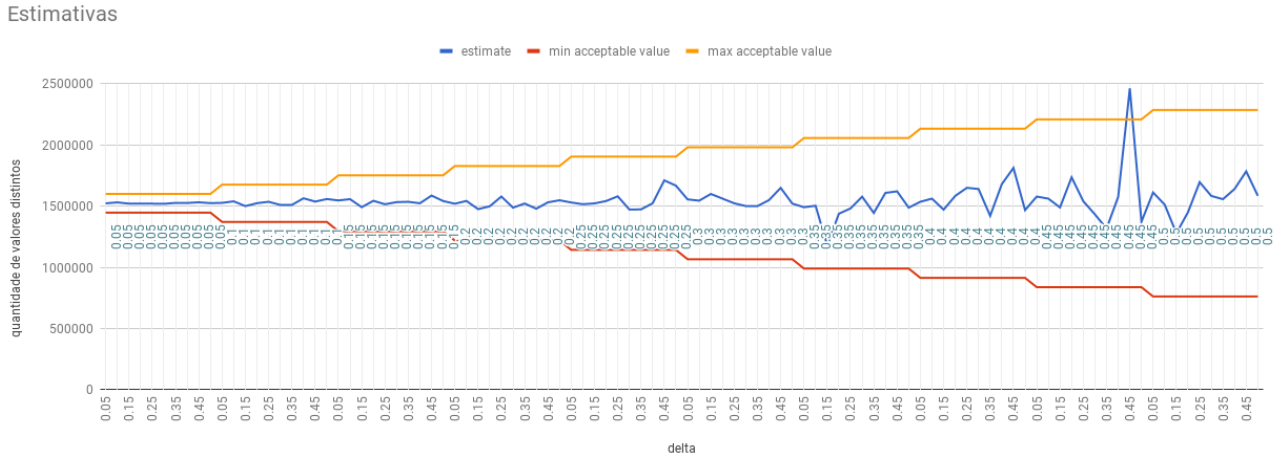
3. Gráfico da quantidade de sketches de acordo com o delta, para a coluna 0



4. Gráfico do K de acordo com o Eps, para a coluna 0



5. Gráfico das estimativas de acordo com o delta e o eps, para a coluna 0



Para ver a imagem em melhor definição clique aqui.

3.4 Discussão dos resultados e conclusão

Count Min:

Como podemos observar nos gráficos **1**, **2** e **3** da seção de gráficos do Count Min, o aumento dos parâmetros **eps** e **delta** influenciaram diretamente no tempo de execução, que teve sua média em torno de 6.5s. Isso era esperado, pois nesse algoritmo o tempo de execução está diretamente relacionado com a complexidade de espaço utilizado. Quanto maior a margem de erro (**eps**) menos espaço o algoritmo precisa para ser executado e, consequentemente, menor vai ser o tempo de execução. O mesmo ocorre para o parâmetro **delta**, cujo aumento significa aumentar a probabilidade de um erro mais discrepante acontecer, o que é diretamente refletido no menor uso de memória pelo algoritmo.

Além disso, pode-se observar no gráfico **4** que em todos os testes registrados, o erro obtido foi muito menor do que o erro aceitável. Assim, concluímos que o algoritmo Count Min funcionou como esperado.

KMV:

Aqui iremos discutir sobre o que está sendo mostrado nos gráficos além de outros comentários pontuais.

O gráfico **1** mostra a quantidade de estimativas cujo resultado ficou dentro da margem de erro, e a quantidade de estimativas cujo resultado ficou fora da margem de erro. Nele podemos perceber que dos 900 testes executados, apenas 3 ficaram fora da margem de erro.

Os seguintes gráficos foram feitos a partir dos dados obtidos com os testes em que o valor da **target** era 0.

O gráfico **2** mostra o tempo de acordo com o delta. O delta é o maior influenciador do tempo que o algoritmo leva para ser executado, pois a partir dele, é calculada a quantidade de sketches que serão gerados para fazer o truque da mediana (**numSketches**). Neste gráfico, o valor do **eps**, que não está sendo mostrado, era 0.05. Porém, para os diferentes valores de **eps**, o resultado do

gráfico de **delta** x **tempo** tem valores similares, mantendo a variância muito parecida. Como esperado, podemos ver que quanto maior o **delta**, menor o **tempo** tomado. Afinal, quanto maior a probabilidade de erro, menos sketches precisam ser gerados para o truque da mediana.

O gráfico **3** mostra a quantidade de sketches usada (**numSketches**) para fazer o truque da mediana, de acordo com o **delta**. A quantidade de sketches depende estritamente do valor do **delta**, por isso achamos interessante a adição desse gráfico. Corroborando com o gráfico **2**, podemos ver que, quanto maior o delta, menor a quantidade de sketches.

O gráfico **4** indica o valor de **k** de acordo com o **eps**. No algoritmo, o valor do **k** depende estritamente do valor do **eps**, portanto, independente do valor do **delta**, esse gráfico se mantêm igual. Como esperado, quanto maior o **eps**, menor é o **k** escolhido. Isso faz sentido pois, quanto maior a margem de erro, menos precisa é a nossa estimativa esperada, logo podemos diminuir o **k** e poupar memória.

O gráfico **5** mostra a estimativa do KMV, indicando se ela ficou dentro ou fora da margem de erro. O eixo x indica o valor do **delta** e no meio do gráfico podemos diversos números, que são um marcador indicando o **eps**. Para cada intervalo [0.5, 0.45] de valores de **delta**, podemos ver que a margem de erro vai aumentando, pois o **eps** está crescendo para a direita. É possível perceber que no geral a estimativa ficou muito próxima do valor real, com a exceção de um pico na estimativa no lado direito do gráfico.

Fora a análise dos gráficos, gostaríamos de pontuar que o tempo médio de execução do algoritmo foi de 19 segundos.